МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное
учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Языки программирования и компиляторы — 2017

Труды конференции
(3–5 апреля 2017 г.)

Под редакцией Д. В. Дуброва

Ростов-на-Дону
2017

УДК 004.4(063)
ББК 32.973я43
Я411

Конференция проводится при финансовой поддержке ООО «СиПроВер» (генеральный директор Е. А. Рыжков), разработчика статического анализатора кода PVS-Studio для поиска ошибок в программах на C, C++ и C# на Windows и Linux (https://www.viva64.com)

Сборник издан при поддержке Российского фонда фундаментальных исследований (проект 17-01-20061)

Я411 **Языки программирования и компиляторы — 2017** : труды конференции / Южный федеральный университет ; под ред. Д. В. Дуброва. — Ростов-на-Дону : Издательство Южного федерального университета, 2017. — 282 с.
ISBN 978-5-9275-2349-8

В сборнике представлены труды конференции «Языки программирования и компиляторы» (Южный федеральный университет, Институт математики, механики и компьютерных наук им. И. И. Воровича, 3–5 апреля 2017 г.).

Конференция посвящается памяти известного ростовского учёного Адольфа Львовича Фуксмана, основоположника технологии вертикального слоения программ, заведующего Вычислительным центром РГУ с 1964 по 1978 годы.

УДК 004.4(063)
ББК 32.973я43

ISBN 978-5-9275-2349-8

Содержание

Штрихи к биографии А. Л. Фуксмана

Налбандян Ю. С.[1], ysnalbandyan@sfedu.ru
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Штрихи к биографии... Именно так. Ибо Адольф Львович Фуксман настолько легендарен, настолько ярок, что каждый абзац, каждое предложение этой короткой заметки можно превратить в отдельную статью.

Адольф Львович Фуксман родился 4 июля 1937 года в Запорожье. Военные годы он провёл в эвакуации на Урале, а в 1944 его семья переехала в Жданов (Мариуполь). Здесь, фактически, прошло его детство, здесь он увлёкся математикой, а летом 1954 года поступил на физмат Ростовского госуниверситета. Это было удивительное время, возможно, одно из самых лучших на факультете. Активно работали преподаватели, пережившие тяжелые военные годы (М. Г. Хапланов, С. Я. Альпер, Е. Л. Литвер, К. К. Мокрищев, Н. М. Несторович, А. А. Сукало, Н. Н. Рожанская), в 1953 году из Казани вместе с группой талантливых учеников приехал профессор Ф. Д. Гахов, в конце 40-х — начале 50-х гг. в состав кафедры теоретической механики влились приехавшие из Москвы молодые кандидаты наук И. И. Ворович, Н. Н. Моисеев, Л. А. Толоконников. В этой обстановке не мог не раскрыться талант увлечённого и яркого студента. Юноша успешно учился, активно занимался в студенческом научном обществе, увлекался музыкой и спортом, не чуждался общественной работы (был членом комитета комсомола), а в 1957 году вместе с другими физматовцами работал на целинных землях в Казахстане.

Научные увлечения А. Л. Фуксмана в то время были связаны с вопросами приближения функций. По воспоминаниям однокурсника и друга, ныне профессора В. П. Захарюты, задачу А. Л. Фуксману о приближении с соблюдением нулевых граничных условий (обобщение

результатов И. Ю. Харрик) поставил Иосиф Израилевич Ворович, а непосредственное руководство этой работой осуществлял Семен Яковлевич Альпер, один из ведущих лекторов и учёных факультета. В аспирантуре (1959–1962 гг.) официальным руководителем Адольфа Львовича был Михаил Григорьевич Хапланов.

В 1960–1962 гг. в Докладах Академии наук СССР по представлению академика В. И. Смирнова были опубликованы три статьи А. Л. Фуксмана. Результаты, изложенные в работах «Приближение функций с сохранением однородных граничных условий» (1960, т. 34, № 2) и «О приближении функций многих переменных с сохранением граничных условий» (1961, т. 141, № 5), существенно опирались на новый метод продолжения функции с сохранением дифференциальных свойств из области, граница которой содержит особые точки определённого вида. В статье «Локальные свойства некоторых аппроксимационных операторов» (1962, т. 142, № 3) обсуждался вопрос о том, в какой мере быстрота сходимости данной последовательности аппроксимирующих функций в некоторой точке определяется свойствами приближаемой функции в некоторой окрестности этой точки.

В 1962 году в Днепропетровском государственном университете А. Л. Фуксманом была досрочно защищена диссертация «Приближение функций многих действительных переменных с сохранением однородных условий на границе области». Пока новоиспечённый кандидат наук успешно работал старшим преподавателем в Ростовском государственном пединституте, в университете разворачивались бурные события, связанные с развитием вычислительного центра (см., например, `http://50.uginfo.sfedu.ru/history.htm`). В этот момент Адольф Львович полностью меняет тематику своих исследований. Его математические работы будут появляться в научных журналах вплоть до 1968 года (Сибирский математический журнал — 1964, Успехи математических наук — 1965, Доклады АН СССР — 1968), однако новым и основным увлечением учёного становятся вопросы, связанные с развитием вычислительной техники — трансляторы, языки программирования, системное программирование.

Его принимают на должность старшего научного сотрудника в ВЦ РГУ, в ноябре 1965 году по конкурсу избирают заведующим Вычислительным центром. Как признавали позднее коллеги, с этого момента началось активное научное становление ВЦ и были заложены основы таких направлений исследований как системное и теоретическое программирование, прикладное программирование, разработ-

ка и построение специализированных средств вычислительной техники. А. Л. Фуксман организует знаменитые школы-семинары, постоянно публикует свои статьи, активно создаёт собственную научную школу. Его технология расслоённого программирования и исследования по слаборазделённым грамматикам завоёвывают широкое признание.

И при этом — постоянная преподавательская деятельность. В 1971 году на мехмате открылось отделение прикладной математики. Созданная кафедра вычислительной математики, которую возглавил И. Б. Симоненко, уже в 1972 году делится на три — кафедра алгебры и дискретной математики (заведующий — профессор И. Б. Симоненко), кафедра вычислительной математики (заведующий — профессор В. И. Юдович), кафедра математического обеспечения ЭВМ и АСУ (заведующий — доцент Г. В. Аржанов). В основном, именно на кафедре Г. В. Аржанова, на полставки, увлечённо работал Адольф Львович. Как заметил в своих воспоминаниях И. И. Голянд, «сам он учился на «пятёрки» и, преподавая, требовал такой же учёбы у студентов». А. Л. Фуксман вёл курсы на мехмате, читал вычислительную математику на других факультетах (в частности, по воспоминаниям И. И. Голянда, на геофаке), руководил вычислительной практикой — и очаровывал всех слушателей своими профессионализмом, увлечённостью, энтузиазмом.

Он готовил к печати монографию «Технологические аспект создания программных систем», был полон творческих и организационных планов, но жизнь распорядилась иначе. После трагической гибели А. Л. Фуксмана в январе 1978 года его дело продолжили друзья и коллеги. Прежде всего они завершили работу над монографией, в предисловии к которой академик А. П. Ершов напишет: «Адольфу Львовичу удалось подвести первый итог его успешной научной и конструкторской работы, выдвинувшей его в ряды ведущих системных программистов. Память о нем навсегда сохранится в сердцах его друзей и товарищей по профессии, и его книга — одна из первых книг в СССР по системному программированию — долго послужит развитию предмета.»

В 2008-м году заведующий кафедрой системного программирования СПбГУ профессор А. Н. Терехов вспомнит о том, как «академик Ершов на одном из собраний нашей группы по Алголу 68 сказал, что мы потеряли большого учёного и должны сделать все, чтобы его школа не развалилась».

Что ж, сегодня можно утверждать, что идеи А. Л. Фуксмана по-

прежнему актуальны, а его ученики и последователи успешно продолжают и развивают дело Учителя.

А. Л. Фуксман и его роль в развитии Вычислительного центра Ростовского государственного университета

Муратова Г. В.[1], muratova@sfedu.ru
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Важная страница жизни Адольфа Львовича Фуксмана связана с Вычислительным центром Ростовского государственного университета.

Вычислительный центр РГУ начинает свою историю с октября 1958 г. Тогда Министерство высшего и среднего специального образования РСФСР выделило Ростовскому госуниверситету одну из первых в СССР серийных электронно-вычислительных машин «Урал 1». В штат лаборатории при кафедре математического анализа были приняты первые 4 сотрудника. День принятия их на работу — 4 октября 1958 г. считается днём основания ВЦ РГУ. 21 января 1959 г. в министерстве был подписан приказ об организации вычислительного центра на правах лаборатории при кафедре мат. анализа физмата РГУ. Штат ВЦ был 9 человек. Это был первый ВЦ на юге России и Северном Кавказе. Научное руководство ВЦ было поручено профессору Ф. Д. Гахову, общее — доцентам Л. А. Чикину, Е. Л. Литверу и М. М. Чепиноге. С первых же дней существования ВЦ его деятельность не ограничилась учебной работой, уже в 1960 г. выполнялись научно-исследовательские работы и первые заказы для промышленности.

В апреле 1961 года Лев Александрович Чикин возглавил новую кафедру — вычислительной математики. ВЦ был закреплён за этой кафедрой как учебный счётно-аналитический центр.

По мере роста объёма работ возникла необходимость увеличения машинного времени, и в конце 1962 г. ВЦ получил новую ЭВМ —

«Минск 12». А 30 мая 1963 г. был подписан приказ «Об утверждении структуры вычислительного центра РГУ». Этим приказом ВЦ выделялся в самостоятельное научно-исследовательское подразделение со штатом в 45 человек.

В январе 1964 г. руководителями трёх лабораторий ВЦ стали старший научный сотрудник, кандидат физ.-мат. наук Адольф Львович Фуксман, старшие инженеры Евгений Георгиевич Ротов и Игорь Анатольевич Николаев. Был утверждён совет ВЦ во главе с Львом Александровичем Чикиным. В том же 64-ом заведующим ВЦ был назначен Адольф Львович Фуксман.

Адольф Львович впервые в Ростове стал заниматься актуальными в то время проблемами системного и теоретического программирования. Он открыл и исследовал класс так называемых слаборазделённых грамматик, которые были положены в основу всех созданных в РГУ трансляторов и систем автоматизации их построения. Эти работы вывели Фуксмана в ряды лидеров теоретического и системного программирования СССР. Важными практическими результатами его научной школы стали трансляторы с языков Алгол 60, Фортран и Симула 67.

В этот период выросли потребности использования вычислительных машин у нейрокибернетиков, физиков, химиков. Значительно возрос объем и собственных научных исследований ВЦ. Большую помощь в развитии математического потенциала ВЦ оказали ведущие учёные механико-математического факультета И. И. Ворович, В. И. Юдович, С. В. Жак и И. Б. Симоненко, под руководством которых проводились научные исследования, были защищены первые кандидатские диссертации.

Параллельно с развитием научного потенциала ВЦ росла его материальная база. В 1966 г. была запущена ЭВМ «Урал 11М» — первая машина второго поколения в Ростовском университете, которая стала полигоном в работах по системному программированию.

В 1972 году началось строительство здания ВЦ в Западном микрорайоне Ростова-на-Дону, а в 1975 г. в новом корпусе была запущена мощная ЭВМ — «БЭСМ 6». Это позволило приступить к созданию регионального вычислительного центра коллективного пользования.

С 1973 года Вычислительный центр РГУ начал проводить ежегодные всесоюзные школы-семинары по системному и теоретическому программированию. Их основателем и научным руководителем был Адольф Львович Фуксман, чей острый ум, человеческое обаяние, широта научных интересов и постоянная готовность к научному контакту

ценились многими учёными страны. Поэтому школы неизменно собирали цвет советской программистской науки.

А.Л. Фуксман — автор оригинального подхода к технологии создания больших программных комплексов. Своё выражение эти идеи нашли в монографии «Технологические аспекты создания программных систем», вышедшей в свет уже после гибели А. Л. Фуксмана в 1978 году. Книга на долгое время стала бестселлером для программистов и памятником этому выдающемуся учёному и практику.

Одно из главных дел жизни А. Л. Фуксмана – развитие Вычислительного центра продолжили его коллеги и ученики.

Е. Г. Ротов руководил ВЦ РГУ с 1978 по 1986 гг. Он был одним из первых сотрудников ВЦ, к моменту назначения являлся, фактически, главным инженером. Основным направлением «инженерное» руководство выбрало сохранение традиций А. Л. Фуксмана и создание условий финансовой независимости ВЦ. В результате резко вырос объём хоздоговорных работ, и ВЦ оснастился большим количеством вычислительной техники: ЭВМ ЕС-1060, две ЭВМ ЕС-1033, две ЭВМ М-220, ПС-2000, несколько машин ряда СМ.

В 1986 году на должность заведующего ВЦ был временно назначен заведующий лабораторией, кандидат технических наук Х. Д. Дженибалаев, проработавший к тому времени в ВЦ более 15 лет. Возглавляемый им коллектив также старался продолжать лучшие традиции, заложенные А. Л. Фуксманом.

Интересный и яркий этап истории ВЦ РГУ связан с деятельностью Игоря Анатольевича Николаева, руководившего центром с 1987 года по 2000 год. Областью научных интересов И. А. Николаева были разработка и исследование методов и средств параллельного программирования; моделирование процессов, описываемых уравнениями в частных производных.

В период руководства ВЦ РГУ Игорем Анатольевичем Николаевым произошла реорганизация вычислительного центра. 20 ноября 1996 г. на базе ВЦ РГУ был создан Южно-Российский региональный центр информатизации (ЮГИНФО). С 29.04.1999 г. ВЦ и ЮГИНФО объединены под названием Южно-Российский региональный центр информатизации РГУ. ЮГИНФО стал преемником и продолжателем лучших научных и образовательных традиций ВЦ РГУ.

Под руководством профессора И. А. Николаева был создан региональный центр высокопроизводительных вычислительных систем на базе параллельной супер ЭВМ nCube 2S. Созданная им научная шко-

ла в области параллельных вычислений и математического моделирования продолжает действовать и приносить весомые результаты в различных областях прикладной математики и вычислительной техники.

С 2000 по 2015 годы ЮГИНФО возглавлял Лев Абрамович Крукиер — доктор физ.-мат. наук, профессор. За прошедшие годы под его руководством было реализовано множество значимых проектов в области информационных технологий и математического моделирования.

В результате реорганизации 2015 года ЮГИНФО ЮФУ (в прошлом ВЦ РГУ) вместе с НИИМ и ПМ им. И. И. Воровича был присоединён к механико-математическому факультету ЮФУ. Создан Институт математики, механики и компьютерных наук им. И. И. Воровича. В ряду актуальных научных направлений, которые реализуются учёными ИММ КН им. И. И. Воровича, возрождаются и развиваются направления, связанные с проблемами системного программирования: современными языками программирования, компиляторами, интегрированными средами разработки, инструментарием для программирования.

Организованная сотрудниками ИММ КН им. И. И. Воровича Всероссийская научная конференция «Языки программирования и компиляторы ’2017» является не только данью памяти А. Л. Фуксману, но и возможностью совместного обсуждения учениками и последователями направлений развития блестящих идей и открытий, предложенных выдающимся учёным Адольфом Львовичем Фуксманом.

Статический анализ кода: от теории к практике

Филипп Хандельянц

Одной из смежных задач при компиляции кода является его статический анализ с целью выявления потенциальных ошибок на как можно раннем этапе разработки. Статический анализ кода реализуется как компиляторах, так и сторонних вспомогательных инструментах. Являясь одним из разработчиков анализатора PVS-Studio хочется поднять важную и интересную тему борьбы с ложно позитивными срабатываниями.

Придумать новое диагностическое правило большого труда не составляет. Например, просто придумать, что следует искать искать циклы, в которых к массиву обращаются по константному индексу. Такой код подозрителен и с большой вероятностью может содержать ошибку. Однако подозрительный код, это не тоже самое что неправильный код. Поэтому самое сложное при статическом анализе суметь как можно точнее отделить случае ошибок от корректного кода, написанного специфическим образом.

Разрабатывая анализатор PVS-Studio, мы накопили большой практический опыт в выявлении корректных и некорректных паттернов кода. Эти знания мы эффективно используем для реализации новых диагностических правил. Мы готовы делиться своими теоретическими и практическими наработками и этот доклад один из способов рассказать о некоторых полезных освоенных нами методиках.

В докладе будут рассмотрен цикл разработки новых диагностических правил. Будет рассказано, как создаются алгоритмы поиска новых ошибок, основанных на таких методологиях как сопоставление с шаблоном, выводе типов, символьном выполнении, анализе потоков

данных, аннотирование методов. Будет затронут вопрос, как сохранять высокую скорость работы анализатора. несмотря на рост количества диагностик. Например, для нас вредно утверждение «преждевременная оптимизация — зло» и как мы работаем с «виртуальными значениями переменных...

Повышенное внимание будет уделено вопросам создания специальных подправил-исключений в диагностиках для подавления ложных срабатываний. Помимо общих сведений, будут разобраны несколько практических примеров.

Будет рассмотрено, какую инфраструктуру мы построили для эффективной разработки новых диагностик и проверки корректности работы анализатора.

Преобразование программ «растягивание скаляров»

Автономов Д. А.[1], avtonomovvv@gmail.com
Штейнберг О. Б.[1], olegsteinb@gmail.com
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

Преобразование программ "Растягивание скаляров" заменяет скаляр внутри тела цикла массивом, что иногда позволяет распараллеливать данный цикл. В работе представлен алгоритм, выполняющий данное преобразование как для скалярных переменных, так и для индексных переменных, не зависящих от счетчика цикла. Также допускается присутствие условных операторов. Алгоритм реализован в рамках Оптимизирующей распараллеливающей системы (ОРС).
Ключевые слова: параллельные вычисления, преобразования программ, разбиение цикла, оптимизирующий компилятор, растягивание скаляров, экспансия массивов.

1 Введение

При распараллеливании программ [5], [7], [3] важную роль занимают программные циклы. В некоторых случаях распараллелить цикл невозможно без применения вспомогательных преобразований. Одним из таких преобразований является «растягивание скаляров», заменяющее вхождения скалярной переменной вхождениями созданного временного массива.

В литературе как правило описывается лишь подход к устранению дуг зависимости, называемый растягиванием скаляров. При этом его

объяснение приводится лишь на примерах, содержащих пару операторов присваивания и пару вхождений скаляров. Что необходимо делать в случае присутствия большего количества вхождений, а также присутствия рекуррентных операторов или условных операторов, не поясняется. В книге Аллена и Кеннеди [5] приведен алгоритм растягивания скаляров, основанный на графе зависимости по данным [6], [5], [7], [3], [2]. Он основан на нахождении так называемых покрывающих определений. Авторами же предлагается немного видоизмененный алгоритм, не содержащий графа зависимости по данным, а также, не использующий покрывающих определений. Алгоритм был реализован в рамках Оптимизирующей распараллеливающей системы (ОРС) [1].

2 Преобразование программ «растягивание скаляров»

«Растягивание скаляров» - это преобразование, состоящее в замене скалярных переменных внутри цикла индексными переменными. Оно используется как вспомогательное для распараллеливания и разбиения циклов [4].

Стоит отметить, что разработанный алгоритм применим к циклам, внутри которыхо нет операторов безусловного перехода **goto**, а также **break** и **continue**. Алгоритм «растягивает» все переменные внутри цикла, не зависящие от его счетчика (при условии что у этих переменных в данном цикле присутствует вхождение осуществляющее запись).

Для каждой из «растягиваемых» переменных C_k создается временный массив $Ctemp_k$ размер которого на единицу больше количества итераций цикла. После этого находится первое появление генератора (т.е. вхождения, осуществляющего запись) переменной C_k. Все вхождения переменной в теле цикла до этого генератора заменяются на вхождения созданного массива с индексным выражением равным i (где i счетчик преобразуемого цикла). Все последующие вхождения, включая генератор, заменяются на вхождения массива с индексным выражением i+1.

3 Расширение области применения алгоритма

В случае когда в цикле присутствует индексная переменная, не зависящая от его счетчика, может применяться аналогичное преобразование называемое «экспансия массивов» (*array expansion*) [6].

Также допускается присутствие внутри тела цикла условного оператора **if**(в алгоритме, описанном в [5], также допускается присутствие условных операторов). Правда, в случае его присутствия могут возникнуть дополнительные действия.

Если до условного оператора **if** не имеется генератора соответствующей «растягиваемой» переменной C_k, а внутри хотя бы одной из веток, таковой присутствует, то, кроме вышеописанных действий, выполняется следующее:

В каждой ветке **if**, все вхождения «растягиваемой» переменной, стоящие до первого генератора и до **if**, заменяются на вхождения созданного массива с индексным выражением равным счетчику цикла ($Ctemp_k[i]$). Все последующие вхождения и вхождения стоящие после **if**, включая генератор, заменяются на вхождения массива с индексным выражением на единицу больше ($Ctemp_k[i+1]$). Если ветка **else** отсутствует, то ее следует создать. После этого, в ветке, в которой нет генератора «растягиваемой» переменной в качестве последнего оператора, добавляется оператор $Ctemp_k[i+1] = Ctemp_k[i]$.

4 Заключение

В данной работе приведен алгоритм, реализующий распараллеливающее преобразование «растягивания скаляров». Описанный алгоритм допускает присутствие условных операторов. В отличие от известного, описанного раннее алгоритма [5], он не использует граф зависимости по данным и покрывающие определения.

Список литературы

1. Оптимизирующая распараллеливающая система. — URL: `http://ops.rsu.ru/about_OPS.shtml`.

2. *Падуа Д.*, *Вольф М.* Оптимизация в компиляторах для суперкомпьютеров // Векторизация программ: теория, методы, реализация. — Москва : Мир, 1991. — С. 7—47. — ISBN 5-03-001943-X.

3. *Штейнберг Б. Я.* Математические методы распараллеливания рекуррентных программных циклов на суперкомпьютеры с параллельной памятью. — Ростов-на-Дону : Издательство Ростовского университета, 2004. — 192 с.

4. *Штейнберг О. Б.* Минимизация количества временных массивов в задаче разбиения циклов // Известия ВУЗов. Северо-Кавказский регион. Естественные науки. — Ростов-на-Дону, 2011. — № 5. — С. 31—35.

5. *Allen R.*, *Kennedy K.* Optimizing compilers for modern architectures. — San Francisco, San Diego, New York, Boston, London, Sidney, Tokyo : Morgan Kaufmann Publishers, 2002. — 790 с.

6. *Feautrier P.* Array Expansion // ICS '88 Proceedings of the 2nd international conference on Supercomputing (St Malo, France). — New York, NY, USA : ACM, 1988. — С. 429—441. — ISBN 0-89791-272-1.

7. *Wolfe M.* High performance compilers for parallel computing. — Redwood city : Addison-Wesley Publishing Company, 1996. — 570 с.

Синтаксический анализ графов и задача генерации строк с ограничениями

Рустам Азимов, rustam.azimov19021995@gmail.com
Семён Григорьев, Semen.Grigorev@jetbrains.com
Лаборатория языковых инструментов JetBrains,
Санкт-Петербургский государственный университет,
Россия, 199034, Санкт-Петербург,
Университетская наб. 7/9

Аннотация

Одной из задач, изучаемых в теории формальных языков, является задача генерации строк, удовлетворяющих заданной системе правил. С другой стороны, существует задача синтаксического анализа графов, то есть задача поиска путей в графе, метки на ребрах которых образуют строку, принадлежащую заданному формальному языку. В данной работе будет показана связь между этими двумя задачами.

Ключевые слова: синтаксический анализ графов, генерация строк, формальные языки, конъюнктивные грамматики.

В таких областях, как графовые базы данных [2; 5], биоинформатика [8], возникают задачи поиска путей в графах, удовлетворяющих определенным ограничениям. В качестве таких ограничений естественно выбрать формальный язык L [1] и искать пути в графе, соответствующие строкам из языка L. Задачи поиска путей в графе, которые используют такие ограничения с формальными языками, называются задачами *синтаксического анализа графов*. Данная задача также возникает при статическом анализе динамически формируемого кода, например динамических SQL-запросов или генераторов Web-страниц. В данном случае графом является представление регулярной аппроксимации множества возможных значений динамически формируемых строк.

Кроме того, существует задача генерации строк, суть которой в построении строк, принадлежащих некоторому формальному языку. В работе [9] приведены формулировки задачи генерации строк с дополнительными ограничениями.

Некоторые вариации задач синтаксического анализа графов могут быть сведены к задаче генерации строк. Так, например, в большинстве задач синтаксического анализа графов недостаточно просто определить существование пути, соответствующего строке некоторого формального языка L, но также требуется предъявить такой путь. Так как все пути в графе соответствуют строкам из некоторого регулярного языка R, то в данной задаче требуется найти путь, соответствующий строке из языка $L \cap R$. Эта задача может быть решена с помощью генератора строк рассматриваемого пересечения языков. В рамках данной работы была поставлена задача исследования связей между задачей генерации строк [9] и некоторыми типами задач синтаксического анализа графов [3; 4], использующие контекстно-свободные и конъюнктивные [7] языки.

Язык, который порождается графом G и выделенными в нем вершинами m, n, обозначим $L(G, m, n)$. А язык, порождаемый грамматикой C, со стартовым нетерминалом a обозначим $L(C, a)$.

В контексте задач синтаксического анализа графов бывает необходимо отвечать на различного рода вопросы, связанные с искомыми в графе путями. Тип вопросов, на которые отвечает задача принято называть *семантикой запроса*.

Использование *relational* семантики запроса означает, что для нетерминала a и графа G необходимо построить множество $\{(m, n) \mid L(C, a) \cap L(G, m, n) \neq \varnothing\}$. В случае использования КС-языка было выявлено отсутствие необходимости в применении генератора строк для поиска ответа на запрос с *relational* семантикой, так как в работе [4] используется аннотированная грамматика, которая порождает язык $L(C, a) \cap L(G, m, n)$ и ее построение автоматически решает поставленную задачу.

Использование *all-path* семантики запроса означает, что для нетерминала a, графа G и его вершин m, n, необходимо предъявить все пути из вершины m в вершину n, такие что метки на ребрах этих путей образуют строку из языка $L(C, a)$. В случае использования КС-языка также было выявлено отсутствие необходимости в применении генератора строк для данной семантики, так как в работе [4] аннотированную грамматику и предлагают в качестве ответа на запрос. Но

также была выявлена возможность использования генератора строк для получения конкретных строк пользователем из полученной аннотированной грамматики.

Использование *single-path* семантики запроса означает, что для нетерминала a, графа G и его вершин m, n, необходимо предъявить какой-нибудь путь (если он существует) из вершины m в вершину n, такой что метки на ребрах этого пути образуют строку из языка $L(C, a)$. Для КС-языков в работе [4] строится аннотированная грамматика, и если она порождает непустой язык, то в ней ищется строка минимальной длины, которая и будет соответствовать искомому пути в графе G. Таким образом, было выявлено, что алгоритм решения задачи синтаксического анализа графов с использованием *single-path* семантики запроса, предложенный в работе [4], и является примером использования генерации строки из КС-языка $L(C, a) \cap L(G, m, n)$.

Также была рассмотрена задача синтаксического анализа графов с использованием конъюнктивной грамматики. Из неразрешимости задачи определения пустоты конъюнктивных языков была получена неразрешимость задачи синтаксического анализа графов с использованием конъюнктивных языков и *relational* семантики запроса, о чем также упоминается в работе [3]. Кроме того, было выявлено, что при использовании конъюнктивных грамматик нельзя гарантировать нахождения хотя бы одной строки из конъюнктивного языка $L(C, a) \cap L(G, m, n)$. Предположим, что найдется хотя бы одна строка, удовлетворяющая рассматриваемым ограничениям. Тогда при использовании *all-path* семантики запроса, применяя алгоритм генерации строки, происходил бы просто перебор всех возможных строк и проверка на принадлежность этих строк к языку $L(C, a) \cap L(G, m, n)$, что не соответствует практическому смыслу задачи. А для задачи синтаксического анализа графов с использованием *single-path* семантики запроса есть возможность сгенерировать некоторую строку непустого языка $L(C, a) \cap L(G, m, n)$. Стоит отметить, что использование конъюнктивных языков в задачах синтаксического анализа графов мало изучено. Полученные результаты могут быть использованы в дальнейших исследованиях данной области. Одной из тем таких исследований, например, является применимость булевых [6] грамматик в синтаксическом анализе графов.

Список литературы

1. *Barrett C.*, *Jacob R.*, *Marathe M.* Formal-language-constrained path problems // SIAM Journal on Computing. — 2000. — Т. 30, № 3. — С. 809—837.
2. Context-free path queries on RDF graphs / X. Zhang [и др.] // International Semantic Web Conference. — Springer. 2016. — С. 632—648.
3. *Hellings J.* Conjunctive context-free path queries //. — 2014.
4. *Hellings J.* Querying for Paths in Graphs using Context-Free Path Queries // arXiv preprint arXiv:1502.02242. — 2015.
5. *Mendelzon A. O.*, *Wood P. T.* Finding regular simple paths in graph databases // SIAM Journal on Computing. — 1995. — Т. 24, № 6. — С. 1235—1258.
6. *Okhotin A.* Boolean grammars // Information and Computation. — 2004. — Т. 194, № 1. — С. 19—48.
7. *Okhotin A.* Conjunctive grammars // Journal of Automata, Languages and Combinatorics. — 2001. — Т. 6, № 4. — С. 519—535.
8. Quantifying variances in comparative RNA secondary structure prediction / J. W. Anderson [и др.] // BMC bioinformatics. — 2013. — Т. 14, № 1. — С. 149.
9. *Охотин А. С.* О сложности задачи генерации строк // Дискретная математика. — 2003. — Т. 15, № 4. — С. 84—99.

Генерация кода для графических ускорителей в ДВОР

Аллазов А. Н.[1], afarallazov@gmail.com
Гуда С. А.[1], gudasergey@gmail.com
Морылев Р. И.[1], rmorylev@gmail.com
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

Портируя программы на CUDA программист сталкивается с множеством трудностей. Ему приходится анализировать зависимости в программе, искать распараллеливаемые циклы, трансформировать код так, чтобы достичь наилучшего отображения на архитектуру видеокарты. Избежать ошибок — невозможно. Описываемый в статье Диалоговый высокоуровневый оптимизирующий распараллеливатель программ (ДВОР) позволяет автоматизировать разработку CUDA-программ. Диалоговый подход имеет ряд преимуществ над полностью автоматическим распараллеливанием: пользователь может выбрать последовательность преобразований программы, попробовать несколько вариантов результирующего кода, задать параметры преобразований, сравнить производительность и выбрать лучшие значения. ДВОР может автоматически находить распараллеливаемые циклы, визуализировать зависимости по данным, выполнять множество преобразований кода (расщепление тела цикла, слияние, гнездование, раскрутка, векторизация циклов, преобразование рекуррентных циклов к распараллеливаемой форме и др.), генерировать CUDA-код, автоматически определять оптимальные параметры запуска задачи на видеокарте.

Ключевые слова: генерация CUDA-кода, диалоговая оптимизация, автоматическое распараллеливание, ДВОР.

В последнее время большую популярность приобрели ускорители параллельных вычислений. На рынке активно конкурируют устройства NVIDIA, AMD и Intel. Разработано множество средств, облегчающих работу программиста: технологии программирования (CUDA, OpenCL), библиотеки программ, прагмы (OpenACC, OpenMP), расширения языков и т.д. В данной статье описано одно из таких средств, основанное на диалоговом подходе к оптимизации кода. Визуальный интерфейс пользователя позволяет удобно выделять фрагменты кода, применять преобразования, делать проверки, строить графы по программе.

Состоящий из более чем 180,000 строк кода на C++, кросплатформенный Диалоговый высокоуровневый распараллеливатель программ (ДВОР) [1] обладает большой базой анализаторов и преобразователей кода, многофункциональным GUI с возможностью выделения участков кода. По своим параметрам он похож на авторапараллеливающие системы: Rose Compiller, SUIF, Cetus, PPCG, Par4All, Parascope Editor. Авторы статьи реализовали в данной распараллеливающей системе автоматический генератор CUDA-кода. ДВОР использует анализ информационных зависимостей в циклах, основанный на полиэдральной модели, с собственной реализацией параметрического метода Гомори [2]. ДВОР поддерживает создание проектов с множеством файлов, позволяет выполнять разнообразные преобразования циклов, производит проверку корректности применяемых преобразований. Так же, как и менеджер памяти времени выполнения [3], ДВОР находит оптимальное расположение операций копирования данных на ускоритель и обратно, предотвращая копирование не измененных данных, интенсивные перемещения данных внутри циклов и копирование ячеек памяти, используемых только на видеокарте. В отличие от менеджера [3], ДВОР определяет все это на этапе компиляции.

Модуль статической профилировки GPU-кода позволяет находить оптимальные размеры блока потоков, запускаемых на видеокарте, и оптимальное отображение циклов гнезда на измерения пространства потоков. ДВОР полагается на встроенный тайлинг циклов, который автоматически получается в результате разбиения видеокартой потоков на блоки, и на автоматическое кеширование, в отличие от программного тайлинга и управления L1-кешем в компиляторе PPCG [4]. Это позволяет снизить накладные расходы и добиться лучшей производительности и компактности кода kernel-функций. Как показывают численные эксперименты, ДВОР позволяет получать преобразо-

ванную программу в компактной форме, производительность которой сравнима с результатами распараллеливающего без участия пользователя компилятора PPCG.

К функциям автоматической генерации GPU-кода в ДВОР организован доступ через веб-интерфейс [5]. Распараллеливаемые циклы в загружаемых на сайт программах должны быть помечены директивой "pragma target".

Список литературы

[1] ДВОР, `http://ops.rsu.ru/about.shtml`

[2] Feautrier, P.: Parametric Integer Programming. RAIRO Recherche Op'erationnelle. 22, 243–268 (1988)

[3] Pai, S., Govindarajan, R., Thazhuthaveetil, M.J.: Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques, pp. 33–42. ACM, New York (2012)

[4] Verdoolaege, S., Juega, J.C., Cohen, A., Gomez, J.I., Tenllado, Ch., Catthoor, F.: Polyhedral parallel code generation for CUDA. ACM Trans. Archit. Code Optim, 9 (2013)

[5] Веб-распараллеливатель, http://ops.opsgroup.ru

Тестирование преобразований программ в компиляторе с заданным критерием качества

Алымова Е. В., langnbsp@gmail.com
Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

В работе рассматривается подход к организации автоматического тестирования преобразований программ в оптимизирующих компиляторах. Перечислены этапы тестирования преобразования тестами, полученными автоматически с помощью параметризуемого генератора. В работе рассмотрено преобразование "Вынос оператора из цикла" приведены настройки генератора тестов и выбор критерия достаточности тестового набора для этого преобразования.

Ключевые слова: оптимизирующий компилятор, автоматизация тестирования, преобразования программ, критерий достаточности тестирования, генератор тестов.

Целью автоматизации тестирования является генерация наборов тестов, проверяющих качество реализации преобразования с заданным критерием достаточности. Преобразование реализовано качественно, если не нарушает функциональной эквивалентности исходной и преобразованной программ. Две программы функционально эквивалентны, если на одних и тех же наборах входных данных они возвращают одинаковые выходные данные.

Вопросами генерации тестов для компиляторов занимаются многие авторы. В ИСП РАН разрабатываются методики и инструменты генерации тестов на основе моделей [2], [3]. В работе [1] рассматривается подход к генерации корректных в отношении типов программ

для компилятора функционального языка программирования. В работе [4] предлагается методика генерации тестов по модели, основанной на темпоральных логиках.

В данной работе генерация тестов основывается на описании условий применимости тестируемого преобразования. Набор тестов удовлетворяет критерию достаточности, если для каждой последовательности операторов из множества всех n-ок выделенных операторов найдётся хотя бы одна программа, содержащая эту последовательность [6].

Тестом для преобразования является программа на языке Си, прошедшая компиляцию, и набор входных данных к ней. Позитивный тест - это такая программа, для которой существует функционально эквивалентная ей преобразованная программа. Позитивная тестовая программа должна содержать фрагмент кода, подходящий для тестируемого преобразования.

Тестирование преобразования на позитивных тестах в Оптимизирующей распараллеливающей системе [7] состоит из следующих этапов: генерация набора позитивных тестовых программ, удовлетворяющего сформулированному критерию достаточности; генерация набора входных данных для каждой тестовой программы; применение преобразования к каждой тестовой программе; установление функциональной эквивалентности исходной и результирующей программ из набора тестов.

Генератор тестов [5] на вход принимает контекстно-свободную грамматику целевого языка и конфигурационный файл в формате XML с описанием условий применимости преобразования: допустимым составом операторов и информационных связей в преобразуемом фрагменте.

В данной работе проводится анализ преобразования, заключающегося в выносе оператора из цикла, и составляется формальное описание условий применимости этого преобразования.

Первичный конфигурационный файл для тестирования выноса оператора из цикла содержит описание цикла, тело которого может быть разбито по крайней мере на три блока:

```
for (int i = 0; i <= N; i++) {
    BLOCK1(i);
    BLOCK2(i);
    BLOCK3(i);
}
```

$BLOCK2$ всегда непустой, а $BLOCK1$ и $BLOCK3$ не могут быть пустыми одновременно. В $BLOCK2$ все генераторы не зависят от счётчика цикла и в этом блоке нет истинных циклически порождённых зависимостей. На основе первичного конфигурационного файла формируются две модификации. В первой модификации нет дуг графа информационных связей, ведущих в блок $BLOCK2$ из остальных блоков. Во второй модификации нет дуг графа информационных связей, ведущих из $BLOCK2$ в остальные блоки.

Набор тестов достаточен, если в нем встречаются все возможные четверки, которые можно составить из оператора присваивания, условного оператора (с одной и двумя ветвями), оператора цикла с предусловием и оператора выбора. Мощность набора тестов: $5^4 = 625$. Пример программы, сгенерированной по конфигурационному файлу:

```
#include <stdio.h>
/* Variable Declarations */
int main() {
  /* Initialization block */
  for(intIndex_0 = 23; intIndex_0 <= 458;
      intIndex_0 = intIndex_0 + 1)
  {
     realArray_3[intIndex_1] =
       intArray_2[intIndex_0] +
       realArray_3[intIndex_1] +
       intArray_0[intIndex_1 + 5];
     realArray_2[intIndex_3 + 1] =
       intArray_12[intIndex_0 - 2] + 1.4 +
       realArray_3[intIndex_1] *
       intArray_9[intIndex_1 + 2];
  }
  /* Result Output Block  */
  return 1;
}
```

Список литературы

1. *Palka M. H.* Testing an optimising compiler by generating random lambda terms. // AST 11 Proceedings of the 6th International Workshop on Automation of Software Test. — 2011. — С. 91—97.

2. *Zelenov S. V.* Automated Generation of Positive and Negative Tests for Parsers. // Formal Approaches to Software Testing. Lecture Notes in Computer Science. — 2005. — № 3997. — С. 187—202.

3. *Zelenov S. V.* Test Generation for Compilers and Other Formal Text Processors. // Programming and Computer Software. — 2003. — № 29. — С. 104—111.

4. *Zhao C.* Automated Test Program Generation for an Industrial Optimizing Compiler. // Proceedings of the 4th International Workshop on Automation of Software Test. — 2009. — С. 36—43.

5. *Алымова Е. В.* Автоматическая генерация тестов на основе конфигурационных файлов для оптимизирующих преобразований компилятора // Известия вузов. Северо-Кавказский регион. Естеств. науки. — 2010. — № 3.

6. *Алымова Е. В.* Критерий полноты тестовых наборов, ориентированных на проверку распараллеливающих преобразований программ. // Информационные технологии. — 2011. — № 9. — С. 19—22.

7. Оптимизирующая распараллеливающая система. — URL: `http://ops.rsu.ru` (дата обр. 18.03.2017).

Мультиязычный обфускатор программ и компилируемая библиотека непрозрачных предикатов

Алымова Е. В.[1], langnbsp@gmail.com
Баглий А. П.[1], taccessviolation@gmail.com
Гуфан К. Ю.[2], k.gufan@niisva.org
Штейнберг Б. Я.[1], borsteinb@mail.ru
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет
[2]ФГАНУ "НИИ Спецвузавтоматика"

Аннотация

Ставится задача мультиязычной обфускации, в том числе для байт-кода. Предлагаемый метод обфускации основывается на использовании преобразований из теории оптимизирующих компиляторов и языково-независимой базы математических тождеств, что позволяет вносить значительное зашумление в код без нарушения функциональной эквивалентности исходной и обфусцированной программ.
Ключевые слова: обфускация, LLVM, MSIL, преобразования программ.

В этой работе описывается создание многоязычного обфускатора, основанного на некоторых компиляторных преобразованиях программ и библиотеке непрозрачных функций. Алгоритм работы этого обфускатора универсален для широкого класса потенциальных входных языков, таких как байт-код LLVM, MSIL или текст JavaScript. Составной частью обфускатора является библиотека непрозрачных функций, код которых используется в преобразованиях. Под непрозрачными здесь понимаются любые функции с заранее известными

свойствами, код которых вставляется в выходной код для затруднения его анализа. Обфускатор предназначен для затруднения ручного или автоматического анализа кода, предотвращения распространения эксплойтов. Аналогичный подход используется, например в [1]. Представляемый обфускатор может генерировать тысячи различных по коду и эквивалентных между собой экземпляров одной программы: каждому пользователю свой уникальный экземпляр. Это позволяет, например, при появлении несанкционированной копии определять источник утечки.

Целью работы является создание универсального обфускатора (или семейства обфускаторов) для различных входных языков. В простейшем виде алгоритм работы обфускатора заключается в:

- последовательном обходе входной программы с поиском фрагментов кода, для которых имеются преобразования.
- последовательном применении случайных преобразований из списка доступных к найденным фрагментам
- замене всех найденных фрагментов на преобразованные.

Перечисленные действия составляют один проход, таких проходов может быть несколько. Существует возможность добавления преобразований для реализации значительной части базовых обфусцирующих преобразований, классифицированных в [3].

Используемые преобразования кода могут усложнять редактирование управляющего графа, но делают проще добавление новых преобразований и проверку их корректности. Допускаются проходы с использованием сторонних инструментов для использования других методов обфускации, в частности для изменения имен и управляющего графа.

Используемая обфускатором Библиотека содержит сотни непрозрачных функций (преобразований). Эта библиотека создается из Библиотеки базовых (более простых) функций. Генерируемые на лету или заранее функции создаются путем композиции простых функций. Генерируемые функции могут включать тождественные или константные функции одного или нескольких аргументов, которые возвращают результаты с определенной погрешностью. Для создания примера таких наборов функций использовались формулы из [2]

Все используемые функции должны быть реализованы через минимально необходимый набор инструкций языка (C#) для упрощения автоматической конвертации в другие языки.

Для использования сгенерированных из библиотеки функций в коде на LLVM или других языках требуется написание достаточного конвертера из C# в каждый из целевых языков (или использование любого существующего). Так, для конвертации MSIL в JavaScript используется JSIL

Используемые на данный момент входные языки - байт-код MSIL, LLVM, или текст JavaScript. Использование радикально различающихся входных языков может затруднять применение одних и тех же преобразований. Например, для корректного применения преобразований требуется проверка типов выражений, которую легко провести в MSIL и LLVM, но не в исходном коде JavaScript и некоторых других языков.

В результате проведенной работы создан расширяемый обфускатор MSIL, LLVM (и частично JavaScript), проведено тестирование обфускатора на стандартной программе, реализующей алгоритм шифрования по ГОСТ Р 34.12-2015, написанной на C# (MSIL) и C++ (LLVM), а также других сборках MSIL.

Список литературы

[1] А.Р. Нурмухаметов. Применение диверсифицирующих и обфусцирующих преобразований для изменения сигнатуры программного кода. Труды Института системного программирования РАН, том 28, вып. 5, 2016, стр. 93-104.

[2] Попов Б.А., Теслер Г.С. Вычисление функций на ЭВМ. Киев: Наукова Думка, 1984. — 600 с.

[3] C. Collberg, C. Thomborson, D. Low. A Taxonomy of Obfuscating Transformations. Departament of Computer Science, the University of Auckland, 1997.

Промежуточное представление программ ОРС для генерации схемы конвейерного вычислителя

Алымова Е. В[1], langnbsp@gmail.com
Баглий А. П.[1], taccessviolation@gmail.com
Дубров Д. В.[1], dubrov@sfedu.ru
Ибрагимов Р. А.[1], mhr112@yandex.ru
Михайлуц Ю. В.[1], aracks@yandex.ru
Петренко В. В.[1], vpetrenko@gmail.com
Штейнберг Б. Я.[1], borsteinb@mail.ru
Штейнберг Р. Б.[1], romanofficial@yandex.ru
Яковлев В. А.[1], vlad309523@gmail.com
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

В данной работе рассматривается разработка компилятора языка Си на процессор с программируемой архитектурой. Такой компилятор включает в себя конвертор языка Си в язык описания электронных схем и библиотеку (или генератор) драйверов. Компилятор разрабатывается на основе Оптимизирующей распараллеливающей системы. Описаны основные элементы структуры промежуточного представления компилятора для генерации конвейерного ускорителя. Для тестирования производительности получаемого кода разработан генератор тестовых программ, вычисляющих свертки.

Ключевые слова: компилятор, высокоуровневый синтез, язык описания электронных схем.

1 Введение

Данная работа является развитием прежних работ группы авторов по созданию компилятора на процессор с программируемой архитектурой [1], [2]. Разработка ведется на основе Оптимизирующей распараллеливающей системы [2], имеющей высокоуровневое внутреннее представление [4]. Существуют инструменты, которые преобразуют входную программу на языке высокого уровня целиком в описание устройства (или его часть). Примерами таких инструментов являются Vivado HLS [5], Catapult HLS [6], Bambu [3] и другие. Есть инструменты, которые разбивают исходную программу на части, предназначенные для вычисления на центральном процессоре и ускорителях. Примерами являются C2H [4] и HaSCoL [9]. Как показывает опыт практического использования инструмента C2H, помимо этого, для значительного ускорения вычислений, может также потребоваться существенное переписывание алгоритмов [10]. Представляемый в данной статье компилятор отличается от известных тем, что разрабатывается на основе распараллеливающей системы с высокоуровневым внутренним представлением (ВП), что упрощает генерацию конвейерного кода и позволяет использовать преобразования ОРС. Наличие конвертора из ВП ОРС в LLVM позволяет генерировать код на ЦПУ из той части исходной программы, которая не конвейеризуется.

2 Внутреннее представление конвейера

Внутреннее представление конвейера (ВПК) — абстрактная модель, хранящая и представляющая вычислительный конвейер удобным образом для построения и обработки. Основной задачей ВПК является компактное и структурированное хранение всей информации, необходимой для генерации HDL-кода. Основной структурной единицей представления является узел - элемент конвейера, ответственный за обработку или хранение данных. Узлы реализованы в рамках иерархии классов C++, которая представляет отдельный узел конвейера в ВПК: PLNodeBase – базовый класс узла, PLNodeCore – класс, представляющий вычислитель (узел), PLNodeBuffer – класс, представляющий буфер (узел), PLNodeVarBuffer - класс, представляющий буфер для хранения переменных значений, PLNodeConstBuffer - класс, представляющий буфер для хранения констант. Здесь под вычислителем понимается модуль, способный произвести установленную операцию

над переданными ему данными и предоставить доступ к результату. Под буфером понимается модуль, способный хранить заданное количество данных и предоставлять к ним доступ на чтение и запись.

Класс вычислителя содержит дополнительную информацию о реализуемой операции и количестве операндов (арности).

Класс буфера, за счет разнообразия вариантов хранения и передачи данных, несет много дополнительной информации о методах хранения и способах обращения к данным. Буфер может снабжать данными неограниченное количество вычислителей, работающих, вообще говоря, разное количество тактов.

Таким образом, буфер содержит информацию о размере своего основного блока, размерах всех подбуферов-задержек и о режиме работы. Буфер констант также содержит информацию о хранимой им константе и по умолчанию работает в режиме постоянной памяти.

Отдельной сущностью представления является порт, содержащий информацию о входных/выходных данных конвейера, их размерах, адресах чтения и записи. Порты определяют интерфейс взаимодействия конвейера и окружения и позволяют окружению воспринимать конвейер как черный ящик. В данной реализации порт содержит информацию о передаваемом объекте (массив, переменная, константа), ссылку на буфер, с которым связан данный порт и направление передачи (из порта в буфер или наоборот).

Сам класс промежуточного представления PipelineBase наследуется от StatementBase в дереве Reprise (ВП ОРС). Это обеспечивает целостное и независимое представление преобразованной программы в Reprise и открывает возможности для дальнейших преобразований, в которых обращение к конвейеру будет представлено как равноправный оператор языка. Структурно класс PipelineBase содержит список узлов, список портов, а также конкретную архитектуру конвейера в виде списка смежности графа узлов. Он обеспечивает удобный интерфейс для построения конвейера, а также позволяет обходить конвейер как в прямую, так и в обратную сторону.

Список литературы

[1] Boris Ya. Steinberg, Denis V. Dubrov, Yury V. Mikhailuts, Alexander S. Roshal, Roman B. Steinberg "Automatic High-Level Programs Mapping onto Programmable Architectures. Proceedings of the 13th International Conference on Parallel Computing Technologies, August

31 – September 4, 2015, Petrozavodsk, Russia"V 9251. p. 474-485. Springer Verlag

[2] Boris Ya. Steinberg, Anton P. Bugliy, Denis V. Dubrov, Yury V. Michiluts, Oleg B. Steinberg, Roman B. Steinberg "A Project of Compiler for a Processor with Programmable AcceleratorProcedia Computer Science 5th International Young Scientist Conference on Computational Science, YSC 2016, 26-28 October 2016, Krakow, Poland, Volume 101, 2016, Pages 435–438.

[3] Оптимизирующая распараллеливающая система www.ops.rsu.ru (дата обращения 08.02.2017)

[4] Петренко В.В. Новое внутреннее представление Открытой распараллеливающей системы. http://ops.rsu.ru/download/ops/VP_diplom_05.pdf (дата обращения 08.02.2017)

[5] Vivado High-Level Synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[6] Catapult High-Level Synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

[7] Bambu. http://panda.dei.polimi.it/?page_id=31

[8] Nios II C2H Compiler. User Guide. https://www.altera.com/en_US/pdfs/literature/ug/ug_nios2_c2h_compiler.pdf

[9] HaSCoL. http://oops.math.spbu.ru/projects/coolkit

[10] Etiemble D., Piskorski S., Lacassagne L. Performance evaluation of Altera C2H compiler on image processing benchmarks. Proceedings on the Fifteenth International Conference on Parallel Architectures and Compilation Techniques, September 16-20, 2006, https://www.lri.fr/~lacas/Publications/TCHA06.pdf

Непроцедурный язык НОРМА и его применение для параллельных архитектур

Андрианов А. Н., and@a5.kiam.ru
Баранова Т. П., bart1950@yandex.ru
Бугеря А. Б., shurabug@yandex.ru
Ефимкин К. Н., bigcrocodile@yandex.ru
Институт Прикладной Математики
им. М. В. Келдыша РАН

Аннотация

В работе предлагается декларативный подход к созданию прикладного программного обеспечения для высокопроизводительных вычислительных систем. Рассматриваются методы автоматического построения программ для различных параллельных архитектур по непроцедурным спецификациям на языке НОРМА. Описана структура созданного на основе рассмотренных методов компилятора и произведена оценка его эффективности. Приводятся результаты применения компилятора для нескольких прикладных задач.

Ключевые слова: параллельное программирование, автоматизация программирования, непроцедурные спецификации, графические процессоры, гибридные архитектуры, язык НОРМА.

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект 15-01-03039-а.

В современном научном сообществе задача разработки эффективных параллельных программ имеет важное стратегическое значение. Несмотря на то, что параллельное программирование появилось уже достаточно давно, успешно развивается и активно исследуется, вопрос создания эффективных параллельных программ для решения расчетных задач до сих пор крайне актуален для программистов и прикладных специалистов. В составе средств разработки программ для новых

параллельных архитектур со временем появляются различные новые инструменты и библиотеки, предоставляющие прикладному программисту эффективные элементарные «кирпичики» для конструирования своей программы. Их наличие, несомненно, существенно облегчает задачу прикладному специалисту, но только в том случае, если все его потребности в высокопроизводительных вычислениях покрываются имеющимися распараллеленными пакетами и/или библиотеками. Если же с помощью таких готовых средств построить решение для своей задачи не удаётся, то иного пути, кроме как изучить специальные инструменты программирования для целевой архитектуры и начать реализовывать свой алгоритм этими средствами на достаточно низком уровне, у прикладного специалиста нет.

Надежды на автоматическое распараллеливание уже написанных последовательных программ на различные виды параллельных архитектур пока не оправдываются, несмотря на то, что фирмы-производители аппаратных решений давно активно поддерживают данное направление исследований. Из уже реализованных подходов можно отметить те, которые базируются на вполне разумном симбиозе распараллеливающего компилятора и подсказок со стороны программиста, выполненных в виде специальных программных директив, например OpenACC [2].

В такой ситуации интерес представляют подходы к построению параллельных программ, точно определяющие границы того, что и как можно автоматически распараллелить и предоставляющие возможности для автоматизированного построения эффективных параллельных программ. Одним из таких подходов является непроцедурный язык НОРМА [5].

При использовании этого подхода прикладной специалист программирует решение вычислительной задачи на непроцедурном языке (понятия, связанные с архитектурой параллельного компьютера, моделями параллелизма и прочие детали целевой системы при этом не используются), а затем компилятор автоматически строит параллельную программу (уже учитывая архитектуру целевого параллельного компьютера, модели параллелизма и прочее). С учетом отмеченных выше проблем привлекательность этого подхода в настоящее время только усиливается, и интерес к идеям непроцедурного декларативного программирования и реализации этих идей в языках программирования неуклонно растет. Но, следует отметить, что в общем случае задача автоматического эффективного синтеза параллельной програм-

мы по непроцедурной спецификации может оказаться не разрешимой.

Идеи декларативного программирования были сформулированы еще в прошлом веке, теоретические исследования этого подхода для класса вычислительных задач проведены в пионерских работах И.Б. Задыхайло еще в 1963 году [3]. Непроцедурный язык НОРМА и система программирования НОРМА [5] [6] [7] разработаны в ИПМ им. М.В. Келдыша РАН также достаточно давно и предназначены для автоматизации решения вычислительных сеточных задач на параллельных компьютерах. Расчетные формулы записываются на языке НОРМА в математическом, привычном для прикладного специалиста виде. Язык НОРМА позволяет описывать решение широкого класса задач математической физики. Программа на языке НОРМА имеет очень высокий уровень абстракции и отражает метод решения, а не его реализацию при конкретных условиях.

Компилятор программ на языке НОРМА [4] позволяет получать исполняемую программу на заданном процедурном языке программирования для определённой модели параллелизма. На данный момент компилятор умеет создавать программы на языках Си или Фортран для следующих вычислительных архитектур:

- последовательные программы.
- для многоядерных систем с общей памятью с использованием технологии OpenMP.
- для распределённых систем с использованием технологии MPI.
- для графических процессоров фирмы NVIDIA с использованием технологии NVIDIA CUDA.

В настоящее время ведутся работы по созданию версии компилятора для гибридных архитектур.

Компилятор программ на языке НОРМА использовался для решения многих практических и тестовых задач. В качестве примера применения компилятора приведём получившиеся результаты для расчетной задачи из области газодинамики, теста CG из пакета NPB (NAS Parallel Benchmarks) [1] и фрагмента задачи с большой вычислительной плотностью. Во всех трёх примерах применения компилятора удалось добиться получения полностью корректного результата и была проведена оценка эффективности получающихся исполняемых программ.

Список литературы

1. NAS Parallel Benchmarks. — URL: `http://www.nas.nasa.gov/publications/npb.html` (дата обр. 26.01.2017).
2. OpenACC. — URL: `http://openacc.org` (дата обр. 26.01.2017).
3. *Задыхайло И. Б.* Организация циклического процесса счета по параметрической записи специального вида // Журн. выч. мат. и мат. физ. — М., 1963. — Т. 3, № 2. — С. 337—357.
4. Модульная архитектура компилятора языка Норма+ / А. Н. Андрианов [и др.] // Препринт ИПМ им. М.В. Келдыша РАН. — М., 2011. — № 64. — С. 16.
5. НОРМА. Описание языка. Рабочий стандарт / А. Н. Андрианов [и др.] // Препринт ИПМ им. М.В. Келдыша РАН. — М., 1995. — № 120. — С. 52.
6. Простые вещи / А. Н. Андрианов [и др.] // Суперкомпьютеры. — М., 2014. — № 2. — С. 58—61.
7. Система НОРМА. — URL: `http://www.keldysh.ru/pages/norma` (дата обр. 26.01.2017).

Разработка операционной семантики языков программирования на основе двухэтапного метода концептуального проектирования информационных систем

Ануреев И. С.[1], anureev@iis.nsk.su
[1]Институт систем информатики имени А. П. Ершова

Аннотация

Статья представляет двухэтапный метод разработки операционной семантики языков программирования, основанный на онтологическом подходе к формальной спецификации языков программирования [1]. Абстрактная машина языка программирования, определяющая его онтологическую операционную семантику, рассматривается как информационная система, для описания которой применяется двухэтапный метод концептуального проектирования информационных систем [2].
Ключевые слова: онтологическая операционная семантика, абстрактная машина, информационная система, информационная система переходов, концептуальная система переходов, онтология языка программирования.

Двухэтапный метод концептуального проектирования информационных систем (далее ИС) представлен в [2].

На первом этапе определяется структура ИС в соответствие с шаблоном, основанном на информационных системах переходов (далее ИСП) — формализме для моделирования ИС. ИСП [2] характеризуется множествами состояний, объектов состояний, информационных запросов, объектов информационных запросов, ответов, объектов ответов, а также функцией интерпретации и экзогенным и эндогенным

отношениями переходов. Состояния моделируют информацию, хранимую в ИС (содержимое ИС), информационные запросы — информацию, передаваемую из окружения в ИС с целью получить ее содержимое, а ответы — информацию, передаваемую от ИС к окружению, инициируемую этими запросами. Объекты состояний, запросов и ответов — это объекты, которые могут наблюдаться в состояниях, запросах и ответах, соответственно. Они описывают наблюдаемую внутреннюю структуру состояний, запросов и ответов. Функция интерпретации моделирует передачу информации от ИС к ее окружению и от окружения к ИС. Она связывает запросы с функциями из состояний в ответы. Экзогенное отношение перехода моделирует изменение содержимого ИС, вызванное ее окружением. Оно связывает запросы с бинарными отношениями на состояниях (называемыми отношениями переходов) и ответами, возвращаемыми парами состояний из этих отношений переходов (называемых переходами). Эндогенное отношение перехода моделирует изменение ИС, вызванное факторами внутри самой системы. Оно определяется как отношение переходов с ответами, возвращаемыми переходами из этого отношения переходов. Описание ИС, полученное на первом этапе, как правило, схематическое, так как его цель — определить, какие элементы и компоненты ИС планируется формализовать на втором этапе и зафиксировать терминологию, используемую для описания этих элементов и компонент. На втором этапе, полученная ИСП формально описывается (моделируется) с помощью концептуальных систем переходов (далее КСП). КСП [2] характеризуются набором концептуальных структур (элементы, концептуали, понятия, атрибуты, концептуальные состояния, концептуальные конфигурации), достаточно универсальным, чтобы моделировать типовые онтологические элементы и обеспечивать полную классификацию онтологических элементов, включая определение их новых видов и подвидов с произвольной концептуальной гранулярностью, а также отношением перехода на концептуальных конфигурациях, основанном на сопоставлении с образцом и переписывании.

Специфика двухуровневого метода концептуального проектирования ИС применительно к задаче разработки операционной семантики ЯП состоит в следующем. На первом этапе посредством ИСП схематически определяется формализуемый фрагмент ЯП, где состояния ИСП специфицируют состояния абстрактной машины (далее АМ), определяющей операционную семантику языка программирования, объекты состояния — элементы состояний АМ, информационные запросы — ин-

струкции АМ, ответы — значения, возвращаемые инструкциями АМ, а объекты запросов и ответов — структурные составляющие инструкций и значений. Функция интерпретации описывает семантику инструкций АМ, не изменяющих состояние и возвращающих значение (например, выражения без побочных эффектов). Экзогенное отношение перехода описывает семантику инструкций АМ, изменяющих состояние. Эндогенное отношение перехода описывает внутренние алгоритмы АМ (например, просачивание исключений, разрешение перегрузки функций и методов и т. п.), которые не инициируются явно инструкциями программы, исполняемой АМ. На втором этапе посредством КСП строится онтологическая операционная семантика ЯП [1], где онтология ЯП специфицируется концептуальными структурами КСП, а операционная семантика инструкций ЯП — отношением перехода КСП, определяемым в терминах этой онтологии. Состояния ИСП моделируются концептуальными конфигурациями КСП, объекты состояний ИСП — концептуальными структурами КСП, а запросы, ответы и объекты запросов и ответов ИСП — элементами КСП.

Предлагаемый подход к разработке операционной семантике ЯП обладает двумя преимуществами. Во-первых, использование онтологического аппарата, основанного на терминологии ЯП, позволяет сделать описание операционной семантики ЯП близким к описанию на естественном языке по таким критериям как компактность и понятность и при этом формальным. Во-вторых, средствами КСП можно описывать аксиоматическую семантику языка программирования, базирующуюся на его операционной семантике, более точно генератор условий корректности для программ на этом языке, что позволяет использовать этот подход в задачах верификации.

Работа частично поддержана грантом РФФИ 15-01-05974.

Список литературы

[1] Anureev I.S. Operational ontological approach to formal programming language specification // Programming and Computer Software. 2009. Vol. 35. N 1. P. 35-42.

[2] Anureev I.S. Formalisms for conceptual design of information systems // System Informatics. 2016. N 8. P. 53-88.

Потоковый механизм вывода графа программы в конструкторе компиляторов RiDE

Бахтерев М. О.[1, 2], m.bakhterev@imm.uran.ru
Куклин И. Ю.[2], kuklin.iy@gmail.com
Савлова А. А.[2], saa1330@gmail.com
[1]Институт Математики и Механики им. Н. Н. Красовского
[2]Уральский Федеральный Университет

Аннотация

С ростом популярности ПЛИС и увеличением доступности средств разработки СБИС растёт и количество проектов процессоров с нестандартными архитектурами. Сложность таких разработок увеличивает жёсткая ориентация существующих систем компиляции на процессоры со стандартной архитектурой. Это обстоятельство повышает актуальность разработки альтернативных средств компиляции, упрощающих разработку компиляторов для новых процессоров. В работе мы предлагаем систему компиляции, построенную вокруг проблемно-ориентированного языка LK, основанного на математической модели потоковых вычислений RiDE [3].

Ключевые слова: модель распределённых вычислений, граф потока данных, конструктор компиляторов.

Современные компиляторы являются высококачественным программным обеспечением, в развитие которого вложены большие ресурсы. Поэтому, На первый взгляд, разработка ещё одного конструктора компиляторов может показаться избыточной. Однако, на альтернативные конструкторы существует запрос со стороны разработчиков процессоров с принципиально новой архитектурой. Здесь принципиальность новизны архитектуры как раз и следует «измерять» уровнем

сложности адаптации к ней существующих систем компиляции. Эта сложность, в основном, возникают из-за жёсткой ориентации этих систем во всех уровнях на традиционные процессоры.

В средствах разработки компиляторов для «альтернативных» процессоров, напротив, требуется гибкость. Она необходима не только в подсистемах оптимизации промежуточного представления программ и его перевода в целевой машинный код, но и в возможностях расширения исходных языков конструкциями, позволяющими использовать особенности разрабатываемой архитектуры. Важно, чтобы система компиляции позволяла экспериментировать как с самими архитектурными решениями, так и с соответствующими расширениями языка. В современных системах такая экспериментальная работа затруднена тем, что многие изменения в исходном языке требуют объёмных модификаций кода компилятора на многих уровнях абстракции.

Строго говоря, не известны точные критерии, позволяющие выбрать математические и технические решения, необходимые для достижения нужных свойств конструктора компиляторов. Поэтому в своей работе мы в качестве эксперимента пробуем программировать процесс компиляции в модели распределённого параллельного исчисления с потоками данных RiDE. Для выбора именно такой модели, то есть, модели параллельных распределённых вычислений, в качестве основы для кодирования процесса компиляции есть некоторые основания.

Во-первых, на трансляцию некоторого выражения $e = (\texttt{kw}\ e_1\ \ldots\ e_n)$[1] можно смотреть как на n *параллельных* процессов трансляции подвыражений e_i, взаимодействующих в синтаксическом и семантическом контекстах выражения e. Некоторые из e_i могут входить подвыражениями в выражения с другой синтаксической и семантической структурой. Это обстоятельство можно интерпретировать как требование к процессам трансляции подвыражений быть явно зависимыми только от определённой локальным контекстом частичной информации о состоянии процесса компиляции всей программы. То есть, эти процессы должны вести себя как *распределённые* процессы. Нам, как разработчикам модели распределённых вычислений, особенно интересно, что в компиляторе взаимодействия между такими процессами интенсивные и насыщенные перекрёстными ссылками, поэтому их описание в рамках некоторой модели быть хорошим тестом на

[1]Нам удобно описывать абстрактный синтаксис s-выражениями, различая синтаксические формы ключевыми словами `kw` [2].

выразительность этой модели.

Во-вторых, некоторые формализмы из теории языков программирования и теории параллельных вычислений изоморфны. Например, алгебраические μ-типы без умножения [4] и рекурсивные взаимодействующие процессы [1], ограниченные только конструктором выбора $(P \mid Q)$.

В-третьих, один из хорошо изученных и популярных на практике видов промежуточного представления программ – графы потока управления, узлами которых являются линейные участки, описываемые графами потоков данных. Такие графы достаточно абстрактны и удобны, чтобы генерировать по ним код для разнообразных процессорных архитектур. Строить такие граф можно теми же методами, что и графы потоков данных в распределённых параллельных вычислениях.

Не вдаваясь в подробности, модель RiDE можно описать следующим образом. Вычисление в RiDE – это динамическое рекурсивное формирования графа потока данных $G = \langle E, V \rangle$. В G некоторые вершины отмечены метками $L : \mathcal{L} \nrightarrow E$. Кроме графа G и функции L состояние вычисления содержит набор r-продолжений K. R-продолжение $\alpha \in K$ – это пара из процедуры p_α и, что естественно для потоковых моделей, условий её вызова $c_\alpha \in \mathcal{L}$. Процедура p_α вычисляет три значения: участок графа $g = \langle e, v \rangle$, новые метки $l : \mathcal{L} \nrightarrow e$ и множество r-продолжений k. После исполнения p состояние вычисления $\langle G, L, K \rangle$ обновляется информацией из g, l и k, после чего вызывается процедура следующего готового к вычислению r-продолжения. R-продолжение $\beta \in K$ становится готовым к вычислению, когда L становится определённой на условии вызова $c_\beta \in \mathcal{L}$. Естественно, параметрами для вызова p_β являются узлы из множества $L(c_\beta)$.

Ядром нашего конструктора компиляторов является предметно ориентированный язык LK с RiDE-семантикой. На LK кодируются процессы трансляции выражений исходного языка, которые естественным для RiDE способом превращают эти исходные выражения в графы. Интересным позитивным результатом такого нашего экспериментального подхода к программированию процесса компиляции стало ускорение работы над компилятором. Главным образом благодаря разделению работ между разработчиками вплоть до уровня программирования процессов трансляций отдельных операторов исходного языка. В среднем, одна итерация разработки такого процесса для одной конструкции Си-99 занимала у нас 1, 2 дня. Это обеспечило высокие темпы экспериментальной проверки аппаратных и языковых решений.

Список литературы

1. *Hoare C. A. R.* Communicating Sequential Processes. — 2003. — Гл. 1.
2. *Turbak F.*, *Gifford D.*, *Sheldon M. A.* Design Concepts in Programming Languages (MIT Press). — Cambridge, 2008. — Гл. 2.
3. *Бахтерев М. О.*, *Васёв П. А.* Об эффективной реализации системы RiDE // XVI Международная конференция «Супервычисления и Математическое Моделирование». Тезисы. — Саров, 2016. — С. 25—27.
4. *Пирс Б.* Типы в языках программирования. — 2010. — Гл. 20.

Реализация сертифицированного интерпретатора для расширения простого типизированного лямбда-исчисления с концепт-параметрами

Белякова Ю. В.[1], julbel@sfedu.ru
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

Разработка сертифицированного интерпретатора предполагает наличие формальной модели соответствующего языка программирования. При реализации формальных моделей в системах интерактивного доказательства теорем, для представления программ обычно используют неэффективные структуры данных, которые удобны для формального рассуждения. Таким образом, задача сертификации интерпретатора порождает конфликт между эффективностью интерпретации и «пригодностью» интерпретатора для формального рассуждения. В данной работе представлено расширение простого типизированного лямбда-исчисления с концепт-параметрами, которые моделируют «модуль-подобные» конструкции языков программирования. На примере этого расширения исследуются проблемы механизированной верификации соответствующего эффективного интерпретатора. В частности, рассматривается использование эффективного словаря на основе бинарного дерева.

Ключевые слова: формальные модели, интерпретаторы, языки программирования, Coq, концепты.

Введение. Системы интерактивного доказательства теорем (СИДТ), такие как Coq или Agda, применяются в том числе

для верификации формальных моделей языков программирования (Featherweight Java [4; 5], Scala-DOT-calculus [8], JSCert [6]), а также разработки сертифицированного программного обеспечения. Примерами последнего могут служить интерпретаторы JavaScripts [1] и ML [6], а также CompCert [2] — сертифицированный компилятор языка C, создание которого заняло порядка 10 лет.

По сравнению с реальными языками программирования (ЯП), формальные модели (то есть определения семантики языка, отношения типизации), как правило, имеют более компактное описание: они концентрируются лишь на ключевых конструкциях и возможностях языка программирования, абстрагируясь от несущественных в данном контексте деталей. Отметим, что формальные модели используются для *анализа* поведения программ, а не *выполнения* программ. Поэтому при реализации формальных моделей в СИДТ не имеет значения, насколько эффективны структуры данных, используемые для представления программ (типов, термов, окружения, и т. п.). Важно другое: насколько они удобны для доказательства *свойств* формальной модели. В то же время, при разработке компилятора/интерпретатора языка программирования, эффективность структур данных, используемых для внутреннего представления программ, играет существенную роль, так как влияет на *время* компиляции/интерпретации. Далее, для простоты, будем говорить только об интерпретаторе, понимая под этим ПО, которое принимает на вход программу на соответствующем языке, выполняет нужные статические проверки этой программы, и выполняет её.

Мы называем интерпретатор *сертифицированным*, если доказано, что он выполняет программы в полном соответствии с формальной моделью языка (как, например, интерпретатор JSRef выполняет JavaScript-код в соответствии с моделью JSCert [1]). Это означает, в том числе, что программа успешно типизируется интерпретатором только тогда, когда она типизируема в формальной модели, и эти типы совпадают. Таким образом, для того, чтобы реализовать сертифицированный интерпретатор, нужно прежде реализовать формальную модель. При этом возникает необходимость использовать одну и ту же структуру данных (в данном случае структуру, представляющую типы) для реализации как интерпретатора, так и формальной модели. То есть нужно либо пожертвовать эффективностью интерпретации, либо верифицировать эффективный интерпретатор, имея дело с менее удобными для формального рассуждения структурами данных.

Концепт-параметры. Многие статически типизируемые языки программирования предоставляют специальные средства отделения интерфейса от реализации (средства абстракции). Примерами служат сигнатуры («интерфейсы») и модули («реализации») в ML, классы типов и экземпляры в Haskell, интерфейсы и классы в объектно-ориентированных языках Java и C#. Это позволяет писать абстрактный код, зависящий только от «интерфейса» (далее называемого концептом), и затем автоматически получать из абстрактного кода конкретный, подставляя вместо «интерфейса» его конкретную «реализацию» (далее называемую моделью[1]).

В упрощённой форме *концепт* можно считать именованным словарём (ассоциативным массивом), который ставит в соответствие идентификаторам типы языка. Соответствующая концепту *модель* представляет собой именованный словарь, отображающий идентификаторы в выражения языка, причём множество ключей модели совпадает со множеством ключей концепта, а типы выражений соответствуют типам, объявленным в концепте. Ниже приведён псевдокод определения концепта `IntMonoid` и двух соответствующих ему моделей, `IMAdd` и `IMMult`.

```
IntMonoid {ident: Int, binop: Int * Int -> Int}
IMAdd of IntMonoid {ident = 0, binop = +}
IMMult of IntMonoid {ident = 1, binop = *}
```

Заметим, что концепты, подобно модулям или классам, задают пространства имён. При формализации ЯП с подобными конструкциями в СИДТ (например, STLC[2] с записями[3]), для представления словарей часто используют списки пар `(<ключ>, <значение>)` [5; 8; 9]. Однако, это представление является неэффективным. В данной работе исследуется возможность верификации в СИДТ Coq интерпретатора, использующего эффективное представление словарей. В частности:

1. Разработано расширение простого типизированного лямбда-исчисления с концепт-параметрами (cpSTLC). Его синтаксис, семантика, и правила типизации формализованы в Coq.

[1] Терминология «концептов» и «моделей» часто используется в области обобщённого программирования, например, в документации к библиотеке шаблонов C++ (STL [10]).

[2] Простое типизированное лямбда-исчисление, Simply Typed Lambda Calculus.

[3] Запись образует пространство имён.

2. Частично реализован сертифицированный интерпретатор cpSTLC[4], где для представления концептов и моделей используется словарь на основе самобалансирующегося бинарного дерева из стандартной библиотеки Coq (`FMapAVL` [7]). Это повышает эффективность интерпретатора, но усложняет формальное рассуждение о его свойствах. (Работа над интерпретатором продолжается, репозиторий с исходным кодом проекта размещён на github [3].)

Описание cpSTLC. Программа cpSTLC состоит из объявлений концептов и моделей (подобно примеру `IntMonoid` выше), а также терма (выражения). Терм `t` принадлежит языку STLC, расширенному тремя новыми конструкциями: концепт-параметром λ`c#C.t`, применением модели `t # M`, где `C` и `M` это идентификаторы (имена) концепта и модели соответственно, а также обращением к элементу концепт-параметра `c.f`. Например:

```
(λc#IntMonoid.λx:Int.if x = c.ident then ...) # IMAdd 5.
```

Модель cpSTLC отражает основные особенности «концепт-подобных» конструкций в реальных языках программирования: концепты и модели создаются на этапе статического анализа (до начала вычисления терма) и образуют пространства имён; соответствие моделей концептам устанавливается в определении модели при помощи явной декларации `of <имя концепта>`, при этом структура модели должна соответствовать структуре указанного концепта (в объектно-ориентированных языках подобным же образом проверяется соответствие класса интерфейсу); обращение к концептам и моделям выполняется с помощью указания их имён. Всё это отличает cpSTLC от STLC с записями, где соответствие значения-записи типу-записи устанавливается исключительно структурно.

Заключение. В данной работе разработано и формализовано в Coq расширение простого типизированного лямбда-исчисления, которое моделирует модуль-подобные конструкции реальных языков программирования. На примере этого расширения исследуется возможность эффективной реализации сертифицированного интерпретатора. В частности, для представления концептов и моделей интерпретатор

[4]Наибольший интерес для нас представляет верификация той части интерпретатора, которая отвечает за статическую проверку программы, анализ концептов и моделей. Эффективное вычисление термов STLC в данном контексте играет второстепенную роль.

использует словари на основе самобалансирующегося бинарного дерева, а не более удобного для формального рассуждения списка пар. Верификация такого интерпретатора требует доказательства ряда утверждений, связанных с выбранных представлением словарей, но не зависящих от языка. Поэтому свойства, доказанные нами для интерпретатора cpSTLC и не связанные с формальной моделью, могут быть использованы в других проектах, требующих разработки сертифицированных интерпретаторов.

Список литературы

1. A Trusted Mechanised JavaScript Specification / Bodin, Martin and Charguéraud, Arthur and Filaretti, Daniele and Gardner, Philippa and Maffeis, Sergio and Naudziuniene, Daiva and Schmitt, Alan and Smith, Gareth // SIGPLAN Not. — New York, NY, USA, 2014. — Янв. — Т. 49, № 1. — С. 87—100. — ISSN 0362-1340. — DOI: 10.1145/2578855.2535876. — URL: http://doi.acm.org/10.1145/2578855.2535876.
2. *Blazy S.*, *Leroy X.* Mechanized semantics for the Clight subset of the C language // Journal of Automated Reasoning. — 2009. — Т. 43, № 3. — С. 263—288. — URL: http://gallium.inria.fr/~xleroy/publi/Clight.pdf.
3. Concept Parameters for STLC. — URL: https://github.com/julbinb/concept-params/blob/master/cpSTLC/cpSTLCa.md.
4. *Delaware B.*, *Cook W.*, *Batory D.* Product Lines of Theorems // SIGPLAN Not. — New York, NY, USA, 2011. — Окт. — Т. 46, № 10. — С. 595—608. — ISSN 0362-1340. — DOI: 10.1145/2076021.2048113. — URL: http://doi.acm.org/10.1145/2076021.2048113.
5. Encoding Featherweight Java with Assignment and Immutability Using the Coq Proof Assistant / J. Mackay [и др.] // Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs. — Beijing, China : ACM, 2012. — С. 11—19. — (FTfJP '12). — ISBN 978-1-4503-1272-1. — DOI: 10.1145/2318202.2318206. — URL: http://doi.acm.org/10.1145/2318202.2318206.

6. *Garrigue J.* A Certified Implementation of ML with Structural Polymorphism // Programming Languages and Systems: 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings / под ред. K. Ueda. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. — С. 360—375. — ISBN 978-3-642-17164-2. — DOI: 10.1007/978-3-642-17164-2_25. — URL: http://dx.doi.org/10.1007/978-3-642-17164-2_25.

7. Library Coq.FSets.FMapAVL. — URL: https://coq.inria.fr/library/Coq.FSets.FMapAVL.html.

8. *Rompf T., Amin N.* Type Soundness for Dependent Object Types (DOT) // SIGPLAN Not. — New York, NY, USA, 2016. — Окт. — Т. 51, № 10. — С. 624—641. — ISSN 0362-1340. — DOI: 10.1145/3022671.2984008. — URL: http://doi.acm.org/10.1145/3022671.2984008.

9. Software Foundations / B. C. Pierce [и др.]. — Electronic textbook, 2016. — Гл. Records. Adding Records to STLC. — Version 4.0. http://www.cis.upenn.edu/~bcpierce/sf.

10. *Stepanov A. A., Lee M.* The Standard Template Library: Technical Report / HP Laboratories. — 11.1995. — 95—11(R.1).

Трассирующая Нормализация, Основанная на Игровой Семантике и Частичных Вычислениях

Даниил Березун[¶], danya.berezun@gmail.com
Neil Jones[2], neil@diku.dk
[1]Санкт-Петербургский Государственный Университет, Россия
[2]DIKU, University of Copenhagen, Denmark

Аннотация

Данная работа сфокусирована на исследовании операционных особенностей игровой семантики. Мы применили подходы, используемые в игровой семантике, к нетипизированному лямбда-исчислению, показав, что произвольный лямбда-терм может быть скомпилирован в достаточно низкоуровневый язык LLL, программы на котором не содержат стандартные для функциональных языков техники, такие как окружения, замыкания и т.п. Скомпилированная программа представляет собой функционал первого класса и имеет предопределенное множество переменных, число которых зависит от размера входного лямбда-терма. Фактически сгенерированный низкоуровневый код на LLL трассирует подвыражения входного лямбда-терма, осуществляя трассирующую нормализацию последнего.
Ключевые слова: Нормализация, лямбда-исчисление, игровая семантика, трассирующая семантика, частичные вычисления, семантика.

[¶]Все исследования автора происходят при непосредственной поддержке компании JetBrains

Задача построения полностью абстрактной (англ. fully abstract) модели для языка PCF была впервые сформулирована Гордоном Плоткиным [2]. Одно из первых решений данной задачи основано на игровой семантике программ [3, 4, 5, 6], после чего были разработаны полностью абстрактные модели для широкого спектра языков программирования, положившие начало целой области для исследований.

Для краткости введём следующие обозначения: ULC (untyped lambda-calculus) и STLC (symply-typed lambda-calculus) для нетипизированнного и простого типизированного лямбда исчисления, соответственно.

Статья [1] использует фреймворк, основанный на игровой семантике, для определения нормализующей процедуры для STLC, используя трассирующую концепцию из [4, 8]. Для краткости мы обозначаем её STNP (simply-typed normalisation procedure). Она содержит полное доказательство корректности алгоритма STNP, основанное на правилах вывода вида $\kappa \vdash M : A \ldots$, где A — тип терма M, а κ — типовое окружение. Доказательство использует типы для построения программно-зависимых арен и категорий, чьими объектами являются сами арены, а морфизмами — наивные стратегии (англ. innocent strategies) над соответствующими аренами. Алгоритм STNP является детерминированным и определяется с помощью правил вывода, основанных на синтаксисе. STNP основан на типах, более того, он требует преобразования входного терма в η-длинную форму.

STNP может рассматриваться как интерпретатор: он нормализует λ-терм M, манипулируя списком его подвыражений, снабжённых указателями назад (backpointers). Одной из особенностей данного подхода является возможность реализации интерпрететора языка без использования стандартных подходов, таких как β-редукции, окружения для связывания переменных и замыкания для вызовов функций и параметров.

Данная работа посвящена исследованию операционных аспектов игровой семмантики и расширению процедуры трассирующей нормализации до бестипового лямбда исчисления. Авторы также считают, что данное исследование позволит предложить новый подход к генерации компиляторов, основанной на семантике.

В данной работе рассматривается бестиповое лямбда исчисление. Термы: $e ::= x \mid e_1 \bullet e_2 \mid \lambda x.e$. Свободные и связанные переменные определяются стандартным образом. В работе приводится алгоритм, получивший название UNP (untyped normalisation procedure), обоб-

щающий процедуру трассирующей нормализации SNTP до нетипизированного λ-исчисления. UNP корректно нормализует любой STLC-терм, лишённый типов [9]. Таким образом, UNP является истинным расширением STNP, поскольку ULC является Тюринг-полным языком, а STLC — нет. В работе приводится пошаговая разработка UNP путём введения пяти последовательных преобразований стандартной семантики большого шага для ULC. Также в работе кратко описывается, как частичные вычисления могут быть использованы для реализации ULC и его компиляции в низкоуровневый машинный язык.

Список литературы

[1] C.-H. Luke Ong. Normalisation by traversals. CoRR, abs/1511.02629, 2015.

[2] Gordon D. Plotkin. LCF considered as a programming language. Theor. Comput. Sci., 5(3):223–255, 1977.

[3] Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. Inf. Comput., 105(2):159–267, 1993.

[4] Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for PCF. In Theoretical Aspects of Computer Software, International Conference TACS '94, pages 1–15, 1994.

[5] S. Abramsky and G. McCusker. Game semantics. In Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School, pages 1–56. Springer Verlag, 1999.

[6] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. Inf. Comput., 163(2):285–408, 2000.

[7] C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In 21th IEEE Symposium on Logic in Computer Science (LICS 2006), pages 81–90, 2006.

[8] Robin P. Neatherway, Steven J. Ramsay, and C.-H. Luke Ong. A traversal-based algorithm for higher-order model checking. In ACM SIGPLAN International Conference on Functional Programming, ICFP'12, pages 353–364, 2012.

[9] D. Berezun and Neil D. Jones. 2017. Compiling untyped lambda calculus to lower-level code by game semantics and partial evaluation (invited paper). In Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2017). ACM, New York, NY, USA, 1-11.

«Яр» — русскоязычный язык программирования с горячей заменой кода

Будяк Д. В.

Аннотация

В статье описан проект русскоязычного компилируемого ЯП «Яр» с BASIC-подобным синтаксисом, статической типизацией, горячей заменой кода, модулями, полиморфными функциями, REPL, реализованного в виде транслятора в Common Lisp. Рассматриваются примеры применения горячей замены кода в современном ПО — сервера БД, оболочки и ядра операционных систем и др. Текущее состояние проекта Яр — работающий прототип.

Ключевые слова: Язык программирования Яр, Горячая замена кода

Введение

Цель проекта «Яр» — создать язык программирования со следующими особенностями: горячая замена кода (возможность изменения программы на ходу); русскоязычный интерфейс и ключевые слова; статическая типизация.

Реализация проекта позволит: минимизировать проблему языкового барьера для русскоязычных пользователей; повысить производительности труда при создании и поддержке крупных программ за счёт горячей замены кода; приобщить российских ИТ-специалистов к культуре «искусственного интеллекта», повлиявшей на многие важные проекты, например, телескоп Хаббл [1], PostgreSQL [2], Maxima, EMACS и др.

Основные черты языка и реализации

Яр — русскоязычный, императивный, компилируемый язык со строгой типизацией и сборкой мусора, с полиморфизмом, с REPL. Яр реализуется как транслятор в Common Lisp и может вызывать библиотеки на Lisp, позволяя своим пользователям приобщиться к культуре "искусственного интеллекта".

Черновик спецификации языка опубликован [3] и дорабатывается.

Синтаксис Яра похож на Бейсик. Ниже приведён пример кода, создающего единичную матрицу.

```
пусть М -- массив = н.массив н.список(Эн,Эн) начальный-элемент 0.0
  кнм
цикл
  для И -- целое от 0 до (: Эн - 1 :)
  записать М[И И] = 1
кнц
```

Горячая замена кода

Горячая замена кода имеет большое значение в ИТ. Примеры: ALTER TABLE; dpkg -i; modprobe; Microsoft Office Basic.

В Яре во время выполнения разрешено переопределять типы, добавлять поля в структуры и удалять их, менять перечень параметров функции и т.п. Горячая замена полезна для ускорения разработки, для обновления программы в производстве без её остановки, для модификации сторонних библиотек, для отладки, для оптимизации, как средство повышения выразительности языка.

Возможность горячей замены кода среда выполнения Яра наследует от Common Lisp. Например, вызов функции производится косвенно, по адресу, записанному в объекте функции. При переопределении функции создаётся новое тело и адрес в объекте функции заменяется. При следующем вызове функции будет исполняться новое тело. Текущие выполнения функции не повреждаются.

Текущее состояние проекта Яр

Имеется прототип языка и среды разработки, демонстрирующий основные инструменты (например, пошаговый отладчик), он опубликован под разрешительной лицензией [3].

Заключение

Разрабатываемый язык программирования «Яр» обладает уникальным сочетанием характеристик - горячей заменой кода, статической типизацией, простым русскоязычный синтаксисом и компиляцией в двоичный код.

Литература

1. Mark D. Johnston and Glenn E. Miller. SPIKE: Intelligent Scheduling of Hubble Space Telescope Observations. – Space Telescope Science Institute [сайт]. URL: http://www.stsci.edu/m̃iller/papers-and-meetings/93-Intelligent-Scheduling/spike/spike-chapter3.html (дата обращения 18.01.2017)

2. Create table [руководство пользователя]

   URL: https://www.postgresql.org/docs/6.4/static/sql-createtable.htm (дата обращения 16.01.2017)

3. Язык программирования Яр [сайт] URL: https://bitbucket.org/budden/yar

Стратегия использования крупных заданий при параллельном обходе дерева

Бурховецкий В. В., Штейнберг Б. Я.
Институт математики, механики и компьютерных наук
им. И. И. Воровича, Южный Федеральный Университет

Аннотация

В работе предлагается использовать двухстороннюю очередь (дек) для параллельного обхода дерева вариантов при древовидном поиске и, в частности, применительно к методу ветвей и границ. В отличие от предшествующих работ, предлагается генерировать параллельно выполняемые задания не только из головы очереди, но и из хвоста (в зависимости от ситуации), для улучшения локализации данных. Поскольку задача обхода дерева возникает во многих приложениях, предполагается использовать предлагаемый подход автоматически в распараллеливающих компиляторах.

Ключевые слова: параллельные вычисления, обход дерева, метод ветвей и границ.

Введение

Параллельный обход дерева состоит в том, что разные ветви дерева обрабатываются разными вычислительными устройствами. Обход дерева вариантов используется для очень широкого класса задач и инструменты ускорения этого метода представляют собой не только теоретический, но и практический интерес. Для получения хорошего ускорения от распараллеливания обхода дерева иногда требуется высокая квалификация программиста. Поэтому автоматизация приемов получения эффективного распараллеливания очень актуальна. Реализация таких приемов уместна в компиляторах в виде директив или

библиотек. Более того, сами компиляторы содержат много функций, использующих обход дерева вариантов (например, обход синтаксических и семантических деревьев), которые также могут быть распараллелены (при компиляции им самого себя).

При распараллеливании обхода дерева возникает проблема неравномерной загрузки вычислительных устройств, поскольку объемы вычислений на каждой ветви заранее не определены, а при использовании метода ветвей и границ некоторые ветви отсекаются в процессе вычислений. Поэтому распараллеливание целесообразно проводить динамически. Для решения проблемы загрузочного баланса можно использовать разбиение на мелкие процессы (или треды). Но для разных задач алгоритмы обхода дерева по-разному работают с памятью: в некоторых случаях в вершине дерева вариантов создается много новых промежуточных данных, а в некоторых — мало. Это означает, что в разных алгоритмах по-разному локализуются данные, т. е. разное соотношение между вычислениями и попаданиями в кэш (при общей памяти) или межпроцессорными пересылками (при распределенной памяти).

В работах [2], [5] описана динамическая структура для параллельного обхода бинарного дерева вариантов в методе ветвей и границ для высокопроизводительного кластера с распределенной памятью. Описан пул выполняемых заданий. Новые задания формируются как задачи обхода ветвей, выходящих из текущей вершины дерева. Сначала эти задания крупные, а с некоторого момента, когда пул заданий пополняется динамически, новые задания могут становиться мелкими. В работе [4] описан эвристический алгоритм неполного обхода дерева, который тоже можно выполнять параллельно. В работе [1] рассмотрен алгоритм обхода дерева, использующий SIMD-ускоритель (векторизатор) и учитывающий локализацию данных (попадание в кэш-память).

В данной работе предлагается для обхода дерева использовать двухстороннюю очередь (дек) и генерировать либо мелкие (с головы очереди), либо крупные задания (с хвоста очереди) в зависимости от ситуации с балансом и локализацией данных. К потребности в генерации крупных заданий авторы пришли после анализа распараллеливания модификации алгоритма Литтла для решения задачи коммивояжера [3]. Задания могут выполняться как процессами, так и тредами. Для многих приложений такой подход может позволить лучше локализовать данные при параллельных вычислениях.

1 Обход поддерева (ветви) дерева вариантов с помощью двухсторонней очереди (дека)

Дерево вариантов будем рассматривать как исходящее дерево. Содержимым дека будет множество вершин некоторого ориентированного пути дерева вариантов (начало пути — хвост очереди, а конец — голова).

В начале работы алгоритма единственный элемент дека — начало (корень) поддерева. Этот элемент является одновременно и головой, и хвостом.

Поддерево (ветвь) дерева вариантов обходится в одном вычислительном устройстве последовательным алгоритмом. Для обхода поддерева создается новый дек, который используется как стек при обходе методом поиска в глубину.

Предположим, что обход поддерева завершен. Это означает, что в деке процесса, выполняющего обход данного поддерева, остался один элемент (вершина дерева вариантов), который является одновременно головой и хвостом, и из этой вершины дерева нет ветвей, которых не обходил данный процесс.

2 Создание нового процесса

Новый процесс может создаваться, если в вычислительной системе существует свободное вычислительное устройство, или если пул заданий не полон. Новый процесс создается из какого-нибудь существующего выполняющегося процесса.

Если требуется создавать много мелких заданий, то новый процесс следует создавать из головы дека, как это делается в [2], [5]. Если преследуется цель создания процессов, выполняющих как можно больший объем работы, то для порождения нового процесса следует взять еще не пройденную ветку, которая ближе к хвосту дека.

Необходимым условием для создания нового процесса из существующего выполняющегося процесса является количество элементов дека этого процесса больше одного.

Заключение и актуальность

Алгоритмы обхода дерева используются для широкого класса переборных задач, имеющих высокую, в т. ч. экспоненциальную, вычислительную сложность. У многих таких алгоритмов высокое отношение количества вычислительных операций к количеству входных данных, что является необходимым условием эффективного распараллеливания для вычислительных систем с общей памятью. Но некоторые такие алгоритмы имеют большое (экспоненциальное) количество промежуточных данных [3]. Если большое количество промежуточных данных сохранять на оперативной памяти, то это погубит ускорение от распараллеливания. Частичным решением проблемы может быть использование многоядерных процессоров, имеющих адресуемую локальную память на своей микросхеме (а не на отдельной микросхеме памяти). Такая память есть, например, у процессора IBM Cell, графических карт Nvidia и у ПЛИС-ускорителей. Использование такой памяти уменьшает дорогостоящие обмены данными между микросхемой процессора и микросхемой памяти. В недалеком будущем ожидается появление новых процессоров с локальной памятью, в том числе и отечественных. Такие процессоры повысят эффективность многих алгоритмов, использующих обход дерева.

Список литературы

1. Scalable Graph Exploration on Multicore Processors / V. Agarwal [и др.] // 2010 Proceeding SC '10 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. — 2010. — Pages 1-11, November 13-19. — URL: `http://russianscdays.org/files/pdf16/41.pdf` ; IEEE Computer Society Washington, DC, USA ©2010.
2. Using simulation for performance analysis and visualization of parallel Branch-and-Bound methods / Y. G. Evtushenko [и др.] // Russian Supercomputing Days 2016. — 2016. — URL: `http://russianscdays.org/files/pdf16/41.pdf`.
3. *Бурховецкий В. В.*, *Штейнберг Б. Я.* «Исследование возможности распараллеливания алгоритма Литтла с модификацией Костюка для решения задачи коммивояжера» на конференции

«НСКФ-2016», Переславль-Залеский с 29 ноября по 1 декабря 2016. — 2016. — URL: http://2016.nscf.ru/TesisAll/04_Reshenie_zadach_optimizatsii/736_BurkhovetskiyVV.pdf.

4. *Макошенко Д. В.* Аналитическое предсказание времени исполнения программ и основанные на нем методы оптимизации: диссертация на соискание степени к. ф.-м. н., выполнена в РГУ, защищена в ИСИ СО РАН, Новосибирск. — 2011.

5. *Посыпкин М. А.*, *Сигал И. Х.* Комбинированный параллельный алгоритм решения задачи о ранце // Труды четвертой международной конференции «ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ И ЗАДАЧИ УПРАВЛЕНИЯ». — 2008. — С. 177—189.

Ostap: синтаксическое расширение OCaml для создания парсер-комбинаторов с поддержкой левой рекурсии

Вербицкая Е. А.,
ekaterina.verbitskaya@jetbrains.com,
Санкт-Петербургский государственный университет,
Лаборатория языковых инструментов JetBrains

Аннотация

Парсер-комбинаторы — привлекательная техника реализации синтаксических анализаторов. Одним из недостатков подхода является невозможность использования левой рекурсии при объявлении парсер-комбинаторов. В докладе будет представлено синтаксическое расширение языка OCaml для создания парсер-комбинаторов, позволяющее реализовывать в том числе леворекурсивные парсер-комбинаторы.
Ключевые слова: парсер-комбинаторы, OCaml, левая рекурсия.

Одним из способов реализации синтаксического анализа (проверки принадлежности предложения заданному языку и построения для него некоторого структурного представления) является техника парсер-комбинаторов [2]. Основная идея подхода заключается в моделировании синтаксических анализаторов функциями высшего порядка и комбинирования их при помощи небольшого множества комбинаторов. Существует множество различных стилей реализации подхода, среди которых особо выделяются монадические парсер-комбинаторы [3], позволяющие осуществлять синтаксический анализ,

зависящий от контекста, что может быть полезно при разборе, например, двумерного синтаксиса. Помимо способности разбирать более широкий класс языков, чем класс контекстно-свободных, парсер-комбинаторы также предоставляют возможность вычислять семантические значения, не строя абстрактное синтаксическое дерево. Другим существенным преимуществом является возможность создавать парсеры высшего порядка, то есть синтаксические анализаторы, которые принимают аргументом другие парсеры. Данная особенность повышает композиционность и сокращает размер реализации синтаксического анализатора.

Библиотека Ostap предоставляет набор парсер-комбинаторов и синтаксическое расширение языка OCaml для упрощения реализации синтаксических анализаторов. Библиотека реализует монадические парсер-комбинаторы с неограниченным предпросмотром, при этом комбинатор выбора игнорирует вторую альтернативу, если анализ с помощью первой завершился успешно. Таким образом, результатом анализа всегда является единственный возможный вывод данного предложения (если он существует). Библиотека Ostap позволяет систематически создавать пользовательские парсеры высшего порядка из небольшого набора стандартных парсер-комбинаторов. Например, можно описать парсер выражений *expr ops opnd*, параметризуемый двумя парсерами: *ops*, разбирающим бинарные операции, и *opnd*, специфицирующим операнды. Далее парсер-комбинатор *expr* можно использовать для спецификации пользовательского парсера арифметических выражений над натуральными числами, над числами с плавающей точкой или, например, лямбда-выражений: для этого достаточно применить *expr* к соответствующим аргументам.

Парсер-комбинаторы Ostap полиморфны относительно типа входного потока. С одной стороны, данная особенность позволяет единообразно анализировать потоки символов из разных источников. С другой стороны, она значительно повышает модульность пользовательских парсер-комбинаторов: если входной поток конкретен, то комбинирование парсеров высшего порядка без их модификации затруднено. Помимо этого, становится возможным реализовывать дополнительную функциональность в объекте, моделирующем вход: например, производить лексический анализ (выделение лексем). В библиотеке есть стандартная реализация потока Matcher, инициализируемая строкой, производящая базовый лексический анализ и содержащая информацию о координатах анализируемой лексемы.

Одним из недостатков нисходящего синтаксического анализа, реализацией идей которого являются парсер-комбинаторы, является невозможность обработки леворекурсивных определений парсеров. Леворекурсивные правила являются наиболее естественным способом описать левоассоциативные операции, поэтому их поддержка может значительно упростить создание синтаксических анализаторов. Существует несколько решений описанной проблемы [1; 6], ни одно из которых не в состоянии обрабатывать леворекурсивные парсеры высшего порядка. Библиотека парсер-комбинаторов Meerkat позволяет использовать левую рекурсию при создании анализаторов, однако достигается это путем явного построения леса разбора для данной строки [4].

Для поддержки левой рекурсии в библиотеке Ostap мы использовали идеи, применённые для её добавления в формализм Parser Expression Grammar [5]. Данный подход оперирует понятием ограниченной рекурсии: такое использование нетерминала в выводе, что количество леворекурсивных вызовов в нем ограничено некоторой константой. Если строка имеет вывод в данной грамматике, то количество вызовов каждого леворекурсивного нетерминала при разборе конечно, поэтому для обработки леворекурсивного нетерминала достаточно найти оптимальное ограничение левой рекурсии, что и осуществляется динамически во время процесса вывода. Во время поиска промежуточные данные сохраняются в таблицу мемоизации, которая должна быть глобальна для всех используемых парсер-комбинаторов, поэтому эта функциональность реализована в абстракции входного потока, являющейся наследником класса Matcher. Для использования левой рекурсии в случае парсеров высшего порядка в библиотеке предусмотрен парсер-комбинатор *fix*. К сожалению, производительность решения на данный момент в случае использования взаимной рекурсии не является удовлетворительной и ее улучшение является задачей на будущее.

Реализация библиотеки с поддержкой леворекурсивных парсер-комбинаторов может быть найдена по ссылке `https://github.com/kajigor/ostap`. Автор принимал участие в проекте под именем kajigor.

Список литературы

1. *Frost R. A.*, *Hafiz R.*, *Callaghan P.* Parser combinators for ambiguous left-recursive grammars // International Symposium on

Practical Aspects of Declarative Languages. — Springer. 2008. — С. 167—181.

2. *Hutton G.* Higher-order functions for parsing // Journal of functional programming. — 1992. — Т. 2, № 03. — С. 323—343.
3. *Hutton G.*, *Meijer E.* Monadic parser combinators: Technical Report / University of Nottingham. — University of Nottingham, 1996. — NOTTCS-TR-96-4.
4. *Izmaylova A.*, *Afroozeh A.*, *Storm T. v. d.* Practical, General Parser Combinators // Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. — St. Petersburg, FL, USA : ACM, 2016. — С. 1—12. — (PEPM '16).
5. *Medeiros S.*, *Mascarenhas F.*, *Ierusalimschy R.* Left Recursion in Parsing Expression Grammars // Programming Languages: 16th Brazilian Symposium, SBLP 2012, Natal, Brazil, September 23-28, 2012. Proceedings / под ред. F. H. de Carvalho Junior, L. S. Barbosa. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. — С. 27—41.
6. *Warth A.*, *Douglass J. R.*, *Millstein T. D.* Packrat parsers can support left recursion. // PEPM. — 2008. — Т. 8. — С. 103—110.

Учебно-макетный вариант инструментальной системы "Преобразователь кода TCJ"

Георгиев В. О.[1] Поликашин Д. С.[1] Рафиков Д. С.[1]
Бурнашев Р. А.[1] Прокопьев Н. А.[1]
[1]Казанский (Приволжский) Федеральный Университет, г. Казань,
кафедра технологий программирования

В работе представляется учебно-макетная версия программной кодовой реализации инструментальной системы «Преобразователь кода TCJ», позволяющая транслировать исходные коды из одного языка программирования в другой без потери качества, с использованием: AST – абстрактного дерево разбора, различных библиотек, заменяющих функционал, которого нет в родных библиотеках другого языка, а также, ряда вспомогательных шаблонов.

Система разработана в ходе проведения лекционных и практических курсов «Проектирование и архитектура программных систем» и «Обьектно-ориентированный анализ и проектирование» читаемых студентам Казанского (Приволжского) Федерального Университета по направлению «Программная инженерия» и «Прикладная математика и информатика».

Актуальность разработки определена тем, что при разработке программного обеспечения (ПО) часто имеет место проблема объединения нескольких подзадач созданных на различных языках программирования, что иногда приводит к большим временным и экономическим затратам. Одним из решений данной проблемы является конвертирование исходных кодов ПО на другие языки программирования, что

помогает в результате снизить временные и экономические издержки при производстве программных продуктов.

В качестве методологических основ в нашей системе используются концепции представленные в работах [1,2]. В настоящее время на интернет-рынке сравнительно немного аналогов нашей системы. Среди прочих можно выделить JLCA, Convert.Net, Janett. Их особенность состоит в узкой направленности только на один язык при преобразовании кода (например, C# <– JAVA или JAVA –> C#).

Предлагаемая нами система позволяет в стандартном режиме преобразовывать исходный код на языке C# в эквивалентный ему код на JAVA, а также, наоборот. Разработанное приложение позволит Вам выполнять преобразование текста той или иной программы, по максимуму сохраняя данные.

Разобраться с программным интерфейсом достаточно просто: окошко приложения представляет собой две обширные зоны. В находящуюся слева вы должны будете ввести, либо вставить копию программного кода, после чего за считанные секунды в правой появится его аналог, полученный в результате конвертации. Для того, чтобы задать направление нужного вам преобразования, необходимо воспользоваться специальным окном направления. Практическое использование разработки предполагает ее использование в учебных курсах перечисленных в начале тезисов, а также в учебном курсе «Конструирование Программного Обеспечения» и в научно исследовательском направлении проводимым кафедрой технологий программирования Института вычислительной математики и информационных технологий Казанского (Приволжского) Федерального Университета «Концептуальные основы построения учебно-модельных макетов сборочного генератора программного обеспечения сложных систем [3,4].

ОСНОВНЫЕ ХАРАКТЕРИСТИКИ ПРЕДСТАВЛЕННОГО ПРОЕКТА

Стадия проекта : разработка и опытная эксплуатация.
Описание технологии:Происходит разбор исходного кода в AST, трансформация конструкций, которые не имеют аналогов в другом языке, замена вызовов функций одного языка на другие(стандартные библиотеки, виртуализация, эмуляция), проверка результирующего кода, внесение необходимых изменений.
Описание новизны: Осуществляется поддержка нескольких языков (C#,Java,C/C++,Python,VB), подключение дополнительных

языков осуществляется дополнительными модулями, которые подключаются дополнительно. Присутствует возможность выполнять все алгоритмические действия на стороне сервера (облегченная версия продукта).

ЛИТЕРАТУРА:

[1] Д.С.ПОЛИКАШИН, А.И.ЕНИКЕЕВ, В.О. ГЕОРГИЕВ ИССЛЕДОВАНИЕ ПРОБЛЕМ АВТОМАТИЗАЦИИ РЕШЕНИЯ ЗАДАЧИ СОВМЕСТИМОСТИ ПРОГРАММНЫХ СИСТЕМ. //Материалы XXII Международной научно-технической конференции "ИНФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ"ИСТ-2016 г. ISBN 978-5-9902087-7-3, стр. 251.
[2] RESEARCHING OF THE PROBLEM OF SOLUTION AUTOMATION OF SOFTWARE SYSTEMS COMPATIBILITY Denis Polikashin, Arslan Enikeev, Victor Georgiev //Europe and MENA Cooperation Advances in Information and Communication Technologies //ISBN 978-3-319-46567-8// s 159-166, 2016 г.
[3] RESEARCH AND OPTIONS OF PRACTICAL SOLUTION OF SOFTWARE SYSTEMS COMPATIBILITY PROBLEM D.S. Polikashin, V.O. Georgiev /International Journal of Pharmacy & Technology IJPT| Sep-2016 | Vol. 8 | Issue No.4 | 24309-24316 Page 24309 ISSN: 0975-766X CODEN: IJPTFI.
[4] В.О. ГЕОРГИЕВ УЧЕБНО-МОДЕЛЬНЫЙ ВАРИАНТ ИНТЕРАКТИВНОЙ СИСТЕМЫ ГЕНЕРАЦИИ ПО СЛОЖНЫХ СИСТЕМ, С ПРЕДВАРИТЕЛЬНОЙ ПРЕДИНТЕРПРЕТАЦИЕЙ ПРОГРАММНЫХ МОДУЛЕЙ. //Материалы XXII Международной научно-технической конференции "ИНФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ"ИСТ-2016 г. ISBN 978-5-9902087-7-3, стр. 248-249.

$\Sigma-$спецификация языков программирования

Глушкова В. Н.
Донской государственный технический университет,
Ростов-на-Дону

Аннотация

Выделен новый класс $\Delta_0 T-$формул в рамках концепции $\Sigma-$программирования [1], составляющий основу логической спецификации семантики языков программирования. Специфика этих формул состоит в том, что их префикс с ограниченными кванторами иерархизирован в соответствии с правилами КС-грамматики языка. Использование $\Delta_0 T-$формул упрощает спецификацию статической семантики языка по сравнению с квазитождествами с отрицанием, применяемыми в [2], для которых требуется наличие дополнительных позитивных предикатов перед их негативными вхождениями. Наличие иерархизированного префикса позволяет более эффективно организовать интерпретацию логической спецификации и получить оценку сложности ее реализации с учетом синтаксиса языка.

Ключевые слова: логическая спецификация семантики языков программирования, формулы многосортного языка ИП 1-го порядка с ограниченными кванторами, КС-грамматика

$\Sigma-$спецификация программы описывает процесс ее выполнения посредством квазитождеств, левая часть которых содержит исполняемый оператор, а правая часть - результат его исполнения. Квазитождества задаются $\Delta_0 T-$формулами с явным использованием переменных сорта "список" многосортного языка исчисления предикатов 1-го порядка. Контекстно-зависимые требования языка для ясности отделены от спецификации правил вычисления результатов исполнения операционных конструкций языка.

Логическая спецификация интерпретируется на многосортных КС-списках, которые представляют деревья разбора программ. Сорт списка определяется символом-меткой корня представляемого дерева. Отношению принадлежности списков соответствует отношение непосредственного подчинения узлов. Рассмотрим арифметические выражения, описываемые правилами:

$$E \to E + T \mid T * F \mid (E) \mid id$$

$$T \to T * F \mid (E) \mid id$$

$$F \to (E) \mid id,$$

где терминальный символ id разпознается на лексическом уровне. Дереву вывода:

$$\begin{aligned} E &\to E + T \to T * F + T \to id_1 * F + T \to id_1 * id_2 + T \\ &\to id_1 * id_2 + id_3 \end{aligned}$$

соответствует индексированный сортами список $\langle\langle Id_1\rangle_T * \langle Id_2\rangle_F\rangle_E + \langle Id_3\rangle_T\rangle_E$.

Для программы $pr = \langle op_1, ..., op_i, ..., op_n\rangle$ спецификация описывает процесс выполнения операторов op_i шаг за шагом с помощью $\Delta_0 T$-формул вида:

$$(\forall\, x_1 \bullet\!\in t_1)\ldots(\forall\, x_m \bullet\!\in t_m)(y_1 \prec z_1)\ldots(y_p \prec z_p)\varphi(\bar{x},\bar{t}) \to \psi(\bar{x},\bar{t})$$

$m \geq 1$, $p \geq 0$, $y_j, z_j \in <\bar{x},\bar{t}>$, $1 \leq j \leq p$; $\bullet\!\in$ обозначает отношение принадлежности элемента списку или его транзитивное замыкание, $\prec$ – отношение "левее". Формулы φ (ψ) конъюнкция атомных формул вида r, $\tau_1 = \tau_2$ (r, $f = \tau$) или их отрицаний; $r, f-$ предикатный и функциональный символы, τ, τ_1, τ_2- термы. Формулами этого вида можно описывать предикаты и функции, определенные на поддеревьях дерева вывода.

Пусть функция $Val : Var \times N \to Q \cup \{\perp\}$ задает значения переменных из множества рациональных чисел Q на $n-$ом шаге вычисления, $\perp-$ неопределенное значение; предикат $Act \subseteq Op \times N$ выделяет исполняемый на $n-$ом шаге работы программы pr оператор $op \in Op$. Оператор присваивания as специфицируется $\Delta_0 T-$ формулой с неограниченным квантором $\forall\, n$ по шагам вычисления:

$(\forall\, n)(\forall\ op\ \bullet \in pr)(\forall\ as \in op)(\forall\ var, exp \in as), (nil \prec var)$

$(Act(as, n) \rightarrow Val(var, n) = Vald(exp, n-1), Ter(as, n), Ter(op, n))$

Результатом исполнения оператора присваивания на $n-$ом шаге вычисления является изменение значения переменной *var* из левой части оператора на значение выражения *exp* правой части оператора, вычисленного на предыдущем $(n-1)-$ом шаге. Функция $Vald-$продолжение функции *Val* по n; $Ter \subseteq Op \times N-$предикат завершения исполнения оператора на шаге n.

Теория из формул рассматриваемого вида интерпретируется на исходном КС-списке по правилу вывода modus ponens, исходя из фактов вида $r(\overline{c}), f(\overline{c}) = d$ для констант $\overline{c}$, d. При интерпретации арифметические операции считаются встроенными, отрицание интерпретируется по принципу "замкнутого мира". Если теория из квазитождеств обладает свойством нётеровости и конфлюентности, то для неё можно построить индуктивно вычислимую модель с иерархической надстройкой из констант.

$\Delta_0 T-$формулами можно специфицировать контекстные свойства и ограничения языка программирования. Пусть сначала определены функции *Type*, *Name*, предикат $\underline{declare - in} \subseteq Id \times Prog$ на определяющих вхождениях идентификаторов и предикат $\underline{use - in}$ на использующих вхождениях идентификаторов. Описание ($\underline{describe - in}$) использующих вхождений специфицируется формулой:

$(\forall\ id_1, id_2\ \bullet \in prog)(id_1\, \underline{declare - in}\ prog, id_2\ \underline{use - in}\ prog,$

$Name(id_1) = Name(id_2) \rightarrow id_2\ \underline{describe - in}\ prog,\ Type(id_2) = Type(id_1))$

Контекстное ограничение "каждый используемый в программной единице идентификатор должен быть описан единственным образом" специфицируется двумя формулами:

$(\forall\ id\ \bullet \in oper)(\forall\ oper\ \bullet \in prog)(id\ \underline{describe - in}\ prog)$;

$(\forall\ id_1, id_2\ \bullet \in descr)(\forall\ descr \in prog)(id_1 \neq id_2 \rightarrow$

$Name(id_1) \neq Name(id_2))$

Сложность проверки $\Delta_0 T-$ формул вида:

$$(\forall\, x_1 \bullet \in t_1) \ldots (\forall\, x_m \bullet \in t_m)\psi(\bar{x}, \bar{t})$$

реализуется со сложностью $O(n^{m+1})$ по времени и $O(n^2)$ по памяти относительно $n-$ количества объектов, составляющих список.

Список литературы

1. Goncharov S.S., Ershov Yu.L., Sviridenko D.I., *Semantic programming* // Information processing. – 1986. – V. 11. – № 10 . – p. 1093–1100.

2. Глушкова В.Н., Ильичева О.А.,*Средства диагностики ошибок и эффективной реализуемости в системах логического моделирования* // Логика и семантическое программирование. Вычислительные системы – 1992. – Вып. 146 . – стр. 3–17.

Сквозная функциональность и её анализ в грамматике языка программирования

Головешкин А. В.
Институт математики, механики и компьютерных наук
им. И. И. Воровича, Южный Федеральный Университет

Аннотация

Сквозной функционал (crosscutting concern) ортогонален существующему разбиению программного кода на единицы модульности. Составляющие сквозной функционал участки кода логически связаны, но физически разбросаны по различным файлам, классам, методам. В работе анализируется понятие сквозного функционала программы, рассматривается формальная модель, пригодная для его представления и анализа, демонстрируется её применение для анализа сквозной функциональности в грамматике языка программирования.
Ключевые слова: сквозная функциональность, прорезающая функциональность, спутанный код, распределённый код, аспект, аспектно-ориентированное программирование, разметка кода, грамматика, компилятор, scattering, tangling, crosscutting concerns, separation of concerns.

1 Введение

Функциональность (concern) — проблема, решаемая совокупностью фрагментов кода, и сама эта совокупность. В случае, если фрагменты рассредоточены по файлам, классам, методам проекта, функциональность называется сквозной или прорезающей (crosscutting). При этом не делается предположений о возможности её вынесения в отдельную единицу модульности, в то время как ключевой идеей АОП является

идея разделения кода аспекта, инкапсулирующего рассредоточенные действия, и основной программы [1]. Понятие прорезающей функциональности восходит к идеям монографии А. Л. Фуксмана [2]: хотя в программе существует зафиксированный на этапе проектирования набор реализующих функций — горизонтальные слои, — в ней также можно выделить слои вертикальные, фрагментарно представленные в горизонтальных. В этом заключается её двумерная структура [3].

Сквозная функциональность, присутствующая в компиляторе, непригодна для представления в виде аспекта. Оформление языковых конструкций, прорезающих этапы компиляции, в виде аспектов затруднительно и усложняет восприятие разрабатываемого языка как единого целого. Между тем, о них можно говорить как о функциональностях языка и компилятора, анализ их взаимосвязей способен облегчить процесс разработки.

2 Формализация сквозной функциональности

В исследовании [4] предложена наиболее общая формализация для представления и анализа сквозной функциональности. В рассмотрение вводятся два множества: S — множество функциональностей, выделяемых в программе, и T — множество элементов программы для той или иной степени детализации (например, классов или методов). Определяются отображение $f : S \to 2^T$, связывающее каждую функциональность со множеством реализующих её элементов, и отображение $g : T \to 2^S$, связывающее элемент со всеми функциональностями, в реализации которых он участвует.

Прорезающий функционал определяется через сочетание распределённости (scattering) и спутанности (tangling). Функциональность s является распределённой, если её отображение во множество T состоит из более чем одного элемента. Элемент $t \in T$ называется спутанным, если участвует в реализации нескольких функциональностей. Прорезание для $s_1, s_2 \in S : s_1 \neq s_2$ имеет место тогда и только тогда, когда

$$\begin{gathered} \mid f(s_1) \mid > 1\,, \\ \exists\, t \in f(s_1) : s_2 \in g(t). \end{gathered}$$

Помимо непосредственных связей функциональностей с элемента-

ми программы допускается учёт опосредованных связей. Под непосредственной связью понимается связь в случае, если элемент программы реализует логику, составляющую функциональность. Опосредованная связь элемента t и функциональности s означает, что существует элемент t', непосредственно связанный с s и некоторым образом зависящий от t. Если t и t' — символы грамматики, под зависимостью можно понимать наличие символа t в правиле для t'. Выявить и проанализировать прорезающий функционал предлагается путём построения матриц специального вида.

3 Эксперименты

Рассмотренная модель была применена для анализа сквозных функциональностей в грамматике языка PascalABC.NET. При помощи интегрированной среды [5] разработчиками языка было размечено 15 функциональностей, связанных с различными языковыми конструкциями. После получения матрицы прорезающего функционала потребовалось более подробное изучение степени взаимного прорезания для пар функциональностей: в процессе разработки периодически возникает необходимость модификации или временного отключения некорректно компилируемой языковой конструкции, при этом важно проверить, что данное действие не нарушит работу позднее реализованных функциональностей.

На основе анализа множеств непосредственно (H) и опосредованно (O) связанных с функциональностями элементов был сделан вывод о том, что возможность отключения напрямую связана с количеством общих элементов разного типа для пар функциональностей. Если функциональности пересекаются своими непосредственно связанными элементами, отключение любой из них повлияет на другую. Иначе, если пересечение объединений их множеств H и O пусто, отключение возможно, причём точность этого утверждения тем выше, чем больше шагов в глубину по опосредованным связям было сделано. Иначе для пары s_i, s_j автором настоящей статьи предлагается рассмотреть значение метрики

$$\textit{Степень зависимости}(s_i, s_j) = \frac{\mid H(s_i) \cap (H(s_j) \cup O(s_j)) \mid}{\mid H(s_i) \mid},$$

где $H(s_i)$, $H(s_j)$ — множества элементов, непосредственно связанных с соответствующими функциональностями, $O(s_i)$, $O(s_j)$ — множества

опосредованно связанных элементов. Значение 0 означает, что функциональность s_i может быть отключена без нарушения работы s_j; чем ближе значение метрики к 1, тем большая часть элементов s_i спутана с s_j и тем сложнее осуществить модификацию или отключение.

4 Заключение

Разметка программы и применение обобщённой формальной модели позволяют рассмотреть прорезающие функциональности независимо от применимости к ним АОП. В частности, такое рассмотрение становится возможным для грамматик языков программирования. При этом в ходе анализа непосредственно и опосредованно связанных с функциональностями символов могут быть выявлены взаимосвязи, влияющие на возможность независимой модификации или отключения функциональности, что позволяет скорректировать дальнейший процесс разработки.

Список литературы

[1] Masuhara H., Kiczales G., Modeling Crosscutting in Aspect-Oriented Mechanisms. // Object-Oriented Programming, ser. LNCS. — Springer-Verlag, 2003. — Vol. 2743. — С. 2–28.

[2] Фуксман А. Л., Технологические аспекты создания программных систем. / А. Л. Фуксман — М: Статистика, 1979. — 184 с.

[3] Горбунов-Посадов М. М., Как растёт программа. // ИПМ им. М. В. Келдыша РАН. URL: http://keldysh.ru/gorbunov/grow.htm (дата обращения: 10.11.2016)

[4] Conejero J. M., Hernández J., Jurado E., Berg K. G., Crosscutting, what is and what is not? // University of Twente. URL: http://doc.utwente.nl/ 64648/1/ConHerJurBer2007.pdf (дата обращения: 25.11.2016)

[5] Головешкин А. В., IDE с аспектной разметкой кода для работы с YACC-грамматиками. // Магистерская диссертация. — Южный федеральный университет, Ростов-на-Дону, 2015. — 53 с.

Программирование в терминах предметной области

Горишний С. А., ipgsa@rambler.ru
ООО «Визуал Дата»

Аннотация

Платформа Visual Data представляет собой вычислительную среду объектного управления данными, включающую в себя объектную СУБД, а также полный комплект инструментальных и коммуникативных средств в архитектуре сервер-клиент, обеспечивающих быстрое создание прикладных программ. В вычислительной среде платформы функции программы выполняют декларативные структуры данных, описывающие как исполнительную модель приложения, так и пользовательский интерфейс.

Введение

Платформа Visual Data разрабатывалась как кросс-платформенная полнофункциональная вычислительная среда, предназначенная для быстрого создания прикладных программ методами простого визуального конструирования, с возможностью их исполнения на самых различных устройствах, включая мобильные.

1 Технологическая основа платформы

В основу реализации платформы легли несколько взаимосвязанных технологий управления данными:

1. Технология *объектного управления данными* - основана на дальнейшем развитии реляционной модели Кодда, при котором в мета-модель СУБД вводится новое понятие - *связь атрибутов*. В расширенной таким образом модели, любая предметная область описывается как совокупность понятийных сущностей (*классов*) и их характеристик (*атрибутов*), а все действующие правила выражены *отношениями* классов и *связью* атрибутов.

2. Технология *объектного представления информации* предоставляет простые, и как следствие - производительные методы организации долговременного хранения, модификации и извлечения логически связанных данных в формате унифицированного объекта, с соблюдением всех требований ACID. Совокупность всех объектов образует объектную базу данных, в которой каждый изолированный объект обладает уникальным идентификатором. Для размещения данных объект предоставляет кортеж контейнеров, что позволяет хранить в объектах как собственно данные, так и все виды мета-данных.

3. Технология *сущность-представление* вводит в мета-модель системы управления данными такое понятие как *интерфейсное представление* сущности. Любая сущность мета-модели, или ее событие, обладает по меньшей мере одним представлением для каждого типа интерфейса - как визуального, так и не визуального. Абстрактное представление унифицировано связывает субъект мета-модели с компонентами интерфейса, экземпляры которых в декларативной форме входят в состав представления, и своими внутренними методами обеспечивает взаимодействие субъекта с этими экземплярами.

4. Технология *трехмерной визуализации метаданных* обеспечивает (в сравнении с плоскими методами визуализации UML) качественно новый уровень наглядности восприятия взаимодействующих субъектов, образующих модель предметной области, и в том числе за счет использования средств анимации и естественной навигации.

5. Технология *визуальных примитивов* позволяет, не прибегая к программированию, простым созданием декларативных экземпляров, производных от всего четырех простых визуальных компонент, реализовать сценарные формы произвольного уровня

сложности, включая интерактивные графики и диаграммы. При этом сильно упрощается и унифицируется процесс администрирования содержимого сценарных форм.

6. Технология *web-преобразования* позволяет на лету конвертировать любую форму визуального интерфейса, созданную по технологии сущность-представление, в интерактивную html-страницу с точным сохранением ее внешнего вида и компоновки при отображении в любом существующем браузере на любом устройстве.

Совокупность перечисленных технологий позволяет создавать прикладные программы без написания программных кодов, что резко увеличивает скорость процесса разработки, одновременно обеспечивая высокую степенью алгоритмической надежности исполнения программы.

2 Вычислительная модель платформы

В вычислительной среде платформы функции прикладной программы выполняет декларативное описание модели данных прикладной задачи, а пользовательский интерфейс образован совокупностью экранных, печатных и файловых форм представления сущностей модели - сценарных ресурсов. Создание программы осуществляется при помощи визуальных конструкторов, с немедленным переходом от режима исполнения к режиму дизайна и обратно по нажатию одной клавиши. Платформа характеризуется многоуровневой вычислительной моделью, в состав которой входят:

1. Интерпретатор исполнительной модели данных, который несет основную вычислительную нагрузку, исполняется непосредственно на уровне базы данных, и полностью удовлетворяет требованиям ACID;

2. Интерпретатор примитивных вычислений, выполняемых на уровне сценарных источников данных;

3. Интерпретатор команд внутреннего языка управление источниками данных (DataSource) сценарных ресурсов;

Все перечисленные интерпретаторы для расширения своих функциональных возможностей позволяют исполнять скрипты, написанные на языке Python, с использованием его библиотеки функций.

3 Архитектура и функциональный состав

Платформа включает в себя систему управления базой данных, а также полный набор инструментальных средств, необходимых для создания и эксплуатации прикладного ПО. Платформа характеризуется функциональным единством, компактностью, и минимальными требованиями к аппаратной части. Платформа использует только базовые ресурсы ОС: файловую систему, память, сокеты Беркли и подсистема графического вывода. Платформа программно реализована на языке C++ в среде разработки Qt, в традиционной архитектуре сервер-клиент с обменом данными по протоколу TCP.

Серверная часть платформы (сервер) включает в себя:

- систему управления объектной базой данных;
- подсистему авторизации, управления подключениями и пользовательскими DataSource;
- собственный менеджер памяти;
- подсистему динамической статистики;
- подсистему обмена по протоколу TCP.

Объектная СУБД обеспечивает долговременное хранение всей информации в формате унифицированного объекта, многопользовательский доступ к объектам, и включает в себя:

- файловое хранилище объектов;
- подсистему разделения доступа к объектам;
- интерпретатор исполнительной модели данных.

Файловое хранилище объектов обеспечивает:

- разбиение дискового пространства на кластеризованные банки рабочей базы данных;
- сохранение объектов в банках с их последующим извлечением ;
- сохранение транзакций в журнале транзакций;
- автоматическое создание контрольных точек восстановления.

При повторном запуске после программно-аппаратного сбоя, файловое хранилище реализует быстрое автоматическое восстановление целостности рабочей базы данных из контрольных точек и журнала транзакций.

Подсистема разделения доступа обеспечивает:

- отображение объектов в оперативную память;
- транзакционный характер создания и модификации объектов;
- сохранение текущего состояния базы данных до завершения выборки начавшим ее пользователем.

Для реализации перечисленных целей подсистема разделения доступа использует таблицу аллокации объектов, cache объектов, таблицу текущих состояний и таблицу управления объектами состояния.

Интерпретатор исполнительной модели данных решает следующие задачи:

- загружает модель данных в память при старте системы;
- реализует бизнес-логику приложения при чтении или модификации данных в соответствии с правилами, установленными декларациями сущностей и связей модели приложения;
- обеспечивает транзакционный характер внесения изменений в собственно модель.

Клиентская часть платформы (клиент) выполняет функцию пользовательского браузера сценария при исполнении, или визуального конструктора при создании (дизайне) приложения, и для этих целей включает в себя:

- трехмерный визуальный конструктор пространственной модели приложения;
- двухмерный визуальный конструктор/интерпретатор экранных, печатных и файловых интерфейсных форм;
- библиотеку визуальных компонент с пиктографическими диалогами управления свойствами компонента;
- подсистему кэширования и управления сценарными ресурсами;

- подсистему экранной визуализации;
- встроенный WEB-сервер;
- менеджер управления печатными отчетами;
- подсистему обмена по протоколу TCP.

Специфичной особенностью клиента является наличие встроенного в него WEB-сервера, что позволяет использовать его как масштабируемый прокси-сервер, обслуживающий множество клиентских подключений, осуществляемых через любой известный Internet-браузер. При использовании WEB-доступа подсистема экранной визуализации клиента динамически конвертирует экранные формы сценария прикладной программы в html-страницы таким образом, что они в любом браузере выглядят абсолютно идентично исходным десктопным.

Заключение

Перечисленный выше набор инструментов и объем встроенной функциональности позволяют использовать платформу как самодостаточную вычислительную среду, обеспечивающую решение самого широкого круга задач разработки и эксплуатации прикладного ПО.

Список литературы

[1] E.F.Codd, The Relational Model for Database Management, Addison Wesley Publishing Company, 1990, ISBN 0-201-14192-2.

Учебный язык параллельного программирования СИНХРО

Городняя Л. В.

Аннотация

Доклад представляет проект язык Синхро, специально ориентированного на начальное обучение параллельному программированию в терминах управления взаимодействием роботов на небольших игровых примерах, показывающих типичные проблемы создания и отладки параллельных программ.

1 Введение

Оценка образовательного значения парадигм программирования выделяет в качестве фундаментальных функциональное, параллельное и императивно-процедурное программирования, что показывает актуальность выбора средств и методов обучения параллельному программированию. Проект учебного языка Синхро нацелен на ознакомление с основными явлениями и моделями параллельного программирования, встречающимися в учебно-методических и научных материалах, языках высокого уровня (ЯВУ). Цель опережающего ознакомления с понятиями и явлениями параллелизма – профилактика жесткого привыкания к принципам традиционного последовательного императивно-процедурного программирования. Организация параллельных процессов часто требует независимости порядка вычислений от последовательности представления действий в программе и дисциплины доступа в памяти [1]. Следует отметить, что изначально система понятий языка начального обучения программированию Робик содержала резерв для изучения параллельных программ. Программа

могла включать/выключать разных исполнителей, обладающих своими системами команд, и назначать исполнителей отдельных действий [4]. Этот резерв не был востребован в конце 1970-ых годов, в наши дни он обретает актуальность [2]. Чуть позже, в конце 1980-ых, увидел свет перевод на русский язык превосходной книги блестящего учёного-исследователя проблем правильности програм Т.Хоара «Взаимодействующие последовательные процессы» [5], в которой на простых примерах управления автоматами показано, что параллельные композиции внешне не сложнее последовательных, если не акцентировать внимание на том, что отладка взаимодействующих процессов много сложнее. Важно, что модель Хоара чувствительна к обнаружению достаточно тонких ошибок при создании программ. Поэтому для целей обучения примеры Хоара дополнены рассмотрением вопросов отладки и методов минимизации ее объёма с помощью выделения типовых, многократно используемых фрагментов, реализуемых с учётом синтаксической правильности результата подстановки фрагментных переменных.

2 Основные решения

Ведущее понятие языка Синхро - исполнитель, способный выполнять команды (унаследовано от языка Робик и школьного курса информатики. Используется как общий синоним терминов «автомат», «процессор», «робот».), исполнителей может быть много и они могут обладать разными системами команд. Основные методы сжатого представления массовых вычислений обычно связаны с использованием неявных циклов, позволяющих избежать выписывания однотипных схем над стандартными структурами данных типа многомерных векторов. Этот механизм в Синхро используется более широко распространением на технику применения функций и вызова исполнителей, что повышает лаконизм выражений. Механизмы применения функций рассматриваются как операции, допускающие просачивание. Кроме того, из бинарных операций можно конструировать фильтры. Результат фильтрации исчезает из аргумента – он переносится в другую структуру данных или сохраняется как промежуточное значение. Структура из фильтров дает структуру из результатов их применения к одному и тому же аргументу. Кроме обычных присваиваний имеются и пересылки, при которых пересылаемые данные могут исчезать. При определении фрагментов кроме обычных переменных ис-

пользуются фрагментные, с помощью которых можно формировать синтаксические макросы, вид параметров которых задаётся как синтаксическое подобие (~~) вхождению фрагментной переменной. Вид такого параметра задается металингвистической формулой используемого языка. Например, фрагментная переменная «b» синтаксически подобна вхождению барьера « b: » или последнего элемента списка « , b)» в определение функции «учет».

```
учет = (  n  b ~~ { b: | , b ) } )
\\ параметризация барьера в заголовке функции
       ЕСЛИ n ТО   b:  учет ( n - 1 , b )
       \\ использование барьера в определении функции
             учет (10  контр )
       \\ задание имена барьера при вызове функции
```

Язык не поддерживает статической иерархии определений и областей действия имён. Локализация имён используется лишь при определении исполнителей и функций. Текст программы формируется как композиция действий и фрагментов. При организации сложных данных и действий используются общие структуры или средства композиции, такие как списки, вектора и множества, обеспечивающие представление отношений «после», «одновременно» и «взаимоисключено», причём последовательность вычисления компонентов может не зависеть от порядка их размещения в программе или в памяти: `(a ; b)` – последовательный доступ к результатам вычисления a ; b в порядке вхождения. `(a , b)` – последовательный доступ к результатам вычисления a , b в произвольном порядке. `(a | b)` – первый из результатов успешного вычисления a ; b в порядке вхождения. `[a , b]` – индексный доступ к результатам вычисления a , b в произвольном порядке. `[a ; b]` – индексный доступ к результатам вычисления a ; b в порядке вхождения. `[a | b]` – пара из номера и первого успешно вычисленного результата a или b. `{a ; b}` – произвольный доступ к различимым результатам вычисления a ; b в порядке вхождения. `{a , b}` – доступ к различимым результатам вычисления a , b в произвольном порядке. `{a | b}` – доступ к результату первого успешно вычисленного результата a или b.

Все команды допускают безусловное и условное выполнение. Действия в языке Синхро приспособлены к варьированию дисциплины доступа к данным и схемы управления процессами обработки комплексов с помощью схемы обхода, выглядящей подобно оператору IF

без ELSE. Действия, измененяющие состояния памяти, подчинены механизму транзакций, т.е. признание их безуспешными влечёт восстановление памяти в состояние, предшествующее этому действию. При отладке формируется ряд контекстов, на которых демонстрируются конкретные свойства фрагментов, из которых собирается полная программа. Это контексты для отдельных потоков, для пар синхронизованных потоков, для интегрированной из потоков программы, а кроме того, контексты для удостоверения наличия-отсутствия информационных связей между фрагментами. Потоки могут отлаживаться и выполняться независимо друг от друга в предположении, что каждый из них можно располагать отдельно. Механизм разложения программы на потоки даёт полигон для формирования навыков аспектно- ориентированной декомпозиции программ. Барьеры выполняют роль точек синхронизации, разбивающих программу на слои. Каждый слой начинает выполняться одновременно и завершается до выполнения следующего слоя. Выражения одного слоя из разных потоков, помеченные одинаковыми барьерами, выполняются одновременно. Описание учебной версии языка использует метафору «фабрика разнотипных роботов для конструирования программно-управляемых игрушек». По этой метафоре конструктор игрушки строит определённую обстановку (контекст) для комплекса взаимодействующих роботов. При создании игрушки конструктор может сценарии робота уточнить, включая изменение системы команд. Каждый поток может выполняться отдельным роботом. Роботы могут различаться по системе команд и другим характеристикам. Имеются общие универсальные команды, известные всем роботам. Для оперирования роботами выделяются команды уровня МикроРобота, устроенные как функции без параметров над простыми данными из общей памяти. Работа МакроРобота похожа на работу препроцессора в производственных системах программирования. Показ общих моделей параллельных вычислений и связанных с их применением явлений выполнен на задачах, описанных в книге Хоара [5], олимпиадных задачах, примерах программ из описаний языков параллельного программирования и учебных примеров из школьных и факультативных курсов информатики [6]. При подготовке примеров для демонстрации учебного языка параллельного программирования были приняты следующее ограничения: -взаимодействия потоков выполняются через синхронизацию; -одновременно исполняемые потоки могут обмениваться данными через общую память; -при взаимоисключении каждый поток работает в своей копии контекста; -поддерживается спи-

сок для восстановления многократно выполняемых фрагментов.

3 Заключение

Предлагаемый язык Синхро представляет собой эксперимент по выбору базовых средств для достаточно полного решения проблем обучения методам реализации параллельных алгоритмов, что может быть полезным при изучении методов создания параллельных программ с акцентом на тестирование, верификацию и отладку, а также, развития средств и методов ясного описания семантики языков параллельного программирования, включая представление программируемых преобразований текста и переносимого кода программы с удостоверением их корректности [3]. Выводы: 1. Язык СИНХРО поддерживает ряд решений по представлению разных схем управления процессами с возможностью синхронизации в терминах барьеров. 2. Для факторизации схем управления предложены средства изображения фрагментных переменных с контролем синтаксического подобия при подстановке их значений. 3. Многопоточные программы ориентированы на реализацию внешнего управления циклами и рекурсией, удобными при программировании сервисных систем. 4. Обучение программированию следует начинать с ознакомления с миром параллелизма. 5. Полноценное решение проблем параллельного программирования может быть выполнено в рамках экспериментальной разработки и эксплуатации учебного языка параллельного программирования.

Список литературы

1 Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. // СПб.: БХВ-Петербург, 2002. 608 с.

2 . Городняя Л.В. О курсе «Начала параллелизм» // Ершовская конференция по информатике. Секция «Информатика образования». 27 июня - июля 2011 года. Новосибирск. с. 51-54.

3 . Городняя Л.В. О проблеме автоматизации параллельного программирования // В сборнике Международной суперкомпьютерной конференции «Научный сервис в сети Интернет: многообразие суперкомпьютерных миров» - URL: http://agora.guru.ru/abrau2014 .

4 Звенигородский Г.А. Первые уроки программирования // Библиотечка Кванта, v.41. М.: Наука, 1985.

5 . Хоар Ч. Взаимодействующие последовательные процессы. // М.: Мир, 1989. 264 с.

6 . Городняя Л.В. Язык параллельного программирования Синхро, предназначеный для обучения //Новосибирск. Препринт ИСИ СО РАН. №180 – 32 с. http://www.iis.nsk.su/files/preprints/gorodnyaya_180.pdf

О модели программирования вычислительной схемотехники

Дбар С. А.[1], ssa@kiam.ru
Лацис А. О.[1], lacis@kiam.ru
[1]Институт прикладной математики
им. М. В. Келдыша РАН

Аннотация

Высокопроизводительные вычисления предъявляют очень жесткие требования к возможности эффективной реализации используемых моделей и технологий программирования. Модель должна быть очень адекватной абстракцией целевого оборудования, иначе технологии на ее основе бесполезны. Гибридные суперкомпьютеры с ускорителями вычислений на ПЛИС крайне своеобразны в архитектурном отношении, и непросты в прикладном программировании. В докладе обобщается примерно 10-летний опыт проектирования, реализации и применения технологий вычислительной схемотехники в ИПМ им. М. В. Келдыша РАН (http://www.kiam.ru/MVS/research/fpga/index.html).
Ключевые слова: Модель программирования, технология программирования, высокопроизводительные вычисления, ПЛИС.

Для высокопроизводительных вычислений уже почти 30 лет используются многопроцессорные вычислительные системы. Многолетний устойчивый рост их быстродействия долгое время обеспечивался как ростом числа процессоров (процессорных ядер) в вычислительной системе, так и ростом быстродействия самого процессорного ядра. Почти 15 лет назад рост быстродействия процессорного ядра практически прекратился [1]. Наращивание быстродействия путем увеличения числа процессорных ядер до десятков и сотен миллионов в одном

суперкомпьютере создает целый ряд трудностей, важнейшая (но не единственная) из которых - громадное энергопотребление [2]. Так возникает потребность в вычислителях гораздо более эффективных (в любой разумной метрике) при выполнении вычислительной работы, чем традиционный процессор общего назначения.

Построить такой процессор можно, но для этого приходится менять архитектуру - все, что можно было сделать в терминах традиционной суперскалярной архитектуры, уже сделано [3]. Так возникает потребность в процессорах нетрадиционной архитектуры.

Первыми такими процессорами были GPGPU, использование которых позволяет поднять эффективность (в метрике флопс/Ватт) почти на порядок по сравнению с традиционным процессором. Это хорошо, но мало.

Наиболее радикальный подход к повышению эффективности заключается в создании процессоров, реализующих прикладной алгоритм непосредственно на аппаратном уровне. Такие "процессоры одной задачи¨ не нуждаются в программе для своей работы, но и не тратят ни времени, ни энергии на адаптацию алгоритма к системе команд по ходу счета. Технически они реализуются на микросхемах программируемой логики (ПЛИС) [4].

К сожалению, сам факт аппаратной реализации алгоритма совсем его не ускоряет, по сравнению с выполнением соответствующей программы на процессоре общего назначения. Ускориться способна не просто схема, а хорошая схема. Вопрос о средствах проектирования именно хороших схем вычислительного характера, таким образом, выходит на первый план.

Описания схем в современных условиях выполняются в текстовом виде, на языках, внешне непоминающих языки программирования. Такие языки, широко используемые в работе профессиональными схемотехниками, существуют уже не один десяток лет, но, по ряду причин, совершенно непригодны для первоначального освоения схемотехники и последующего использования прикладными программистами. В то же время, именно эти языки, как показал многолетний опыт, позволяют создавать эффективные схемы. При ближайшем рассмотрении можно обнаружить, что языки профессиональной схемотехники реализуют совершенно особую, самобытную модель программирования, которая в высшей степени адекватна целевому оборудованию, в отличие, например, от языков программирования традиционной, императивной модели, таких как Си, Фортран и им подобные [3, 4].

Именно эта модель программирования (мы назвали ее схемотехнической) и содержит в себе в явном виде все те понятия, в терминах которых можно описать эффективно работающую схему. К сожалению, языки профессиональной схемотехники спроектированы настолько скверно, что упомянутая модель программирования в них, на первый взгляд, просто не видна. Увидеть и использовать ее в этих языках способен только специалист, уже овладевший схемотехническими понятиями каким-то другим способом, то есть имеющий профессиональную подготовку, весьма далекую от программистской [3].

В нашей работе мы попытались спроектировать и реализовать язык описания схем вычислительного характера - Автокод Stream - реализующий схемотехническую модель программирования в явном виде. Предварительно упростив язык применительно к приложениям вычислительного характера, мы затем снабдили его высокоуровневыми средствами построения вычислительных конвейеров непосредственно по записи формул в традиционном виде [5]. Попутно пришлось поставить и решить ряд проблем использования ПЛИС в высокопроизводительных вычислениях общего плана.

Об этом и будет более подробно рассказано в докладе.

Литература:

1. http://www.top500.org Дата обращения 13.02.2017

2. Свежий взгляд из-за океана. Интервью И. Левшина с Дж. Донгаррой. "Суперкомпьютеры¨ №2(14) март-апрель 2013

3. А.О. Лацис. Параллельная обработка данных. Издательский центр "Академия¨ Москва, 2009.- 336с. ISBN 978-5-7695-5951-8

4. С.С. Андреев, С.А. Дбар, А.О. Лацис, Е.А. Плоткина. Почему новые вычислительные архитектуры такие неудобные. Труды XXI Всероссийской конференции "Теоретические основы и конструирование численных алгоритмов решения задач математической физики¨ посвященной памяти К.И. Бабенко. Абрау-Дюрсо, сентябрь 2016г

5. С.С. Андреев, С.А. Дбар, А.О. Лацис, Е.А. Плоткина. Гибридный реконфигурируемый вычислитель. Руководство программиста на языке Автокод. http://www.kiam.ru/MVS/research/fpga/progman Дата обращения 13.02.2017

Разработка масштабируемой системы оптимизации времени связывания

Долгорукова К. Ю., unerkannt@ispras.ru
Институт системного программирования РАН

Современные программы зачастую состоят из нескольких файлов с исходным кодом, а для больших сложных программ число файлов может составлять тысячи. В этом случае для получения эффективно работающей исполняемой программы компилятору недостаточно только локальных внутрипроцедурных оптимизирующих преобразований: необходимо анализировать связи между процедурами и оптимизировать программу с их учетом, – такие оптимизации называются межпроцедурными.

Традиционный метод сборки программ из исходного кода состоит из двух этапов: компиляции и связывания. В первом этапе все модули программы компилируются отдельно и независимо в объектные файлы, которые потом связываются компоновщиком. В этом случае у компилятора нет программного кода других модулей, поэтому эффективность межпроцедурных оптимизирующих преобразований ограничены одним файлом с исходным кодом. Если же есть возможность оптимизировать все входящие в программу модули вместе, эффективность межпроцедурных преобразований значительно возрастает. Межпроцедурные оптимизации, проводимые на всей программе целиком, называются межмодульными оптимизациями.

На втором этапе сборки компоновщик получает все скомпилированные файлы программы и файлы библиотек, разрешает коллизии и зависимости между ними и строит исполняемый файл. Практика показала, что целесообразно проводить межпроцедурные оптимизации на этапе связывания, так как именно на этом этапе системе сборки доступна вся программа целиком: для этого в объектные файлы программ, полученные на этапе компиляции, добавляется некоторая информация

о программе, достаточная, чтобы построить промежуточное представление программы, используемое компилятором. Такие системы называются системами оптимизаций времени связывания.

Межмодульные оптимизирующие преобразования, проводимые над всеми модулями программы, практически всегда требуют построения промежуточного представления для всей программы. В случае больших приложений этот процесс может потребовать огромных ресурсов. Для сборки с оптимизациями времени связывания таких программ, как операционные системы или интернет-браузеры, состоящих из нескольких тысяч файлов исходного кода, может требоваться объем оперативной памяти, которым не обладают не только обычные настольные компьютеры, – но и далеко не все серверные архитектуры способны собрать такие программы без задействования отгрузки на диск. К примеру, сборка состоящего из 36,5 тысяч исходных C/C++ файлов браузера Firefox на компиляторах GCC и LLVM с оптимизацией времени связывания требует от 6 до 34 Гб ОЗУ в зависимости от настроек, и работает от 11 до 26 минут на x86-64. Для офисного текстового редактора LibreOffice, состоящего почти из 20 тыс C/C++ файлов, эти числа будут иметь значения 8-14 Гб и 61-68 минут соответственно [1] [2].

Методы регулирования потребления ресурсов системой для различных архитектур называются масштабированием системы.

Масштабирование может быть проведено как по времени, так и по памяти, ускоряя процесс сборки – или уменьшая количество потребляемой памяти. Масштабирование с целью запуска на многоядерных архитектурах называется горизонтальным масштабированием.

Данная работа посвящена разработке масштабирумой системы оптимизации времени связывания на основе LLVM для работы на больших приложениях, способной работать как на системах с ограниченными ресурсами, так и на многоядерных архитектурах.

Оптимизация времени связывания в LLVM[3] проводится с использованием конвейера анализирующих и оптимизирующих проходов, последовательно проводимых над программой в промежуточном представлении, находящимся в памяти. Схема работы оптимизации времени связывания в LLVM показана на рисунке 1.

В данной работе представлен общий метод масштабирования, который заключается в разделении стадий анализа (МПА) и оптимизации (МПО) таким образом, чтобы было возможно проводить оптимизации параллельно, а анализ – не имея всего промежуточного представления

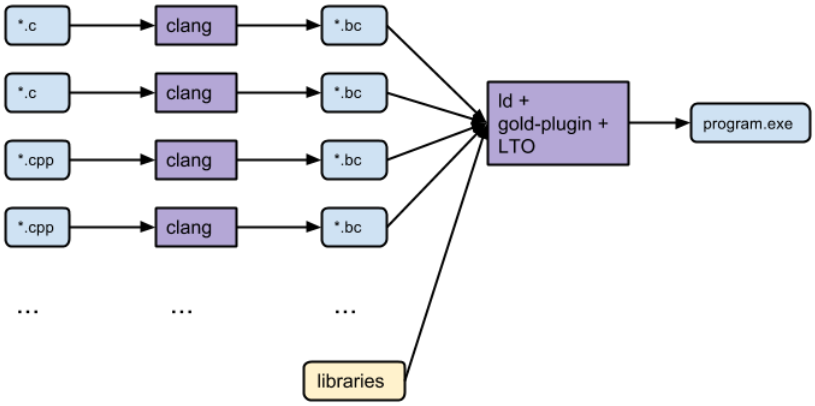

Рис. 1: Схема сборки программы с помощью утилит LLVM

в памяти. Для этого часть необходимого для оптимизации времени связывания анализа переносится на этап компиляции исходного кода в промежуточное представление; и промежуточное представление аннотируется результатами анализа таким образом, чтобы можно было на его основе проводить оптимизации, не загружая всё промежуточное представление в память. Схема работы разработанной системы представлена на рисунке 2.

Целью доклада является демонстрация применения разработанных

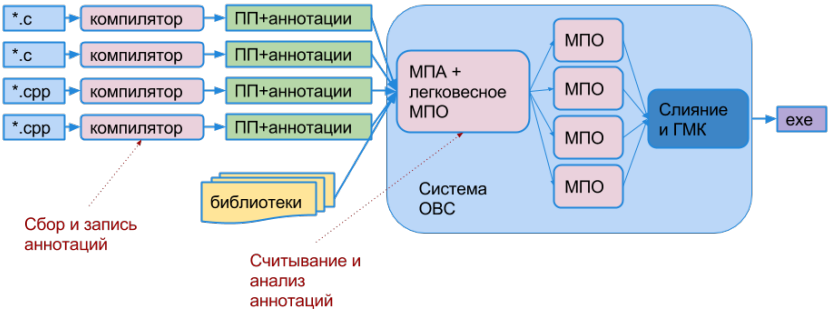

Рис. 2: Предлагаемая архитектура системы оптимизаций времени связывания

ранее методов масштабирования [4] [5] [6] к оптимизации крупных приложений – браузера Mozilla Firefox и офисного редактора Libre Office. В данный момент работа еще ведется, но согласно результатам предварительного тестирования, ускорение сборки на архитектуре x86-64 при распараллеливании на 4 потока достигает 36% при падении производительности программ относительно скомпилированных в 1 поток на 3%.

Список литературы

[1] Honza Hubicka. Linktime optimization in GCC, part 2 - Firefox. `http://hubicka.blogspot.ru/2014/04/linktime-optimization-in-gcc-2-firefox.html`

[2] Honza Hubicka. Linktime optimization in GCC, part 3 - LibreOffice. `http://hubicka.blogspot.ru/2014/09/linktime-optimization-in-gcc-part-3.html`

[3] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO04), March 2004.

[4] К. Ю. Долгорукова. Обзор масштабируемых систем межмодульных оптимизаций. Труды Института системного программирования РАН Том 26. Выпуск 3. 2014 г. Стр. 69-90. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2014-26(3)-3

[5] К. Ю. Долгорукова. Разработка и реализация метода масштабирования по памяти для систем межмодульных оптимизаций и статического анализа на основе LLVM. Труды Института системного программирования РАН Том 27. Выпуск 6. 2015 г. Стр. 97-110. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2015-27(6)-7

[6] К. Ю. Долгорукова, С. В. Аришин. Ускорение оптимизации программ во время связывания. Труды Института системного программирования РАН, том 28, вып. 5, 2016, стр. 175-198. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2016-28(5)-11

Интеграция компилятора clang со сторонней библиотекой оптимизирующих преобразований

Дубров Д. В.[1], dubrov@sfedu.ru
Патерикин А. Е.[1], gradgerh@gmail.com
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

Настоящая работа посвящена разработке преобразователя внутреннего представления программ из формата Оптимизирующей распараллеливающей системы (ОРС) в формат компилятора clang. Работа по развитию системы ОРС ведётся в институте математики, механики и компьютерных наук ЮФУ. Разработанный преобразователь открывает возможность использования кодогенераторов LLVM после оптимизации программ при помощи ОРС, что повышает возможности тестирования корректности и эффективности преобразований ОРС. Для пользователей компилятора clang результаты данной работы помогут расширить набор высокоуровневых оптимизаций реализованными в ОРС.
Ключевые слова: компиляция, внутреннее представление программ, LLVM, clang.

1 Введение

В процессе разработки библиотек оптимизирующих преобразований программ возникает задача их интеграции с существующими генераторами кода, поскольку разработка собственного кодогенератора

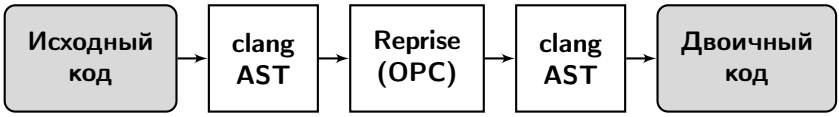

Рис. 1: Схема работы встраивания

с нуля является слишком трудозатратной. Основными промышленными наборами компиляторов с открытыми исходными кодами, в которых реализованы кодогенераторы высокого качества для большого количества целевых платформ, являются GCC и LLVM.

Оптимизирующая распараллеливающая система (OPC) является набором библиотек и инструментов, предназначенных для разработки оптимизирующих, в том числе распараллеливающих компиляторов [1]. OPC реализует большое количество методов анализа и оптимизирующих преобразований, работающих с программой в высокоуровневом внутреннем представлении Reprise [2]. В настоящее время OPC использует front-end компилятора clang [3] для обработки программ на языке C. В OPC реализовано несколько кодогенераторов (back-end), позволяющих получить результаты преобразований в виде исходных текстов на языках C, C+OpenMP, VHDL и т. д. То есть, OPC в настоящее время работает по принципу «source-to-source translation».

Целью настоящей работы является реализация модуля преобразования внутреннего представления Reprise в представление clang (clang AST). Далее полученное представление можно подать на вход соответствующего модуля clang, который выполняет понижение высокоуровневого представления до представления LLVM (LLVM IR) и далее использует LLVM для генерации выходного двоичного кода (рис. 1).

2 Проблемы преобразования внутренних представлений

Совместное использование компонент различных компиляторов часто затрудняется принципиальными различиями в их внутренних представлениях. Хотя представления Reprise и clang AST внешне схожи (оба являются высокоуровневыми абстрактными синтаксическими деревьями), различия в задачах, ставившихся при их проектировании, отразились на определённых конструктивных решениях в их

реализации. Представление Reprise разрабатывалось как по возможности не зависящее от языка программирования, на котором написана исходная программа. Это позволило использовать в OPC front-end языка Fortran. С другой стороны, clang AST ориентировано на представление программ на языках, поддерживаемых этим компилятором: C, C++ и других. По этой причине оно по возможности приближено к стандартам C99 и т. д. В частности, в нём хранятся неявно используемые по стандарту преобразования типов данных. Так например, в представлении clang AST индексного выражения с участием массива, например, `a[i]`, используется дополнительный узел `ArrayToPointerDecay` — преобразование из массива к указателю, аналога которому нет в Reprise. Другим серьёзным различием между представлениями является подход к хранению объявлений переменных. В языках Fortran и C89 объявления переменных находятся перед телом функции, и их область видимости и время жизни совпадает с границами тела. С другой стороны, в стандарте C99 переменные могут объявляться внутри блочных операторов. Поэтому в Reprise объявления переменных хранятся наиболее общим образом — перед телом функции. Такая модель не соответствует модели clang AST, где с каждой переменной связан декларатор — оператор объявления, который находится в теле функции.

Приведение преобразования Reprise к виду, совместимому с clang AST, сделало бы неработоспособным большинство существующих модулей OPC. К тому же, реализацию преобразований пришлось бы усложнять для поддержки дополнительных узлов внутреннего представления, засоряющих семантическую модель программы. Также пришлось бы отдельно обрабатывать внутреннее представление Reprise, получаемое из программ на Fortran.

Реализованный ранее в OPC модуль `ClangParser`, будучи запущенным после работы clang front-end, преобразует создаваемый им clang AST в Reprise, отображая узлы одного синтаксического дерева в их аналоги из другого. После этого в результирующем представлении теряется информация о неявных преобразованиях и местах объявлений локальных переменных внутри функций. Реализованный в рамках настоящей работы модуль `RepriseToClang`, являясь обратным преобразованием по отношению к `ClangParser`, переводит узлы Reprise в соответствующие узлы clang AST. Если получаемое на этом этапе внутреннее представление передать clang для печати в виде текста на C («pretty-print»), компилятор выведет корректную программу. Однако

кодогенератор clang отвергнет это же представление как ошибочное.

Ввиду этого, в настоящей работе были дополнительно реализованы преобразования, применяемые к генерируемому представлению clang AST. Они добавляют в древовидные представления выражений программы недостающие узлы, которые обозначают неявные преобразования, в соответствии со стандартом C99:

- lvalue к rvalue;
- массивы к указателям;
- целочисленные продвижения (integer promotions);
- обычные арифметические преобразования (usual arithmetic conversions).

3 Заключение

В рамках настоящей работы был реализован модуль преобразования внутреннего представления системы OPC (Reprise) в clang AST, позволяющий использовать кодогенератор совместно с библиотекой оптимизаций OPC. Для этого пришлось решить проблемы различий в способах представления программ обеих систем. В частности, пришлось реализовать восстановление информации о требуемых по стандарту C99 неявных преобразований в выражениях. Реализованное преобразование применимо к любой входной программе на языке C (стандарт C99). Тестирование преобразователя на ряде типичных примеров, использующих неявные преобразования и локальные объявления, показывает работоспособность использованного подхода.

Список литературы

1. Оптимизирующая распараллеливающая система. — URL: `http://ops.rsu.ru/about_OPS.shtml` (дата обр. 08.02.2017).

2. *Петренко В. В.* Внутреннее представление Reprise распараллеливающей системы // Труды Четвертой Международной конференции «Параллельные вычисления и задачи управления» PACO‘2008 (Москва, 27—29 окт. 2008). — М. : Институт проблем управления им. В. А. Трапезникова РАН, 2008. — С. 1241—1245. — ISBN 978-5-91450-016-7. — URL: `http://ops.rsu.ru/download/works/PACO2008-petrenko.pdf` (дата обр. 08.02.2017).

3. clang: a C language family frontend for LLVM. — URL: `http://clang.llvm.org/` (дата обр. 08.02.2017).

Преобразование программных циклов “retiming”

Ивлев И. А.[1], ivlev_1996@mail.ru
Штейнберг О. Б.[1], olegsteinb@gmail.com
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

В работе рассматривается распараллеливающее преобразование программ “retiming”. Это преобразование позволяет увеличить степень распараллеливаемости программных циклов. Преобразованием используется граф зависимостей по данным, вершинам и дугам которого присваиваются веса. Далее, исходя из полученных весов, строится результирующий код. В работе рассматривается область применения преобразования, а также вспомогательные преобразования, способные ее расширить. Данное преобразование может быть использовано оптимизирующими компиляторами как для автоматического распараллеливания, так и для конвейеризации. Преобразование добавлено в Оптимизирующую распараллеливающую систему (ОРС).
Ключевые слова: оптимизирующий компилятор, retiming, параллельные вычисления, преобразования циклов.

1 Введение

При параллельном выполнении программ немаловажную роль играют программные циклы. Однако не в каждом цикле можно выполнить даже несколько итераций независимо друг от друга. В некоторых случаях для этого требуются вспомогательные преобразования. Одним из таких преобразований является **retiming** [5]. Это преобразование было доработано авторами данной статьи и встроено в Оптимизирующую распараллеливающую систему [2].

2 Преобразование программных циклов “retiming”

Преобразование описываемое в данной работе нацелено на увеличение количества итераций программного цикла, пригодных для параллельного выполнения. Предполагается, что преобразуемый программный цикл содержит постоянные верхнюю и нижнюю границы. Счетчик цикла после каждой итерации увеличивается на единицу. В теле цикла содержатся только операторы присваивания, в которых могут быть лишь константы и индексные переменные, зависящие от счетчика цикла. При этом рекуррентные зависимости в операторах присваивания не допускаются. Преобразование выполняется в три этапа.

Этап 1. Построение взвешенного графа зависимостей по данным

На первом этапе по коду программы строится граф зависимостей по данным (ГЗД) [4], [6]. Каждой дуге этого графа присваивается вес [1], [3], равный разности индексных выражений образующих её вхождений.

Этап 2. Преобразование взвешенного графа зависимостей по данным

Второй этап состоит в определении весов вершин графа и последующем пересчете весов дуг. Веса вершин определяются по-разному в зависимости от количества циклов в ГЗД.

В случае, когда ГЗД не содержит циклов, веса вершин определяются из системы неравенств:

$$r(u) \ + \ w(u,\ v) \ - \ r(v) \geq k, \forall (u, v) \in E,$$

где k - минимальное количество итераций программного цикла, пригодных для параллельного выполнения; r(u), r(v) – веса вершин u и v соответственно, w(u, v) – вес дуги ГЗД, идущей из u в v.

Далее веса дуг пересчитываются по формуле:

$$w_{new}(u,\ v) \ = \ r(u) \ + \ w(u,\ v) \ - \ r(v),$$

Сложность алгоритма, находящего веса, является полиномиальной.

Случай, когда ГЗД содержит ровно один цикл, серией преобразований сводится к случаю с ациклическим графом.

В случае, когда ГЗД содержит более одного цикла, для нахождения весов вершин и дуг решается задача линейного целочисленного программирования. Алгоритм, решающий эту задачу, имеет экспоненциальную сложность.

Этап 3. Преобразование кода программы

В теле цикла к индексным выражениям каждого оператора присваивания прибавляется вес соответствующей вершины ГЗД. Далее, для того чтобы результирующий код был эквивалентным исходному циклу, количество итераций цикла уменьшается на число, равное весу самой "тяжёлой"вершины графа. Затем до и после результирующего цикла добавляются операторы присваивания, вычисляющие недостающие итерации исходного цикла.

3 Добавление преобразования “retiming” в OPC

Добавление описанного преобразования в OPC состояло в создании кода, использующего существующие функции проверки входных данных, построения графовой модели программы и создания новых узлов внутреннего представления. Также был написан алгоритм, реализующий второй этап преобразования. Итого, в проделанной работе можно выделить ряд блоков, отличающихся с точки зрения реализации:

(a). **Проверка входных данных на применимость преобразования.** В OPC существует большой набор функций, способных осуществлять разные проверки. Для проверки применимости данного преобразования использовались функции, проверяющие, что:

- цикл является простым (с постоянными верхней и нижней границей и счётчиком, увеличивающимся на единицу)
- в цикле присутствуют только операторы присваивания
- в операторах присваивания цикла присутствуют только константы и индексные переменные, зависящие от счётчика цикла, подаваемого на вход преобразованию

Кроме описанных функций, уже реализованных в ОРС, была добавлена проверка на отсутствие рекуррентных зависимостей внутри операторов присваивания.

(б). **Построение ГЗД.** В ОРС присутствует инструмент, строящий граф информационных связей, но для описанного преобразования требуется ГЗД, являющийся фактор-графом [1],[3] графа информационных связей [4], [6], [2]. Ввиду этого авторами реализовано получение ГЗД по имеющемуся графу информационных связей.

(в). **Получение весов дуг.** В данном преобразовании вес дуги определяется как разность индексных выражений вхождений образующих дугу. Они получались с помощью служебных функций предназначенных для работы с внутренним представлением ОРС.

(г). **Выполнение второго этапа преобразования.** Для реализации второго этапа преобразования был написан код к основе которого лежат различные действия производимые над числовыми матрицами (матрицами весов графа).

(д). **Создание результирующего кода.** Для создания результирующего кода использовались функции, изменяющие существующие узлы внутреннего представления (например, при изменении индексных выражений переменных операторов тела цикла) и добавляющие операторы до и после тела цикла, необходимые для обеспечения эквивалентности преобразования.

4 Заключение

В данной работе описано, реализованное в ОРС, оптимизирующее преобразование, нацеленного на увеличение степени распараллеливаемости цикла.

Список литературы

1. *Зыков А. А.* Основы теории графов. — Москва : Вузовская книга, 2004. — 662 с.

2. Оптимизирующая распараллеливающая система. — URL: `http://ops.rsu.ru/about_OPS.shtml`.
3. *Харари Ф.* Теория графов. — Москва : Мир, 1973. — 300 с.
4. *Allen R.*, *Kennedy K.* Optimizing compilers for modern architectures. — San Francisco, San Diego, New York, Boston, London, Sidney, Tokyo : Morgan Kaufmann Publishers, 2002. — 790 с.
5. *Liu D.* [и др.]. Optimal Loop Parallelization for Maximizing Iteration-Level Parallelism // International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09) (Grenoble). — 2009. — С. 67—76.
6. *Wolfe M.* High performance compilers for parallel computing. — Redwood city : Addison-Wesley Publishing Company, 1996. — 570 с.

Программная инфраструктура семантического анализа программ на C++

Зуев Е. А.[1], e.zuev@innopolis.ru
[1]Университет Иннополис

Аннотация

В статье рассматриваются актуальные проблемы, приводящие к необходимости повышения качества и глубины анализа программ на языке C++. Обсуждаются возможные пути решения проблем повышения надежности и безопасности программ. В качестве основы различных систем анализа предлагается концепция **семантического представления** (СП). Обсуждаются преимущества СП, определяются основные требования к ней, важнейшие характеристики и возможные сферы применения.
Ключевые слова: язык C++, анализ программи, семантическое представление, компиляторы

1 Проблема

В настоящее время размеры и сложность программных продуктов и систем увеличились многократно. Фактически, сложность программ вплотную подошла к пределу, за которым традиционные методы оценки корректности и надежности программ и их соответствия исходным требованиям становятся принципиально неприменимыми. В то же время возросшая ответственность программных систем настоятельно требует серьёзных гарантий их качества.

Решение этой проблемы лежит на пути создания автоматизированных и автоматических систем статического и динамического анализа (аудита) программ.

Данная проблематика имеет достаточно общий характер и актуальна практически для любого современного языка программирования,

используемого в разработке ПО. Предлагаемая статья, однако, ограничивается обсуждением проблем семантического анализа программ на языке C++. Этот язык, будучи одним из наиболее распространенных инструментальных средств, страдает многими хорошо известными недостатками, которые не способствуют созданию по-настоящему надежного ПО.

2 Актуальные направления

Представляется, что наиболее актуальными направлениями в этой сфере являются следующие:

- Аудит исходных текстов программ на предмет выявления потенциальных уязвимостей, которые могут вызывать нерегламентированное поведение программы, в том числе вызвать сбои и/или поломки программ или среды, в которой они работают.

- Оптимизационный анализ текстов программ с целью совершенствования их структуры или функциональных свойств, повышения их производительности и ослабления чрезмерно высоких требований на потребляемые ресурсы.

- Анализ программ на предмет заимствований фрагментов исходного текста из сторонних программ (инструменты «антиплагиата»).

- Динамическое профилирование – тестовое исполнение программы в модельной среде, что может выявить специфические ошибки, связанные с этапом выполнения программ и не выявляемые методами статического анализа исходных текстов.

- Автоматизация тестирования. Автоматическое создание тестовых покрытий и управление тестированием.

3 Текущее состояние

Одним из первых подходов к решению проблемы анализа программ следует считать комплекс ASIS (Ada Semantic Interface Specification [1]), задающий множество высокоуровневых интерфейсов доступа к

Ада-программам. ASIS определяет стандартную совокупность запросов в терминах входного языка, оставляя их реализацию на усмотрение разработчиков.

К настоящему времени важность и актуальность проблемы анализа программ вполне осознана как их пользователями (корпоративными и индивидуальными), так и сообществом разработчиков программных средств. Рынок инструментов анализа переживает явный подъем. Сейчас доступен весьма широкий спектр программ систем различного качества, масштаба и ассортимента функций – от небольших свободнодоступных инструментов до мощных многофункциональных систем коммерческого характера.

Однако, практически все известные системы статического анализа базируются либо на некотором внутреннем представлении программ, порождаемом компиляторами третьих компаний, либо на собственных программах разбора.

Оба подхода страдают фундаментальными недостатками. Промежуточное представление программ, генерируемое сторонними компиляторами, практически никогда не содержит адекватной информации о семантике исходных программ, что очень сильно ограничивает возможности их анализа. Кроме того, использование промежуточного представления приводит к сильной зависимости разработчика анализатора от текущего состояния этих компиляторов: формат промежуточного представления, как правило, не фиксирован, может изменяться в непредсказуемые моменты времени произвольным образом, и сами поставщики компиляторов, как правило, не рекомендуют полагаться на этот формат (или прямо запрещают это делать). Примером описанного подхода может служить проект Pivot [2], развиваемый при участии создателя C++ Б.Страуструпа.

Многие производители инструментов анализа используют собственные средства разбора программ. Однако в большинстве случаев такой разбор осуществляет весьма поверхностный анализ исходных текстов, опять-таки, игнорируя большое число семантических особенностей, существенных для качественного и глубокого анализа. Среди немногочисленных исключений следует отметить систему llvm/clang, которая включает статический анализатор [3], построенный на собственном компиляторе переднего плана C++.

Из сказанного следует, что достичь решающего преимущества на пути создания высококачественного и конкурентноспособного инструментария для анализа программ можно было бы на пути **повышения**

степени глубины анализа программ.

4 Базовая идея

В основе проекта лежит фундаментальная идея, которая заключается в том, чтобы создать **унифицированное, регулярное и удобное для программного использования семантическое представление** (далее – **СП**) для программ, написанных на языке C++.

Идея некоторого специального представления программ, альтернативного исходному текстовому виду, не нова. Компилятор любого ЯП в процессе обработки входной программы преобразует её в некоторое *промежуточное или внутреннее представление* – с тем, чтобы обеспечить контроль синтаксической и семантической корректности и выполнить генерацию целевого кода для некоторой аппаратной архитектуры.

Фундаментальные отличия предлагаемого СП от известных промежуточных представлений программ заключаются в следующих аспектах:

- СП принципиально ориентируется на широкий спектр применений, не ограничиваясь внутренними задачами конкретного компилятора по генерации целевого кода.

- СП содержит полную семантическую информацию об исходной программе, а не только подмножество этой информации, необходимое для создания кода.

- Семантическая информация представляется строго в терминах входного языка, а не во внутренних терминах, специфичных для данного компилятора.

- СП включает *скрытую семантику*, то есть, такую информацию, которая непосредственно не выражена в исходной программе, но подразумевается согласно определению ЯП.

  Типичным примером скрытой семантики служат вызовы деструкторов C++ для объектов классовых типов, объявленных в некоторой области действия, при выходе из этой области действия. Такие вызовы не присутствуют явно в программе, однако подразумеваются семантикой языка.

- СП является открытым и снабжается удобным программным интерфейсом. Иными словами, СП – это набор классов и методов, посредством которых можно получить любую информацию о структуре программы, ее компонентах, связях и отношениях между компонентами с любой степенью детализации.

- СП обеспечивает двусторонний доступ: как по чтению, так и по записи. Это означает, что с помощью СП можно как получать семантическую информацию о программе, так и модифицировать его, добавляя или удаляя те или иные компоненты.

- СП является полностью независимой системой, которая не использует каких-либо сторонних программных анализаторов, компиляторов или иных инструментов.

  Это свойство принципиально отличает СП от систем и проектов аналогичного назначения, которые в подавляющем большинстве представляют собой конгломерат различных продуктов (в частности, свободно-доступных компиляторов) и технологий, слабо связанных между собой посредством «фильтров», «переходников» и других средств. Подобные комплексы характеризуются весьма сложными технологическими схемами использования и являются сильно ограниченными по своим возможностям.

Семантическое представление, основные особенности которого перечислены выше, вместе с программным интерфейсом для доступа к нему, будет дальше называться **Semantic API**.

5 Область применения Semantic API

Semantic API может использоваться для реализации широкого спектра операций над программами. Природа и назначение таких операций могут быть произвольными и, в принципе, ничем не ограниченными. Ниже следует список возможных операций.

- Выявление потенциальных уязвимостей, скрытых ошибок, «мёртвого» кода, «закладок», заимствованного кода.

- Статический анализ программ в широком смысле; снятие метрических характеристик, статическое профилирование и т.д.

- Рефакторинг; оптимизация программ по различным критериям.

- Конвертирование программ: преобразование с одного ЯП на другой с сохранением семантической эквивалентности и функциональности.
- Разработка компиляторов ЯП.
- Генерация тестов, контроль тестовых покрытий.
- Формальная верификация программ.
- Генерация документации и отчетов.
- Интерпретация программ; динамическое профилирование; разработка систем отладки.

Приведенный список является, очевидно, неполным; архитектура Semantic API допускает эффективное использование и для многих других областей анализа.

6 Ссылки

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

[2] The Pivot: A brief overview. Bjarne Stroustrup and Gabriel Dos Reis, https://parasol.tamu.edu/pivot/publications/reis1.pdf (последнее обращение 19.03.2017).

[3] Clang Static Analyzer, https://clang-analyzer.llvm.org/ (последнее обращение 19.03.2017).

Beyond C++: проект современного языка программирования общего назначения

Канатов А. В.[1], a.kanatov@samsung.com
Зуев Е. А.[2], e.zuev@innopolis.ru
[1]Samsung Research and Development Institute
[2]Университет Иннополис

Аннотация

В статье рассматриваются тенденции развития современных языков программирования и предлагаются к обсуждению несколько концепций, которые должны повысить простоту и удобство проектирования и разработки программного обеспечения, в рамках статической проверки правильности программ. В частности дается нетрадиционный подход к базовому строительному блоку программного кода – контейнеру атрибутов и подпрограмм, альтернативная схема наследования, как простого и единственного механизма расширения уже существующего ПО, концепция мультитипа, как расширение наследования, а также полное решение для проблем нулевых указателей и неинициализированных данных.

Ключевые слова: языки программирования, компиляторы

1 Предпосылки

Язык программирования, со своей инфраструктурой – системами поддержки времени выполнения, стандартными и прикладными библиотеками, инструментами и средами разработки (компиляторы, редакторы связей, IDE) представляет собой ключевой инструмент создания программного обеспечения и, тем самым, служит определяющим фактором обеспечения эффективного, безопасного и надежного функционирования многочисленных и разнообразных электронных устройств в современном мире.

Существующее положение в сфере языков программирования и, шире, в области инструментария современного программирования весьма далеко от идеала. Большинство языков программирования, наиболее широко используемых в настоящее время, создано более двадцати лет назад и в настоящее время совершенно не адекватно практически ни одному из современных требований, предъявляемых к инструментам разработки современного ПО. Эти языки архаичны, неуклюжи, громоздки, неудобны и сложны в практическом использовании, не способствуют надежности и эффективности создаваемого ПО, зачастую несут явный отпечаток вкусовых пристрастий и причудливых взглядов их создателей [1].

Новые языки программирования, в изобилии появляющиеся в последние годы, пытаются преодолеть указанные недостатки, однако в значительной степени повторяют устаревшие и порочные подходы проектирования, берущие своё начало в восьмидесятых годах прошлого столетия [2], [3].

2 Существо проекта

Основной смысл предлагаемого проекта заключается в том, чтобы спроектировать и реализовать оригинальный язык программирования (предварительное название – **SLang**) вместе с необходимой экосистемой: компилятором, подсистемой поддержкой времени выполнения, стандартными библиотеками, редактором связей (комплексатором).

Основной смысл проекта заключается в том, чтобы предложить разработчикам программного обеспечения XXI века инструмент, который бы позволил им решать разнообразные задачи наиболее простым и надежным способом в соответствии с предъявляемым требованиям и применительно к разнообразным уровням квалификации разработчиков.

Язык SLang воплощает адекватное понимание существа процесса проектирования и разработки современного ПО. Он включает свойства, обеспечивающие надежность, безопасность и эффективность программ, создаваемых с его использованием. В то же время он достаточно прост для обучения, освоения и использования, что обеспечит «гладкий» процесс разработки и сопровождения, а также предоставит возможность включить в сферу разработки ПО более широкие, нежели в настоящее время, сообщества разработчиков, в том числе, непрофессионалов.

Язык SLang носит принципиально *универсальный* характер, и потому пригоден для успешного создания ПО в различных сферах применений – от микроустройств с минимальными характеристиками производительности и памяти (что особенно существенно в сфере «интернета вещей», IoT), мобильных устройств, приложений для сети Интернет, до серверных систем, суперкомпьютеров и систем реального времени.

Язык SLang – *масштабируемый*; под этим подразумевается его пригодность для реализации программных систем различного масштаба и сложности: от многофункциональных, логически насыщенных систем с повышенными требованиями к надежности, до коротких программ («скриптов»), решающих простые прикладные задачи.

Помимо языка, проект будет включать необходимый (на первом этапе – минимальный, в итоге – исчерпывающий) набор инструментов, в совокупности обеспечивающих полный жизненный цикл программ.

Язык спроектирован как *машинно-независимый*. Его семантика имеет высокий уровень и не ориентирована на особенности какой-либо аппаратной или программной платформы. В то же время, язык может быть эффективно реализован для любой распространённой в настоящее время среды.

Замечание об импортозамещении. В настоящее время достаточно остро стоит вопрос создания отечественных версий программного обеспечения различного рода. Для обеспечения полноты импортозамещения в сфере разработки ПО, необходим современный отечественный язык программирования вместе со всеми необходимыми инструментами. Проект языка программирования SLang и системы программирования на его основе призван заполнить существующий пробел и предложить мощное и современное решение в указанной сфере.

3 Краткая характеристика языка SLang

Модульность. В отличие от многих современных языков, программа на SLang формируется по модульному принципу: строительными блоками любой программы служат модули – независимо определяемые, независимо хранимые и независимо компилируемые единицы со строго определенными интерфейсами, согласно которым они могут вступать в различные отношения друг с другом: использование, агрегация, наследование и т.д.

В языке определены два основных вида модулей: контейнеры и подпрограммы. Контейнеры представляют агрегацию логически связан-

ных ресурсов (данных и подпрограмм-членов), подпрограммы реализуют некоторую функциональность и, в свою очередь, могут представлять собой процедуры или функции.

Продолжая эту логику, естественно считать, что компонент любого из указанных видов может служить *единицей компиляции*, то есть, допускать раздельную компиляцию. Отсутствие ограничений способствует созданию композиции программы, адекватной требованиям и особенностям ее использования. Программа может представлять собой, по сути, произвольную композицию единиц, от простого набора взаимодействующих подпрограмм до сложной комбинации контейнеров различных видов.

В предельном случае единица компиляции или вся программа может представлять собой единственную единицу – простую последовательность операторов. Если необходимо написать несколько строк кода, которые будут служить реакцией на нажатие клавиши мыши в некотором средстве просмотра Интернет-страниц, то нет необходимости писать подпрограмму – достаточно просто задать последовательность операторов, которая выполнит нужное действие. Если же решение задачи предполагает более сложную логику, то результат можно получить, комбинируя отдельные подпрограммы, контейнеры и блоки. Таким образом, единая языковая нотация может быть использована для решения максимально широкого класса задач.

Строгая типизация. Понятие типа является одним из базовых понятий любого языка. Под *типом* некоторого объекта, существующего в программе, понимается тройственная сущность, определяемая множеством *значений*, которые может принимать данный объект, связанным с ним множеством *операций*, допустимых над значениями данного типа, а также множеством *отношений* между данным типом и другими типами.

Язык SLang представляет собой язык со *статической типизацией* объектов. Это означает, что тип является статически неизменным свойством объекта; это свойство присуще объекту с момента его возникновения в программе и не может измениться во время жизни этого объекта. В терминах программирования это означает, что тип объекта назначается объекту (явно или неявно) при его объявлении, и не существует возможностей (языковых конструкций), позволяющих изменить тип в процессе выполнения программы. Статическая типизация является в настоящее время признанным средством обеспечения надежности и высокой производительности программ. Тип может

быть (явно) приписан объекту программистом при объявлении объекта, либо (неявно) выведен компилятором из контекста объявления этого объекта. Примером контекста в данном случае может служить тип инициализирующего выражения из объявления объекта.

Поддержка различных парадигм программирования. SLang – мультипарадигменный язык, в том смысле, что в нём воплощены важнейшие современные концепции программирования, включая объектно-ориентированное, обобщённое (generic) и функциональное программирование.

В языке имеется понятие типа (класса), которое реализуется посредством языковой конструкции «контейнер» (см. ниже), со всеми традиционными свойствами – инкапсуляцией, наследованием, полиморфизмом. Имеется возможность задавать абстрактные классы, а также реализовывать между классами отношения множественного наследования. При этом язык предлагает оригинальный подход к разрешению возможных конфликтов при множественном наследовании.

Любой контейнер или подпрограмма может быть параметризована типами или константами. Настройка обобщенных программных единиц предполагает задание конкретных типов и/или значений. Тем самым, в языке обеспечивается полная поддержка парадигмы *обобщенного программирования*, что позволяет проектировать компоненты программы в виде, максимально независимом от контекстов их использования.

В языке поддерживается необходимый набор средств *функционального подхода* к программированию, включая функциональные типы, лямбда-выражения и замыкания. Эти свойства основаны на трактовке функций как значений, а также предполагают свободное использование понятия неизменяемых (immutable) объектов.

Контейнер: модуль, класс и тип в одном флаконе. Важнейшими концепциями, используемыми при разработке программного обеспечения (ПО), служат понятия атрибутов (данных) и подпрограмм (действий). Атрибуты могут изменяться подпрограммами в процессе работы программы; они образуют ее вычислительный контекст, в то время как подпрограммы задают алгоритм решения задачи. Между атрибутами и подпрограммами есть логические связи, и объединяя атрибуты и подпрограммы в единый именованный контейнер, мы просто фиксируем эту связь. Таким образом, понятие контейнера (английский термин, выбранный для его наименования, – unit) можно считать простым средством агрегации логически связанных данных и действий в

единое целое.

Более строго, *контейнер* (*unit*) можно определить как поименованную совокупность атрибутов и подпрограмм, которая может быть параметризована типами или константами, и может быть использована для задания типов, конструирования новых контейнеров при помощи наследования или для прямого использования атрибутов и подпрограмм данного контейнера в других контейнерах и отдельно-стоящих подпрограммах.

Контейнер можно рассматривать как определение множества данных и операций над ними – то есть, как задание некоторого типа. Тем самым, можно определить объект, тип которого будет контейнером. Во-вторых, можно предоставить открытое (общедоступное) содержимое контейнера для *использования* в некотором программном коде, то есть, включить его ресурсы в некоторый контекст. Наконец, атрибуты и подпрограммы контейнера могут (пере)использоваться при создании нового контейнера. Такой механизм носит название *наследования*. Таким образом, различные варианты использования контейнера приводят к понятиям *типа*, *модуля* и *класса*. В языке SLang сохраняются преимущества единой нотации задания контейнеров, с возможностью явного задания различных способов использования контейнеров.

Однородная система типов. Система типов языка SLang является однородной. Это означает, что в языке отсутствует деление на различные категории типов (например, «встроенные в язык» и «определяемые пользователем»). Любой тип, используемый в программе, определяется единообразно, посредством универсальной конструкции «контейнер» (unit). Единственное различие в системе типов заключается в том, что некоторые наиболее часто используемые на практике типы – целый, вещественный, булевский, а также такие структуры, как массивы, списки, словари и т.д. – определены в качестве библиотечных и могут использоваться в программе «по умолчанию». Типы, определяемые пользователем, используются наравне с библиотечными типами без каких-либо ограничений.

Контрактное программирование. Подход к проектированию программ на основе понятия *контракта* (Design by contract (c))[2], изначально разработанный и обоснованный Б.Майером и реализованный в языке Эйфель, в настоящее время является общепринятым средством повышения надежности и верифицируемости ПО. Подход в том или ином объеме реализован во многих современных ЯП. Язык SLang поддерживает полный спектр механизмов *контрактного программи-*

рования, включая пред- и постусловия для подпрограмм и инварианты контейнеров и циклов. Система поддержки времени выполнения обеспечивает эффективную (параллельную, где это возможно) проверку условий и инвариантов.

Параллельное программирование. В отличие от большинства современных языков, где поддержка параллельности реализована на уровне библиотек и носит ограниченный и слабо верифицируемый характер, SLang включает удобный и достаточно надежный механизм распараллеливания программ на уровне самого языка. В языке имеется простой и компактный набор конструкций и спецификаторов для задания многопоточности и синхронизации по доступу к данным. Кроме того, семантика языка допускает автоматическое распараллеливание исполнения.

Безопасность. Проблема, связанная с неконтролируемым использованием нулевых указателей («пустых» или «повисших» ссылок), является одной из наиболее распространенных в практике программирования, а также одной из самых опасных по своим последствиям с точки зрения обеспечения надежности программ. В то же время, контроль доступа по таким указателям не имеет удовлетворительного решения в традиционных языках.

В языке SLang проблема пустых указателей трактуется не как самостоятельная проблема, а как часть более общей проблемы некорректной работы с *неинициализированными атрибутами*. Пустая ссылка считается разновидностью неинициализированного атрибута, и в языке имеются механизмы, которые строго ограничивают случаи, когда действительно нужны неинициализированные атрибуты, от ситуаций, когда всякая сущность должна иметь определенное значение. В дополнение к этому имеется надежный механизм перехода от потенциально неинициализированных атрибутов к инициализированным – своего рода мостик от «опасного» мира в «безопасный». Эта схема работает как для ссылочных типов, так и для типов-значений.

4 Ссылки

[1] ISO International Standard ISO/IEC 14882:2014(E) – Programming Language C++.

[2] https://itunes.apple.com/us/book/the-swift-programming-language
/id881256329?mt=11. (Последнее обращение 19.03.2017).

[3] https://golang.org/ref/spec. (Последнее обращение 19.03.2017).

[4] Б.Мейер. Почувствуй класс. Учимся программировать хорошо с объектами и контрактами, М.: НОУ "Интуит Бином, 2011, ISBN: 978-5-9963-0573-5

Теоретико-графовые методы и системы программирования

Касьянов В. Н., kvn@iis.nsk.su
Институт систем информатики им. А. П. Ершова СО РАН
Новосибирский государственный университет

Аннотация

Доклад посвящен теоретико-графовым методам и системам программирования, работа над которыми ведется в лаборатории конструирования и оптимизации программ ИСИ СО РАН при финансовой поддержке Российского фонда фундаментальных исследований.
Ключевые слова: визуализация, графы, графовые алгоритмы, системы программирования.

Современное состояние программирования нельзя представить себе без теоретико-графовых методов и алгоритмов. Широкая применимость графов в программировании связана с тем, что они являются очень естественным средством объяснения сложных ситуаций на интуитивном уровне [3].

Среди первых работ, существенно использующих теоретико-графовые методы в решении задач программирования, можно отметить широко известные работы А. П. Ершова по организации вычисления арифметических выражений (1958 г.), граф-схемной модели для императивных программ в виде операторных алгоритмов (1958 – 1962 гг.), теории схем Янова с использованием их графового представления и концепции так называемой разметки (1966 – 1968 гг.) и граф-схемной теории экономии памяти (1961 – 1966, 1972 гг.). Первой книгой, посвященной применению графов в программировании, была изданная в 1977 г. монография А. П. Ершова [1], в которой он рассмотрел две

классические задачи теоретического программирования, решения которых и развитые на этих решениях методы привели к созданию теоретического программирования как самостоятельной математической дисциплины. Это задача экономии памяти в операторных схемах Лаврова и задача построения полной системы преобразований в схемах Янова. В данной книге, написанной в виде беседы с читателем, Андрей Петрович продемонстрировал применение графовых методов к решению задач программирования в действии, начиная с элементарных постановок решаемых задач и кончая полным решением проблем во всей их сложности.

Программирование, по словам А. П. Ершова, — это новый вид универсальной деятельности, при которой человек должен вложить в ЭВМ все, что видит, слышит, знает, и научить ее всему, что делает сам. Важнейшим свойством информационной модели или управляющей системы является ее структура, или говоря математическим языком, совокупность бинарных отношений на наборах элементарных единиц данных и действий. Эти структуры данных и структуры действий являются единственными ипостасями программ и обрабатываемой ими информации, в которых они могут существовать в воображении программиста во чреве компьютера. Вот почему, утверждал Андрей Петрович, графы являются основной конструкцией для программиста. Он считал, что "графы обладают огромной, неисчерпаемой изобразительной силой, соразмерной масштабу задачи программирования", и говорил, что "программисту о графах нужно много знать, при этом с большим запасом по отношению к любой конкретной задаче".

Поэтому не случайно, что в отличие от Москвы, где, начиная с работ Юрия Ивановича Янова, в большей степени развивался логический подход к программированию, или Киева, где в работах Виктора Михайловича Глушкова и его учеников явно прослеживается приоритет алгебраических методов, Новосибирск стал центром применения теоретико-графовых моделей и методов в программировании. Созданная в Новосибирске академиком А. П. Ершовым и его учениками авторитетная школа программирования, пользующаяся мировой известностью, внесла значительный вклад в становление и развитие теоретического и системного программирования с использованием теоретико-графовых методов [2].

Это направление по-прежнему продолжает активно развиваться и в наши дни, теперь уже в работах сотрудников Института систем инфор-

матики СО РАН, носящего имя академика А. П. Ершова. В докладе я остановлюсь на некоторых из этих работ, выполняемых в лаборатории конструирования и оптимизации программ ИСИ СО РАН при финансовой поддержке Российского фонда фундаментальных исследований (грант N 15-07-02029).

Описываются системы WikiGRAPP и WEGA, предназначенные для широкого круга специалистов, использующих методы теории графов при решении своих задач на компьютерах, в первую очередь для системных и прикладных программистов. WikiGRAPP — это вики-словарь по теории графов и ее применениям в информатике и программировании. WEGA является вики-энциклопедией, которая содержит описание теоретико-графовых методов и алгоритмов решения задач информатики и программирования, в том числе задач трансляции и конструирования программ. Словарь и энциклопедия открыты для доступа, пополнения и развития и предназначены для накопления и систематизации знаний по прикладной теории графов.

Рассматриваются задачи визуализации графов и методы решения задач визуализации структурированной информации большого объема с использованием иерархических графовых моделей.

Описывается система Visual Graph для визуализации атрибутированных иерархических графов большого размера. Система ориентирована на визуализацию структур данных, возникающих в компиляторах, позволяет одновременно работать с ними как в графовой, так и в текстовой форме и обеспечивает плавность выполнения основных операций над графами, содержащими до 100000 элементов (вершин и дуг). Система использует для спецификации входного (визуализируемого) графа стандартный язык описания графов GraphML и предоставляет богатые возможности для навигации по графовой модели и анализа ее структурных свойств, для работы с атрибутами ее элементов, а также для настройки системы на нужды конкретного пользователя.

Рассматриваются задачи конкретизации (оптимизации в контексте) и редукции программ, и описываются методы их решения на основе аннотирования программ и трансформационного подхода, использующего конкретизирующие и редуцирующие преобразования аннотированных программ.

Описывается система Reduce для минимизации компиляторных тестов, являющихся C/C++ и Fortran программами. Reduce поддерживает расширяемый набор преобразований, направленных на редукцию

программ-тестов с сохранением воспроизводимости ошибок компилятора. Эти ошибки могут проявляться как на стадии трансляции, так и во время исполнения оттранслированной программы. Например, такой ошибкой может быть разница в результатах исполнения программ, полученных из одной и той исходной с применением и без применения оптимизаций при трансляции. Преобразования выполняются системой Reduce на внутреннем представлении программы-теста в виде так называемого гибридного абстрактного синтаксического дерева. В отличие от обычного синтаксического дерева в гибридном дереве те части программы, которые заведомо не будут преобразовываться, могут не раскрываться в виде поддеревьев, а оставаться в виде текстовых вершин.

Описывается визуальная среда CSS параллельного программирования на базе функционального языка Cloud Sisal, поддерживающего аннотированное программирование. Задача среды — предоставить любому пользователю, имеющему выход в Интернет, возможность без установки дополнительного программного обеспечения на своем рабочем месте в визуальном стиле создавать и отлаживать переносимые параллельные программы на языке Cloud Sisal, а также в облаке осуществлять эффективное решение своих задач, исполняя на некотором супервычислителе, доступном ему по сети, созданные и отлаженные переносимые Cloud-Sisal-программы, предварительно адаптировав их под используемый супервычислитель с помощью облачного оптимизирующего кросс-компилятора, предоставляемого средой.

Система CSS использует внутреннее представление Cloud-Sisal-программ в виде атрибутированных иерархических графов, построенных из простых и составных вершин, дуг, портов и типов. Вершины соответствуют вычислениям. Простые вершины обозначают литералы или операции, такие как сложение или деление. Составные вершины представляют составные конструкции, такие как структурные выражения и циклы. Порты — это вершины основного графа, которые используются для изображения операндов вычислений. Они делятся на входные и выходные порты. Дуги соединяют порты и изображают передачу значений от одного операнда к другому. Типы — это атрибуты дуг, которые представляют типы значений, передаваемых по этим дугам.

Список литературы

1. *Ершов А. П.* Введение в теоретическое программирование (беседы о методе). — М. : Наука, 1977. — 288 с.
2. *Касьянов В. Н.* Ершов и графы в программировании // Андрей Петрович Ершов – ученый и человек. — Новосибирск : Изд-во СО РАН, 2006. — С. 157—160.
3. *Касьянов В. Н.*, *Евстигнеев В. А.* Графы в программировании: обработка, визуализация и применение. — СПб. : БХВ-Петербург, 2003. — 1104 с.

Методы и системы дистанционного обучения программированию

Касьянова Е. В., kev@iis.nsk.su
Институт систем информатики им. А. П. Ершова СО РАН
Новосибирский государственный университет

Аннотация

Описываются исследования, ведущиеся в лаборатории конструирования и оптимизации программ Института систем информатики им. А. П. Ершова СО РАН и на кафедре программирования Новосибирского государственного университета по разработке адаптивных методов и системы дистанционного обучения программированию в рамках проблемного подхода, а также по созданию визуальной облачной среды для поддержки обучения параллельному и функциональному программированию.
Ключевые слова: адаптивная гипермедиа, дистанционное обучение, облачные вычисления, обучение программированию.

Стремительное развитие информационных технологий и проникновение их во все стороны жизни общества и во все сферы производственной деятельности приводит к тому, что выпускнику вуза, чтобы стать успешным в своей дальнейшей деятельности, не достаточно освоить существующие пользовательские технологии и получить навыки поиска готовых решений, а необходимо научиться решать возникающие задачи с помощью программирования. Однако овладение умением программировать все еще остается весьма сложной задачей для многих студентов.

Системы дистанционного обучения в настоящее время активно исследуются и развиваются. Выгоды сетевого обучения ясны: аудиторная и платформенная независимости. Сетевое обучающее программное обеспечение, один раз установленное и обслуживаемое в одном

месте, может использоваться в любое время и по всему миру тысячами учащихся, имеющих любой компьютер, подключенный к Интернету. Тысячи программ сетевого обучения и других образовательных приложений стали доступны в сети Интернет за последние годы. Проблема состоит в том, что большинство из них являются не более чем статичными гипертекстовыми страницами и слабо поддерживают проблемный подход к обучению. Появившиеся в последнее время адаптивные гипермедиа-системы существенно повышают возможности обучающих систем. Целью этих систем является персонализация гипермедиа-системы, ее настройка на особенности индивидуальных пользователей.

Всё большую популярность набирают облачные сервисы, которые предоставляют различные возможности, в том числе и для обучения. Такие облачные хостинги, как Amazon и Cloud9, предоставляют вычислительные ресурсы, не требуя дополнительной установки библиотек поддержки времени исполнения и позволяя выполнять программный код внутри браузера, причем безопасно для пользователя. Растущее количество сервисов в сети Интернет, предоставляющих различные услуги, прямо или косвенно связанные с вычислениями, говорит о популярности такого подхода. Очень привлекательно, не обладая компилятором для того или иного языка программирования, иметь возможность в любой момент зайти на соответствующую страницу в сети и выполнить интересующий код на этом языке. Подобные сервисы варьируются от совершенно аскетических, рассчитанных на исключительно короткие программы, до предоставляющих богатую среду разработки с группировкой по проектам, подсветкой синтаксиса и т. д.

Цель исследований, представленных в данном докладе, — разработать методы и средства дистанционного обучения программированию, которые позволят сделать процесс обучения программированию более индивидуальным, доступным и эффективным. Описываются исследования, ведущиеся в лаборатории конструирования и оптимизации программ Института систем информатики им. А. П. Ершова СО РАН и на кафедре программирования Новосибирского государственного университета по разработке адаптивных методов и системы дистанционного обучения программированию в рамках проблемного подхода, а также по созданию визуальной облачной среды для поддержки обучения параллельному и функциональному программированию [1],[2],[3].

Работа выполняется при частичной финансовой поддержке Россий-

ского фонда фундаментальных исследований (грант РФФИ N 15-07-02029).

Список литературы

1. *Kasyanov V. N.*, *Kasyanova E. V.* Cloud system of functional and parallel programming for computer science education // Proceedings of 2015 2nd International Conference on Creative Education (ICCE 2015), June 27-28, 2015, London, UK. — SMSSI, 2015. — С. 270—275.
2. *Kasyanov V. N.*, *Kasyanova E. V.* WAPE – a system for distance learning of programming // Learning to Live in the Knowledge Society : Proceedings of the 20th IFIP World Computer Congress. — Boston : Springer, 2008. — С. 255—256.
3. *Касьянова Е. В.* Адаптивные методы и средства поддержки дистанционного обучения программированию. — Новосибирск : ИСИ СО РАН, 2007. — 170 с.

Дизайн и реализация языка программирования с обобщенными множествами, типами и отображениями в качестве значений первого класса

Квачев В. Д., rasielll@gmail.com
Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

Целью настоящей работы является разработка экспериментального языка программирования с выразительной системой типов, обеспечивающей предотвращение ошибок с сохранением читаемости и удобства написания кода. Это достигается унифицированным подходом к типам, множествам и отображениям и возможностью их использования в качестве значений первого класса. В докладе будет представлен обзор основных возможностей языка и некоторых деталей реализации.
Ключевые слова: языки программирования, системы типов, уточняющие типы, типы как множества, типы как предикаты, градуальная типизация.

Многие современные языки программирования, такие как Haskell, Scala, Agda, используют систему типов как средство отсеивания дефектов, указания ограничений на сущности и проводимые с ними операции, а также повышения выразительности и удобства использования. Среди актуальных в настоящее время инструментов систем типизации можно рассматривать уточняющие типы, представляющие собой типы, дополненные логическими предикатами [2]. Системы типов, основанные на уточняющих типах, обладают высокой степенью автоматизации и выразительности при верификации программного кода.

Они не требуют подробного приведения доказательств, как в системах с зависимыми типами, и позволяют пользователю легко конструировать ограничения [8]. Во многих существующих реализациях систем с уточняющими типами, уточняющие типы ортогональны основной системе типов языка. Например, в расширении LiquidHaskell языка Haskell конструкции с уточняющими типами приводятся в специальных аннотациях исключительно для статической проверки корректности кода [9]. При этом даже если базовые типы можно вывести автоматически, они обязательны к указанию. Встраивание логических предикатов в систему типов и построение языка вокруг уточнений потенциально дает новые способы решения задач, именно такая попытка предпринята в настоящей работе.

Синтаксические и семантические возможности систем с уточняющими типами можно расширить за счет унификации подхода к использованию типов, множеств и отображений. В проектируемом языке программирования работа с этими элементами ведется одними и теми же обобщенными средствами: операциями пересечения [1; 5], объединения [1] и другими, включая определенные пользователем. При этом важной особенностью является возможность использования произвольных функций, в том числе анонимных, на уровне типов. Приведем фрагмент текста программы, иллюстрирующий данные особенности.

```
Nat = Int, (>0)
Even x = (x % 2 = 0)

myFunction y (x : Even, Nat, (<y), {1, 2, 3, 4, 5}) : {Int & (>1)} =
    {x + 1}
```

В этом примере описаны типы натуральных и четных чисел, а также функция, оперирующая ими. Функция *myFunction* принимает 2 аргумента, y и x. Второй аргумент является четным натуральным числом, принадлежащим множеству, и меньшим, чем первый аргумент y. Возвращаемое значение функции является множеством целых чисел больше 1. Определено это значение как множество всех возможных x, увеличенных на 1.

Как видно из примера, предикаты и множества могут быть использованы буквально на месте типов. Чтобы позволить такое поведение, сущность, обобщающая предикаты, типы и множества, должна принимать значение и возвращать его же при выполнении условия (соответственно: удовлетворения предикату и принадлежности типу или множеству). За неудовлетворение предикату считается невозвращение

значения по завершении вычисления.

В языке в типовых аннотациях не запрещено использовать и другие отображения, если для них возможно вычислить обратное отображение. В части случаев это тривиально и делается автоматически (например, увеличение на число, или возврат значения, удовлетворяющего условию), в других случаях поведение должно быть определено пользователем.

В качестве прототипной реализации языка программирования был разработан интерпретатор [10], представляющий собой заготовку для статической проверки программы. Интерпретатор преобразует текст программы в систему утверждений о переменных, которая может быть нормализована до конкретных значений, если для этого хватает информации, тем самым вычисляя выражения на языке программирования. Система может быть проверена SMT-решателем, и нормализована до конкретных значений, если для этого хватает информации, тем самым динамически вычисляя выражения на описываемом языке программирования.

Использование произвольных функций в типовых аннотациях языка со статической типизацией невозможно из-за того, что во время статической проверки типов может быть недостаточно информации о динамических значениях. Чтобы сохранить эту возможность, необходимо ввести элементы динамических языков: использование градуальной типизации (gradual typing) позволило бы бесшовно сочетать статические и динамические проверки, перенося последние на этап выполнения [3; 7]. Она может быть использована с уточняющими типами [4].

В планы на будущее входит задача отделения этапа статических проверок от проверок на этапе выполнения с внедрением градуальной типизации и исследование возникших осложнений.

[heading=bibintoc,title=Список литературы]

Список литературы

1. *Backes M.*, *Hriţcu C.*, *Maffei M.* Union, Intersection, and Refinement Types and Reasoning About Type Disjointness for Secure Protocol Implementations // Journal of Computer Security. — 2014.
2. *Freeman T.*, *Pfenning F.* Refinement types for ML // PLDI. — 1991.
3. *Garcia R.*, *Clark A. M.*, *Tanter É.* Abstracting Gradual Typing // POPL. — 2016.

4. *Lehmann N.*, *Tanter É.* Gradual Refinement Types // POPL. — 2017.
5. *Pereira M.*, *Alves S.*, *Florido M.* Liquid Intersection Types // EPTCS. — 2015.
6. Refinement types for secure implementations / J. Bengtson [и др.] // ACM TOPLAS. — 2011.
7. *Siek J. G.*, *Taha W.* Gradual Typing for Functional Languages // Scheme and Functional Programming Workshop. — 2006.
8. *Vazou N.*, *Bakst A.*, *Jhala R.* Bounded Refinement Types // ICFP. — 2015.
9. *Vazou N.*, *Seidel E. L.*, *Jhala R.* LiquidHaskell: Experience with Refinement Types in the Real World // Haskell Symposium. — 2014.
10. Репозиторий с реализацией интерпретатора. — URL: `http://github.com/rasie1/c-of-x`.

О парадигме универсального языка параллельного программирования

Климов А. В.[1], arkady.klimov@gmail.com
[1]Институт проблем проектирования
в микроэлектронике РАН

Аннотация

Существует много разных платформ параллельных вычислений, и для каждой требуется писать программу практически с нуля. Мы предлагаем единый универсальный язык, на котором можно один раз записать алгоритм решения задачи, отладить его, а затем выводить с минимальными усилиями эффективный код для каждой платформы. Пользователю, возможно, придется сообщать компилятору дополнительную информацию о способе отображения вычислений на ресурсы платформы.
Ключевые слова: параллельное программирование, графический язык программирования, потоковая модель вычислений, синтез программ.

Для параллельного программирования существует много разных платформ, и даже различных парадигм: SMP, PGAS, MPI, CSP, Active Messages (AM), GPGPU. Они предъявляют разные требования к структуре и способам организации алгоритма, в связи с чем, переходя к новой парадигме, приходится придумывать алгоритм практически заново.

Мы предлагаем проект нового языка и инструментальной системы параллельного программирования, используя которые можно записать один раз математическое описание алгоритма, а затем систематически выводить из этого описания эффективные программы для выбранной вычислительной платформы. Поэтому будем называть наш язык UPL (Universal Parallel Language).

Под «выводить» мы понимаем не обязательно автоматически, но, возможно, с привлечением дополнительной информации от программиста. Как правило, эта информация будет относиться к вопросу о способе отображения данной схемы алгоритма на выбранную платформу. В частности, в ней могут содержаться указания по распределению вычислений и данных по пространству и времени, по агрегации данных и вычислений с целью снижения накладных расходов, способам вычисления редукций и т.п. Иными словами, нашей целью будет отделить основу алгоритма от деталей, привносимых задачей отображения алгоритмов на ту или иную платформу, причем эти детали могут быть либо вынесены в отдельные файлы, либо разбросаны по основному алгоритму в формате псевдокомендариев.

Существующие средства и языки параллельного программирования плохо отвечают данной цели: они основаны на модели последовательного программирования, которая вынуждает программиста при записи алгоритма принимать лишние решения о порядке действий. Это осложняет дальнейшую задачу анализа и оптимизации кода, не говоря об увеличении трудозатрат. Хорошая форма записи должна побуждать автора фиксировать только математическую (информационную) структуру алгоритма, его вычислительный граф. Иначе говоря, это что от чего и как зависит, а не порядок вычислений или иные аспекты их организации (распределение по процессорам, блочность и т.п.). Эти дополнительные аспекты должны привноситься в программы на более поздних стадиях, с учетом характеристик используемой вычислительной платформы и, по возможности, автоматически. И если математический алгоритм был отлажен, то эти дополнения не должны нарушать его правильность.

Такую парадигму принято называть WORE (или WOCA): write once – run everywhere (compute anywhere). В последние годы наблюдается растущий интерес к данной теме в мире. Ниже приводятся известные автору проекты. Примечательно, что все они так или иначе опираются на потоковую (dataflow) модель вычислений.

LabVIEW (National Instruments) [5] – развивается с конца 80-х, имеет графический входной язык в парадигме потоков данных, транслируется в C, сейчас есть компиляторы для OpenMP и CUDA. Разрабатывается для FPGA, а в перспективе для MPI. Позиционируется как среда программирования для инженеров.

Syngle Assingment C (SAC)[6] – разработка ряда европейских университетов. Язык имеет C-подобный синтаксис, развитые средства ра-

боты с многомерными массивами как значениями (в духе map/reduce), имеет семантику единственного присваивания. Позиционируется как функциональный язык для научных вычислений.

Concurrent Collections (CnC) [1] – разработка ряда фирм и университетов США, восходит к работе [4]. Вычислительный граф представляется как набор коллекций, а каждая коллекция это набор однотипных вершин, индексированных тегами. Вершины (соответсвенно, коллекции) делятся на два сорта: действия (steps) и значения (items). Между коллекциями действий и значений проходят дуги записи и чтения. Этот проект по форме наиболее близок к нашему языку UPL, но есть и существенные отличия. В докладе будем сравниваться с ним подробнее.

Polyhedron model [2] – промежуточное представление распараллеливающих компиляторов для аффинных гнезд циклов. Может считаться прообразом всех упомянутых моделей (включая нашу) для данного класса программ. Есть генераторы кодов для OpenMP, CUDA, FPGA.

ПИФАГОР [8] – проект, развиваемый в Красноярском техническом университете. Он основан на своеобразном функционально-потоковом языке ПИФАГОР, который имеет спицифический синтаксис и предназначен скорее для исследовательских целей (в плане архитектурнонезависимого программирования), чем для практического применения. По внутренней логике он близок к SAC.

Важным отличием языка UPL от всех перечисленных предшественников является его опора на парадигму раздачи в отличие от парадигмы сбора, лежащей в основе большинства современных языков. Мы говорим, что язык или программа сделаны в *парадигме сбора* (gather), если при описании некоторого вычислительного фрагмента мы задаем явно, откуда взять аргументы, но не указываем, где будут использованы результаты: они просто кладутся в некоторое место в памяти. Для *парадигмы раздачи* все наоборот: в рамках фрагмента все аргументы уже лежат «под рукой», но результаты должны быть переданы в места их будущего использования. Хотя парадигма раздачи является непривычной, и потому на первый взгляд трудной, но она упрощает дальнейшую деятельность по отображению алгоритма на параллельную и, особенно, на распределенную архитектуру.

Charm++ [3] – Это объектный язык распределенного программирования в парадигме активных сообщений, компилируемый в C++. И, хотя он не преследует цель WORE, он нам здесь интересен, поскольку тоже «работает» в парадигме раздачи.

Язык UPL воплощает потоковую (dataflow) модель вычислений, но не классическую, а приведенную к парадигме раздачи. Она была получена как обобщение модели вычислений параллельной потоковой вычислительной системы (ППВС), разработанной акад. В.С. Бурцевым и его командой [9; 10].

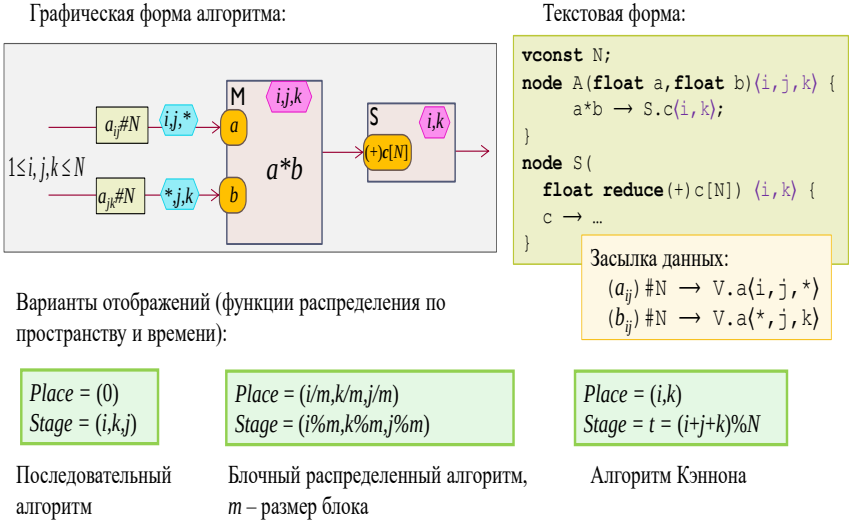

Рис. 1: Алгоритм умножение матриц NxN

У языка UPL наряду с обычным текстовым представлением имеется графическая форма. В качестве иллюстрации на Рис.1 показан алгоритм умножения квадратных матриц. В программе два узла: M умножает пару аргументов, S суммирует результаты. Узел M имеет трехмерный тег $\langle i, j, k\rangle$. Элемент a_{ij} (b_{jk}) подается в виде токена на узлы $\mathrm{M}\langle i, j, *\rangle$ (соответственно, $\mathrm{M}\langle *, j, k\rangle$). Символы $\#N$ означают, что кратность токена равна N. Токены «встречаются» на узле $\mathrm{M}\langle i, j, k\rangle$. Произведение отправляется на узел $\mathrm{S}\langle i, j\rangle$, где суммируются N произведений от узлов $\mathrm{M}\langle i, *, k\rangle$ (здесь всюду $*$ означает «произвольное значение»).

Снизу приведены три варианта отображения, каждое задается своей парой функций распределения place и stage. Обе имеют аргументом тег, значение place – номер процессора, значение stage – номер этапа.

На выходе будут получены три разных, в традиционном понимании, алгоритма (когда будет изготовлен соответствующий компилятор).

Более детально графическая форма языка UPL (под старым названием DFL-G) представлена в [7]. Пока проект находится в начальной стадии и не имеет реализаций, помимо обобщаемого им языка DFL, работающего на эмуляторе вышеупомянутой системы ППВС. Но уже есть много примеров написанных на нем алгоритмов в самых разных областях: разностные уравнения, сортировка, обработка графов (задачи BFS, SSSP, MST, разбиения на сообщества), задача N тел, SAT-задача, молекулярная динамика. И каждый такой пример помещается буквально "на одном экране".

Список литературы

1. Concurrent collections programming model / M. G. Burke [и др.] // Encyclopedia of Parallel Computing / под ред. D. Padua. — Springer Verlag, 2011. — С. 364—371.
2. *Feautrier P.*, *Lengauer C.* Polyhedron model // Encyclopedia of Parallel Computing / под ред. D. Padua. — Springer Verlag, 2011. — С. 1581—1592.
3. *Kale L.* Charm++ // Encyclopedia of Parallel Computing / под ред. D. Padua. — Springer Verlag, 2011. — С. 256—264.
4. *Knobe K.*, *Offner K.* TStreams: a model of parallel computation: Technical Report / HP Cambridge Research Lab. — 2005. — № 78.
5. LabVIEW. — URL: http://russia.ni.com/labview.
6. *Scholz S.-B.* Single Assignment C - Functional Programming Using Imperative Style // In John Glauert (Ed.): Proceedings of the 6th International Workshop on the Implementation of Functional Languages (IFL'94). — Norwich, England, UK : University of East Anglia, 1994. — С. 21.1—21.13.
7. *Климов А. В.*, *Окунев А. С.* Графический потоковый метаязык для асинхронного распределенного программирования // МЭС-2016, сборник трудов, часть II, 3–7 октября 2016 года. — М.: ИПМ РАН, 2016. — С. 151—158.

8. *Легалов А. И.* Функциональный язык для создания архитектурно-независимых параллельных программ // “Вычислительные технологии”. — 2005. — Т. 10, № 1. — С. 71—89.

9. Параллелизм вычислительных процессов и развитие архитектуры суперЭВМ / под ред. В. С. Бурцев. — Москва : ИВВС РАН, 1997.

10. Параллельная потоковая вычислительная система – дальнейшее развитие архитектуры и структурной организации вычислительной системы с автоматическим распределением ресурсов / А. Л. Стемпковский [и др.] // “Информационные технологии”. — 2008. — № 10. — С. 2—7.

Краткая история суперкомпиляции в России

Климов А. В., klimov@keldysh.ru
Романенко С. А., romansa@keldysh.ru
Институт прикладной математики
им. М. В. Келдыша РАН*

Аннотация

Суперкомпиляция — это метод преобразования программ, предложенный В. Ф. Турчиным в 1970-е годы и далее развиваемый его учениками и последователями. В докладе дается краткое введение в суперкомпиляцию, справка о работах В. Ф. Турчина, подробнее о работах последних десятилетий в России, отмечены современные проблемы метавычислений и дальнейшие направления развития.
Ключевые слова: суперкомпиляция, специализация программ, верификация программ, метавычисления.

Введение *Суперкомпиляция* — это метод преобразования программ, предложенный Валентином Федоровичем Турчиным (1931-2010) в 1970-е годы, основанный на идее развертки графа путей вычисления программы (*прогонка*) и выполнения над ним операций, обеспечивающих в конце концов получение конечного графа, являющегося представлением остаточной программы, эквивалентной исходной (*конфигурационный анализ*). История работ В. Ф. Турчина с библиографией описана в статье [10]. Популярное изложение метода суперкомпиляции можно найти в работе [9].

*Работа поддержана грантом РФФИ № 16-01-00813-а.

Из-за ограниченности объема здесь не обозреваются многочисленные зарубежные работы по суперкомпиляции, приведен сокращенный список литературы и многие работы упоминаются без ссылок на публикации.

Суперкомпиляторы для языка Рефал В. Ф. Турчин начал разрабатывать суперкомпилятор для языка Рефал в середине 1970-х годов, пользуясь реализацией Рефала-2 на БЭСМ-6, но лишь после эмиграции в США смог опубликовать описание метода и завершить первые работающие версии суперкомпилятора Рефала. Библиография его работ приведена в [10].

С 1993 года дальнейшим развитием суперкомпилятора Рефала занимался А. П. Немытых. К началу 2000-х годов суперкомпилятор, названный SCP-4, приобрел законченный вид, и А. П. Немытых подготовил и опубликовал его подробное описание [14].

Исследовательские и экспериментальные суперкомпиляторы Параллельно шел поиск более простых систем метавычислений и суперкомпиляторов. Первой такой системой был реализованный С. А. Романенко в конце 1970-х годов простой специализатор для Рефала-2. Его принципы были позднее описаны в [11].

В начале 1990-х Анд. В. Климов и R. Glück выделили ядро суперкомпиляции для модельного функционального языка [1]. Затем С. М. Абрамов описал на Haskell'е другие компоненты суперкомпиляции, включая *окрестностый анализ*, и на его основе предложил метод *окрестностного тестирования*, а также реализацию *универсального решающего алгоритма* и методику *инверсного программирования* на основе суперкомпиляции [7].

Линию простых суперкомпиляторов продолжил И. Г. Ключников расширив суперкомпиляцию на языки с функциями высшего порядка [12].

Корректность суперкомпиляторов Первые подробные доказательства корректности отдельных алгоритмов метода суперкомпиляции были проведены С. М. Абрамовым [7]. Анд. В. Климов разработал формальные спецификаций простых суперкомпиляторов с целью последующего использования их для доказательств корректности. Первое доказательство корректности простого суперкомпилятора в системе Coq, проверяемое на компьютере, разработал Д. Кръстев. Вслед за

ним И. Г. Ключников и С. А. Романенко реализовали альтернативный метод проверки на компьютере корректности суперкомпиляции, назвав его *сертифицирующей суперкомпиляцией*: вместе с остаточной программой суперкомпилятор порождает доказательство ее эквивалентности исходной на входном языке системы Coq [4].

Формальные свойства «свистков» (правил остановки суперкомпилятора) исследовала Антонина Н. Непейвода [6].

Приложения к верификации программ Первый практический результат по верификации коллекции моделей протоколов с помощью SCP-4 получили А. П. Немытых и А. П. Лисица [13]. Впоследствии эти эксперименты были повторены с помощью Java-суперкомпилятора [2] и простого суперкомпилятора, запрограммированного на языке Scala. Анализируя их, Анд. В. Климов определил компактный предметно-ориентированный суперкомпилятор для счетчиковых систем переходов [3].

И. Г. Ключников успешно применил суперкомпилятор функционального языка высшего порядка [12] для проверки эквивалентности программ и использовал его для построения *двухуровеневого суперкомпилятора.*

Дальнейшего повышения мощности и эффективности проверки эквивалентности программ добился С.А. Гречаник путем синтеза методов *многорезультатной суперкомпиляции* и *насыщения равенствами* [8].

Многорезультатная суперкомпиляции В суперкомпиляции, как и во многих методах оптимизации программ, много степеней свободы. Вместо изобретения эвристик можно перебирать всевозможные решения, порождая множество остаточных программ, и выбирать из них «лучшую» по некоторому критерию или использовать их все для каких-то задач. Отсюда возникла идея и реализация *многорезультатной суперкомпиляции* [5].

Оказалось, что успех в применении суперкомпиляторов для верификации протоколов использовал многорезультатность путем варьирования эвристик руками, а специализированный алгоритм для счетчиковых систем [3] проводит автоматический частичный перебор с учетом специфики данного класса задач.

Суперкомпилятор языка Java В 1999–2003 годах Анд. В. Климов, Арк. В. Климов и А. Б. Шворин разработали суперкомпилятор JScp для языка Java (актуальной в то время версии 1.4) [2]. Прогонка была проработана достаточно глубоко с целью сохранения как можно большей информации о значениях ссылочных переменных. Конфигурационный анализ был упрощен: он обрабатывал только явные циклы в программе, а рекурсивными функциями не занимался. Всевозможные степени свободы в стратегиях суперкомпиляции были выведены на пользователя в виде опций.

Основная проблема, с которой столкнулись попытки практического применения Java-суперкомпилятора, — необходимость вручную подбирать опции стратегий суперкомпиляции, изучая остаточный код и догадываясь, как прошел процесс суперкомпиляции и как его направить по-другому.

Направления будущих работ

- разработка упрощенных суперкомпиляторов — как демонстрационных для теоретических исследований и преподавания, так и практических проблемно-ориентированных;
- реализация суперкомпиляторов для реальных функциональных языков (Haskell, SML и т.п.);
- перенос достижений с функциональных языков на практические объектно-ориентированные типа Java;
- теория суперкомпиляции и построение доказательств важных свойств суперкомпиляторов (особенно — проверяемых на компьютере), методы верификации суперкомпиляторов, сертифицирующей суперкомпиляции;
- приложения суперкомпиляции к верификации программ;
- многоуровневая суперкомпиляция;
- многорезультатная суперкомпиляция;
- суперкомпиляция с насыщением равенствами;
- интерактивные метавычисления.

Поясним последний пункт. Мы пришли к убеждению, что автоматические методы преобразования и оптимизации программ подошли некоторому пределу, барьеру сложности. Многорезультатность и использование мощных компьютеров для перебора может несколько отодвинуть эту границу. Тем не менее, практическое применение суперкомпиляции и других методов метавычислений требует вовлечения человека и разработки систем с удобным человеко-машинным интерфейсом, дающим программисту информацию в понятном ему виде о том, как его программа анализируется и преобразуется, что получается на выходе и как связан остаточный код с исходным. Только сам программист знает, с какой целью он занимается преобразованиями своих программ и какие аспекты стоит оптимизировать, а какие не надо. Такие средства должны быть не слабее широко используемых диалоговых отладчиков. Это направление работ мы называем *интерактивными метавычислениями.*

Список литературы

1. *Glück R.*, *Klimov A. V.* Occam's razor in metacomputation: the notion of a perfect process tree // Static Analysis. Lecture Notes in Computer Science. T. 724. — Springer, 1993. — С. 112—123.
2. *Klimov A. V.* A Java Supercompiler and its application to verification of cache-coherence protocols // Perspectives of Systems Informatics. PSI 2009. Lecture Notes in Computer Science. T. 5947. — Springer, 2010. — С. 185—192.
3. *Klimov A. V.* A simple algorithm for solving the coverability problem for monotonic counter systems // Automatic Control and Computer Sciences. — 2012. — Т. 46, № 7. — С. 364—370.
4. *Klyuchnikov I. G.*, *Romanenko S. A.* Certifying supercompilation for Martin-Löf's type theory // Perspectives of System Informatics. PSI 2014. Lecture Notes in Computer Science. T. 8974. — Springer, 2015. — С. 186—200.
5. *Klyuchnikov I. G.*, *Romanenko S. A.* Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions // Perspectives of Systems Informatics. PSI 2011. Lecture Notes in Computer Science. T. 7162. — Springer, 2012. — С. 201—226.

6. *Nepeivoda A. N.* Turchin's relation for call-by-name computations: a formal approach // Electronic Proceedings in Theoretical Computer Science. — 2016. — Т. 216. — С. 137—159.

7. *Абрамов С. М.* Метавычисления и их приложения. — М.: Наука, 1995.

8. *Гречаник С. А.* Доказательство свойств функциональных программ методом насыщения равенствами // Программирование. — 2015. — № 3. — С. 44—61.

9. *Климов А. В.* Введение в метавычисления и суперкомпиляцию // Будущее прикладной математики: Лекции для молодых исследователей. От идей к технологиям. — М.: КомКнига, 2008. — С. 343—368.

10. *Климов А. В.* О работах Валентина Федоровича Турчина по кибернетике и информатике // Труды SORUCOM-2011. — 2011. — С. 149—154.

11. *Климов А. В.*, *Романенко С. А.* Метавычислитель для языка Рефал. Основные понятия и примеры: Препринт / М.: ИПМ им. М.В. Келдыша АН СССР. — 1987. — № 71.

12. *Ключников И. Г.* Суперкомпиляция функций высших порядков // Программные системы: теория и приложения. — 2010. — Т. 4, № 4. — С. 37—71.

13. *Лисица А. П.*, *Немытых А. П.* Верификация как параметризованное тестирование (эксперименты с суперкомпилятором SCP4) // Программирование. — 2007. — № 1. — С. 22—34.

14. *Немытых А. П.* Суперкомпилятор SCP4: общая структура. — М.: Эдиториал УРСС, 2007.

Динамически формируемый код: синтаксический анализ контекстно-свободной аппроксимации

Ковалев Д. А., dmitry.kovalev-m@ya.ru
Григорьев С. В., semen.grigorev@jetbrains.com
Санкт-Петербургский государственный университет,
Россия, 199034, Санкт-Петербург,
Университетская наб. 7/9;
Лаборатория языковых инструментов JetBrains

Аннотация

Многие программы в процессе работы формируют из строк исходный код на некотором языке программирования и передают его для исполнения в соответствующее окружение (пример — dynamic SQL). Для статической проверки корректности динамически формируемого выражения используются различные методы, одним из которых является синтаксический анализ регулярной аппроксимации множества значений такого выражения. Аппроксимация может содержать строки, не принадлежащие исходному множеству значений, в том числе синтаксически некорректные. Анализатор в данном случае сообщит об ошибках, которые на самом деле отсутствуют в выражении, генерируемом программой. В докладе будет описан алгоритм синтаксического анализа более точной, чем регулярная, контекстно-свободной аппроксимации динамически формируемого выражения.

Ключевые слова: синтаксический анализ, динамически формируемый код, контекстно-свободные грамматики, GLL, GFG, dynamic SQL

Современные языки программирования общего назначения поддерживают возможность работы со строковыми литералами, позволяя

формировать из них выражения при помощи строковых операций. Строковые выражения могут создаваться динамически, с использованием таких конструкций языка, как циклы и условные операторы. Данный подход широко используется, например, при формировании SQL-запросов к базам данных из программ, написанных на Java, C# и других высокоуровневых языках.

Недостаток такого метода генерации кода заключается в том, что формируемые выражения не проходят статические проверки на корректность и безопасность, что приводит к ошибкам времени исполнения и усложняет сопровождение системы. Включение обработки динамически формируемых строковых выражений в фазу статического анализа осложняется тем, что такие выражения, в общем случае, невозможно представить в виде линейного потока, который принимают на вход традиционные алгоритмы лексического/синтаксического анализа.

Для решения данной проблемы были разработаны специальные методы статического анализа множества значений формируемого выражения. Как правило, язык, на котором написана исходная программа, тьюринг-полон, что делает невозможным проведение точного анализа. Ряд существующих методов использует для анализа *регулярную аппроксимацию* — множество строк, генерируемых программой, аппроксимируется сверху регулярным языком, и анализатор работает с его компактным представлением, таким как регулярное выражение или конечный автомат.

В магистерской диссертации [5] был описан алгоритм, позволяющий проводить синтаксический анализ регулярной аппроксимации (детерминированного конечного автомата) множества значений динамически формируемого выражения. Основой для данного алгоритма служит алгоритм обобщенного синтаксического анализа Generalized LL (GLL, [7]). Такой подход позволяет получать конечное представление множества деревьев вывода (SPPF, [6]) корректных строк, содержащихся в аппроксимации. Это представление может быть использовано для проведения более сложных видов статического анализа и для целей реинжиниринга.

GLL позволяет работать с произвольными КС-грамматиками, в том числе неоднозначными. Такая возможность достигается путем использования специальной структуры стека (GSS) и механизма *дескрипторов*. Дескриптор хранит в себе информацию о состоянии анализатора в определенный момент времени, достаточную для про-

должения процесса анализа с этого момента. Оригинальный GLL-алгоритм был модифицирован для работы с нелинейным входом (конечный автомат представляется в виде графа). Дескрипторы нового алгоритма хранят номер вершины входного графа вместо позиции в линейном потоке. Также, на шаге исполнения просматривается не единственный входной символ, а все ребра, исходящие из текущей вершины.

Мы расширили описанный подход, изменив алгоритм таким образом, чтобы он мог обрабатывать более точную, чем регулярная, *контекстно-свободную аппроксимацию* значений динамически формируемого выражения. Использование более точной аппроксимации позволяет снизить количество ложных синтаксических ошибок, возникающих в результате того, что аппроксимирующее множество содержит строки, осутствующие среди значений искомого выражения. Под контекстно-свободной аппроксимацией здесь подразумевается грамматика, описывающая контекстно-свободный язык, который содержит в качестве подмножества возможные значения выражения. Идея использования такой аппроксимации была заимствована из существующих в данной области работ [1; 2].

Наш алгоритм принимает на вход графовое представление аппроксимирующей КС-грамматики. В качестве такого представления был выбран Grammar Flow Graph (GFG, [4]). Алгоритм последовательно обходит узлы GFG, производя синтаксический анализ порождаемых им строк. Для правильного построения таких строк алгоритм должен манипулировать дополнительным стеком. Информация о текущей вершине данного стека записывается в дескрипторы. Другой особенностью работы с GFG является то, что он, в отличие от регулярной аппроксимации — детерминированного конечного автомата, допускает возможность неоднозначного выбора пути обхода. Подобная ситуация возникает при наличии в исходной грамматике нескольких продукций, содержащих в левой части одинаковый нетерминал. Механизм дескрипторов позволяет решать проблему недетерминированного выбора пути — для каждого из возможных вариантов создается отдельный дескриптор, который добавляется в очередь исполнения.

Алгоритм был реализован на языке программирования F# в рамках проекта YaccConstructor. Исходный код доступен по ссылке: `https://github.com/YaccConstructor/YaccConstructor`. Результаты проведенных синтетических тестов показали снижение количества ложных синтаксических ошибок при анализе динамически фор-

мируемого кода. В дальнейшем планируется провести аппробацию на реальных данных. Задача синтаксического анализа графов также возникает в области биоинформатики [5]. Предполагается, что наш алгоритм может увеличить производительность анализа метагеномных сборок за счет использования более компактного представления входных данных — из конечного автомата, описывающего сборку, может быть извлечена КС-грамматика [3], графовое представление которой содержит меньше состояний.

Список литературы

1. *Doh K.-G.*, *Kim H.*, *Schmidt D. A.* Abstract LR-Parsing // Formal Modeling: Actors, Open Systems, Biological Systems: Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday / под ред. G. Agha, O. Danvy, J. Meseguer. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. — С. 90—109. — ISBN 978-3-642-24933-4. — DOI: 10.1007/978-3-642-24933-4_6. — URL: http://dx.doi.org/10.1007/978-3-642-24933-4_6.
2. *Minamide Y.* Static approximation of dynamically generated web pages //. — In Proceedings of the 14th International Conference on World Wide Web, WWW '05. ACM, 2005. — С. 432—441.
3. *Nevill-Manning C. G.*, *Witten I. H.* Identifying Hierarchical Structure in Sequences: A Linear-time Algorithm // J. Artif. Int. Res. — USA, 1997. — Сент. — Т. 7, № 1. — С. 67—82. — ISSN 1076-9757. — URL: http://dl.acm.org/citation.cfm?id=1622776.1622780.
4. *Pingali K.*, *Bilardi G.* A Graphical Model for Context-Free Grammar Parsing // Compiler Construction: 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings / под ред. B. Franke. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2015. — С. 3—27. — ISBN 978-3-662-46663-6. — DOI: 10.1007/978-3-662-46663-6_1. — URL: http://dx.doi.org/10.1007/978-3-662-46663-6_1.
5. *Ragozina A.* GLL-based relaxed parsing of dynamically generated code : Master's Thesis / Ragozina Anastasiya. — SPbU, 2016.
6. *Rekers J. G.* Parser generation for interactive environments : PhD Thesis / Rekers Joan Gerard. — Citeseer, 1992.

7. *Scott E.*, *Johnstone A.* GLL parsing // Electronic Notes in Theoretical Computer Science. — 2009. — Т. 253, № 7. — ISSN 1571-0661. — DOI: 10.1016/j.entcs.2010.08.041.

Уменьшение цены абстракции при типобезопасном встраивании реляционного языка программирования в OCaml

Дмитрий Косарев, Dmitrii.Kosarev@protonmail.ch
Санкт-Петербургский государственный университет
Математико-механический факультет

Аннотация

В данной работе затронуты детали OCanren — типобезопасной реализации реляционного языка программирования из семейства miniKanren на OCaml, а именно, два подхода для организации типов логических значений. Первый, наивный, подход позволил получить типобезопасную реализацию, однако привел к понижению производительности на стадии унификации по сравнению с реализацией miniKanren на Racket. Второй подход не страдает этим недостатком.

Ключевые слова: OCaml, miniKanren, унификация, цена абстракции.

Реляционный язык программирования miniKanren позволяет записывать программы как формулы, состоящие из отношений (relations). Относительная простота miniKanren привела к появлению целого семейства реализаций, большинство из которых были сделаны либо для различных диалектов LISP, либо на типизируемых языках в бестиповой манере. OCanren является типизированной реализацией miniKanren на OCaml — языке программирования со строгой статической типизацией.

В OCanren разрешается унифицировать между собой только те логические переменные и значения, которые инкапсулируют значения одинакового типа. Это позволяет находить на стадии компиляции случаи, при которых унифицируются значения разных типов, и,

следовательно, унификация не может завершиться успешно ни при каких условиях. Также компилятор, теоретически, может генерировать вызов специализированной для конкретного типа унификации вместо обобщенной.

При создании новой реализации miniKanren необходимо обдумать два аспекта: в каком порядке будут вычисляться результаты (порядок поиска), и как именно будет происходить процесс унификации. В OCanren была реализована истинно полиморфная унификация: единая (для всех типов агументов) функция позволяет унифицировать любые значения одинаковых типов. Унификация производится путём сравнения ориентированных графов (в случае miniKanren — деревьев), т.к. все значения OCaml хранятся в памяти именно в таком виде.

Однако наивный подход, связанный с объявлением алгебраического типа логических значений `'a logic`, который содержит в себе либо логические переменные (конструктор `Var`), либо обычные значения (конструктор `Value`), приводит к тому, что размер представлений (деревьев) значений в памяти увеличивается. Например, целые числа типа `int` хранятся в виде одного блока памяти. Логическое представление для этих чисел будет состоять из двух блоков памяти: один для конструктора `Value` и один непосредственно для числа, тем самым увеличивая высоту дерева представления в два раза: с единицы до двух.

Для списков (и других *рекурсивных* структур данных) высота деревьев также увеличивается в два раза. Например, список чисел из одного элемента будет занимать в памяти три блока и образовывать дерево высотой два. Логический эквивалент такого же списка будет представляться в памяти деревом размером 6 и высотой 4, что будет сказываться на производительности унификации. Таким образом, общая производительность OCanren оказывается в разы меньше чем изначальный вариант.

Альтернативный подход не использует алгебраических типов данных при объявлении типа логических значений (если говорить упрощенно, то `type 'a injected = 'a`). Таким образом, не появляется дополнительных блоков памяти в представлении данных, размер деревьев не увеличивается, и производительность унификации увеличивается по сравнению с первым случаем. Данный подход требует некоторых преобразований получившихся значений, если они содержат в себе свободные логические переменные. Эти преобразования, однако, происходят один раз в конце вычислений и влияют на производительность

существенно меньшим образом. Также подход требует специфической работы на уровне системы типов, а именно использования примитивов для конструирования логических значений.

```
val lift: 'a -> ('a,'a) injected
val inj: ('a, 'b) injected -> ('a, 'b logic) injected
val distribute: ('a,'b) injected t -> ('a t, 'b t) injected
```

Первые два примитива универсальны, последний вид примитивов не объявлен заранее для каждого вида `t`, его можно сгенерировать, если тип `t` представим как функтор.

Список литературы

[1] Jason Hemann, Daniel P. Friedman. μKanren: A Minimal Core for Relational Programming // Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme '13).

[2] Dmitrii Kosarev, Dmitri Boulytchev. Typed Embedding of a Relational Language in OCaml // Workshop on ML, 2016.

Имитация процедурно-параметрической парадигмы с применением библиотеки макроопределений

Косов П. В., Копцев А. Е.

Аннотация

В работе приведено описание библиотеки моделирования процедурно-параметрических конструкций, позволяющих гибко развивать функционал программных продуктов, написанных на процедурных языках программирования.

Ключевые слова: процедурно-параметрическое программирование, эволюционная разработка.

При разработке больших программных продуктов необходимо предусматривать возможность последующего расширения или модификации ранее написанного кода. В настоящее время данная задача решается преимущественно в рамках объектно-ориентированной парадигмы программирования. Однако, существуют и альтернативные подходы, одним из которых является процедурно-параметрическая парадигма. Ее особенность проявляется в инструментальной поддержке эволюционного расширения ранее написанного кода за счет использования обобщенных записей и обобщающих параметрических процедур. Это позволяет добавлять новую функциональность без изменения ранее написанных исходных текстов, что обеспечивает динамическое связывание альтернатив за счет использования процедурно-параметрического полиморфизма, что в свою очередь повышает гибкость при разработке программ и уменьшает число ошибок, вносимых во время модификации уже написанных исходных текстов.

Технически механизмы процедурно-параметрической парадигмы реализуются за счет независимого расширения абстрактных типов

данных и связанных с ними процедур, что позволяет легко реализовать их как в процедурных, так и в функциональных языках. Появляющиеся при этом возможности обеспечивают эволюционное прямое расширение программ даже в случае множественного полиморфизма. Вместе с тем полная поддержка процедурно-параметрической парадигмы без избыточного дублирования других языковых конструкций может быть обеспечена лишь при помощи создания новых языков программирования. Основная проблема при этом возникает в том, что новые языки тяжело внедрить в широкое использование ввиду поддержки малого количества платформ, длительного периода разработки, а также необходимости постоянной поддержки в виде дополнительных библиотек, фреймворков. Сравнительно небольшая известность как языка, так и парадигмы, лежащей в его основе, будет способствовать нескорому получению качественной обратной связи от сообщества.

Одним из путей более быстрой апробации новой парадигмы является использование специализированной библиотечной поддержки предлагаемых ею конструкций в уже существующих языках. Это позволяет оценить возможности нового стиля и обеспечить более гибкое создание программ. В работе рассматривается одно из возможных решений, ориентированное на использование макропроцессора и шаблонов языка программирования C++. С их помощью создана библиотека макроопределений, предоставляющая обертку над программными объектами данного языка. Рассмотрены особенности реализации основных конструкций, поддерживающих процедурно-параметрическую парадигму. Показаны возможности использования этих конструкций при разработке эволюционно расширяемых программ.

Моделирование расширяемых данных. Расширение данных, в ряде случаев соответствующее построению обобщенных записей процедурно-параметрической парадигмы, можно получить за счет применения наследования к обычным структурам. При этом отсутствие внутри структур виртуальных методов и конструкторов не позволяет использовать объектно-ориентированный полиморфизм, но избавляет данные структуры от избыточности, обеспечивая реализацию полиморфизма за счет имитации процедурно-параметрического полиморфизма внешними функциями. Добавление каждой специализации и ее регистрация могут осуществляться в процессе расширения программы поэтапно и независимо за счет подключения к проекту дополнительных заголовочных файлов и единиц компиляции со специализациями без изменения ранее написанного кода, что обеспечивает требуемое

эволюционное расширение данных.

Добавление параметрических процедур. Семантическая модель параметрической процедуры формируется из моделей обобщенной параметрической процедуры и обработчиков специализаций. Обобщенная параметрическая процедура задает общий интерфейс для всех добавляемых обработчиков каждой из специализаций. В общем случае ее реализация опирается на задание многомерного массива, размерность которого определяется числом обобщенных аргументов в вызове процедуры и зависит от размерности реализуемого мультиметода. Регистрация каждого из обработчиков специализаций может осуществляться в своей единице компиляции. Регистрируемые функции, являющиеся обработчиками специализаций, используют для фиксации своего местоположения в массиве в качестве индексов значения признаков, автоматически сформированных во время регистрации специализаций обобщения.

Добавление мультиметодов. Мультиметоды, осуществляющие обработку двух или более обобщенных параметров сначала должны пройти процедуру регистрации, после которой осуществляется выбор одной из комбинаций входных параметров через массив указателей на функции-обработчики конкретных отношений двух специализаций. Для изменения мультиметода (добавления в него новых комбинаций входных параметров) необходимо после добавления специализации реализовать код мультиметода для нее в отдельной единице компиляции.

Список литературы

[1] Legalov A, Kosov P. Evolutionary software development using procedural-parametric programming. // CEE-SECR '13 Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia. ACM New York, NY, USA ©2013. ISBN: 978-1-4503-2641-4. Article No. 3.

[2] Легалов И. А. Применение обобщенных записей в процедурно-параметрическом языке программирования // Науч. вестн. НГТУ. 2007. No 3 (28). с. 25–38.

[3] Легалов А.И. Мультиметоды и парадигмы. // Открытые системы, № 5 (май) 2002, с. 33-37.

[4] Легалов А.И., Косов П.В. Эволюционное расширение программ с использованием процедурно-параметрического подхода // Вычислительные технологии. 2016. Т. 21. № 3. с. 56-69.

[5] clang: a C language family frontend for LLVMclang: a C language family frontend for LLVM. - Адрес доступа: https://clang.llvm.org/ (дата обращения 20.01.2017)

[6] Using the Meta-Object Compiler (moc). - Адрес доступа: http://doc.qt.io/qt-4.8/moc.html (дата обращения 20.01.2017)

Кроссплатформенное средство разработки программного обеспечения «Платформа ДОМИНАНТА»

Кручаненко А. Ю., mag@platdom.ru
ООО «Платформа ДОМИНАНТА»

Аннотация

Рассмотрены современные концепции разработки программного обеспечения. Изложены характеристики кроссплатформенного средства разработки «Платформа ДОМИНАНТА».
Ключевые слова: компилятор, средства разработки, парадигмы программирования, ООП, обобщённое программирование, событийное программирование, управляемый код.

По мере совершенствования вычислительной техники возрастает сложность разрабатываемого программного обеспечения (ПО). Растёт объём исходного кода системных и прикладных программ. Одновременно в современных рыночных условиях к инструментам разработки предъявляется требование увеличения производительности разработки ПО. Для выполнения этих требований научным сообществом и коммерческими участниками отрасли предлагаются новые концепции разработки – парадигмы программирования. Безусловно актуальным является синтез хорошо зарекомендовавших себя концепций и оригинальных идей в новых средствах разработки ПО.

Для решения задач промышленного производства было создано средство разработки приложений «Платформа ДОМИНАНТА».

Цель настоящей работы состояла в запуске третьей версии платформы на линейке оборудования на основе архитектур ARM, x86-x64, VLIW, MIPS. Было необходимо обеспечить функционирование платформы под разрабатываемой концерном КРЭТ российской операционной системой реального времени и под широко распространёнными

десктопными и мобильными операционными системами такими как Linux, Windows, Android, iOS, Tizen. Ещё одной задачей стала оптимизация выделений памяти и обеспечение производительности исполнения на аппаратных архитектурах, а также подготовка архитектуры системы к использованию для вычислений ресурсов GPU.

За 12 лет развития, впервые в полностью российском продукте, в комплексе реализованы самые современные подходы разработки ПО. Это объектно-ориентированное, обобщённое, событийное программирование, управляемый код. Все компоненты платформы: компилятор, виртуальная машина и фреймворк созданы «с нуля», без каких-либо заимствований стороннего программного кода.

В «Платформе ДОМИНАНТА» за основу принят синтаксис Java[1] и C#[2]. Они хорошо известны широкому кругу программистов. Дополнительно используются производительные парадигмы известные по C++[3], например множественное наследование. Язык программирования платформы дополнен языковыми расширениями запросов данных.

Созданы собственный кроссплатформенный компилятор и виртуальная машина. Архитектура системы спроектирована таким образом, чтобы перенести максимум вычислительной нагрузки на компилятор и упростить виртуальную машину.

Распространённый алгоритм периодического вызова сборщика мусора для освобождения неиспользуемых участков памяти усложняет виртуальную машину. Сборка мусора является ресурсоёмкой операцией [4][5]. Поэтому «Платформе ДОМИНАНТА» реализована сборка мусора подсчётом ссылок на объект. Подсчёт ссылок выполняется в общем потоке, он распределен по времени и не требует пауз в работе приложения.

Гарантируется своевременность удаления неиспользуемых объектов. Для этого в байт-коде указаны переходы для любых ситуаций времени исполнения. Такой подход переносит вычислительную нагрузку на время компиляции и алгоритмически упрощает виртуальную машину. Этот механизм не зависит от архитектурных особенностей среды исполнения. Обеспечивается стабильность, своевременность высвобождения неиспользуемых ресурсов и производительность. Это позволяет использовать при разработке такие идиомы ООП как RAII (Resource Acquisition Is Initialization) [6] располагая код освобождения ресурсов в деструкторе.

Возникновение исключений в конструкторах и деструкторах до-

пускается и обрабатывается корректно.

Для создания объектов сложной структуры, для которых обычно требуется множество ресурсоёмких операций выделения памяти, предназначен способ создания объединённых объектов. Память для них выделяется за один раз.

Одной из наиболее прогрессивных парадигм разработки используемых в C++ является множественное наследование [7]. В «Платформе ДОМИНАНТА» реализовано полноценное множественное наследование.

В Java и C# механизмы обобщенного программирования [8] были упрощены по сравнению с C++, для них существует множество ограничений. В «Платформе ДОМИНАНТА» используются шаблоны – подход принятый в C++, поэтому ограничения при использовании обобщенного программирования отсутствуют.

Исправлены архитектурные недостатки языков-предшественников, такие как возможность вызова виртуального метода производного класса из конструктора родительского, хотя производный класс в этот момент ещё не инициализирован, что приводит к аварийной остановке программы.

При сравнительном тестировании с Java и C#, компилятор «Платформы ДОМИНАНТЫ» показывает производительность компиляции выше до 40%, и при исполнении, производительность виртуальной машины в сценариях комплексного использовании свойств языка выше до 2-х раз.

«Платформа ДОМИНАНТА» относится к современным кроссплатформенным средствам разработки и является полностью российским продуктом. Платформа реализует объектно-ориентированное, обобщённое, событийное программирование, управляемый код. Её использование позволяет осуществлять производительную разработку ПО для широкого спектра оборудования и операционных систем. Они включают как десктопные и серверные вычислительные системы, так и устройства комплексов пилотажно-навигационного оборудования летательных аппаратов, телекоммуникационное оборудование, системы управления робототехническими комплексами, медицинские устройства и устройства «интернета вещей». Архитектура платформы облегчает добавление новых классов устройств.

ЛИТЕРАТУРА

1. Гослинг Д. Язык программирования Java SE 8, М., 2015

2. Шилдт Г. C# 4.0 полное руководство. М., 2011

3. Лафоре Р. Объектно-ориентированное программирование в C++. СПб., 2017

4. Правильно освобождаем ресурсы в Java // URL: https://habrahabr.ru/post/178405/

5. Инициализаторы объектов в блоке using // URL: https://habrahabr.ru/post/144240/

6. RAII (Resource Acquisition Is Initialization) // URL: https://goo.gl/zhLeAc

7. Множественное наследование // URL: https://goo.gl/FKn7NY

8. Обобщённое программирование // URL: https://goo.gl/Qia2tW

Языковая поддержка архитектурно-независимого параллельного программирования

Легалов А. И.
Сибирский федеральный университет, Красноярск

Аннотация

Рассматриваются особенности концепции архитектурно-независимого параллельного программирования, опирающейся на функционально-потоковую модель параллельных вычислений.

Параллельное программирование давно перестало быть прерогативой высокопроизводительных вычислений. Вместе с тем, подходы к разработке параллельных программ практически мало изменились по сравнению с 80-ми годами прошлого века. Как и раньше основным направлением является написание кода с учетом особенностей используемых архитектур параллельных вычислительных систем, что снижает переносимость и требует переписывания кода при появлении новых вычислительных систем или при модификации существующих. Основным фактором, определяющим особенности методов программирования, является наличие ресурсных ограничений, которые явным образом необходимо учитывать программисту. Эти ограничения проявляются через системы команд и затрагивают процессоры, память, коммутаторы, а также отображаются в особенностях языков параллельного программирования.

В основе архитектурно-независимого параллельного программирования лежит отказ от явного управления вычислениями и исключение каких-либо ресурсных ограничений на уровне языка программирования. Предполагается, что абстрактный вычислитель имеет неограниченные вычислительные ресурсы, которыми не нужно специально

управлять. Ограничениями являются только информационные зависимости между данными. Исходя из этой концепции предполагается, что создаваемая программа может обладать максимально возможным параллелизмом, а ее переносимость обеспечивается сжатием этого параллелизма с учетом ресурсных ограничений конкретной вычислительной системы. Можно выделить следующие характеристики модели параллельных вычислений и языка функционально-потокового параллельного программирования [1]:

- ориентация на вычисления в неограниченных ресурсах, что достигается использованием принципа единственного выполнения каждой операции в сочетании с принципом единственного присваивания;
- программа определяется в виде ациклического информационного графа (циклические процессы описываются на основе рекурсии);
- отсутствие явно выраженных ветвлений, что позволяет рассматривать программу как безусловный информационный граф, на котором по результатам выполнения все дуги имеют разметку;
- распараллеливание на уровне элементарных операций;
- описание только информационных зависимостей, определяющих неявное управление вычислениями на уровне модели вычислений и языка программирования;
- асинхронный параллелизм с управлением по готовности данных;
- отсутствие переменных, что позволяет избежать конфликтов, связанных с совместным использованием памяти;
- наличие алгебры преобразований позволяет оптимизировать структуру программы до начала вычислений.

Предложенный язык программирования обеспечивает написание кода за счет соединения функций программно-формирующими операторами, которые обеспечивают размножение данных и их слияние в различные виды списков. Формально в языке существует только один вид функции: оператор интерпретации.

Создаваемые программы не предполагается непосредственно применять при выполнении реальных расчетов. Также нецелесообразно

разрабатывать параллельный компьютер, ориентированный на функционально-потоковую модель. Ключевая идея заключается в преобразовании этих программ в программы для целевой архитектуры. При этом исходную программу также можно использовать для более простого и гибкого решения различных промежуточных задач. Процесс разработки может включать:

1. Написание необходимых функций. На данном этапе предполагается использование интегрированной среды, поддерживающей трансляцию функций и их сохранение в одном из репозиториев, обеспечивающем хранение без привязки к файловой структуре [2].

2. После написания функции, осуществляется ее трансляция в промежуточное представление, определяющее реверсивный информационный граф (РИГ). В отличие от обычного информационного графа дуги направлены от оператора, принимающего данные, к операторам-источникам данных. Это впоследствии позволяет использовать граф без изменений во время выполнения программ, образуя при запуске оператора прямой доступ к аргументам. Помимо этого данный граф используется для построения управляющего графа (УГ), который определяет передачу управления между операторами [3]. Трансформация управляющего графа позволяет делать первый шаги по адаптации функции к целевой архитектуре.

3. Промежуточные представления могут быть оптимизированы без изменения семантики функции. В ряде случаев возможно преобразование хвостовой рекурсии к итерациями [4].

4. Сформированный РИГ может также использоваться для проведения формальной верификации функции [5]. Полученные в ходе анализа результаты позволяют подтвердить корректность логики программы до ее трансформации в ресурсно зависимую форму. Использование при анализе ресурсно неограниченной модели обеспечивает отсутствие побочных эффектов, связанных с тупиковыми ситуациями и ресурсными конфликтами.

5. Также еще до переноса программы на конкретную архитектуру возможно ее выполнение и отладка на разнообразных тестовых наборах. Для этого используется соответствующий интерпретатор функционально-потоковых параллельных программ [6].

Созданная функционально-потоковая параллельная программа может в дальнейшем быть преобразована к программе для целевого вычислителя с использованием дополнительно разработанных инструментальных средств. Корректная реализация инструментов преобразования позволит избавиться от необходимости проводить специфическую отладку и верификацию применительно к конкретным архитектурам.

Список литературы

[1] Легалов А.И. Функциональный язык для создания архитектурно-независимых параллельных программ // Вычислительные технологии. 2005. № 1 (10). С. 71-89.

[2] Легалов А.И., Матковский И.В., Анкудинов А.В. Особенности хранения функционально-потоковых параллельных программ. / Вестник Сибирского государственного аэрокосмического университета. Вып. 4 (50). 2013. С. 53-57.

[3] Легалов А.И., Савченко Г.В., Васильев В.С. Событийная модель вычислений, поддерживающая выполнение функционально-потоковых параллельных программ. // Системы. Методы. Технологии. № 1 (13). - 2012. - С. 113-119.

[4] Legalov A.I., Nepomnyaschy O.V., Matkovsky I.V., Kropacheva M.S. Tail Recursion Transformation in Functional Dataflow Parallel Programs. // Automatic Control and Computer Sciences, 2013, Vol. 47, No. 7, pp. 366–372. ISSN 0146-4116. © Allerton Press, Inc., 2013.

[5] Kropacheva M., Legalov A. Formal Verification of Programs in the Pifagor Language. / Parallel Computing Technologies, 12th International Confernce PACT September-October, 2013. – St. Petersburg, Russia. // Lecture Notes in Computer Science 7979, Springer, 2013. – Pp. 80-89.

[6] Матковский И.В., Легалов А.И. Инструментальная поддержка трансляции и выполнения функционально-потоковых параллельных программ. / Ползуновский вестник, № 2. - 2013. С. 49-52.

Эволюционная разработка программ с применением процедурно-параметрической парадигмы

Легалов А. И.
Сибирский федеральный университет, Красноярск

Аннотация

Рассматриваются языковые и инструментальные средства, обеспечивающие эволюционную поддержку множественного полиморфизма при процедурно-параметрическом программировании.

Необходимость использования эволюционной разработки программ во многом обуславливается спецификой их применения в различных областях. Часто добавление новых конструкций может осуществляться уже в процессе эксплуатации систем, что ставит актуальной задачу расширения ранее написанного кода. Для инструментальной поддержки безболезненного наращивания программ была предложена процедурно-параметрическая парадигма программирования [1, 2].

Появление процедурно-параметрической парадигмы привело к созданию новых языковых конструкций, расширяющих процедурное программирование. Были предложены обобщенные записи, специализации обобщений, обобщающие параметрические процедуры, обработчики параметрических специализаций.

Обобщенные записи и специализации. Обобщенная запись [3] является основой общей для всех специализаций. Она содержит общие поля по аналогии с вариантной записью языков Паскаль, Модула2. Расширение обеспечивается за счет специализаций, которые могут добавляться во вновь создаваемых модулях. В основе операций над обобщенными записями лежат операции обработки расширяемых

записей языка программирования Оберон-2. Допускается статическое и динамическое создание специализированных записей, указателей на обобщенные и специализированные записи, явное приведение обобщенного типа к типу специализации, проверка типа. Обобщенную запись и ее специализации допускается использовать в операторах присваивания. При наличии эквивалентных специализаций в левой и правой частях операторов присваивания, осуществляется присваивание всех полей записи, стоящей справа от знака «:=», полям, расположенным в его левой части. Если специализации различны, то присваивание осуществляется только для общих полей обычной записи. Допускается также прямое присвоение полям специализации обобщенной записи полей основы специализации. Аналогичная ситуация возможна и при использовании в левой части оператора присваивания основы специализации, когда в правой его части располагается соответствующая специализированная запись. В этом случае поля основы заполняются соответствующими полями специализации.

Обобщающие параметрические процедуры и обработчики специализаций. Обобщенные записи могут использоваться в качестве параметров в обобщающих параметрических процедурах [1, 2]. Тело такой процедуры содержит обработчик по умолчанию, если для всех обобщающих параметров существует тип по умолчанию. В противном случае оно содержит обработчик исключений. Тело обобщающей процедуры может отсутствовать, что задается «приравниванием» его нулевому значению (по аналогии с чистыми функциями языка программирования C++). В этом случае необходимы обработчики специализаций для всех комбинаций обобщенных параметров.

Обработчики специализаций обеспечивают реализацию различных комбинаций специализаций, сопоставляемых с обобщениями из списка обобщающих параметров. Комбинация, на которую «настроен» конкретный обработчик, задается значениями признаков. Каждый элемент списка специализированных параметров должен задавать конкретное значение признака. Специализации должны поэлементно соответствовать параметрам обобщающей процедуры. В обязательном теле процедуры кодируется конкретный метод обработки. Можно использовать один из двух способов задания специализаций, более удобный в рассматриваемом контексте: несколько одинаковых специализаций в группе или несколько разных специализаций в группе.

Подключаемые модули. Идея подключаемых модулей [4] схожа с использованием механизма наследования вместо прямого вклю-

чения типов во вновь формируемый программный объект. В случае с наследованием формируется описание нового типа, расширяющего базовый тип дополнительными понятиями. За счет принципа подстановки осуществляется использование метода производного класса, который может обрабатывать свои собственные дополнительные данные. При использовании подключаемых модулей ситуация отличается тем, что вместо множества экземпляров базового и производного классов в программе существуют только по одному экземпляру разных модулей. Поэтому данный метод не является аналогом наследования. Вместо этого разработанное расширение модуля может подключаться к уже существующему базовому модулю, образуя вместе с ним единое пространство имен. Это отличает подключение модуля от его импорта, при котором внутренние пространства имен модулей не пересекаются. Подключаемый модуль может импортироваться из других модулей, обеспечивая им передачу своего интерфейса и интерфейса расширяемого модуля. Главной особенностью подключаемого модуля является возможность описания в нем расширений обобщенных записей и обобщающих параметрических процедур.

Возможности процедурно-параметрического подхода на примере простых ситуаций. Эволюционная разработка характеризуется добавлением новых программных объектов в уже написанный код. Эти добавления могут быть комплексными и порождать разнообразные комбинации. Можно выделить ряд типичных ситуаций расширения кода, часто встречающихся на практике [5]:

1. расширение обобщений специализациями и, как следствие, расширение обрабатывающих их обобщающих процедур;

2. добавление новых процедур, обеспечивающих дополнительную функциональность;

3. добавление новых полей данных в существующие типы и изменение в соответствии с этим процедур, осуществляющих обработку измененных программных объектов;

4. добавление новых процедур, предназначенных для обработки только одной из специализаций некоторого обобщения;

5. создание нового обобщения на основе существующих специализаций;

6. добавление в программу мультиметодов, осуществляющих обработку двух или более обобщенных параметров;

7. изменение мультиметодов при добавлении в обобщения новых специализаций, используемых в качестве аргументов мультиметодов.

Показано, что процедурно-параметрическое программирование обеспечивает безболезненное эволюционное расширение практически во всех ситуациях. Представляет интерес использование возможностей предлагаемого подхода и в более сложных случаях, образуемых комбинациями простых ситуаций, например, при реализации основных патернов проектирования.

Полученные результаты показывают, что предлагаемый подход обеспечивает прямую поддержку эволюционного расширения, что повышает эффективность процесса разработки программного обеспечения, особенно в случаях использования гибких методов разработки программ.

Список литературы

[1] Легалов А.И. процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? / Красноярск. 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. 43 с.

[2] Легалов А.И. Методы поддержки параметрического полиморфизма. / Научный вестник НГТУ, № 3 (18), 2004. С. 73-82.

[3] Легалов И.А. Применение обобщенных записей в процедурно-параметрическом языке программирования. / Научный вестник НГТУ. 2007. № 3 (28). С. 25–38.

[4] Легалов А.И., Бовкун А.Я., Легалов И.А. Расширение модульной структуры программы за счет подключаемых модулей. / Доклады АН ВШ РФ, № 1 (14). – 2010. – С. 114-125.

[5] Legalov A., Kosov P. Evolutionary software development using procedural-parametric programming. / CEE-SECR '13 Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia. ACM New York, NY, USA ©2013. ISBN: 978-1-4503-2641-4. Article No. 3.

Конвертация функций высшего порядка в реляционную форму

Лозов П. А., lozov.peter@gmail.com
Санкт-Петербургский государственный университет
Математико-механический факультет
Кафедра системного программирования
Лаборатория языковых инструментов JetBrains

Аннотация

В данной работе рассматривается задача преобразования типизированных функциональных программ высшего порядка в реляционные программы.
Ключевые слова: функциональное программирование, функции высшего порядка, реляционное программирование, трансляторы.

1 Введение

Реляционное программирование [5] является привлекательной технологией, основанной на построении программы как отношения. В результате реляционные программы могут быть исполнены в различных "направлениях", что делает возможным, например, моделирование вычисления обратных функций. Помимо того, что это интересно с теоретической точки зрения, такой подход имеет практическое значение: некоторые задачи выглядят гораздо проще, если рассматривать их как запросы к реляционной спецификации. Существует целый ряд примеров, подтверждающих это наблюдение: реляционная программа проверки типов для просто-типизированного лямбда-исчисления может быть использована для вывода типов или решения проблемы

населенности какого-либо типа; реляционный интерпретатор позволяет генерировать “квайны” [4] — программы, результатом исполнения которых являются они сами; реляционная сортировка списка может быть использована в качестве генератора всех перестановок.

Многие логические языки программирования, такие как Prolog, Mercury[1] или Curry[2] в некоторой степени являются реляционными. Существует, однако, язык — miniKanren[3] — который специально разработан в качестве модели для реляционного программирования. Изначально довольно минималистичный DSL для языка Scheme/Racket (его минимальная реализация [6] содержит меньше сотни строк кода), miniKanren нашел применение, будучи встроенным в десятки языков программирования, среди которых Haskell, Standard ML и OCaml [7].

Непосредственное написание реляционных программ иногда бывает весьма утомительным; однако во многих случаях желаемое описание может быть получено из некоторой функциональной программы достаточно регулярным образом. Таким образом мы рассматриваем проблему конвертации функций высшего порядка в реляционную форму. Отметим, что данная работа является обобщением существующего метода [3] конвертации, который применим только для функций первого порядка.

2 Обзор литературы

Проблема конвертации функциональных программ в реляционную форму была рассмотрена в предшествующих работах по miniKanren [3; 5]. Данный подход заключается в преобразовании n-мерной функции в $(n+1)$-мерное отношение, где $(n+1)$-ый аргумент соответствует результату исходной функции. Данный метод, однако, корректно работает только с программами первого порядка.

Мы утверждаем, что в случае программ высшего порядка конвертация не может быть выполнена для нетипизированных термов, а также предлагаем подход для типизированного случая.

[1] `https://mercurylang.org`
[2] `http://www-ps.informatik.uni-kiel.de/currywiki`
[3] `http://minikanren.org`

3 Методология и текущие результаты

Наш подход существенно опирается на знание типов термов. Мы начали с просто-типизированного случая (STLC [2] с конструкторами, сопоставлением с образцом и примитивными типами), который впоследствии обогатили реляционными структурами и правилами типизации для них.

Также мы используем функцию для конвертации типов $[\bullet]$, которая каждому типу сопоставляет реляционный аналог. Например,

$$\begin{aligned} [\texttt{int}] &= \texttt{int} \to \mathfrak{G} \\ [\texttt{string} \to \texttt{int}] &= (\texttt{string} \to \mathfrak{G}) \to (\texttt{int} \to \mathfrak{G}) \end{aligned}$$

где $\mathfrak{G}$ (*тип цели*) обозначает уникальный тип для замкнутых чисто реляционных выражений. Другими словами, мы преобразовываем примитивные значения в одноместные отношения, а функции в отношения высшего порядка.

Мы представляем конструктивные правила для термов, которые проектируют их в чисто реляционное подмножество и доказываем, что для исходной типизированной программы мы всегда получим реляционную форму, тем самым подтверждая статическую корректность. Существует некоторая сложность при обосновании динамической корректности, так как miniKanren в настоящее время не имеет формально описанной семантики. Мы доказываем, что наш подход обеспечивает расширение для известного случая первого порядка, а также демонстрируем целесообразность в случае высшего порядка набором примеров, что оправдывает корректность в *ad hoc*-манере.

4 Текущая и будущая работа

В данный момент мы работаем над расширением нашего подхода для полиморфного случая; кажется, что не каждой полиморфной программе соответствует реляционный аналог, поэтому первый шаг заключается в ограничении системы типов до подмножества “полезных” типов. После завершения описания правил конвертации на “бумаге” мы собираемся реализовать транслятор и использовать его для получения некоторых полезных реляционных форм (основными задачами являются, конечно же, некоторые интерпретаторы, а также miniKanren с ограничениями [1]) Также рассматривается возможность

верификации доказательства с помощью Coq — системы автоматизированного доказательства теорем.

В качестве альтернативного направления для исследований рассматривается разработка собственных систем типов для реляционного программирования.

Список литературы

1. *Alvis C. E.*, *Willcock J. J.*, *Byrd W. E.* cKanren: miniKanren with Constraints // Proceedings of the 2011 Workshop on Scheme and Functional Programming (Scheme '11). —.
2. *Barendregt H.* Lambda Calculi with Types. — Handbook of Logic in Computer Science, Volume II, Oxford University Press, 1993.
3. *Byrd W. E.* Relational Programming in miniKanren: Techniques, Applications, and Implementations. — Indiana University, Bloomington, 2009.
4. *Byrd W. E.*, *Holk E.*, *Friedman D. P.* miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl) // Proceedings of the 2012 Workshop on Scheme and Functional Programming (Scheme '12). —.
5. *Friedman D. P.*, *E.Byrd W.*, *Kiselyov O.* The Reasoned Schemer. — MIT Press, 2005.
6. *Hemann J.*, *Friedman D. P.* μKanren: A Minimal Core for Relational Programming // Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme '13). —.
7. *Kosarev D.*, *Boulytchev D.* Typed Embedding of a Relational Language in OCaml // ACM SIGPLAN Workshop on ML. — 2016.

Облачная среда программирования однородных вычислительных систем

Лукин Н. А.[1], n.a.lukin@urfu.ru
Филимонов А. Ю.[1], a.filimonov@urfu.ru
Тришин В. Н.,[2] trishinvn@yandex.ru
[1]Уральский Федеральный Университет
[2]Институт Машиноведения УрО РАН

Аннотация

Развитие интеллектуальных мобильных объектов требует совершенствования встроенных вычислительных систем. Такие системы должны обеспечивать максимальную производительность и отказоустойчивость. Однородные вычислительные среды позволяют удовлетворить этим требованиям и создавать реконфигурируемые вычислительные системы с элементами искусственного интеллекта. Для обеспечения реализации однородных систем, предназначенных для работы в составе мобильных объектов, необходимо решить ряд задач, связанных с созданием соответствующего программного обеспечения. В работе излагается возможный подход к их решению, основанный на применении комплекса облачных сервисов.
Ключевые слова: Однородные вычислительные системы, параллельное программирование, облачные сервисы

Встроенные вычислительные средства мобильных платформ различного назначения должны удовлетворять требованиям максимальной производительности (режим жесткого реального времени) и отказоустойчивости (обеспечение максимально возможного времени безотказной работы в неблагоприятных внешних условиях). Архитектуры

современных мобильных вычислительных систем основаны на использовании концепции фон Неймана и ее разновидностей. Для выполнения жестких эксплуатационных требований подобные решения используют параллельную обработку данных (максимизация производительности) и структурное резервирование (максимизация отказоустойчивости). Однако практическая реализация данных решений сталкивается с ограничениями на аппаратурные и энергетические ресурсы вычислительных систем, что вызывает необходимость оптимизации, которая ввиду отсутствия единой методологии для систем реального времени для большинства систем приобретает индивидуальный и зачастую непрогнозируемый характер. Одним из вариантов преодоления возможных трудностей и противоречий при создании мобильных вычислительных систем может являться применение однородных вычислительных сред (ОВС).

Базовая структура ОВС представляет собой двумерный массив процессорных элементов (ПЭ), каждый из которых программно настраивается на выполнение операций и соединений с соседними элементами [4], [3]. Основным достоинством ОВС является массовый параллелизм вычислений, а регулярность их структур позволяет наращивать вычислительные возможности путем простого увеличения размеров матрицы и организовать достаточно простые методы контроля и восстановления работоспособности ОВС в процессе эксплуатации.

Современный этап развития ОВС (когда массив ПЭ представляет собой кремниевый кластер СБИС) позволяет достичь малых величин энергопотребления при максимальной производительности, что делает этот подход весьма привлекательными для реализации компонентов технологического комплекса IoT (Internet of Things). Более того, специфика архитектуры ОВС дает возможность производить ее реконфигурацию непосредственно в процессе эксплуатации, что позволяет создавать бортовые системы, аппаратно и программно - адаптируемые к изменениям как внешних условий. Адаптация может осуществляться как внешним управляющим комплексом, так и локально, что обеспечивает дополнительные возможности по самообучению и самоорганизации компонентов IoT.

Технологический процесс программирования ОВС представляет собой настройку массива ПЭ на выполнение операций обработки и передачи данных, который завершается «укладкой» графа потоков данных решаемой задачи в вычислительную решетку [1].Это неразрывно связано с проектированием соответствующего функционально

- ориентированного процессора (ФОП) и сопровождается отладкой временных диаграмм работы ансамблей ПЭ, верификацией логико-временного функционирования на математических и физических макетах. Применению традиционных подходов для программирования подобных вычислителей препятствует жесткое соответствие императивных языков программирования особенностям организации вычислительного процесса на вычислителях с архитектурой фон Неймана [**S4**]. По мнению авторов, платформа программирования, построенная на декларативном языке, способна адекватно описывать и учитывать особенности организации вычислительного процесса ОВС и обеспечивать при этом не только привычные для ЯВУ инструменты отладки программ, но и дополнительные возможности отладки, в том числе — с использованием имитационного и физического моделирования.

Важная особенность компонентов IoT на базе ОВС состоит в том, что они способны работать длительное время, реализуя принцип управляемой деградации, что может быть обеспечено путем удаленного программирования ОВС непосредственно в ходе эксплуатации. Особенности описанной выше цепочки технологических процессов определяют необходимость оперативного взаимодействия с платформой программирования/проектирования, что существенно ограничивает применение локальных приложений из-за возможного разрыва этой цепочки. В последнее время, благодаря развитию систем телекоммуникаций, активное развитие получили платформы программирования, которые реализуются с использованием облачных сервисов (например, Cloud9 и Wolfram). Подобные сервисы сегодня находят все более широкое применение в платформах удаленного управления (технологический комплекс Software Defined Network). Авторы считают, что комплексная технологическая платформа, сочетающая в себе возможности систем программирования и удаленного управления способна обеспечить эффективное решение большинства проблем, которые возникают в процессе программирования и последующей эксплуатации ФОП ОВС. Предложенный подход может быть использован для самых различных вариантов реализации ОВС-ФОП (FPGA, ASIC и т. д.) в составе некоторых типов мобильных систем реального времени.

В качестве первого этапа создания подобной платформы рассматривается система визуального программирования ОВС на базе проекта MTera 2.0 [2]. Применение облачного сервиса программирования ОВС в данном случае обеспечивает независимость от пользовательской ОС, управление доступом пользователей и возможность командной работы

программистов. Web - интерфейс представляет собой редактор поля ПЭ. Каждая клетка поля, изображающая ПЭ — это встроенный SVG-элемент, являющийся копией скрытого, используемого как библиотечный. Видимость символов состояния ПЭ контролируется таблицей стилей.Применяя указатель, пользователь может изменять состояние регистра команд, программируя тем самым матрицу ОВС. На стороне клиента выполняется JavaScript-код передающий данные о действии пользователя на сервер и получающий в ответ новые правила для таблицы стилей. На стороне сервера выполняются Perl-скрипты, обращающиеся посредством SQL-запросов к БД в которой хранятся данные о состояниях процессорных элементов. В ходе дальнейшего развития платформы предполагается разработка и внедрение комплекса программирования ОВС на ЯВУ который будет использоваться совместно с системами моделирования и визуального программирования в итерационном процессе программирования ОВС. Завершающим этапом создания платформы станет ее интеграция с системой удаленного управления.

Список литературы

1. Вычислительные наноструктуры. Часть 1. Задачи, модели, структуры / Г. М. Алакоз [и др.]. — М : Национальный открытый университет ИНТУИТ, 2013.
2. *Лукин Н. А.* Бортовые функционально-ориентированные процессоры на основе однородных вычислительных сред для мобильных систем реального времени // Фундаментальные исследования. — 2015. — № 12.
3. *Лукин Н. А.* Реконфигурируемые процессорные массивы для систем реального времени: архитектуры, эффективность, области применения // Известия ТРТУ. — 2004. — № 9. — С. 36—45.
4. Семейство многопроцессорных вычислительных систем с динамически перестраиваемой архитектурой / Н. Н. Дмитриенко [и др.] // Вестник компьютерных и информационных технологий. — 2009. — № 6.

Построение синтаксических анализаторов на основе алгебраических эффектов

Лукьянов Г. А., glukyanov@sfedu.ru
Пеленицын А. М., apel@sfedu.ru
Институт математики, механики и компьютерных наук
им. И. И. Воровича, Южный Федеральный Университет

Аннотация

Целью работы является исследование применимости современных методов контроля вычислительных эффектов к задаче построения функциональных синтаксических анализаторов. В работе рассматривается язык программирования `Frank` [7], реализованные в нём концепции алгебраических эффектов и обработчиков эффектов и их применение к конструированию функциональных парсеров на основе парсер-комбинаторов. Описывается текущее состояние разрабатываемой авторами с помощью языка `Frank` комбинаторной библиотеки.

Ключевые слова: синтаксический анализ, парсер, комбинаторы парсеров, функциональное программирование, вычислительные эффекты, алгебраические эффекты, обработчики эффектов.

Введение

Чисто функциональные языки программирования позволяют формально рассуждать о свойствах программ, но представляют сложность для организации побочных эффектов, которые требуются в большинстве программ. В данной статье обсуждаются два известных подхода к управлению эффектами, и второй из них, более новый, применяется в задаче построения парсер-комбинаторных библиотек, которая позволяет оценить удобство и гибкость в использовании данного подхода.

1 Вычислительные эффекты

Возможность статического контроля побочных эффектов является одним из принципиальных преимуществ статически типизированных языков программирования с богатыми системами типов. Отделение «чистых» функций от вычислений, обременённых взаимодействием с глобальным состоянием (например, подсистемой ввода-вывода), позволяет с с большей уверенностью рассуждать о надёжности разрабатываемой программной системы.

Зачастую бинарного разделения на «чистые» и «эффективные» вычисления недостаточно: чтобы увеличить степень контроля над поведением программы, вводят более точную классификацию эффектов. К примеру, различают вычисления с изменяемым состоянием и те, которым требуется только фиксированная конфигурация; другой пример: можно отделять файловый ввод-вывод от межпроцессного взаимодействия. Далее, имея базовые блоки-эффекты, важно иметь удобный механизм для их комбинирования и построения сложных вычислений с несколькими побочными эффектами. Для решения этих проблем существует как минимум два класса методов, общая сведения о которых приведены в данном разделе.

1.1 Монадический подход

Исторически первым подходом к типизации вычислений с побочными эффектами является монадический подход [4]. В данном подразделе кратко рассмотрен монадический подход к описанию и комбинированию вычислительных эффектов с примерами на языке `Haskell`.

Монадами в языке `Haskell` являётся типы, которые имеют экземпляр класса типов `Monad`, удовлетворяющий трём законам.

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

join . return = id
join . fmap return = id
join . join = join . fmap join
```

Описание экземпляра монады даёт абстрактному интерфейсу интерпретацию с помощью некоторого конкретного типа (например, тип списка имеет свой экземпляр класса `Monad`).

Дополнительная классификация эффектов в монадическом подходе осуществляется уточнением интерфейса класса типов `Monad`. Типы, используемые для реализации интерфейсов одновременно нескольких монадических классов, называются *стеками монад* (или монадическими стеками). Для конструирования монадических стеков применяются трансформеры монад [3] — типы, добавляющие функционал некоторой конкретной монады к любой другой. Для языка `Haskell` существует популярная библиотека `mtl` [5], предоставляющая базовые средства для программирования с трансформерами монад.

Монадический подход в целом и трансфермеры монад в частности являются зрелой технологией описания вычислений с побочными эффектами. Однако, ряд присущих ей недостатков, связанных с комбинированием эффектов, например, проблема необходимости описания большого числа однотипных экземпляров классов типов для монадических трансформеров, стимулирует к поиску других методов.

1.2 Алгебраические эффекты и язык `Frank`

Перспективным методом описания вычислений с побочными эффектами, активно развивающимся в последние годы, является теория алгебраических эффектов и обработчиков эффектов [1] – рассмотрим их основные особенности с примерами на экспериментальном языке программирования `Frank` [7], который имеет встроенную поддержку указанных концепций.

Основной особенностью алгебраических эффектов является явное разделение интерфейса и реализации эффекта. Синтаксис эффекта задаётся сигнатурой, содержащей перечисление предоставляемых операций. Для синтаксиса языка, определяемого интерфейсом эффекта, нужно задать семантику — описать интерпретатор. Для этого применяется концепция обработчиков эффектов (effects handlers).

2 Синтаксический анализ как вычисление с эффектами

В статье [2] показано, что парсеры могут быть представлены как вычисления, комбинирующие эффекты изменяемого состояния и некорректного завершения. Монадический подход предоставляет необходимые средства для кодирования этой идеи, что успешно использу-

ются в реализациях библиотек функциональных комбинаторов парсеров (самая известная из таких библиотек — Parsec [6]).

Алгебраические эффекты предоставляют альтернативный трансформерам монад подход к описанию вычислений с несколькими побочными эффектами. Описание парсеров в терминах алгебраических эффектов и процесса синтаксического разбора в виде обработки этих эффектов позволяет более ясно увидеть преимущества этих концепций.

2.1 Эффекты для синтаксического разбора и их обработчики в терминах языка Frank

Мы представляем синтаксический анализ как два зависимых эффекта: парсеры единичных символов и парсеры последовательностей.

Интерфейс парсера для одиночных символов представлен тремя командами: остановкой вычисления в случае обнаружения синтаксической ошибки, распознаванием символа, удовлетворяющего предикату, и выбором между двумя парсерами.

```
interface Parser = fail : ParseError -> Char
                 | sat : {Char -> Bool} -> Char
                 | choose : {[Parser] Char} ->
                                    {[Parser] Char}
                                            -> Char
```

Для реализации обработчика эффекта `Parser` производится сопоставление с образцом для команд интерфейса и состояния входного потока. Команде выбора между двумя парсерами назначается следующая семантика: производится попытка применения первого парсера, в случае неудачи — второго, а если и второй парсер применить не удаётся — разбор завершается с ошибкой.

Для описания разбора последовательностей символов предлагается эффект `MultiParser`, содержащий две команды: обёртку для разбора одиночного символа и команду для разбора последовательности символов произвольной длины, разбираемых данным парсером одиночных символов.

```
interface MultiParser = singleton: {[Parser] Char} ->
                                                  List Char
                      | many : {[Parser] Char} -> List Char
```

Имея в распоряжении механизм для разбора последовательностей символов, можно реализовать парсеры более высокого уровня, которые в последствии могут быть использован для описания прикладных инструментов, например, для разбора языков разметки (`Markdown`, `HTML` и др.).

```
letter : [Parser]Char
letter! = sat isLetter

word : [MultiParser](List Char)
word! = many letter
```

Заключение

В данной работе показано, что традиционный монадический подход к построению парсер-комбинаторных библиотек может быть переработан с помощью альтернативной концепции алгебраических эффектов и их обработчиков. Следующим шагом в этом исследовании могло бы стать развёрнутое сравнение преимуществ и недостатков обоих подходов. Однако, большинство очевидных вопросов, которые тут возникают — например, вопрос о производительности полученной в работе библиотеки — упираются в развитость языка `Frank`. Последний, как было показано в работе, предоставляет удобные средства для эксплуатации идеи алгебраических эффектов, однако, прикладное программирование требует большей зрелости от реализации языка: наличия системы модулей, улучшения сообщений об ошибках компиляции и более развитой инфраструктуры, наконец, замены интерпретатора компилятором.

Список литературы

1. *Bauer A.*, *Pretnar M.* Programming with algebraic effects and handlers // J. Log. Algebr. Meth. Program. — 2015. — Т. 84, № 1. — С. 108—123. — URL: `http://dx.doi.org/10.1016/j.jlamp.2014.02.001`.
2. *Hutton G.*, *Meijer E.* Monadic Parser Combinators // Technical Report NOTTCS-TR-96-4. — 1996.

3. *Liang S.*, *Hudak P.*, *Jones M.* Monad Transformers and Modular Interpreters // Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — San Francisco, California, USA : ACM, 1995. — С. 333—343. — (POPL '95). — ISBN 0-89791-692-1. — URL: http://doi.acm.org/10.1145/199448.199528.

4. *Moggi E.* Computational Lambda-calculus and Monads // Proceedings of the Fourth Annual Symposium on Logic in Computer Science. — Pacific Grove, California, USA : IEEE Press, 1989. — С. 14—23. — ISBN 0-8186-1954-6. — URL: http://dl.acm.org/citation.cfm?id=77350.77353.

5. Monad transformers library. — URL: http://hackage.haskell.org/package/mtl (дата обр. 18.10.2016).

6. Parsec, универсальная библиотека монадических комбинаторов парсеров. — URL: https://github.com/aslatter/parsec.

7. Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017 / под ред. G. Castagna, A. D. Gordon. — ACM, 2017. — ISBN 978-1-4503-4660-3. — DOI: 10.1145/3009837. — URL: http://doi.acm.org/10.1145/3009837.

Свободные би-стрелки, или Как генерировать варианты учебных заданий по программированию

Марченко А. А.[1] Зиятдинов М. Т.[1]

[1]Казанский Федеральный университет

Аннотация

Многовариантные задания могли бы более широко использоваться в учебном процессе, однако процесс их подготовки трудоёмкий, монотонный и рутинный. Мы предлагаем решение для автоматизации этого процесса — язык для описания многовариантных заданий с возможностью генерации постановок задач в текстовом виде, тестовых сценариев для проверки корректности решений и эталонного решения для каждого из вариантов.
Ключевые слова: свободные би-стрелки, бестэговое терминальное кодирование, многовариантные задания.

При обучении программированию часто применяются многовариантные учебные задания одинакового уровня сложности для проведения контрольных работ и отработки у студентов навыков решения задач по определенной теме. Несмотря на все плюсы использования такого рода заданий, их подготовка и последующая проверка решений являются очень трудоемкими, что может сильно ограничивать их применение. Решением проблемы является использование инструментов автоматической генерации многовариантных заданий. Некоторые подходы к генерации заданий рассматривались в работах [5; 6]. Мы предлагаем язык описания учебных заданий на основе свободных бистрелок и использование интерпретаторов для генерации вариантов.

Стрелки являются моделью вычисления — обобщением монад. Они были введены в статье Hughes [1]. Для более удобной работы со стрелками Paterson [3] предложил расширение синтаксиса языка Haskell. Alimarine и др. [4] ввели понятие би-стрелок для описания обратимых вычислений.

Мы представляем многовариантное задание в виде схемы (см. рис. 1), в котором каждый элемент является набором преобразований. Фиксируя одно преобразование в каждом наборе, мы получаем один вариант задания.

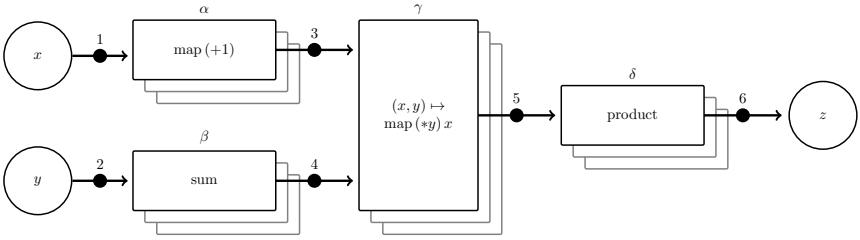

Рис. 1: Задание как схема из наборов преобразований

Это позволяет при помощи небольшого количества преобразований получить большое количество вариантов. Текст варианта можно получить, соединяя тексты отдельных преобразований в топологическом порядке обхода схемы. Аналогично можно получить эталонное решение варианта.

При генерации тестовых сценариев необходимо, чтобы граничные условия отдельных преобразований были учтены в тестовых сценариях для всего варианта в целом, поэтому для каждого элемента задания граничные условия нужно передавать предшествующим и последующим преобразованиям, чтобы получить соответствующие входные и выходные данные. Для этого мы представляем каждое преобразование в виде изоморфизма входных и выходных данных вместе с набором граничных условий и текстом.

Представив схему в виде бестэгового терминального кодирования [2] свободной би-стрелки, мы описываем три её однотипных интерпретатора.

Разработанный пакет программ доступен в репозитории пакетов языка Haskell по ссылке `http://hackage.haskell.org/package/multivariant`.

Список литературы

1. *Hughes J.* Generalizing monads to arrows // Science of Computer Programming. — 2000. — Т. 37, № 1. — С. 67—111. — DOI: 10.1016/S0167-6423(99)00023-4. — arXiv: arXiv:1011.1669v3.
2. *Kiselyov O.* Typed tagless final interpreters // Generic and Indexed Programming. — Springer, 2012. — С. 130—174.
3. *Paterson R.* Arrows and computation // The Fun of Programming. — 2003. — С. 201—222.
4. There and back again / A. Alimarine [и др.] // Proc. of the 2005 ACM SIGPLAN workshop on Haskell. Т. 66. — ACM Press, 2005. — С. 86—97.
5. *Зорин Ю.* Интерпретатор языка построения генераторов тестовых заданий на основе деревьев и/или // Докл. Томского гос. ун-та систем управления и радиоэлектроники. — 2013. — Т. 1(27). — С. 75—79.
6. *Посов И.* Обзор генераторов и методов генерации учебных заданий // Образовательные технологии и общество. — 2014. — Т. 17, № 4.

Транслятор для функционально-потоковых параллельных программ.

Матковский И. В.
Сибирский федеральный университет, Красноярск

Аннотация

Описывается устройство и работа модуля трансляции для параллельных программ на функционально-потоковом языке. **Ключевые слова:** функционально-потоковое параллельное программирование, язык программирования, инструментальные средства

Язык Пифагор [1] представляет собой язык функционально-потокового параллельного программирования и предназначен для изучения способов использования архитектурно-независимого подхода при решении практических задач. Для программирования на языке был разработан ряд утилит, среди которых есть как принципиально важные, отвечающие за основной жизненный цикл программ, так и вспомогательные. В рамках данного доклада будет подробно рассмотрено устройство транслятора.

В системе инструментальных средств языка Пифагор транслятор выступает в качестве первого этапа обработки программ. Используется он двумя способами: для преобразования программного кода на языке Пифагор в промежуточные представления, подходящие для дальнейшей обработки и выполнения и для подготовки входных аргументов вызываемых пользователями функций перед выполнением. Результатом работы транслятора являются реверсивные информационные графы (РИГ)[2] - ациклические ориентированные графы, хранящие образующие программу операторы и информационные связи между ними. Для описания конкретной последовательности выполнения данных операторов отдельной утилитой создается управляющий

граф. Реверсивный информационный граф и управляющий граф вместе содержат достаточно информации для выполнения описываемой ими функции; интерпретатор языка использует эту пару графов и подготовленный с помощью транслятора входной аргумент программы.

При разработке транслятора были сформулированы следующие требования:

- Кроссплатформенность. Язык Пифагор изначально ориентирован на создание архитектурно-независимых программ и привязка инструментальных средств к конкретной платформе не позволяет анализировать поведение разработанных программ в различных условиях

- Отделенность от остальных утилит. В ранних версиях языка Пифагор выполнение программ осуществлялось единой утилитой, объединявшей в себе функции транслятора и интерпретатора. Это не позволяло эффективно взаимодействовать с промежуточными представлениями программ и модифицировать их перед исполнением.

- Использование удобного формата промежуточных представлений. Новый формат промежуточного представления реверсивных информационных графов (РИГ) программ позволял как эффективно работать с ним утилитами, так и модифицировать графы вручную, с помощью текстового редактора.

Транслятор был разработан на языке C++ с использованием библиотеки Qt - это позволило обеспечить необходимую кроссплатформенность. Оформленный отдельной утилитой, он обрабатывал программы на языке Пифагор, создавая из них РИГ и сохраняя полученные РИГ в текстовых файлах. Созданные файлы через специальный модуль репозитория размещались в каталоге, из которого их могли извлечь другие утилиты.

На рисунке 1 показано устройство транслятора программ на языке Пифагор и его взаимодействие с входными и выходными данными.

В трансляторе можно выделить три основных раздела – блок трансляции, транслятор констант и транслятор функций. Блок трансляции, состоящий из лексического и синтаксического анализаторов, отвечает за преобразование входных данных в реверсивный информационный граф; блоки трансляции констант и функций модифицируют процесс преобразования образом, соответствующим обрабатываемым

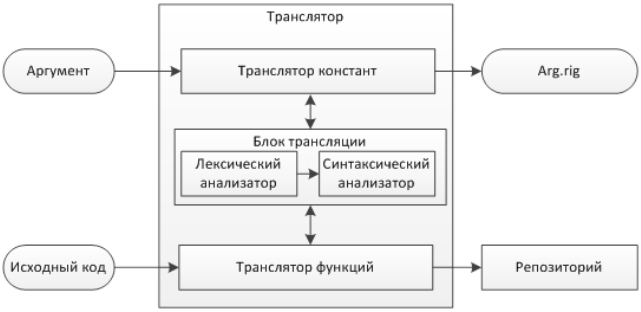

Рис. 1: Транслятор

входным данным. Устройство блока трансляции не представляет особой сложности, тогда как трансляторы констант и функций заслуживают отдельных пояснений.

Файл с программой на языке Пифагор с точки зрения транслятора состоит из объектов двух типов: функций и пользовательских констант. Объекты обоих типов в ходе трансляции преобразуются в РИГ.

Одной написанной на языке Пифагор функции соответствует один РИГ. В РИГ функции гарантированно присутствует минимум две вершины – вершина для входного аргумента и вершина для возвращаемого значения. Предполагается, что РИГ всегда должен быть полностью связным; в дальнейшем возможно появление графов с отделенными от главного тела функции частями. Работа этих частей может никак не влиять на результат самой функции, производя некий побочный эффект. Для поддержания общности вершина для входного аргумента создается даже в тех функциях, которые не получают аргументов извне; в таких случаях вершина аргумента просто остается не связанной с остальной частью РИГ.

Пользовательские константы представляют собой аналог глобальных констант из других языков программирования. Они определяются единожды, и в дальнейшем при необходимости могут быть использованы любой функцией.

Построенный в ходе трансляции функции РИГ является конечным результатом данной трансляции и передается для сохранения в модуль репозитория. Конечным результатом трансляции пользователь-

ской константы является корневая вершина РИГ - в которой, по ходу трансляции, формируется соответствующее константе значение. В случае со массивами данных это может означать, что вместе с корневой вершиной в описании константы хранятся и формирующие её отдельные элементы, однако исполняющая система напрямую оперирует именно с корневой вершиной.

Транслятор используется не только для подготовки заранее написанного программного кода на языке Пифагор, но и для преобразования входных аргументов, с которыми запускаются отдельные функции. В ходе таких преобразований входной аргумент обрабатывается, как одиночное константное значение и оборачивается в временный РИГ. Готовый РИГ сохраняется в служебном файле (arg.rig), откуда его впоследствии извлекает интерпретатор. Необходимости обрабатывать входные аргументы функций, запускаемых по ходу интерпретации не возникает, так как эти аргументы уже будут являться подготовленными для обработки фрагментами данных.

Дальнейшее развитие транслятора может ведется в следующих направлениях:

- Оптимизация функций в ходе трансляции. Оптимизация программ может производиться как через модификацию управляющего графа, так и через изменение информационных связей в РИГ. Возможно предварительное проведение тривиальных вычислений и отключение гарантированно ненужных частей графа.

- Введение поддержки новых языковых конструкций. Развитие языка предполагает введение в него новых конструкций и механизмов; транслятору необходимо понимать такие конструкции и уметь преобразовывать их в части РИГ программы.

Список литературы

[1] Легалов А.И. Функциональный язык для создания архитектурно-независимых параллельных программ // Вычислительные технологии. 2005. № 1 (10). С. 71-89.

[2] Легалов А.И, Непомнящий О.В., Матковский И.В., Фарков. М.А. Особенности преобразования и выполнения функционально-потоковых параллельных программ сборник “Труды НПО 2011”, материалы Ершовской конференции по информатике 2011. С. 146-153

Основанная на OPC система обучения преобразованиям программ «Тренажер параллельного программиста»

Метелица Е. А. Морылев Р. И. Петренко В. В.
Штейнберг Б. Я.
Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

Тренажер параллельного программиста — система для обучения разработчиков оптимизирующих компиляторов. В её основе лежит реальная компилирующая система. В Тренажере визуализируются графовые зависимости программ, влияющие на оптимизацию и распараллеливание. Также визуализируется выполнение преобразований программ, включая преобразование к параллельному коду. Имеется контекстная справка и справочные материалы. Все это позволяет сократить время входа нового разработчика в оптимизирующую компиляцию. Предполагается, что Тренажер параллельного программиста может стать основой диалогового режима компилятора для оптимизации и распараллеливания.
Ключевые слова: Параллельное программирование, компиляторы, визуализация, графовые модели, преобразования программ, решетчатый граф.

В данном проекте описана система визуализации программных зависимостей и эквивалентных высокоуровневых преобразований программ, которые используются в оптимизирующей (и распараллеливающей) компиляции.

Представленная визуализация может использоваться в проектируемом на основе ОРС (Оптимизирующей распараллеливающей системы)[1] диалоговом режиме компиляции. В данный момент эта визуализация используется в «Тренажере параллельного программиста» (ТПП) – электронной системе обучения разработчиков оптимизирующих компиляторов.

При разработке и поддержке оптимизирующих компиляторов возникает проблема ввода новых сотрудников. Оптимизирующий компилятор для современных вычислительных систем – это сложный наукоемкий программный продукт, создание и поддержка которого предполагают высокую специальную квалификацию разработчиков. Введение в группу разработчиков нового программиста требует значительного времени для освоения специфики предстоящих работ. ТПП позволяет на практике наблюдать влияние программных зависимостей на возможность выполнения преобразований и предназначен для сокращения времени знакомства со спецификой разработки оптимизирующего компилятора. Кроме того, ТПП, очевидно, может использоваться как электронное обучающее средство в университетских курсах по оптимизирующей компиляции.

ТПП допускает на входе программы (фрагменты программ) языка Си. В ТПП можно ознакомиться с тем, как работают анализаторы программных зависимостей на входном фрагменте, а также преобразования из имеющейся библиотеки. Тренажер снабжен справочной системой и содержит контекстные подсказки для пользователя. Все текущие наработки ОРС могут быть добавлены в ТПП. В частности, в первую версию ТПП пока не входят генераторы параллельного кода[2].

В основе Тренажера параллельного программиста лежит реальная распараллеливающая система ОРС. Визуализация демонстрирует преобразования программ (ведущих к оптимизации) на небольших по объему фрагментах кода (для простоты восприятия). На экране одновременно представлены исходный и результирующий фрагменты. Пользователь может внести изменения в исходный код, применить преобразования и увидеть на экране изменения (если они допустимы) в результирующем фрагменте. Например, небольшие изменения в индексных выражениях массивов могут изменить граф информационных связей и нарушить эквивалентность преобразования.

Графовые модели программ в ТПП включают в себя: граф информационных связей, решетчатый граф программы, граф потока управ-

ления, граф вычислений и граф вызовов подпрограмм.

Граф информационных связей – основной инструмент контроля эквивалентности сложных оптимизирующих и распараллеливающих преобразований[3]. Для более тонкого анализа зависимостей может быть использовать решетчатый граф[4]. Визуализации решетчатых графов позволяют увидеть возможности параллельного выполнения некоторых итераций гнезда (двумерного) циклов, если ни один цикл гнезда не допускает распараллеливания.

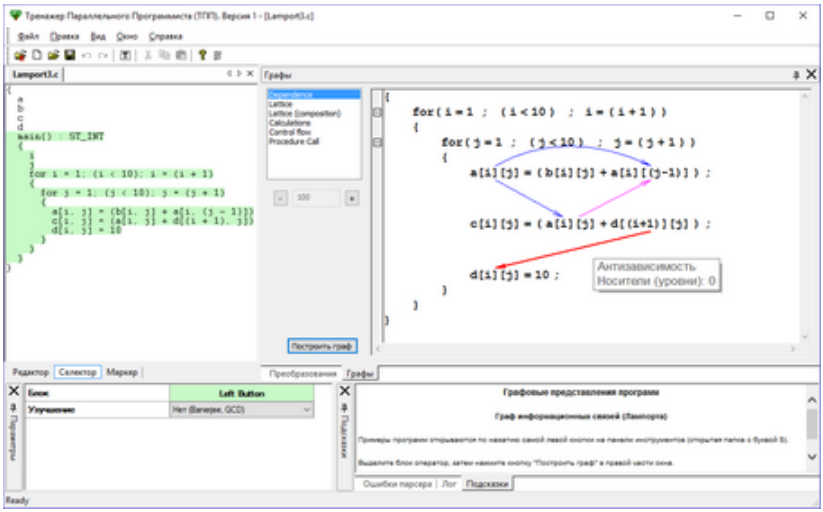

Рис. 1: Граф информационных связей.

Граф потока управления и граф вызовов подпрограмм представляет информацию о последовательности выполнения программы и необходимы для определения корректности преобразований.

Граф вычислений – это промежуточное представление выражений с информацией о задержках для генерации конвейерного кода.

В данный момент, ТПП выполнен на основе внутреннего представления OPC4[5]. Рассматривается вопрос о переносе ТПП на более совершенное внутреннее представление OPC5. При этом может быть добавлен граф укладки вычислений в решетку с поярусной оптимизацией (Mesh Embedding Graph). Он полезен для генерации конвейерного кода и является развитием идей графа вычислений. Также может быть добавлен ряд преобразований: упрощение выражений, улучшен-

ные преобразования условных операторов и операторов циклов и другие[6].

Перенос ТПП на новое внутреннее представление также включает в себя обновление справочной системы, которая будет представлять собой не только руководство по использованию тренажера, но и более полный учебник с описанием графовых моделей и преобразований программ с упражнениями для лучшего освоения материала.

Литература.

1. Оптимизирующая распараллеливающая система `www.ops.rsu.ru` (Дата обращения 27.01.2017)

2. Web-автораспараллеливатель программ `http://ops.opsgroup.ru/` (Дата обращения 27.01.2017)

3. Тренажер параллельного программиста, основанный на OPC4 `www.ops.rsu.ru/about.shtml` (Дата обращения 27.01.2017)

4. R. Allen, K. Kennedy Optimizing compilers for Modern Architectures // Morgan Kaufmann Publisher, Academic Press, USA, 2002, 790 p

5. Feautrier P. Dataflow analysis of scalar and array references // International Journal of Parallel Programming. V. 20(1). February 1991, P. 23-52

6. Muchnick Steven S. Advanced Compiler-Design and Implementation. Morgan Kaufmann Publishers, An Imprint of Elsevier. 1997. – 860 p.

Анализ программного кода в объектных файлах Delphi, скомпилированных под платформу .NET

Михайлов А. А.[1], mikhailov@icc.ru
Хмельнов А. Е.[1], hmelnov@icc.ru
[1]Федеральное государственное бюджетное учреждение науки
Институт динамики систем и теории управления имени В. М. Матросова

Аннотация

В работе рассматриваются некоторые аспекты анализа программного кода объектных файлов Delphi. Данный формат является закрытым и может содержать в себе программный код разного уровня абстракции от «инлайн» байт-кода до машинных команд x86. В работе описаны основные особенности формата и виды программного кода, встречающегося в этих файлах. Рассмотрены детали реализации декомпилятора объектных файлов Delphi, скомпилированных под платформу .NET.

Файлы DCU (delphi compiled unit file) технически можно отнести к объектным файлам, поскольку в дальнейшем с использованием редактора связей из них собирается загрузочный модуль. С другой стороны, файлы DCU содержат больше сведений, чем типичные объектные файлы: там кодируется вся информация, полученная компилятором из исходных текстов модуля, и, в том числе, информация об определенных в этом модуле типах данных. Файл DCU может полностью заменить исходный текст для той версии компилятора, при помощи которой он был создан. Этой особенностью активно пользуются разработчики программных модулей, которые часто распространяют их

в формате DCU без предоставления исходных текстов, в особенности тогда, когда это делается на коммерческой основе. В том случае, когда разработчик прекращает развитие своих программных модулей, отсутствие исходных текстов не позволяет применить эти модули с новыми версиями компилятора. Также становится невозможным: исправить обнаруженные ошибки, проанализировать качество кода модуля, не говоря уже о его доработке. Таким образом, задача исследования и, в идеале, декомпиляции файлов DCU часто становится очень актуальной для тех разработчиков, которые используют файлы DCU без исходных текстов.

Следует отметить что, Delphi продолжает развиваться, и в настоящее время поддерживает компиляцию для наиболее распространенных платформ: Windows, OS X, iOS, Android. Кроме того, в ряде версий (8.0 – 2006) выполнялась компиляция для .NET. В таблице 1 представлены виды байт-кода встречающегося в объектных файлах Delphi и достигнутый к настоящему времени уровень его анализа.

Таблица 1: Достигнутый уровень декомпиляции

Платформа	**Код**	**Версия**	**№**	**Уровень**
Win 32	x86	2.0 – 7.0, 2005 –	2 – 7,9 –	Дизассемблер
Win 64	x64	XE2 –	16 –	Дизассемблер
OS X,32	x86	XE2 –	16 –	Дизассемблер
iOS,Simulator	x86	XE4 –	18 –	Дизассемблер
iOS,Device	ARM 32	XE4 –	18 –	Нет
iOS,Device 64	ARM 64	XE8 –	22 –	Нет
Android	ARM 32	XE5 –	19 –	Нет
.NET	CIL	8.0 – 2006	8 – 10	Декомпилятор
*	Inline	2005 –	9 –	Декомпилятор

Данная работа посвящена декомпиляции объектных файлов Delphi, полученных с помощью компиляторов версии 8.0 – 2006 (таблица 1). Несмотря на то, что в текущей версии Delphi платформа .NET не поддерживается, для нас она представляет значительный интерес: поскольку существуют работоспособные декомпиляторы в C# для загрузочных модулей .NET (например, ILSpy [1], NetRefclector [2]). Существование таких декомпиляторов объясняется достаточно высокоуровневым характером инструкций байт-кода CIL и использованием исключительно стека для представления результатов промежуточных вычислений (что существенно облегчает задачу восстановления выра-

жений). По аналогии с этими декомпиляторами нами был разработан декомпилятор DCUIL2PAS (рис. 1) для файлов скомпилированных под платформу .NET, особенности реализации и результаты тестирования которого будут рассмотрены более подробно в докладе.

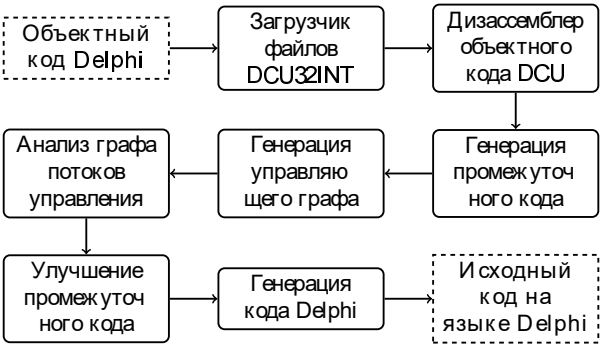

Рис. 1: Схема декомпилятора DCUIL2PAS.

Список литературы

[1] ILSpy .NET Decompiler. // http://ilspy.net

[2] .NET Reflector // http://www.red-gate.com/products/dotnet-development/reflector

Драйверы для обеспечения взаимодействия ускорителя с реконфигурируемой архитектурой и центрального процессора вычислительной системы

Юрий Михайлуц Владислав Яковлев
Руслан Ибрагимов Данил Каримов
Институт математики, механики и компьютерных наук
им. И. И. Воровича, Южный Федеральный Университет

Аннотация

Описание драйвера, обеспечивающего связь универсального вычислителя и специализированного конвейерного ускорителя на ПЛИС.
Ключевые слова: Драйвер, ПЛИС, FPGA, ускоритель.

1 Введение

Данная работа выполнена в рамках проекта разработки компилятора с высокоуровневого языка на вычислительную систему с реконфигурируемой архитектурой [1]. Данный компилятор представляет собой компилятор типа “source-to-source”, но он может генерировать исполняемые бинарные файлы, при помощи сторонних компиляторов для целевого ЦПУ и ПЛИС. Реализован данный компилятор на базе Оптимизирующей Распараллеливающей Системы (ОРС) [2].

Основной целью проекта является разработка такого компилятора с высокоуровневого языка, например, Си, который позволит получать

готовые исполняемые файлы, способные выполняться на вычислительной системе с ускорителем, имеющим реконфигурируемую архитектуру. При этом Компилятор должен поддерживать достаточно большое множество различных вариантов построения данной вычислительной системой (ВС). Близкие проекты [3, 4].

2 Компилятор для ВС с ускорителем, имеющим реконфигурируемую архитектуру

Вычислительная система (ВС) с реконфигурируемым ускорителем представляет собой компьютер на базе универсального процессора, который посредством некоторого интерфейса передачи данных взаимодействует с ускорителем вычислений, имеющим реконфигурируемую архитектуру. Как правило, в роли такого ускорителя выступает программируемая интегральная вычислительная схема (ПЛИС). В такой ВС основная программа работает на универсальном процессоре, но некоторые вычислительно интенсивные подзадачи вынесены в реконфигурируемый ускоритель.

Кроме основной программы на универсальном процессоре и вычислительных IP-ядер данной вычислительной системе требуется подсистема, обеспечивающая взаимодействие частей программы, реализованных на разных архитектурах. Такая подсистема называется драйвером.

3 Задачи драйвера

В задачи драйвера входит:

1. Управление IP-ядрами на ПЛИС.
2. Преобразование передаваемых на ПЛИС данных из формата, в котором они хранятся в программе, в формат, удобный для передачи по выбранному интерфейсу.
3. Распаковка протокола передачи
4. Буферизация отправляемых и получаемых данных.
5. Оптимизация порядка передачи данных, с целью уменьшения задержек в работе IP-ядер.

6. Синхронизацию потоков данных.

4 Разделение драйвера на статическую и генерируемую части

Функционал драйвера, реализованный в виде библиотеки, представляет собой, с одной стороны, процессорные функции для работы с выбранным интерфейсом передачи данных, а с другой стороны, IP-ядра, для работы с выбранным интерфейсом передачи данных на ПЛИС.

Функционал драйвера, который не может быть помещен в библиотеку, должен генерироваться в виде функции. Драйвер совместим с вышеописанными функциями передачи данных между ПЛИС и ЦПУ.

5 Особенности драйверов на разных целевых архитектурах

Тестирование драйвера проводилось как на однокристальных системах, так и на системах с отдельной ПЛИС. Для тестирования ВС с отдельной ПЛИС был реализован тестовый стенд на базе ПЛИС Xilinx Virtex 4 и ПК на базе процессора Intel Core 2 Duo. ПК и ПЛИС были объединены посредством Ethernet. В качестве протокола передачи данных использовался стек протоколов UDP/IP/MAC. Тесты производительности показали, что прирост скорости возможен при вычислительной интенсивности от 100 операций на каждый байт данных.

В составе однокристальной вычислительной системы с реконфигурируемым конвейерным ускорителем и Soft-процессором MIPS используется реализация драйвера на языке C, обеспечивающая пополнение буферов конвейера при помощи шины AHB, с использованием технологии "Memory mapping что дает значительный прирост скорости, ограниченный в основном шириной шины (32 бита).

Список литературы

[1] Boris Ya. Steinberg, Denis V. Dubrov, Yury V. Mikhailuts, Alexander S. Roshal, Roman B. Steinberg "Automatic High-Level Programs Mapping onto Programmable Architectures. Proceedings of the 13th

International Conference on Parallel Computing Technologies, August 31 – September 4, 2015, Petrozavodsk, Russia"V 9251. p. 474-485. Springer Verlag

[2] Оптимизирующая распараллеливающая система www.ops.rsu.ru (дата обращения 08.02.2017)

[3] Bambu. http://panda.dei.polimi.it/?page_id=31

[4] Nios II C2H Compiler. User Guide. https://www.altera.com/en_US/pdfs/literature/ug/ug_nios2_c2h_compiler.pdf

Проблемы реализации синтаксически сахарных конструкций в компиляторах

Михалкович С. С.[1], miks@sfedu.ru
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

Один из способов расширения языка программирования — метод синтаксического сахара: новые конструкции описываются на синтаксическом уровне через конструкции базового языка. Основная проблема здесь — необходимость в нужный момент проводить семантические проверки. В статье рассматриваются современные исследования в этой области и приводятся несколько паттернов реализации для компилятора PascalABC.NET, а также приводятся рекомендации для реализации синтаксического сахара в других компиляторах.
Ключевые слова: компилятор, расширение языка программирования, синтаксический сахар

Введение. Естественный путь развития языка программирования — его расширение новыми конструкциями. В большинстве случаев новые конструкции выражаются через конструкции базового языка, поэтому расширения можно трактовать как синтаксический сахар и на специальном проходе компилятора очищать полученное AST-дерево от сахарных конструкций. Однако для полноценной проверки ошибок необходимы семантические проверки, которые желательно проводить в терминах расширенного языка.

Наиболее известные исследования в этой области связаны с системой SugarJ и ее производными (Sugar*, SoundX) [1]–[3]. Несмотря на гибкость указанных систем, они являются внешними инструментами,

не затрагивающими ядро компилятора, что с одной стороны обеспечивает гибкость, с другой — приводит к проблемам с производительностью и затрудняет поддержку. С другой стороны, ряд промышленных компиляторов (например, Roslyn [4]) активно используют метод синтаксического сахара, интегрированный в компилятор, однако описание используемых подходов практически не встречается в научной литературе.

Основная задача настоящей работы — описать ряд паттернов проектирования, позволяющих встраивать синтаксически сахарные конструкции в действующий компилятор. Для реализации предложенных идей выбран реализуемый под руководством автора свободно распространяемый компилятор PascalABC.NET.

Реализация синтаксического сахара. Важными особенностями архитектуры PascalABC.NET являются разделение внутреннего представления на синтаксическое и семантическое дерево, автоматически генерируемые классы визиторов для обхода деревьев и автоматически генерируемые классы узлов деревьев с рядом сервисных методов.

Реализация синтаксического сахара опирается на построение синтаксического поддерева для соответствующей конструкции. Пусть базовый язык без расширений представлен синтаксическими узлами с типами $t_1, \ldots, t_n$. В процессе расширения языка появляются новые синтаксические узлы с типами $s_1, \ldots, s_k$, которые мы будем называть сахарными. На этапе устранения синтаксического сахара специальный визитор обходит каждый сахарный узел и заменяет его на поддерево, состоящее из узлов базового языка. Пусть узел sug расширенного языка имеет подузлы $n_1, \ldots, n_r$, принадлежащие вообще говоря к расширенному языку. Соответствующий метод visit(sug) выглядит следующим образом:

Сконструировать узел $unsug$ =
$= UnsugType(sug.n_1, \ldots, sug.n_r)$
Заменить в синтаксическом дереве sug на $unsug$
visit($sug.n_1$); ...; visit($sug.n_r$);

Нетрудно видеть, что данный алгоритм посещает все сахарные узлы, уменьшая всякий раз их количество на 1, поэтому он конечен. Основная проблема — необходимость дополнительных семантических проверок, которые невозможно выполнить на синтаксическом уровне. Как правило, это проверки типов для подузлов узла sug. Например, при реализации распаковки кортежей в переменные сахарная конструкция

$(a, b) := t$ нуждается в следующих семантических проверках: t имеет тип кортежа и количество элементов этого кортежа не меньше, чем количество переменных в левой части. Подобные проверки невозможно осуществить на этапе построения синтаксического дерева, поэтому они откладываются до этапа преобразования в семантику.

Предлагается два основных способа реализации подобных проверок. Первый способ предусматривает проведение семантической проверки в визиторе преобразования синтаксического дерева в семантическое непосредственно перед обходом сахарного узла. В данном случае поддерево для сахарной конструкции заменяется на поддерево для несахарной «на лету» и сразу же обходится. Семантические проверки здесь проводятся перед генерацией поддеревьев, что максимально просто: семантические типы подузлов на данном этапе уже известны или могут быть легко получены. Сложность данного способа заключается в том, что обычно помимо обхода построенного «на лету» несахарного поддерева необходимо заменить им в основном дереве соответствующую сахарную конструкцию. Подобные замены в процессе обхода требуют особой аккуратности: возникающие при неверной замене ошибки труднонаходимы. Кроме того, некоторые конструкции базового языка требуют несколько обходов синтаксического дерева, что также необходимо учитывать в данном способе.

Второй способ устранения синтаксического сахара заключается в преобразовании синтаксического дерева на раннем этапе: до построения семантического. Здесь специальный проход компилятора переводит синтаксическое дерево с сахарными узлами в синтаксическое дерево без сахарных узлов, вставляя дополнительные проверочные узлы, которые будут вызывать проверочные действия на этапе преобразования в семантику. Показано, что для сахарных конструкций уровня оператора и уровня выражения такие дополнительные узлы реализуются по-разному. Основное достоинство данного способа — выделение desugaringa в отдельный этап, что облегчает отладку. Основной недостаток сказывается в ситуации, когда надо генерировать разный несахарный код в зависимости от семантических типов подузлов сахарного узла. В этом случае на синтаксическом уровне предлагается использовать визиторы для накопления легковесной семантики по синтаксическому дереву. Такие визиторы могут устанавливать некоторые простейшие характеристики имён: например, определять, является ли данное имя локальной переменной. Недостаток данного способа состоит в маломощности алгоритмов легковесной семантики.

По-существу, водораздел между способами 1 и 2 реализации синтаксического сахара проходит именно по алгоритмам легковесной семантики: если устранение синтаксического сахара проходит с использованием простейших алгоритмов легковесной семантики, то используется способ 2, если же эти алгоритмы сложны или невозможны, то используется способ 1 с бóльшими техническими сложностями в процессе изменения синтаксического дерева в процессе обхода и учётом многократных обходов поддеревьев.

Заключение. В работе разработаны два способа реализации синтаксически сахарных конструкций, предназначенные для интеграции в существующий компилятор. К внутренним представлениям компилятора предъявляются минимальные требования: разделение на синтаксическое и семантическое деревья и сервисные классы и методы для обхода деревьев и их изменения. На конкретных примерах показана применимость каждого способа и их ограничения.

Основное преимущество перед работами [1]–[3] состоит в том, что предложенный в настоящей работе подход реализован интеграцией в компилятор, а не внешней утилитой с повторным анализом семантики и проблемами при изменении базового языка.

Список литературы

[1] S. Erdweg, T. Rendel, C. Kästner, K. Ostermann. SugarJ: library-based syntactic language extensibility // Proceedings of the 2011 ACM International Conference on Object-oriented Programming, Systems, Languages, and Applications. ACM, 2011. Pages 391–406.

[2] S. Erdweg and F. Rieger. A Framework for Extensible Languages // In Proceedings of Conference on Generative Programming: Concepts & Experiences (GPCE). ACM, 2013. Pages 3–12.

[3] Florian Lorenzen and Sebastian Erdweg. Sound Type-Dependent Syntactic Language Extension // Proceedings of Symposium on Principles of Programming Languages (POPL). ACM, 2016. Pages 204–216.

[4] The .NET Compiler Platform "Roslyn". URL: https://github.com/dotnet/roslyn (дата обращения: 01.02.2017).

Независимая от компилятора библиотека точной сборки мусора для языка C++

Евгений Моисеенко, evg.moiseenko94@gmail.com
Даниил Березун, danya.berezun@gmail.com
Санкт-Петербургский государственный университет
Математико-механический факультет

Аннотация

В данной работе представлен подход к сборке мусора для языка C++, основанный на использовании умных указателей. Реализованный алгоритм сборки мусора является точным и не требует поддержки со стороны компилятора, может работать с многопоточными приложениями, а также поддерживает сжатие и параллельную маркировку (*concurrent marking*). Представленный в работе подход позволяет совмещать использование трассирующего сборщика мусора с другими методами управления памятью в C++, в том числе с ручным управлением памятью и методами основанными на использовании умных указателей `std::unique_ptr` и `std::shared_ptr`.

Ключевые слова: C++, динамическое управление памятью, сборка мусора.

Большинство современных языков программирования активно использует *динамическое распределение памяти*, при котором выделение памяти осуществляется во время исполнения программы. *Автоматическое управление памятью* избавляет программиста от необходимости вручную освобождать выделенную память, устраняя тем самым целый класс возможных ошибок и увеличивая безопасность исходного кода программы.

Язык C++ разрабатывался с расчетом на использование ручного управления памятью[1]. Тем не менее, в последние годы прослеживается тенденция к добавлению в язык средств автоматического управления памятью. Так, в стандартную библиотеку C++11 были добавлены

умные указатели `std::unique_ptr` и `std::shared_ptr`[2]. Первый реализует управление памятью на основе *уникального владения ресурсом*, а второй использует *подсчёт ссылок*[3; 4].

Оба подхода имеют свои ограничения и недостатки. Например, `std::shared_ptr` не может быть использован в структурах данных с циклическими ссылками. Для разрешения циклических зависимостей предлагается использовать класс *невладеющего* указателя — `std::weak_ptr`. То есть, ответственность за разрешение циклических зависимостей перекладывается на программиста.

Таким образом, наличие трассирующей сборки мусора в C++ могло бы стать полезным инструментом в случае, когда другие методы автоматического управления памятью не могут быть использованы в силу их ограничений.

В нашей работе представлена реализация *трассирующей сборки мусора* для языка C++ на уровне библиотеки. Пользователю библиотеки предоставляется набор примитивов, в том числе класс умного указателя `gc_ptr` для хранения указателей на управляемые объекты и функция `gc_new` для создания управляемых объектов в куче.

Насколько нам известно, наше решение является первым полностью точным трассирующим сборщиком мусора для C++ на уровне библиотеки без поддержки компилятора. Для решения проблемы *утечки умных указателей*[4] мы добавили в библиотеку новый примитив `gc_pin`, который служит для отслеживания разыменованных указателей и закрепления объектов, на которые они указывают (под закреплением подразумевается запрет на перемещение объекта сборщиком мусора). При каждом явном или неявном разыменовании `gc_ptr` создается объект `gc_pin`.

Для хранения управляемых объектов используется собственная реализация кучи. Поддерживается *сжатие* кучи для уменьшения *фрагментации* памяти. Наш сборщик выполняет *остановку мира* (т.е. приостановку приложения) для сборки мусора. Также поддерживается *параллельная маркировка* (*concurrent marking*). Пользователь библиотеки может активировать эту опцию, в этом случае фаза маркировки проходит параллельно с работой приложения, без необходимости его приостановки. Параллельная маркировка уменьшает время паузы на сборку мусора, однако дополнительные накладные расходы на синхронизации приложения и сборщика могут увеличить суммарное время работы.

Предложенный в работе подход не лишен недостатков. Вследствие

отказа от какого-либо взаимодействия с компилятором, сборщику мусора приходится поддерживать множество структур данных и выполнять множество проверок во время исполнения программы, что приводит к увеличению времени работы приложения. Стоит отметить, что производительность кода, не взаимодействующего со сборщиком, не изменится, что вполне соответствует одному из принципов C++ — "don't pay for what you don't use". Кроме того, сборщик мусора не может разрешить циклические зависимости между областями памяти, находящимися под управлением различных менеджеров памяти.

Таким образом, в нашей работе было показано, что использование богатых языковых возможностей C++ позволяет реализовать сборку мусора без поддержки со стороны компилятора или расширения языка.

Список литературы

1. *Ellis M. A.*, *Stroustrup B.* The annotated C++ reference manual. — Addison-Wesley, 1990.
2. *ISO.* ISO/IEC 14882:2011 Information technology — Programming languages — C++. — Geneva, Switzerland : International Organization for Standardization, 02.2012. — 1338 (est.) — URL: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`.
3. *Jones R.*, *Hosking A.*, *Moss E.* The Garbage Collection Handbook: The Art of Automatic Memory Management. — 1st. — Chapman & Hall/CRC, 2011. — ISBN 1420082795, 9781420082791.
4. *Jones R.*, *Lins R. D.* Garbage Collection: Algorithms for Automatic Dynamic Memory Management. — Wiley, 1996. — ISBN 0471941484.

Анализ эффективности векторизующих компиляторов на архитектурах Intel 64 и Intel Xeon Phi*

Молдованова О. В. Курносов М. Г.
Сибирский государственный университет телекоммуникаций и информатики

Аннотация

В работе выполнен экспериментальный анализ возможностей современных оптимизирующих компиляторов Intel C/C++ Compiler, GCC C/C++, LLVM/Clang и PGI C/C++ по автоматической векторизации циклов на архитектурах Intel 64 и Intel Xeon Phi. В качестве целевого набора тестовых циклов использован Extended Test Suite for Vectorizing Compilers. В ходе работы установлены ускорения для различных типов данных и определены классы циклов, автоматическая векторизация которых указанными компиляторами затруднена. Экспериментальная часть работы выполнена на двухпроцессорном NUMA-сервере (2 x Intel Xeon E5-2620 v4, микроархитектура Intel Broadwell) с сопроцессором Intel Xeon Phi 3120A.

В работе представлены результаты анализа эффективности подсистем автоматической векторизации циклов в современных компиляторах Intel C/C++ Compiler, GCC C/C++, LLVM/Clang, PGI C/C++ на архитектурах Intel 64 и Intel Xeon Phi. Учитывая отсутствие информации о реализуемых коммерческими компиляторами методах векторизации, анализ выполнен методом «черного ящика» – на тестовом наборе циклов из пакета Extended Test Suite for Vectorizing Compilers

*Работа выполнена при поддержке РФФИ (проекты 16-07-00992, 15–07–00653).

[1, 2, 3, 4]. Глобальные массивы в пакете ETSVC были выравнены на границу 32 байта для процессора Intel Xeon и 64 байта – для Intel Xeon Phi. Эксперименты выполнены для массивов с элементами типов `double`, `float`, `int` и `short`.

Исследования проводились на двух системах. Первая система – сервер на базе двух процессоров Intel Xeon E5-2620 v4 (архитектура Intel 64, микроархитектура Broadwell, 8 ядер, Hyper-Threading включен, поддержка набора векторных инструкций AVX 2.0), 64 Гбайта оперативной памяти DDR4, операционная система GNU/Linux CentOS 7.3 x86-64 (ядро linux 3.10.0-514.2.2.el7). Вторая система – установленный в сервер сопроцессор Intel Xeon Phi 3120A (микроархитектура Knights Corner, 57 ядер, поддержка AVX-512), 6 Гбайт оперативной памяти, MPSS 3.8.

Установлено, что рассматриваемые компиляторы способны векторизовать от 39 % до 77 % циклов от общего числа в пакете ETSVC. Наилучшие результаты показал Intel C/C++ Compiler, а наихудшие – LLVM/Clang.

Наиболее проблемными оказались циклы, содержащие условные и безусловные переходы, вызовы функций, индуктивные переменные, переменные в границах цикла и шаге выполнения итераций, а также такие идиоматические конструкции, как рекуррентности 1-го или 2-го порядка, поиск первого подходящего элемента в массиве и свертка цикла.

В случаях, когда компиляторы не могут принять решение об эффективности или возможности векторизации цикла (например, при наличии в цикле нескольких переменных, которые могут ссылаться на одну и ту же область памяти), они генерируют скалярную и одну или несколько векторизованных версий цикла (многоверсионный код, multiversioning). Решение о том, какая версия будет использоваться, принимается во время выполнения программы. Как показали результаты экспериментов, чаще всего в этом случае решение принимается в пользу скалярной версии цикла. Возможное решение – это аннотирование кода специальными директивами, указывающими на отсутствие зависимостей по данным, отсутствие множественных ссылок на одну область памяти.

Направление дальнейших работ – анализ и разработка методов эффективной векторизации установленного класса проблемных циклов, анализ возможностей применения JIT-компиляции [5] и оптимизации по результатам профилирования (profile-guided optimization).

Список литературы

[1] Maleki S., Gao Ya. Garzarán M.J., Wong T., Padua D.A. An Evaluation of Vectorizing Compilers // Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11), 2011. IEEE Computer Society, pp. 372–382.

[2] Extended Test Suite for Vectorizing Compilers // URL: `http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz`.

[3] Callahan D., Dongarra J., Levine D. Vectorizing Compilers: A Test Suite and Results // Proceedings of the ACM/IEEE conference on Supercomputing (Supercomputing'88), 1988. IEEE Computer Society Press, pp. 98–105.

[4] Levine D., Callahan D., Dongarra J. A Comparative Study of Automatic Vectorizing Compilers // Journal of Parallel Computing. 1991. Vol. 17. pp. 1223–1244.

[5] Rohou E., Williams K., Yuste D. Vectorization Technology To Improve Interpreter Performance // ACM Transactions on Architecture and Code Optimization. 2013. 9 (4). pp. 26:1-26:22.

Перенос вычислений на акселераторы NVIDIA в компиляторе GCC

Монаков А. В., amonakov@ispras.ru
Иванишин В. А., vlad@ispras.ru
Кудряшов Е. А., kudryashov@ispras.ru
Институт системного программирования РАН

Аннотация

Начиная с версии 4.0, в наборе языковых расширений OpenMP появилась возможность переноса части вычислений на отдельные устройства-акселераторы. В докладе описывается реализация OpenMP для NVIDIA-акселераторов в компиляторе GCC, опирающаяся на существующую инфраструктуру: генерацию PTX-кода и библиотеку поддержки времени выполнения libgomp. Демонстрируются использованные методы трансляции OpenMP SIMD-регионов для акселераторов, использующих SIMT-параллелизм.

Ключевые слова: компиляторы, GCC, OpenMP, GPU.

Стандарт поддержки параллельного программирования OpenMP [3] представляет собой набор языковых расширений в виде прагм для языков C, C++, Fortran, которые позволяют запись параллельных алгоритмов на общей памяти. Использование прагм открывает возможность поддержки как последовательной, так и параллельной версии алгоритма в рамках единого исходного кода.

В компиляторе GCC поддержка OpenMP выполнена в составе трех компонент: в языковых фронт-эндах (для распознавания и трансляции прагм), в нескольких начальных фазах трансляции (для понижения высокоуровневого представления OpenMP-конструкций) и в библиотеке поддержки libgomp, в которой реализованы как пользовательские

функции OpenMP API, так и специальные функции, обеспечивающие поддержку OpenMP-конструкций [1].

Начиная с версии 4.0, стандарт OpenMP добавил ряд расширений, предназначенных для обеспечения возможности использования специализированных устройств-акселераторов, имеющих архитектуру, оптимизированную под массивно-параллельные вычисления, но потенциально работающие в отдельном пространстве памяти. Соответственно, новые прагмы в OpenMP были необходимы для указания региона кода, который может выполняться на акселераторе (прагмы target и declare target), для указания данных, которые должны быть выделены в памяти акселератора, и для указания данных, которые необходимо копировать в память акселератора или из неё. Кроме того, была расширена модель параллельного выполнения, чтобы позволить запуск множества независимых групп нитей (прагма teams).

В GCC вынос OpenMP-кода на акселераторы был сначала реализован для акселераторов Intel MIC: они имеют x86-совместимую архитектуру и поддерживают интерфейсы Linux, что позволило полностью переиспользовать существующую поддержку в компиляторе и libgomp. Кроме того, в GCC была реализована поддержка OpenACC (акселераторные расширения OpenMP во многом близки к OpenACC) для акселераторов NVIDIA. При этом в GCC была реализована генерация GPU-кода (в виде высокоуровневого ассемблера PTX [2]).

В Институте системного программирования разрабатывалась поддержка выноса OpenMP-кода для NVIDIA-акселераторов [4]. При этом существующая компиляторная поддержка: генерация PTX-кода и загружаемый модуль для управления акселератором через драйверный интерфейс CUDA для libgomp — была переиспользована, но трансляция OpenMP-конструкций требовала поиска новых подходов.

В связи с большим разнообразием OpenMP-конструкций было принято решение переиспользовать стратегии кодогенерации для параллелизма на уровне нитей и команд (teams) и, таким образом, также переиспользовать часть нетривиальной логики времени выполнения, реализованной в libgomp. Для этого было выполнено портирование libgomp на архитектуру PTX. Основные изменения в портировании произошли в логике управления нитями (для процессорных архитектур libgomp запускает нити динамически через интерфейс Pthreads, на PTX же акселераторый код сразу выполняется как иерархия групп нитей) и примитивах синхронизации (на Linux libgomp использует futex-операции для эффективного ожидания, на GPU в большинстве слу-

чаев можно использовать только активное ожидание, но для части удалось использовать эффективную синхронизацию через инструкцию bar.sync).

Аналогично OpenACC, GCC в OpenMP отображает команды OpenMP-нитей на CUDA-блоки, а логические OpenMP-нити на синхронные группы (warps), состоящие из 32 контекстов выполнения на GPU. Это связано с тем, что при выполнении для всех нитей в одной синхронной группе выдается одна и та же инструкция; PTX-нити не являются полностью независимыми, и их выполнение близко к векторному параллелизму (NVIDIA использует термин SIMT — single instruction, multiple threads — по аналогии с SIMD). Отдельные же нити в рамках синхронной группы в OpenMP могут выполнять разные вычисления только в рамках SIMD-региона.

Для отображения логических OpenMP-нитей на синхронные группы было реализовано два новых подхода к трансляции кода. Во-первых, для хранения стековых переменных были организованы управляемые GCC стеки (в PTX явного указателя на стек нет), которые вне SIMD-регионов могут располагаться в глобальной памяти акселератора и быть общими для синхронной группы, а при входе в SIMD-регион переключаться на регионы в локальной памяти. Во-вторых, для поддержки всех 32 нитей активными до входа в SIMD-регион был реализован режим выполнения (uniform-simt), в котором наблюдаемые эффекты происходят так, как если бы был активен только один поток из 32: для этого достаточно немного дополнить трансляцию атомарных инструкций.

Наконец, для SIMD-регионов в компиляторе были реализованы новые стратегии понижения кода, ориентированные на использование SIMT-параллелизма. Для этого компилятор трансформирует пространство итераций исходных циклов чтобы PTX-нити выполняли каждую 32-ю итерацию. Далее необходимо обеспечить корректную трансляцию приватных данных (в частности, клауз reduction и lastprivate) и обеспечить, чтобы адресуемые приватные данные, включая добавленные в SIMD-регион поздно в результате инлайн-подстановки, не попадали на общие для синхронной группы стеки.

Реализация предложенных подходов была принята в основную ветвь разработки GCC и будет доступна в версиях 7.x.

Список литературы

1. GNU libgomp. — 2017. — URL: https://gcc.gnu.org/onlinedocs/libgomp/ (дата обр. 07.02.2017).
2. *NVIDIA Corporation*. Parallel Thread Execution ISA Version 5.0. — 2017. — URL: https://docs.nvidia.com/cuda/parallel-thread-execution/ (дата обр. 07.02.2017).
3. *OpenMP Architecture Review Board*. OpenMP Application Program Interface Version 4.0. — 07.2013. — URL: http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf (дата обр. 07.02.2017).
4. *Монаков А. В.*, *Иванишин В. А.* Поддержка стандарта OpenMP 4.0 для архитектуры NVIDIA PTX в компиляторе GCC // Труды Института системного программирования РАН. — 2016. — Т. 28, № 4. — С. 169—182.

Обещающая компиляция в ARMv8

Антон Подкопаев[1] Ори Лахав[2]
Виктор Вафеядис[2]
[1]СПбГУ, JetBrains, Россия
[2]Институт Макса Планка: Программные Системы, Германия

Аннотация

Поведение многопоточных программ в современных окружениях описываются "слабыми" моделями памяти. При этом модели процессоров и языков высокого уровня часто существенно разнятся, что делает их формальное сопоставление сложным. В работе представлено доказательство корректности компиляции подмножества "обещающей" семантики Kang et al. в модель памяти ARMv8 Flur et al.
Ключевые слова: многопоточность, корректность компиляции, слабые модели памяти.

Семантики языков программирования являются достаточно широко изученной областью с многолетней историей. Одной из важных открытых проблем в области является описание семантики многопоточных программ над общей памятью. Так широко известно, что наивный подход [4], который предполагает *последовательную консистентность* (sequential consistency, SC), не способен описать поведения оптимизированных программ на современных архитектурах.

Поведение программы на уровне машинного кода полностью зависит от целевой процессорной архитектуры. К сожалению, поставщики и разработчики наиболее популярных архитектур (x86, Power, ARM) предоставляют спецификации, которые лишь в общих чертах описывают многопоточное исполнение программ. В последние годы научным

сообществом были разработаны формальные модели для эти архитектур (x86-TSO [7], Power [2; 6], ARMv8 [5]), как результат широкой дискуссии с разработчиками архитектур и тестирования существующих процессоров.

Эта работа рассматривает формальную модель архитектуры ARMv8 [5]. Данная модель является одной из самых новых и продвинутых. Она моделирует широкий класс низкоуровневых особенностей архитектуры, влияющих на многопоточное исполнение программ. Среди этих особенностей: топология кэш-памяти процессора, переупорядочивание обращений к памяти, внеочередное исполнение инструкций, предсказание переходов и т.д. При этом модель ARMv8 существенно *слабее*, т.е. допускает больше поведений для одних и тех же программ, чем упомянутые выше модели. Например, абстрактная машина ARMv8 позволяет первому потоку прочитать 1 из x в следующей программе (подробнее см. [3]):

$$\begin{array}{l} a := [x]; \\ [x] := 1; \end{array} \quad //1 \quad \Big\| \quad \begin{array}{l} b := [x]; \\ [y] := b; \end{array} \quad \Big\| \quad \begin{array}{l} c := [y]; \\ [x] := c; \end{array} \qquad \text{(ARM-weak)}$$

где переменные изначально записаны 0. Кратко, такое поведение достигается следующим образом: первый поток отправляет запись $[x] := 1$ второму потоку, после чего второй поток отправляет запись $[y] := 1$ третьему потоку, а третий поток, произведя чтение из этого сообщения, отправляет запись $[x] := 1$ в основную память, делая её видимой для первого потока. После чего первый поток может произвести чтение $a := [x]$, получив $a = 1$.

В отличие от моделей процессоров, семантики языков программирования должны балансировать между поддержкой эффективной компиляции в широкий набор архитектур и компиляторных оптимизаций с одной стороны, и предоставлением высокоуровневых гарантий для программиста с другой стороны. В работе [1] предложена абстрактная машина, которая, как утверждается авторами, удовлетворяет этим требованиям. Эта модель включает одно необычное правило — в момент исполнения поток может "пообещать" сделать определенную запись в память. После этого обещание становится доступным для чтения другими потоками, но не для самого обещающего потока. Через некоторое время поток обязан выполнить своё обещание. Этот механизм позволяет получить $a = 1$ в рассмотренной программе следующим образом: первый поток обещает $[x] := 1$, вторый и третий потоки читают из этой записи и производят записи $[y] := 1$ и $[x] := 1$

соответственно, после чего первый поток читает из последней, получая $a = 1$, и выполняет обещание $[x] := 1$.

Для обещающей модели не было показано корректности компиляции в модель ARMv8 — в работе [1] корректность компиляции показана только для гораздо более простых моделей x86-TSO и Power. К сожалению, данные доказательства существенно полагаются на результат [3], не применимый для модели ARMv8 (подробнее см. [3]).

Эта работа посвящена доказательству корректности компиляции подмножества обещающей модели, состоящего из расслабленных записей и чтений (relaxed accesses), высвобождающих (release) и приобретающих (acquire) барьеров памяти, в модель ARMv8. У данной задачи есть две основных сложности. Во-первых, обещающая и ARMv8-модели очень сильно отличаются друг от друга. В модели ARMv8 инструкции могут выполняться не по порядку и за несколько шагов. В обещающей модели все операции выполняются за один шаг и, за исключением самих обещаний, в естественном программном порядке. Во-вторых, несмотря на то, что обе модели — операционные, корректность компиляции не может быть показана помощью стандартной техники прямой симуляции. Причина заключается в том, что в модели ARMv8 записи в ячейку памяти могут не быть упорядочены во время исполнения программы, но упорядочены в конце исполнения. В то время как обещающая модель упорядочивает записи в одну ячейку памяти сразу же, как записи исполняются соответствующими потоками. Так симуляционное доказательство должно “угадывать” правильный порядок между записями.

Для решения данной проблемы мы вводим инструментированную версию ARM-машины, которая поддерживает порядок на записях в одну ячейку в течение всего исполнения программы. Для инструментированной машины мы доказываем, что она обладает тем же пространством поведений, что и изначальная машина. На базе этого результата мы доказываем корректность компиляции из обещающей машины в модель ARMv8.

Вторым результатом нашей работы является формализация модели ARMv8 и доказательство нескольких свойств, которые могут быть полезны в контексте доказательств корректности компиляции из других моделей памяти.

Список литературы

1. A Promising Semantics for Relaxed-Memory Concurrency / J. Kang [и др.] // POPL 2017. — ACM, 2017.
2. *Alglave J.*, *Maranget L.*, *Tautschnig M.* Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory // ACM Trans. Program. Lang. Syst. — 2014. — Т. 36, № 2. — 7:1—7:74.
3. *Lahav O.*, *Vafeiadis V.* Explaining Relaxed Memory Models with Program Transformations // FM 2016. — Springer, 2016.
4. *Lamport L.* How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs // IEEE Trans. Computers. — 1979. — Т. 28, № 9. — С. 690—691.
5. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA / S. Flur [и др.] // POPL 2016. — ACM, 2016. — С. 608—621.
6. Understanding POWER multiprocessors / S. Sarkar [и др.] // PLDI. — ACM, 2011. — С. 175—186.
7. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors / P. Sewell [и др.] // Commun. ACM. — 2010. — Т. 53, № 7. — С. 89—97.

Преобразование по уплотнению кода в LLVM

Скапенко И. Р.[1], skapenko@sfedu.ru
Дубров Д. В.[1], dubrov@sfedu.ru
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

В настоящее время существует множество вычислительных систем с ограниченным количеством памяти. Автоматическое уменьшение размера программного кода может избавить разработчиков программного обеспечения от большого количества рутинной работы и необходимости реализовывать программы на языках низкого уровня. Целью настоящей работы является реализация межпроцедурных оптимизаций уменьшения размера кода [2] на основе библиотеки LLVM, а также изучение их эффективности. В работе рассматривается межпроцедурная оптимизация вынесения одинаковых базовых блоков в отдельную функцию, которая реализуется на уровне промежуточного представления LLVM.
Ключевые слова: LLVM, code compaction, basic block abstraction.

1 Введение

Код, создаваемый компиляторами языков высокого уровня, часто бывает практически неприменим ко многим встраиваемым системам из-за его избыточности. Необходимость уменьшения размера кода в компиляторе LLVM появилась с момента создания кодогенератора для

архитектур ARM, MIPS, AVR и других, которые активно используется во встраиваемых системах. Благодаря грамотной инфраструктуре LLVM, имеется возможность писать оптимизации, не зависящие от языка программирования и конечной архитектуры на уровне промежуточного представления (LLVM IR). Также возможно написание низкоуровневых архитектурно-зависимых оптимизаций на уровне машинных команд (LLVM MIR).

Целью настоящей работы является реализация прохода LLVM, выполняющего уплотнение кода путём слияния одинаковых базовых блоков и вынесения их в отдельные функции (т. н. процедурная абстракция базовых блоков). Исходный код данной оптимизации для LLVM 4.0 выложен авторами в открытый доступ на ресурсе GitHub [1].

2 Процедурная абстракция базовых блоков

В LLVM существуют различные виды проходов, но для задач межпроцедурных оптимизаций на уровне LLVM IR подходит тип прохода `ModulePass`, так как только из данного типа имеется возможность просматривать и изменять содержимое функций во внутреннем представлении. Пример применения оптимизации абстрагирования базовых блоков на уровне LLVM IR с помощью `ModulePass` представлен на рис. 1.

Из-за особенностей промежуточного кода невозможно заменить базовые блоки исходной программы целиком, поскольку Phi-инструкции используют значения других базовых блоков, доступ к которым возможен только внутри функции. Terminator-инструкции обязательно должны находиться в конце корректно сформированных базовых блоков. Поэтому в реализованном преобразовании Phi- и Terminator-инструкции не рассматриваются при работе с базовыми блоками, и при вынесении базовых блоков эти инструкции не заменяются.

3 Алгоритм слияния базовых блоков

Алгоритм можно разделить на 2 основных шага:

1. Сравнение базовых блоков.
2. Слияние базовых блоков.

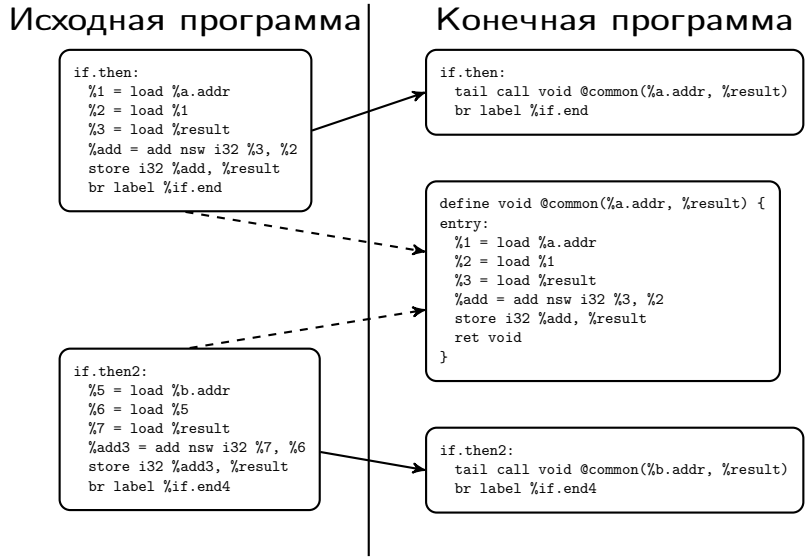

Рис. 1: Пример работы прохода вынесения базовых блоков

Как уже было сказано, при работе с базовыми блоками не учитываются Termination- и Phi-node-инструкции, так как их вынесение в отдельную функцию невозможно.

Большая часть алгоритма сравнения базовых блоков была взята из LLVM-прохода `MergeFunctions` [3]. Одной из модификаций оригинального алгоритма является добавление свойства коммутативности у бинарных операций сложения и умножения. Например, в оригинальной версии инструкции следующего вида считались различными:

```
%mul = mul nsw i32 %k, 3
%mul = mul nsw i32 3, %k
```

Следующей модификацией является исключение некоторых атрибутов при сравнении функций базовых блоков. Исключаются атрибуты, не влияющие на генерацию кода.

Остальные изменения были произведены из-за особенностей сравнения базовых блоков, таких как пропуск Termination- и Phi-инструкций.

Рассмотрим подробнее этап слияния базовых блоков. На данном этапе поочерёдно рассматриваются множества эквивалентных базовых

блоков.

1. Для каждого базового блока находится список его входных и выходных параметров.

2. Количество и типы входных параметров должны совпадать, иначе базовые блоки не эквивалентны. Список выходных аргументов может отличаться, поэтому находится комбинация всех выходных переменных.

3. Проверяется наличие подходящей функции среди множества эквивалентных базовых блоков. Если функция была найдена, то переход к шагу 6.

4. Определяется, произойдёт ли уменьшение размера машинного кода при создании функции и замене базовых блоков на её вызов.

5. Создаётся функция, основанная на одном из базовых блоков множества. Аргументами функции являются входные параметры и указатели на каждый из выходных параметров. При создании функции после каждого создания выходной переменной она помещается в соответствующий выходной аргумент функции. Также устанавливаются атрибуты функции, уменьшающие её конечный размер.

6. Далее происходит замена каждого базового блока на вызов функции с использованием оператора `bitcast` для типобезопасности. Для каждой выходной переменной резервируется место на стеке, а после вызова функции переменные загружаются из стека, если она используется далее.

4 Тестирование преобразования

Для тестирования реализованного преобразования были использованы исходные коды следующих открытых проектов:

- `TinyXML2`: `https://github.com/leethomason/tinyxml2`
- `curl`: `https://github.com/curl/curl`
- `FatFs`: `http://www.elm-chan.org/fsw/ff/00index_e.html`

Каждый из этих проектов был скомпилирован при помощи clang с оптимизацией кода по размеру (ключ `-Oz`) с генерацией результата в LLVM IR (проект `TinyXML2` дополнительно компилировался в версии без поддержки исключений). Далее получаемое внутреннее представление подавалось на вход разработанному преобразованию, после чего обе версии (до и после преобразования) обрабатывались компоновщиком LLVM для получения объектных модулей. Затем сравнивался размер полученных объектных модулей. В качестве целевых архитектур были использованы X86-64 и ARM. Кроме сравнения размеров были реализованы тесты корректности преобразования при помощи анализа LLVM IR.

Данный эксперимент показал, что уменьшение размера кода в результате работы преобразования для всех проектов составляет от 0 до 1 %. Наибольший эффект от преобразования был достигнут на проекте `TinyXML2` с отключённой поддержкой исключений (1 %), наименьший — на `FatFs`, где преобразование оставило код без изменений.

5 Заключение

Из-за неоднородного преобразования промежуточного кода в машинный, одной из главных трудностей алгоритмов оптимизации размера кода на уровне промежуточного представления является определение того, как изменится конечный размер кода (шаг 4 алгоритма). Для определения как можно более точного размера кода необходим доступ к функциональности кодогенератора. В противном случае для получения приемлемого результата нужно производить проходы понижения уровня кода самостоятельно и для разных кодогенераторов использовать разную логику определения размера кода, что является лишней и ненужной работой.

В настоящее время в LLVM не существует хороших средств, определяющих конечный размер кода инструкции/базового блока. Вместо этого LLVM предоставляет доступ к интерфейсу кодогенератора `TargetTransformInfo`. Однако он возвращает величину, являющуюся чем-то средним между размером инструкции и её временем выполнения. Также эта величина является эвристикой и результаты, полученные исключительно с её помощью оказались отрицательными, то есть конечный размер программы увеличивался.

Таким образом, оптимизация размера кода на уровне LLVM IR имеет смысл только для крупных участков кода, таких как базовый блок,

регион, функция.

Список литературы

1. Code compaction. — URL: `https://github.com/skapix/codeCompaction` (дата обр. 08.02.2017).
2. *Debray S. K.*, *Evans W.* Compiler Techniques for Code Compaction // ACM Transactions on Programming Languages and Systems. — 2000. — Март. — С. 378—415. — URL: `http://users.elis.ugent.be/~brdsutte/research/publications/2000TOPLASdebray.pdf` (дата обр. 08.02.2017).
3. MergeFunctions pass. — URL: `http://llvm.org/docs/MergeFunctions.html` (дата обр. 08.02.2017).

Библиотека парсер-комбинаторов для синтаксического анализа графов

Смолина С. К., sov-95@mail.ru
Вербицкая Е. А., kajigor@gmail.com
Санкт-Петербургский государственный
электротехнический университет «ЛЭТИ»
им. В. И. Ульянова (Ленина) (СПбГЭТУ «ЛЭТИ»)

Аннотация

Многие задачи, возникающие в области графовых баз данных и статического анализа динамически формируемых выражений, можно сформулировать как задачу синтаксического анализа графа, то есть задачу нахождения путей в графе, которые описывают цепочки, выводимые в данной грамматике. В докладе будет рассмотрена модификация библиотеки парсер-комбинаторов Meerkat для синтаксического анализа графов.
Ключевые слова: синтаксический анализ графов, парсер-комбинаторы, графовые базы данных.

Графы и графовые базы данных имеют широкое применение во многих областях: в сферах биоинформатики, логистики, социальных сетей и других. Одной из проблем в данной сфере является задача поиска путей в графе, удовлетворяющих некоторым ограничениям. Ограничения часто формулируются некоторой контекстно-свободной грамматикой. В таком случае задача сводится к поиску путей в графе, которые бы соответствовали строкам в контекстно-свободном языке.

Существуют различные подходы к синтаксическому анализу графов (например, [3], [2], [5]). Однако такие подходы не удобны при работе с графовыми базами данных, поскольку усложняется формирование запроса внутри целевой программы. Этот недостаток можно исправить при помощи техники парсер-комбинаторов. К сожалению,

большинство существующих библиотек парсер-комбинаторов анализируют только линейный вход (строки), поэтому их непосредственное использование для решения данной задачи невозможно. В рамках данной работы была поставлена задача разработки библиотеки парсер-комбинаторов для работы с графами. За основу решения была выбрана библиотека парсер-комбинаторов Meerkat[1] [4], реализованная на языке Scala. Данная библиотека осуществляет построение леса разбора Binarized Shared Packed Parse Forest (SPPF) [1] для произвольных (в том числе неоднозначных) контекстно-свободных грамматик.

В библиотеке реализованы четыре базовых комбинатора: terminal, epsilon, seq и rule. Первые два комбинатора представляют собой базовые распознаватели для терминала и пустой строки. Комбинатор seq выполняет последовательную композицию распознавателей. Комбинатор rule используется для задания правила вывода в терминах контекстно-свободных грамматик: нетерминала и соответствующих ему альтернатив. С помощью этих комбинаторов можно задать произвольную контекстно-свободную грамматику.

Библиотека осуществляет поиск всех возможных способов разбора строки. Это удалось достичь за счет использования техники Continuation-Passing Style (CPS) и специальной процедуры мемоизации. Идея программирования в стиле Continuation-Passing состоит в передаче управления через механизм продолжений. Продолжение в данном контексте представляет собой состояние программы в конкретный момент времени, которое возможно сохранить и использовать для перехода в данное состояние. Для реализации данного подхода был создан такой тип данных как Result[T], который представляет собой монаду и реализует следующие три метода: map, flatMap, orElse. Таким образом, любой результат работы распознавателя можно представить как композицию двух функций, используя метод flatMap, или как комбинацию результатов, используя метод orElse. Для избежания экспоненциальной сложности и для обработки лево-рекурсивных нетерминалов используется техника мемоизации, сохраняющая информацию о том, какие продолжения уже вычислялись, а также их результаты. Продолжение для конкретного символа вычисляется один раз и в дальнейшем возвращается лишь результат.

Для решения поставленной задачи потребовалось изменить тип входных данных на граф и преобразовать базовый комбинатор, разбирающий терминал. В отличие от линейного входа при анализе гра-

[1]https://github.com/meerkat-parser/Meerkat

фа позицией во входном потоке является вершина графа, а понятие "следующего символа" заменяется на множество символов, написанных на исходящих из данной вершины ребрах. Синтаксический разбор при этом продолжается по тому пути, который начинается с ребра с соответствующим символом. Продолжения и его результаты сохраняются для конкретной вершины и переиспользуются, при попадании в ту же вершину. Таким образом решается проблема с графами, которые содержат циклы. Для случая, когда из вершины исходят ребра с одинаковыми символами, результат комбинируется с помощию метода orElse. После построения SPPF вычисляются семантика. На данный момент семантика возможно посчитать для случая, когда SPPF является деревом.

В дальнейшем планируется интегрировать библиотеку[2] с промышленной графовой СУБД Neo4J, что позволит использовать данное решение в таких сферах как биоинформатика, логистика, социальные сети.

Список литературы

1. *Afroozeh A.*, *Izmaylova A.* Faster, Practical GLL Parsing // Compiler Construction: 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings / под ред. B. Franke. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2015. — С. 89—108. — ISBN 978-3-662-46663-6. — DOI: 10.1007/978-3-662-46663-6_5. — URL: http://dx.doi.org/10.1007/978-3-662-46663-6_5.
2. *Grigorev S.*, *Ragozina A.* Context-Free Path Querying with Structural Representation of Result // arXiv preprint arXiv:1612.08872. — 2016.
3. *Hellings J.* Conjunctive context-free path queries. — 2014.
4. *Izmaylova A.*, *Afroozeh A.*, *Storm T. v. d.* Practical, General Parser Combinators // Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. — St. Petersburg, FL, USA : ACM, 2016. — С. 1—12. — (PEPM '16). — ISBN 978-1-4503-4097-7. — DOI: 10.1145/2847538.2847539. — URL: http://doi.acm.org/10.1145/2847538.2847539.

[2] https://github.com/sofysmol/Meerkat

5. *Sevon P.*, *Eronen L.* Subgraph queries by context-free grammars // Journal of Integrative Bioinformatics. — 2008. — Т. 5, № 2. — С. 100.

Платформа РуСи для обучения и создания высоконадежных программных систем

Терехов А. Н.[1], a.terekhov@spbu.ru
Терехов М. А.[1], st054464@student.spbu.ru
[1]Санкт-Петербургский государственный университет
Кафедра системного программирования

Аннотация

Проект РуСи преследует две цели – облегчить изучение программирования школьникам и студентам и создать среду для разработки высоконадежного программного обеспечения, максимально защищенного от попыток проникновения вирусов и других атак. Эти цели достигаются путем введения ограничений на входной язык и существенным расширением статического и динамического контроля среды программирования. Хотя ограничений (и расширений тоже!) входного языка довольно много, в основе лежит широко распространенный язык Си, что позволяет надеяться на широкое применение проекта РуСи.

Ключевые слова: язык Си, защищенное программирование, контроль типов, динамический контроль, среды программирования.

Введение

В области создания критически важных программно-аппаратных систем сложилась парадоксальная ситуация: сначала программисты пишут программы на языках, не обеспечивающих достаточный контроль ошибок пользователей, затем они же или специально обученные специалисты долго и трудно работают над поиском и исправлением этих ошибок. Ведущие ученые и специалисты из промышленности поняли эту проблему много лет назад, были придуманы языки и системы

программирования, ориентированные на создание программ с максимально возможным контролем ошибок как периода компиляции (статические), так и периода счета (динамические). Упомянем для примера языки Оберон Никлауса Вирта [1] и Эйфель Бертрана Мейера [2], однако, как и многие другие языки, созданные с той же целью, эти так и не стали массовыми. К сожалению, в программировании так часто бывает, и причины надо искать в областях инженерной психологии и экономики (сколько и кем было вложено средств в продвижение языка). Самым известным примером в ряду таких не принятых обществом программистов языков является Алгол 68 [3] — первый в истории язык высокого уровня с не только точно определенным синтаксисом (сейчас этим никого не удивить), но и семантикой, а это и сегодня для многих языков является проблемой. Многие языковые черты, впервые появившиеся в Алголе 68, например, полный видовой контроль периода компиляции, последовательные предложения и условные выражения, выдающие значение, операторы с присваиванием типа +:=, рекурсивные типы и т.д., затем с успехом были применены в Паскале, Си, Аде и других языках, появившихся позднее.

В данном проекте мы пошли другим путем. Мы не стали выдумывать новый язык, а взяли за основу широко распространенный язык Си, слегка, на наш взгляд, изменив его с целью повышения защищенности от ошибок пользователя, и выполнили собственную его реализацию, причем не только компилятора, но и полноценной среды разработки. Этот проект получил название РуСи.

Первоначально проект возник из потребностей преподавателей школьных кружков робототехники. Уже довольно давно они применяют нашу графическую технологию ТРИК-студия, особенно удачно она подходит для школьников младших и средних классов, но при этом хотелось бы, чтобы школьники могли прочитать те программы, которые генерируются из графических диаграмм, то есть образовать своеобразный “мостик” к урокам информатики. Сейчас во многих странах популярен подход к программированию на основе графических моделей с автоматической генерацией кода на целевом языке. Традиционно для этого применяется язык Си [4], но нельзя забывать, что школьники младших и средних классов практически не знают английского языка. И так-то учиться программировать тяжело, а тут — незнакомые слова, незнакомые сообщения и т.п. Поэтому возникла идея разработать компилятор языка Си в коды выдуманной нами виртуальной машины с русскими сообщениями, ключевыми словами и идентификаторами и

интерпретатор этой машины (но никто не запрещает использовать и английские ключевые слова и идентификаторы). Виртуальная машина обеспечивает легкую переносимость на любые платформы и, хотя работает несколько медленней за счет накладных расходов на дешифрацию кодов, но, если хорошо продумать ее архитектуру, то эти накладные расходы становятся незначительными. Кроме того, появляется хорошая возможность отладки, которая не во всем возможна на реальных ЭВМ без специальной аппаратной поддержки — например, остановка по записи в какую–то ячейку и т.п.

Как компилятор, так и интерпретатор реализованы на стандартном Си, поэтому легко переносятся на все платформы, в частности, интерпретатор перенесен на конструктор роботов ТРИК [5, 6], который разрабатывается сотрудниками и студентами кафедр системного программирования и теоретической кибернетики математико-механического факультета СПбГУ.

Постепенно рамки проекта РуСи расширились: сначала оказалось, что на примере этого проекта удобно проводить занятия со студентами мат-меха по технике трансляции, затем этим проектом заинтересовались военные и другие заказчики, которым была важна надежность создаваемых программных систем, которая в РуСи поддерживалась отказом от адресной арифметики, обязательным контролем индексов в массивах, да и просто подробной сигнализацией об ошибках пользователя. В данном докладе мы сосредоточимся на особенностях нашего варианта языка и реализационных деталях, обеспечивающих существенно более полный контроль ошибок пользователя.

1 Контекст работы

Вообще-то, введение ограничений на входной язык при реализации компилятора — это плохой тон, свидетельство низкой квалификации авторов реализации, поэтому мы с некоторым трепетом рассматривали варианты ограничений языка Си с целью повышения защищенности авторов программ от их собственных ошибок. Однако оказалось, что мы на этом пути далеко не первые. Многие авторы пытались “улучшить” старый добрый Си, назовем хотя бы языки D [7], Cyclone [8], много ссылок на такие работы есть в классической книге Роберта Сикорда [9]. Все эти языки и системы программирования преследуют несколько иную по сравнению с нами цель — предотвратить проникновение вредоносного кода в программы, написанные на языке Си.

Мы же ставим перед собой несколько более скромную задачу — по мере возможности не давать программисту совершать многие типичные ошибки и облегчить локализацию и исправление тех, которые ему все же удалось сделать.

Интересно, что эти две цели довольно близки друг к другу, так как в большинстве случаев вредоносные программы проникают в атакуемые приложения именно через те места приложений, где их авторы безосновательно понадеялись на компилятор или на среду исполнения, не вставили надлежащий контроль данных, то есть совершили и вовремя не нашли ошибку. Но эти ошибки и без всяких вирусов могут принести авторам много неприятностей!

Чтобы пояснить, о чем идет речь, приведем несколько характерных черт языков D и Cyclone, добавленных ради повышения защищенности программ на языке Си.

1. В языке D программист имеет возможность указать, что данная функция является безопасной. В таком случае компилятор вставляет многочисленные дополнительные проверки, например:

   - Нет приведений типов, не являющихся указателями, к типу указателя и наоборот
   - Нет арифметики над указателями
   - Нет ассемблерных вставок
   - Не используются адреса локальных переменных и параметров безопасной функции
   - Внутри безопасных функций нет вызовов небезопасных, в частности, стандартных библиотечных функций

2. В языке Cyclone предусмотрено несколько ограничений и расширений языка:

   - Предусмотрен специальный тип для указателей, которые не могут принимать значение NULL. Все остальные динамически проверяются на неравенство NULL при разыменовании
   - Арифметические операции разрешены только для специального класса (так называемых «толстых») указателей, которые содержат в себе не только адрес, но и возможные границы значений
   - Вводятся дополнительные проверки на отсутствие “висячих” указателей, ссылающихся на уже освобожденную память

- Нельзя передать управление по goto внутрь составных конструкций (циклы, переключатели, условные операторы)

2 Что мы изменили в языке Си

1. Мы отказались от union в структурах. Во-первых, это редко используемая черта языка Си, а, во-вторых, она приводит к понижению уровня безопасности.

2. В РуСи нет арифметики указателей. Пользователь может описать переменную-указатель, создать указатель с помощью операции &, присвоить указатель переменной подходящего типа, но не может, например, прибавить к указателю константу и что-либо записать/прочитать по полученному адресу. Все операции динамического отведения памяти имеют своим операндом тип значения, для которого отводится память. Таким образом, любой указатель знает тип значения, на который он ссылается.

3. Разыменование указателя (то есть получение значения по его адресу) в момент, когда он равен NULL, — одна из частых и весьма болезненных ошибок, с другой стороны, вставка проверки на NULL при каждом обращении к указателю резко ухудшает эффективность кода. Несколько лет назад было придумано, как обойти эту трудность. В дополнение к обычному типу указателя (например, int *) вводится еще один тип указателя (например, int @). Второй тип принято называть Never Null Pointer, то есть указатель, который никогда не принимает значение NULL. Переменные такого типа всегда должны инициализироваться, причем значением, отличным от NULL. Использование таких указателей не требует проверки на NULL, если переменной такого типа присвоить обычный указатель, то компилятор автоматически вставит проверку на NULL. В результате, например, цепные списки будут реализовываться с помощью обычных указателей, но там и так проверка на NULL необходима для определения конца списка, а, например, файлы после открытия будут иметь второй тип указателя, поэтому в массовых операциях типа scanf или printf файлами можно пользоваться безопасно и без проверки на NULL.

4. Массив РуСи — это нормальный языковый объект, хотя в базовом языке Си это совсем не так. Его можно описать, в том числе

с динамически вычисляемыми границами (в Си границы только статические), вырезать из него элемент, например a[i][j], или даже a[i] для многомерного массива, присвоить другому массиву, ввести или напечатать (работа с массивом целиком, а не поэлементно, также не поддержана в Си). При размещении в памяти перед нулевым элементом массива по каждому измерению хранится число элементов массива по данному измерению, это дает возможность динамической проверки выхода индекса за границы массива, обычно трансляторы этим пренебрегают, а это одна из самых трудно обнаруживаемых ошибок пользователей. В процессе анализа литературы по проблемам защищенности от ошибок мы наткнулись на публикацию одного российского автора (не хотим давать ссылку на нее), в которой давалась рекомендация не пользоваться индексацией массивов, а использовать только арифметику указателей, мол, так это эффективнее. На наш взгляд, этот совет очень опасен, а такую же эффективность индексации можно обеспечить довольно простыми оптимизациями.

5. За долгие годы существования языка Си появилось большое количество способов описания формальных параметров функций. Мы выбрали один, наиболее выразительный, на наш взгляд, и ужесточили проверки соответствия фактических параметров формальным.

```
int f (int a, float b)
{
...
}
```

Если функция имеет параметром другую функцию, то идентификаторы её параметров не указываются. Ни один известный нам транслятор с языка Си не проверяет соответствие этих идентификаторов у функций, переданных фактическими параметрами.

```
int f (int a, int(* g) (int, float))
{
...
}
```

6. Наконец, много усилий было потрачено, чтобы обеспечить подробную и понятную систему выдачи ошибок с привязкой к месту возникновения ошибки.

3 Среда программирования РуСи

Изменения коснулись не только языка, но и всей системы программирования. Транслятор проекта РуСи производит лексический, синтаксический и семантический анализ исходной программы и строит промежуточное представление в виде линейной развертки дерева разбора. По этому промежуточному представлению может быть сгенерирован код придуманной нами виртуальной машины с широкими возможностями отладки, которых трудно достичь, работая на обычной ЭВМ, а может быть сгенерирован код LLVM — стандарта де-факто для промежуточных языков, этот код может быть откомпилирован в практически любую архитектуру ЭВМ.

Первый вариант компилятора был реализован на платформе MacOS в среде Xcode в консольном режиме. Для пропуска одиночных тестов этого вполне хватало, но при первом же использовании компилятора школьниками и студентами стало ясно, что необходима нормальная IDE (Integrated Development Environment), то есть среда программирования, включающая в себя текстовый редактор, удобную систему компиляции и запуска программ и диалоговый отладчик. Было решено реализовать эту среду на платформе Windows.

Изначально планировалось сделать её кроссплатформенной и для этого была выбрана легковесная и простая в использовании библиотека элементов интерфейса FLTK [10], поддерживающая и MacOS, и Windows, и Linux. Но почти сразу стало ясно, что нельзя просто взять и запустить один и тот же код на разных системах. Дело в том, что по ряду причин компилятор, интерпретатор и сама среда должны быть скомпилированы в разные файлы. Назовем эти причины:

1. Транслятор и среда разрабатывались и тестировались разными людьми (даже на разных операционных системах), поэтому встраивать код первого во вторую было бы проблематично, так как это потребовало бы сильного изменения транслятора. Несколько изменить его все же пришлось.

2. При автономном тестировании компилятор всегда запускается один раз, тогда как среда требует возможности многократного

компилирования различных текстов за один запуск комплиятора. Поскольку в компиляторе используется большое количество глобальных переменных, при встраивании пришлось бы каждый раз их заново инициализировать вручную, а следить за появлением новых переменных проблематично, в ранних версиях это создавало трудно отлавливаемые ошибки.

3. Наконец, среда, компилятор и интерпретатор решают абсолютно разные и практически независимые задачи, так что разбиение на отдельные исполняемые файлы необходимо для стройного построения проекта, деления его на логически завершенные части.

Но вместе с делением всего РуСи на отдельные исполняемые файлы возникает необходимость "общения" этих файлов между собой: например, среда должна выводить на экран сообщения компилятора или вывод интерпретатора виртуальной машины.

Эти проблемы можно решить, например, записью всех сообщений РуСи и вывода программы в файл с последующим чтением этого файла средой, хотя такое решение не совсем удачно, пользователь должен ждать завершения работы своей программы, чтобы увидеть весь вывод. Удобнее и информативнее было бы наблюдать, что программа выводит, параллельно с её исполнением. А вот со вводом таким образом поступить не получится. Если требуется прочитать значение, интерпретатор не может продолжать работу, пока это значение не будет получено, а точное количество информации, которое ему потребуется, неизвестно до начала работы. Более того, пользователь может написать интерактивную программу и в зависимости от ее вывода вводить различные данные, так что обработка ввода/вывода в режиме оффлайн неприемлема.

Для корректного ввода какого-то значения определенного типа интерпретатор должен запросить соответствующие данные у среды, а ей нужно взять это значение из своего потока ввода (для этого, возможно, потребуется ожидать, пока пользователь введет его) и сообщить это значение интерпретатору.

Механизмы обмена информацией между различными запущенными исполняемыми файлами уже не могут быть независимыми от платформы. В системе Windows они реализуются посредством так называемых именованных каналов (named pipes), из которых можно читать записанные туда сообщения или записывать туда что-либо. Для взаимодействия среды с компилятором удобно использовать односто-

ронний канал, так как нам нужно только выводить сообщения РуСи. Однако, для взаимодействия с интерпретатором, как было показано, требуется обмен данными в обе стороны. Подобные механизмы есть и в других системах, но они, конечно, обладают другой семантикой.

Даже для того, чтобы просто запустить исполняемый файл компилятора или интерпретатора по нажатию кнопки, среде необходимо выполнить системный вызов, так что при её написании в любом случае приходится иметь дело с платформой, для которой мы программируем.

Несмотря на то, что от операционной системы полностью абстрагироваться не получается, можно локализовать платформозависимые участки кода. Для этого мы вынесли в отдельные процедуры создание и удаление канала, подключение к нему, передачу и прием сообщений. Тогда компилятор и интерпретатор для среды отличаются от консольных версий лишь заменой обычного ввода/вывода на эти процедуры.

4 Заключение

Как уже было сказано, первоначально проект РуСи был ориентирован исключительно на обучение школьников и студентов. Эта цель была успешно достигнута. Особенно хочется поблагодарить студентов 102 группы отделения математики математико-механического факультета СПбГУ. Поскольку эти студенты ориентированы, в основном, на чистую математику и далеки от программирования, они писали такие странные программы, которые и в голову не могли прийти нормальному программисту (в том числе и авторам проекта РуСи), часто ломая наш компилятор и среду. Студенты 3 курса мат-меха проходили практические занятия по курсу “Трансляция языков программирования” (CS 240), изучая исходные коды системы, иногда мы получали от них весьма ценные замечания.

Первые изменения языка Си опять же имели только одну цель — облегчить поиск ошибок обучающимися (например, отказ от арифметики указателей и обязательный контроль индексов массивов) и чуть упростить написание программ (например, оператор print с аргументом произвольного типа вместо громоздкой форматной печати). Но когда об этом проекте узнали заказчики ООО Ланит-Терком, жизненно заинтересованные в надежности и защищенности программного обеспечения критически важных систем, пришлось пересмотреть цели и задачи проекта РуСи. Именно тогда мы стали работать над гаран-

тированно ненулевыми указателями, сделали обязательным указание типов в процедурах динамического захвата памяти, расширили список статических и динамических проверок. Данный доклад является первой попыткой рассказа о работах в этом направлении.

Список литературы

[1] *Niklaus Wirth, Jürg Gutknecht* Project Oberon The Design of an Operating System and Compiler, Edition 2005, http://www.ethoberon.ethz.ch/WirthPubl/ProjectOberon.pdf

[2] *Bertrand Meyer* Touch of Class, Learning to Program Well with Objects and Contracts, Springer-Verlag Berlin Heidelberg, 2009 (есть русский перевод: Бертран Мейер, Почувствуй класс, Учимся программировать хорошо с объектами и контрактами, Интуит, Бином, Москва, 2011)

[3] Revised Report on the Algorithmic Language ALGOL 68, Springer Berlin Heidelberg, 1976 (есть русский перевод: Пересмотренное сообщение об Алголе 68, Москва, Мир, 1979)

[4] *Kernighan, B.W., and D.M. Ritchie.* The C Programming Language. Englewood Cliffs, NJ: Prentice Hall, 1978, 2nd edition 1988 (есть русский перевод: Брайан Керниган, Деннис Ритчи. Язык программирования C. – Москва: Вильямс, 2006. – 304 с. – ISBN 5845908914)

[5] *Terekhov Andrey, Luchin Roman, Filippov Sergey* Educational Cybernetical Construction Set for Schools and Universities, Advances in Control Education, Volume# 9 | Part# 1

[6] http://blog.trikset.com/

[7] *Alexandrescu, A.* The D programming language, Boston: Adison-Wesley, 2010 (есть русский перевод: Андрей Александреску, Язык программирования D, Санкт-Петербург- Москва, Символ, 2012)

[8] *Grossman, D., M.Hicks, J.Trevor, and G.Morrisett.* "Cyclone: A Type-Safe Dialect of C." C/C++ Users Journal 23, no. 1 (2005): 6-13

[9] *Robert C. Seacord* Secure Coding in C and C++, Second Edition, Addison-Wesley, 2013 (есть русский перевод: Роберт С.Сикорд,

Безопасное программирование на C и C++, второе издание, Вильямс, 2016)

[10] http://www.fltk.org/

Верификация и доказательство завершения функционально-потоковых параллельных программ

Удалова Ю. В.[1], uuuu82@inbox.ru
Ушакова М. С.[1], ksv@akadem.ru
[1] Сибирский федеральный университет

Аннотация

В статье представлены метод верификации программ на функционально-потоковом языке параллельного программирования Пифагор с помощью интервальных формул и метод анализа завершения рекурсивных программ на языке Пифагор.
Ключевые слова: функционально-потоковые параллельные программы, язык программирования Пифагор, верификация с помощью интервальных формул, доказательство завершения рекурсий.

В настоящее время параллельные вычисления, как правило, основаны на многопроцессных или многопоточных императивных программах. Существуют и альтернативные подходы к организации параллельных программ, например, производящие распараллеливание функциональных программ. В работе рассматриваются вопросы верификации программ на функционально-потоковом языке параллельного программирования Пифагор [1], модель вычислений которого предполагает распараллеливание на неограниченных ресурсах и отсутствие привязки разработчика к программированию в терминах процессов или потоков. Оригинальность модели вычислений языка Пифагор требует отдельной проработки вопросов верификации программ на данном языке.

Метод верификации программ на языке Пифагор с помощью интервальных формул.

Представленный метод верификации функционально-потоковых параллельных (ФПП) программ основан на адаптации метода индуктивных утверждений, изначально сформулированного для императивных программ, к ФПП модели [2].

Для описания спецификации программы и утверждений, выведенных на итерациях метода, предложены оригинальные формулы, использующие интервальные константы. Обход операторов ФПП программы производится в соответствии с информационными зависимостями между вершинами-операторами информационного графа программы.

Для спецификации ФПП программ выбран формат, в котором обязательно наличие входного утверждения, а промежуточные утверждения не обязательны. Промежуточные утверждения предлагается приписывать к операторам-вершинам графа, что повышает эффективность локализации некорректных вычислений на графе и далее в коде.

Спецификация входных данных позволяет описывать их через конкретные значения и специальные интервальные формулы, среди которых: ∼gt A — число большее, чем указанное число A, ∼lt A — число меньшее, чем указанное число A, ∼A interval B — число, лежащее в указанном интервале [A, B].

Промежуточные утверждения прикрепляются пользователем к произвольным вершинам-операторам графа и являются выражениями на языке Пифагор, способными включать как интервальные формулы, так и дополнительные обозначения:

ARG — аргумент функции,

NODE — значение той вершины-оператора графа функции, к которой добавлено пользовательское условие,

NODE<натуральное число> — значение оператора с указанным номером (номера операторов назначаются автоматически перед началом верификации и видны пользователю).

Пользователь может писать произвольные операторы языка Пифагор в промежуточных утверждениях, но рекомендуется выстраивать их так, чтобы они возвращали булевы значения, — это позволяет при прохождении процесса верификации отмечать на графе корректные и некорректные операторы.

Описанный метод верификации реализован в инструментальной среде для разработки, отладки и верификации программ на языке Пифагор.

Доказательство завершения программ на языке Пифагор.

В языке Пифагор все функции являются всюду определёнными, поэтому единственная причина по которой программа может не завершиться — бесконечная рекурсия. Пусть $f(x)$ — рекурсивная функция. Если функция f получает в качестве аргумента x некоторое значение x_0, то это значение называется *текущим аргументом*. Если x_0 приводит к выполнению рекурсивной ветви алгоритма, то произойдёт вызов функции f с некоторым аргументом $x_1 = g(x_0)$, где g — некоторая функция, а x_1 называется *рекурсивным аргументом*. Очевидно, что таких аргументов может быть несколько.

Процесс доказательства корректности рекурсивной программы разбивается на две части: доказательство частичной корректности (соответствия программы своей спецификации, если она завершает работу) и доказательство завершения программы.

Для доказательства частичной корректности используется метод, описанный в [3] и основанный на тройках Хоара и разметке дуг графа программы формулами. При этом, корректность рекурсивной функции доказывается по индукции. Базой индукции является истинность всех ветвей вычислений не содержащих рекурсивные вызовы функции. Индуктивное предположение — рекурсивный вызов возвращает значение, удовлетворяющее постусловию для рекурсивного аргумента. А шагом индукции будет доказательство корректности всех ветвей с рекурсивным вызовом, при условии, что рекурсивный аргумент удовлетворяет предусловию рекурсивной функции.

Для доказательства завершения программы на языке Пифагор, модифицируем метод Флойда так, чтобы его можно было применить для рекурсии. Задаётся некоторое фундированное множество S — частично упорядоченное множество, любое непустое подмножество которого имеет минимальный элемент. С каждым вызовом рекурсивной функции связывается частичная функция, принимающая значения в этом фундированном множестве, аргументы которой совпадают с аргументами рекурсивной функции. Назовём такую функцию *ограничивающей*.

Для доказательства того, что функция на языке Пифагор завершается, необходимо показать, что любой допустимый текущий и рекурсивный аргумент будет принадлежать области определения ограничивающей функции, и значение функции для текущего аргумента будет больше чем для рекурсивного.

Аргументы ограничивающей функции совпадают с аргументами рекурсивной функции, поэтому условия завершения функции могут

быть добавлены в предусловие рекурсивной функции и тогда доказательство истинности тройки Хоара для рекурсивной функции станет доказательством тотальной корректности: программа завершается, для всех допустимым предусловием входных аргументов, и возвращает верный результат, удовлетворяющий постусловию. Такое условие завершения может быть использовано в системе поддержки формальной верификации ФПП программ [4].

Список литературы

[1] Легалов А.И. Функциональный язык для создания архитектурно-независимых параллельных программ // Вычислительные технологии. 2005. № 1 (10). С. 71-89.

[2] Удалова Ю.В., Легалов А.И. Верификация функционально-потоковых параллельных программ методом индуктивных утверждений // Доклады Академии наук высшей школы Российской Федерации. 2014. № 2-3 (23-24). С. 125-132.

[3] Кропачева М.С., Легалов А.И. Формальная верификация программ, написанных на функционально-потоковом языке параллельного программирования // Моделирование и анализ информационных систем. 2012. Т. 19, № 5, С. 81–99.

[4] Ushakova M.S., Legalov A.I. Automation of Formal Verification of Programs in the Pifagor Language // Modeling and Analysis of Information Systems. 2015. Vol. 22. N. 4. P. 578–589.

Новые возможности системы HomeLisp

Файфель Б. Л., catstail@narod.ru
Саратовский государственный технический
университет имени Ю. А. Гагарина

Аннотация

Описан ряд возможностей, добавленных в ядро HomeLisp, которые приближают систему к стандартам Common Lisp. В числе этих возможностей - рациональные и комплексные числа, многозначные функции (формы, имеющие множество значений), универсальные итераторы и структуры. Кратко упоминается интерфейс с Win32API. Описанные возможности заметно расширяют функциональность системы HomeLisp и могут способствовать применению системы для обучения Лиспу.

Ключевые слова: PLC, HomeLisp, Лисп, Рациональные числа, Комплексные числа, Итераторы, Многозначные функции, Структуры, Win32API.

Основная задача проекта HomeLisp [1-3] по мысли автора, состояла в том, чтобы создать простой, лёгкий в установке программный продукт, содержащий все необходимые средства как для обучения Лиспу, так и для реализации типовых программ на Лиспе (работа с файлами, строками, простая графика, графический пользовательский интерфейс с возможностью создания exe-файлов). Известные автору реализации Лиспа либо просто не обеспечивали эти возможности (muLisp), либо делали решение приведенных задач (графика, графический интерфейс пользователя) весьма трудными для начинающих.

Проект первоначально предназначался для использования в учебном процессе при обучении языку Лисп. Впоследствии оказалось, что система может быть полезной и при изучении математики [4]. Первоначально версия языка соответствовала стандарту Lisp 1.5, затем было принято решение переработать язык и приблизить его к стандарту Common Lisp.

В настоящем сообщении приведены основные изменения, внесенные в ядро HomeLisp:

- Арифметика рациональных и комплексных чисел;
- Многозначные функции;
- Новые итераторы;
- Структуры;
- Интерфейс с WinAPI.

Реализация рациональных чисел может оказаться полезным в процессе обучения, поскольку обработка рациональных чисел выполняется без погрешности. Это дает возможность обучаемому разобраться, к примеру, в довольно тонких вычислительных аспектов сходимости рядов (там, где использование чисел с плавающей точкой затруднено из-за накопление погрешности).

Многозначные функции (и специальные формы для работы с ними) введены в язык для облегчения использования кода, подготовленного в стандарте Common Lisp.

Универсальный итератор ITER, введенный в ядро HomeLisp, реализует достаточно большое подмножество функций макро ITERATE [5]. Итератор ITER весьма прост и нагляден в использовании.

Структуры как контейнерный тип данных присутствуют практически во всех современных языках программирования. Синтаксические формы работы со структурами в HomeLisp в целом соответствуют стандарту Common Lisp. Отличия касаются формы внутреннего представления (структуры представляются списками).

Интерфейс с WinAPI обеспечивает вызов функций из библиотек динамической компановки (dll), что заметно расширяет возможности HomeLisp. Подробному описанию интерфейса с WinAPI посвящена работа [6].

Все описанные изменения не привели к заметному возрастанию объёма ядра HomeLisp – система осталась достаточно компактной и простой в использовании.

БИБЛИОГРАФИЯ

1. Интернет-ресурс: http://homelisp.ru
2. Конференция ICIT-2012. “HomeLisp is a simple language implementation of Lisp for education” - p. 50. ISBN 978-5-77433-2489-7.

3. “HomeLisp - простая реализация языка Лисп 1.5 для целей обучения”. Вестник НГУ. Серия “Информационные технологии”. 2012, том 10, выпуск 3, стр. 105. ISSN 1818-7900.

4. В.А.Болотюк, Л.А.Болотюк “О применении HomeLisp в процессе обучения математике” Интернет-журнал “Науковедение” ISSN 2223-5167 http://naukovedenie.ru/ Том 7, №5 (2015)

5. Интернет-ресурс: https://common-lisp.net/project/iterate/doc

6. Файфель Б. Л. “Реализация интерфейса Win32API в системе HomeLisp” Первый открытый статистический конгресс, Новоси бирск, 2015.

Трансляция проблемно-ориентированного языка Green-Marl в параллельный код на Charm++ на примере задачи поиска сильно связных компонент в ориентированном графе

Фролов А. С.[1], frolov@nicevt.ru
Симонов А. С.[1], simonov@nicevt.ru
[1]АО «Научно-исследовательский центр электронной вычислительной техники»

Аннотация

В докладе будут представлены результаты работы по реализации генератора кода на Charm++ в компиляторе проблемно-ориентирован-ного языка Green-Marl, предназначенного для параллельного анализа статических графов. В качестве примера использован параллельный алгоритм поиска сильно связных компонент в ориентированном графе на основе раскраски вершин графа.
Ключевые слова: параллельный анализ графов, вычислительные модели с управлением потоком данных, компиляторы, проблемно-ориентированные языки программирования.

Параллельный анализ статических графов с использованием многопроцессорных вычислительных систем (суперкомпьютеров) представляет значительный практический интерес в связи с появлением большого количества задач, в которых требуется выполнение анализа больших массивов слабоструктурированных данных, часто представляемых в виде семантических графов, а также задач, в которых графы

присутствуют в явном виде (биоинформатика, анализ социальных сетей, анализ веб-графа и т.п.).

Вместе с тем эффективная параллельная обработка графов сопряжена с рядом проблем, возникающих вследствие нерегулярной природы графов, их несбалансированности, а также сложности реализации графовых алгоритмов из-за необходимости использования специфических оптимизаций, таких как агрегация сообщений, переупорядочивание матрицы смежности, статическая и динамическая балансировка вычислительной нагрузки и пр.

В данной работе представлен компилятор проблемно-ориентированного языка Green-Marl [1], предназначенного для параллельной обработки статических графов, расширенный возможностью трансляции программ в параллельную вычислительную модель Charm++ с управлением потоком сообщений [2]. Детально генерация кода на Charm++ в компиляторе Green-Marl представлена в работе [5].

В качестве примера работы компилятора рассматривается реализация алгоритма поиска сильно связных компонент в ориентированном графе на основе раскраски вершин графа. Данный алгоритм был описан на языке Green-Marl, после чего с помощью компилятора была получена его версия на Charm++. Для сравнения производительности использовалась референсная реализация алгоритма на Charm++, выполненная вручную.

Для тестирования производительности полученных реализаций использовались синтетические графы двух типов (RMAT и Random) разного размера (от 2^{20} до 2^{26} вершин). В качестве суперкомпьютера использовался кластер Ангара-К1 на базе отечественной сети Ангара [3; 4]. Результаты экспериментов показали, что время работы программы на Green-Marl сопоставимо со временем работы программы, реализованной вручную на Charm++, что позволяет говорить о приемлемом уровне эффективности компилятора Green-Marl.

Работа выполнена при поддержке гранта РФФИ №15-07-09368.

Список литературы

1. Green-Marl: a DSL for easy and efficient graph analysis / S. Hong [и др.] // ACM SIGARCH Computer Architecture News. Т. 40. — ACM. 2012. — С. 349—362.

2. *Kale L. V.*, *Krishnan S.* CHARM++: A Portable Concurrent Object Oriented System Based on C++ // SIGPLAN Not. — New York, NY, USA, 1993. — Окт. — Т. 28, № 10. — С. 91—108. — ISSN 0362-1340. — DOI: 10.1145/167962.165874. — URL: http://doi.acm.org/10.1145/167962.165874.

3. Первое поколение высокоскоростной коммуникационной сети «Ангара» / А. Симонов [и др.] // Наукоемкие технологии. — 2014. — Т. 15, № 1. — С. 21—28.

4. Результаты оценочного тестирования отечественной высокоскоростной коммуникационной сети Ангара / А. Агарков [и др.] // Тр. международной конференции Суперкомпьютерные дни в России. — 2016. — С. 626—639.

5. *Фролов А.* Использование программной модели Charm++ в качестве целевой платформы для компилятора проблемно-ориентированного языка для обработки статических графов // Вычислительные методы и программирование. — 2017. — Т. 18, № 2. — С. 103—114.

Синтез операторов предикатной программы

Шелехов В. И., vshel@iis.nsk.su
Институт Систем Информатики СО РАН,
г. Новосибирск

Аннотация

Рассматривается программный синтез операторов предикатной программы в интеграции с дедуктивной верификацией и обычным стилем конструирования программы в редакторе Eclipse. Анализируются методы синтеза операторов на примере эффективной программы сортировки методом простых вставок. Предлагается архитектура интегрированной системы дедуктивной верификации и программного синтеза.

Ключевые слова: формальная операционная семантика, программный синтез, дедуктивная верификация, SMT-решатель, система доказательства PVS, сортировка

Язык предикатного программирования P [1] является языком доказательного программирования. *Предикатная программа* $H(x:\ y)$ с аргументами x и результатами y является предикатом в форме вычислимого оператора. Для языка P построена формальная операционная семантика таким образом, что результат y вычисляется для аргумента x тогда и только тогда, когда предикат $H(x:\ y)$ является истинным [5]. *Тотальная корректность* программы $H(x:\ y)$ относительно спецификации в виде *предусловия* $P(x)$ и *постусловия* $Q(x,\ y)$ определяется истинностью формул *частичной корректности* $P(x)\ \&\ H(x:\ y) => Q(x,\ y)$ и *тотальности* $P(x) => \exists\, y.\ H(x:\ y)$.

Для операторов языка P разработана система правил доказательства корректности [3], существенно упрощающая процесс доказательства корректности программы. В системе предикатного программирования реализован генератор формул корректности программы. Значительная часть формул доказывается автоматически SMT-решателем

*CVC*3. Оставшаяся часть формул генерируется для системы интерактивного доказательства *PVS*.

Задача программного синтеза заключается в нахождении тотального предиката H на языке P, обеспечивающего истинность формулы $P(x)\ \&\ H(x:\ y) => Q(x,\ y)$. Наряду с данной формулой можно также использовать любое из правил доказательства корректности. Таким образом, задача синтеза сводится к задаче разрешимости формулы относительно неизвестных предикатов и термов: необходимо найти такие предикаты и термы в позициях неизветстных, чтобы формула стала истинной.

В настоящее время, автоматический синтез программ по формальным спецификациям, несмотря на значительные достижения в этой области, возможен лишь для простых коротких программ. В нашей работе [2] рассматривается синтез небольших фрагментов предикатной программы. Синтез фрагментов реализуется в интеграции с дедуктивной верификацией [3, 4, 6] в среде редактора Eclipse. Однако синтез фрагментов даже простых программ часто блокируется ввиду невозможности используемых SMT-решателей *CVC*3 и *Z*3 разрешить формулы корректности, поставляемые от правил синтеза.

В настоящей работе задача синтеза операторов исследуется на примере вполне реалистичной эффективной программы сортировки массивов методом простых вставок.

Разработка в стиле доказательного программирования предполагает определение базисных теорий для типов данных программы, написание предусловий и постусловий в составе соответствующих теорий вместе с набором лемм, определяющих основные свойства, используемые в процессе доказательства формул корректности. Для задачи сортировки теория *SortDecl* определяет тип массива элементов произвольного типа T с отношением линейного порядка. Теория *Perm* определяет предикат перестановочности массивов; леммы *pe*1 и *pe*2 представляют свойства рефлексивности и транзитивности. Теория *Sort* определяет свойство сортированности.

Сначала исходная задача сортировки $sort(a:\ a')$ сводится к более общей задаче сортировки $sort1(a,\ m:\ a')$, в которой начальная часть массива с индексами от 0 до m является отсортированной. В теории *Sort* доопределяется свойство сортированности начальной части массива:

$formula\ sorted(Arn\ a,\ natn\ m) = \forall\, i, j = 0..m.\ i < j => a[i] \leq a[j];$

Для сведения к более общей задаче $sort1$ используется простое пра-

вило, в соответствии с которым требуется найти тотальную функцию $B(a)$, при котором формула $sorted(a,\ B(a))$ была бы истинной. Терм для функции $B(a)$ определяется перебором. Такими термами могут быть: $0,\ 1,\ n,\ n-1,\ len(a)$. При $B(a) = 0$ в формуле $sorted(a,\ 0)$ кванторная часть вырождается, и она становится тривиально истинной. Истинность $sorted(a,\ 0)$ может быть установлена обращением к SMT-решателю.

Программа $sort1(a,\ m:\ a')$ вызывает подпрограмму $pop\_into(a, m, e : a')$ для вставки очередного элемента e внутрь уже отсортированной части массива для элементов с индексами $0..m$. Представим программу $sort1$ с двумя позициями $[*]$, которые должны быть синтезированы.

$$if\ (m = n)\ [*]\ else\ \{\ pop\_into(a,\ m,\ a[m+1]:\ Arn\ c);\ \ sort1([*])\ \}$$

Формула корректности для первого синтезируемого фрагмента определяется следующим образом:

$$sorted(a,\ m)\ \&\ m = n\ \&\ X(a,\ m:\ a') => perm(a,\ a')\ \&\ sorted(a')$$

Здесь $X(a,\ m:\ a')$ — неизвестный предикат, который следует выбрать так, чтобы эта формула стала истинной. Необходимо независимо обеспечить истинность двух конъюнктов $perm(a,\ a')$ и $sorted(a')$. В соответствии с леммой pe1:

```
pe1: lemma perm(a, a);
```

истинность $perm(a,\ a')$ обеспечивается использованием оператора присваивания $a' = a$. Подсистема синтеза должна уметь найти такое решение. Истинность второго конъюнкта следует из другой леммы.

Простейший способ реализации программы *pop_into* – последовательный обмен очередного элемента e с предыдущими соседними элементами пока не достигнем отсортированного состояния. Это простой способ, но не эффективный. Предпочтительней вместо обмена элементов переместить очередной соседний элемент массива на одну позицию. Такое решение приводит к обобщению программы *pop_into*, в которой появляется дополнительный параметр k, определяющий позицию «пустого» элемента после перемещения очередного элемента на одну позицию вперед. В программе *pop_into* выделено три синтезируемых фрагмента:

```
 (a[k-1] <= e) [*] else
{ Arn b = a with (k: a[k-1]);
  if (k = 1) [*] else pop_into([*])}
```

Для отмеченных фрагментов порождаются несколько длинных формул корректности; некоторые до трех строк. Требуется пять дополнительных нетривиальных лемм. Их невозможно сформулировать без анализа сгенерированных формул корректности. Даже при наличии лемм процесс синтеза с применением SMT-решателей становится проблематичным. Таким образом, задача автоматического синтеза реальных программ оказывается принципиально нереализуемой.

Анализ особенностей синтеза операторов в программе сортировки дает возможность сформулировать предложение об архитектуре подсистемы верификации и синтеза в системе предикатного программирования. Необходим собственный специализированный решатель, работающий интерактивно в контексте создаваемой программы и набора теорий. Решатель проводит преобразования и удобную визуализацию генерируемых формул корректности. Преобразования решателя: унификация термов, перебор термов, подстановки, обеспечивающие истинность формул с использованием лемм, и другие.

Если снабжать формулы корректности необходимыми леммами, то число формул, которые могут быть доказанными SMT-решателями, возрастает с 20% до 90%. Это и другие свойства специализированного решателя способны примерно вдвое ускорить процесс дедуктивной верификации предикатных программ. Сейчас трудоемкость дедуктивной верификации примерно в 10 раз больше в сравнении с обычным стилем программирования. Отметим, что дедуктивная верификация самой быстрой программы сортировки [4], состоящей всего из 20 строк, потребовала более месяца.

Иммется полная версия настоящей работы [7].

Работа выполнена при поддержке РФФИ, грант 16-01-00498.

Список литературы

[1] Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. Новосибирск, 2010. 42с. (Препр. / ИСИ СО РАН; N 153).

[2] Чушкин М.С., Шелехов В.И. Методы синтеза фрагментов предикатных программ // Конф. «Компьютерная безопасность и криптография» SIBECRIPT'16 / Прикладная дискретная математика. Приложение. — 2016, №.9, С.126-128.

[3] Чушкин М.С. Система дедуктивной верификации предикатных программ // «Программная инженерия», № 5, 2016. – С. 202-210.

[4] Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. – Новосибирск, 2012. – 30с. – (Препр. / ИСИ СО РАН. N 164). http://www.iis.nsk.su/files/preprints/164.pdf

[5] Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. — Новосибирск, 2015. — 13с. http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf

[6] Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. — С. 14-21.

[7] Шелехов В.И. Синтез операторов предикатной программы // Институт Систем Информатики, Новосибирск, 2017. — 14с. http://persons.iis.nsk.su/files/persons/pages/sints.pdf

Задачи оптимизирующей компиляции для процессоров близкого будущего

Штейнберг Б. Я.[1], borsteinb@mail.ru
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

В статье рассмотрена проблема отставания высокопроизводительного программного обеспечения от развития вычислительных систем. В статье предложен ряд задач, решение которых может способствовать развитию оптимизирующих распараллеливающих компиляторов для вычислительных систем близкого будущего. Эти задачи ориентированы, в первую очередь, на оптимизацию использования структуры памяти. В частности, предлагается также рассмотрение диалогового режима компиляции, построение фактор-графов решетчатых графов, отображение программ на программируемые архитектуры.
Ключевые слова: автоматическое распараллеливание, оптимизирующий компилятор, структура памяти.

1 Введение

Отметим изменения в развитии компьютеров, которые оказывают влияние на развитие программного обеспечения (ПО).

1. Производительность памяти в десятки раз меньше производительности процессора, а 40-50 лет назад было наоборот [11].

2. Растет разнообразие вычислительных архитектур, обостряя проблему переносимости ПО с сохранением адекватной эффективности.

3. На одном кристалле могут уживаться одновременно MIMD, SIMD и программируемые вычислители и элементы памяти. Создается

проблема эффективного отображения программ на такие гибридные вычислительные архитектуры.

Цена достижения быстродействия ПО: большие затраты времени и высокая (дорогая) квалификация разработчиков. Смягчить проблему может развитие оптимизирующих компиляторов.

2 Оптимизация использования памяти

Минимизации обменов данными между модулями памяти служит стратегия разбиения задач на подзадачи. Идея этого метода состоит в том, чтобы все данные, которые читаются в подзадаче, помещались в одном модуле памяти (кэш-памяти или локальной памяти). Подзадачи, в свою очередь, тоже могут разбиваться на более мелкие подзадачи, для попадания в модуль памяти следующего уровня... Такая иерархия подзадач должна приближаться к повторению иерархии памяти вычислительной системы. Разбиение задач на подзадачи (алгоритмов на подалгоритмы) иногда называют переходом к блочному коду в программе или тайлингом [3].

Блочное распределение массивов в оперативной памяти – дополнение к блочному коду, дающее дополнительное ускорение [8].

Компилятор должен автоматически преобразовывать программу к блочному коду и блочно размещать матрицы в оперативной памяти.

Архитектуры вчерашних суперкомпьютеров сегодня (или близком завтра) повторяются в архитектурах процессоров. В частности, уже появляются многоядерные процессоры, у которых каждое ядро имеет свою локальную (адресуемую) память (IBM Cell, Tile64,...). Таким образом, приемы программирования на системы с распределенной памятью становятся актуальными для программирования систем на кристалле. Основная проблема автоматического распараллеливания на системы с распределенной памятью – автоматическое размещение и переразмещение данных [6].

Всякое данное, которое имеется в кэш-памяти высокого уровня, представлено также и в оперативной памяти (на другом кристалле). Альтернативой кэш-памяти может быть локальная память – это адресуемая память, которая находится на одном кристалле с процессором. Алгоритм Литтла и его модификации [4] для решения задачи коммивояжера являются примером алгоритмов, у которых много (экспоненциальное количество) промежуточных данных, запись и чтение которых создают большие обмены между микросхемой оперативной

памяти и микросхемой процессора. Сохранение этих промежуточных данных компилятором в локальной памяти процессора могло бы существенно ускорить процесс решения задачи.

3 Многообразие вычислительных архитектур и проблема высокопроизводительной переносимости

Растет многообразие архитектур вычислительных систем. Поддержание на различных вычислительных архитектурах одного языка программирования высокого уровня создает возможность переносимости ПО (с перекомпиляцией). В качестве такого параллельного языка программирования многие производители процессоров поддерживают OpenCL. Однако, практика показала, что параллельная программа, написанная на OpenCL, может переноситься с одной архитектуры на другую с большой (до двух порядков) потерей производительности. Очевидным объяснением этого являются различные структуры памяти, на которые не перенастраивается переносимое параллельное ПО. Возникает проблема высокопроизводительной переносимости ПО [1]. Вероятно, эта проблема может как-то решаться с использованием автоматического перехода к блочному коду в компиляторах.

4 Отображение программ на программируемые архитектуры

Процессор с программируемой архитектурой – это может быть, например, система на кристалле, содержащая как универсальные вычислительные ядра, так и программируемую логику. Компилятор на такую систему на кристалле должен содержать в своем составе конвертор с входного языка на язык описания электронных схем и библиотеку драйверов (или генератор драйверов). Компилятор C2H [9] – пример реализации компилятора на программируемый процессор. Этот компилятор был привязан к одному процессору NIOS II и обладает многими недостатками (требует много ручного дописывания прагм). Другой такой проект представлен в [13].

5 Решетчатые графы

Информационные зависимости описываются решетчатыми графами. Анализ зависимостей в гнездах циклов привел к появлению метода гиперплоскостей [12]. Метод гиперплоскостей (распараллеливание) зачастую дает существенное замедление, но в комбинации с тайлингом, дает ускорение. Это приводит к задаче выделения блоков на решетчатом графе и построения фактор-графа решетчатого графа в компиляторе. В [2], [10] описываются методы хранения решетчатых графов в компиляторе.

6 Преобразование программ и диалоговое уточнение зависимостей

В [5] показано, что программная реализация алгоритма Флойда-Уоршала, не может быть автоматически распараллелена, но, если программисту известно, что матрица содержит только неотрицательные элементы, то программист может ее распараллелить (вручную). В этой работе предлагается диалоговый режим компилятора для распараллеливания.

7 Автоматическая оптимизация специальных классов алгоритмов

Известны классы алгоритмов, которые сегодняшними компиляторами автоматически не распараллеливаются, но которые распараллеливаются (оптимизируются) вручную. Рассмотрим несколько таких классов.

Алгоритмы обхода дерева используются во многих переборных задачах и задачах дискретной оптимизации. Обход дерева используется, например, при методе ветвей и границ [4]. Для параллельного выполнения обхода дерева разным вычислительным устройствам раздаются разные ветви.

Рекуррентные циклы представляют собой последовательный процесс, во многих часто встречаемых на практике случаях такие циклы могут быть преобразованы к виду, допускающему распараллеливание

[7]. Эффективность такого параллельного выполнения зависит от вычислительного устройства, на котором алгоритм выполняется.

Итерационные алгоритмы используются, например, для численного решения задач математической физики. Многие такие задачи решаются на суперкомпьютерах с распределенной памятью. К некоторым таким задачам можно применить метод размещения данных с перекрытием [3]. Для автоматического применения этих методов требуется, в первую очередь, распознать по тексту программы, что такой метод используется.

Список литературы

1. *Абу-Халил Ж. М.*, *Гуда С. А.*, *Штейнберг Б. Я.* Перенос параллельных программ с сохранением эффективности // Открытые системы. СУБД. — 2015. — № 4. — С. 18—19.
2. *Воеводин В. В.* Математические модели и методы в параллельных процессах. — Москва : Наука, 1986. — 296 с.
3. *Гервич Л. Р.*, *Штейнберг Б. Я.*, *Юрушкин М. В.* Разработка параллельных программ с оптимизацией использования структуры памяти. — Ростов-на-Дону : издательство Южного федерального университета, 2014. — 120 с.
4. *Костюк Ю. Л.* Эффективная реализация алгоритма решения задачи коммивояжера методом ветвей и границ // Прикладная дискретная математика. — 2013. — Июнь. — Т. 20, № 2. — С. 78—90.
5. *Морылев Р. И.*, *Шаповалов В. Н.*, *Штейнберг Б. Я.* Символьный анализ в диалоговом распараллеливании программ // Информационные технологии. — 2013. — № 2. — С. 40—45.
6. *Штейнберг Б. Я.* Оптимизация размещения данных в параллельной памяти. — Ростов-на-Дону : издательство Южного федерального университета, 2010. — 255 с.
7. *Штейнберг О. Б.*, *Суховерхов С. Е.* Автоматическое распараллеливание рекуррентных циклов с проверкой устойчивости // Информационные технологии. — 2010. — № 1. — С. 40—45.
8. *Юрушкин М. В.* Двойное блочное размещение данных в оперативной памяти при решении задачи умножения матриц // Программная инженерия. — Москва, 2016. — Т. 7, № 3. — С. 132—139.

9. NIOS II & C2H Compiler. User Guide. — URL: https://www.altera.com/en_US/pdfs/literature/ug/ug_nios2_c2h_compiler.pdf (дата обр. 02.07.2017).
10. *Feautrier P.* Parametric Integer Programming // RAIRO Recherche Opérationnelle. — 1988. — Сент. — № 22. — С. 243—268.
11. *Graham S. L.*, *Snir M.*, *Patterson C.* Getting Up To Speed: The Future Of Supercomputing. — National Academies Press, 2005. — 289 с.
12. *Lamport L.* The parallel execution of DO loops // Communications of the ACM. — New York, NY, USA, 1974. — Т. 17, № 2. — С. 83—93.
13. *Steinberg B. Y.* [и др.]. Automatic High-Level Programs Mapping onto Programmable Architectures // Proceedings of the 13th International Conference on Parallel Computing Technologies. Т. 9251 (Petrozavodsk, Russia). — Springer Verlag, 2015. — С. 474—485.

Web-ориентированная интегрированная среда разработки программ на языке EasyFlow описания композитных приложений

Штейнберг Р. Б., romanofficial@yandex.ru
Алымова Е. В., langnbsp@gmail.com
Баглий А. П., taccessviolation@gmail.com
Коненко А. С., yadummer@gmail.com
Колесников С. О., kolsolv@gmail.com
Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

Предметно-ориентированный язык EasyFlow используется для описания вычислительных задач в виде композитного приложения, готового к запуску в облачной распределенной среде МИТП CLAVIRE. В работе изложены подходы к реализации интегрированной среды разработки программ на языке EasyFlow, доступной через web-браузер. В составе реализованной среды рассматриваются текстовый редактор с развитой системой синтаксических и семантических контекстно-зависимых подсказок, а также компонент визуализации композитного приложения в виде абстрактного и исполняемого графов потока заданий.

Ключевые слова: DSL, облачные вычисления, интегрированная среда разработки, web IDE.

Язык EasyFlow [2] является предметно-ориентированным и разработан специально для описания вычислительных задач в виде потока заданий (workflow) без учёта архитектурных особенностей распределенной вычислительной среды. Программа на языке EasyFlow – это

высокоуровневое описание задачи в форме композитного приложения. Под композитным приложением понимается последовательность обращений к сервисам, взаимодействующим в распределенной среде для совместного решения общих задач. Композитные приложения, описанные на языке EasyFlow, предназначены для запуска в многопользовательской инструментально-технологической платформе CLAVIRE [1], которая разрабатывается в Санкт-Петербургском национальном исследовательском университете информационных технологий, механики и оптики под руководством А.В. Бухановского.

МИТП CLAVIRE рассчитана на обработку данных больших объемов и ориентирована на пользователей, заинтересованных в ресурсоёмких вычислениях для исследований в разных научных областях. Одной из целей разработки CLAVIRE является предоставление пользователю удобной среды для формирования вычислительной задачи, её запуска и получения результатов. При этом пользователь имеет возможность сконцентрироваться на самой задаче, а не технических деталях её исполнения в вычислительной среде.

С точки зрения пользователя вычислительная задача представляет собой последовательность вызовов подпрограмм из проблемно-ориентированных вычислительных пакетов, предоставленных в виде сервисов, и может быть визуализирована с помощью графа потока заданий (abstract workflow). В графе потока заданий узлам соответствуют обращения к подпрограммам вычислительных пакетов. Два узла в графе соединены дугой, если обращение к одному пакету происходит только после окончания работы другого пакета (зависимость по исполнению), или если один пакет в качестве входных параметров использует выходные параметры другого пакета (зависимость по данным). Синтаксически и семантически корректно описанная задача готова к исполнению в вычислительной среде. Процесс исполнения задачи визуализируется графом конкретного потока заданий (concrete workflow). В графе конкретного потока заданий узлам соответствуют запуски подпрограмм вычислительных пакетов с конкретными наборами параметров.

Интерфейс среды разработки программ на языке EasyFlow реализован в виде многооконного web-приложения, выполненного по технологии HTML5. В состав интерфейса входит менеджер проектов, текстовый редактор на базе AceEditor, компонент визуализации графовых представлений workflow, компонент загрузки данных о доступных вычислительных пакетах.

В текстовый редактор интегрирован ANTLR-парсер языка EasyFlow. Абстрактное синтаксическое дерево парсера используется для: проверки синтаксиса и семантики входных скриптов; формирования контекстно-зависимых подсказок; визуализации графов.

Интерфейс разработан с учётом следующих требований:

- доступность - для использования приложения нужен браузер и подключение к Интернет;
- удобство разработки программ - с помощью базы пакетов и контекстно-зависимых подсказок пользователь всегда видит входные и выходные параметры доступных ему пакетов;
- наглядность - программа, запущенная на вычисление, представляется в виде интерактивного графа конкретного потока заданий, состояния узлов которого отображают процесс выполнения программы.

В ходе реализации интегрированной среды разработки использовалась система тестирования Karma для AngularJS приложений и генератор документации JSDoc.

Список литературы

1. CLAVIRE: облачная платформа для обработки данных больших объемов / В. Н. Васильев [и др.] // Информационно-измерительные и управляющие системы. — 2012. — Т. 10, № 11. — С. 7—16.
2. *Князьков К. В.*, *Ларченко А. В.* Предметно-ориентированные технологии разработки приложений в распределенных средах // Известия вузов. Приборостроение. Специальный выпуск: Перспективные технологии распределенных вычислений. — 2011. — Т. 10, № 10. — С. 36—43.

Проблемно-ориентированый язык быстрого поиска нуклеотидных последовательностей минимальной длины, удовлетворяющих различным топологиям связывания азотистых оснований

Юрушкин М. В.[1], m.yurushkin@gmail.com
Гервич Л. Р.[1], lgervith@gmail.com
Бачурин С. С.,[2] bachurin.rostgmu@gmail.com
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет
[2]НИИ Физической и Органической Химии ЮФУ

Аннотация

Создан новый язык быстрого поиска последовательностей минимальной длины, удовлетворяющих различным топологиям связывания азотистых оснований. Также к предлагаемому языку разработан компилятор. С помощью данного компилятора были найдены одноцепочечные нуклеотидные последовательности, способные образовывать 4 типа неканонических топологий связывания азотистых оснований: G-квадруплес, i-мотив, шпилька и триплекс.

Ключевые слова: нуклеотидные последовательности, неканонические структуры, азотистые основания, регулярные выражения, теория автоматов

1 Введение

В настоящее время экспериментальными и теоретическими методами были обнаружены следующие неклассические топологии свя-

зывания (т.н. неканонические структуры азотистых оснований): G-квадруплекс, i-мотив, триплекс, шпилька [1]. Для возможности теоретических исследований особенностей образования таких топологий в первую очередь необходимо определить, какие из участков нуклеиновых кислот способны образовывать задаваемые топологии. Учитывая, что известны отдельные последовательности азотистых оснований, которые способны образовывать заданную топологию, представляется возможным рассчитать нуклеотидную последовательность минимальной длины, которая теоретически может сформировать все заданные топологии. Нахождение такой строки даст большому числу специалистов (медиков, химиков, биологов) инструмент для теоретических и экспериментальных изысканий.

2 Алгоритм нахождения минимальной строки, удовлетворяющей заданным топологиям

2.1 Язык описания геномных топологий

Описание топологии представляет собой набор правил, левая часть которых является нетерминальным символом, а правая состоит из нетерминальных и терминальных символов и следующих операций:

- Итерация (замыкание Клини) обозначается символом $*$.
- Конкатенация обозначается пробелом или пустой строкой.
- Объединение обозначается символом $|$.
- Выражение $X\{m\}$ обозначает повторение m раз символа X.
- Выражение *min(I)* означает операцию поиска всех минимальных строк, удовлетворяющих топологии I.

Значение в фигурных скобках может быть параметром, тогда исходное выражение эквивалентно объединению выражений по всевозможным значениям параметра.

Рассмотрим пример программы на предлагаемом языке:

Пример 1. Поиск всех минимальных строк, каждая из которых является одновременно G-квадруплексом, i-мотивом, шпилькой и триплексом.

```
// G-квадруплекс:
I1 = X* d{m} X{3}X* d{m} X{3}X* d{m} X{3}X* d{m} X*, m=[1:20]
// i-мотив:
I2 = X* c{a} X{3}X* c{a} X{6}X* c{a} X{3}X* c{a} X*, a=[1:20]
// шпилька:
I3 = X* a{a}b{b}c{c}d{d} X{4}X* c{d}d{c}a{b}b{a} X*, a=[1:1],
     b=[1:1], c=[1:1], d=[1:1]
// триплекс:
I4 = X* b X{4}X* a X{3}X* c X*
X = (a|b|c|d)
result = min(I1 I2 I3 I4)
```

2.2 Быстрый поиск минимальной строки, удовлетворяющей набору топологий

В предлагаемом компиляторе был реализован алгоритм:

1. На первом шаге каждый используемый паттерн, заданый во входной программе, конвертируется в недетерменированный конечный автомат (NFA). Затем каждый полученный недетерменированный автомат конвертируется в детерменированный конечный автомат (DFA).

2. Строится граф вычислений, в котором вершинами являются операции над автоматами (объединение, пересечение и т.д.).

3. Над каждым полученным DFA производится минимизация с помощью алгоритма Мура [3].

4. В результате выполнения всех операций в графе вычислений получается результирующий DFA. Все строки, которые полученный DFA допускает, удовлетворяют заданным топологиям. Для того, чтобы найти все минимальные строки, полученный DFA рассматривается как ориентированный граф $G(V, E)$. В графе $G(V, E)$ осуществляется поиск минимального пути с помощью алгоритма Дейкстры из вершины, соответствующей начальному состоянию DFA.

3 Результаты

С помощью разработанного инструмента были найдены все одноцепочные нуклеотидные последовательности, способные образовывать любые из следующих неканонических структур: G-квадруплекс, i-мотив, шпилька и триплекс. Результаты расчетов показали, что минимальная искомая строка имеет длину 17 символов. Всего таких строк было найдено 2496. С исходным кодом предлагаемого компилятора, а также примерами программ для него, можно ознакомиться здесь [2].

Список литературы

1. A bouquet of DNA structures: Emerging diversity / M. Kaushik [и др.] // Biochemistry and Biophysics Reports. — 2016. — Март. — Т. 5. — С. 388—395. — DOI: 10.1016/j.bbrep.2016.01.013. — URL: https://doi.org/10.1016%2Fj.bbrep.2016.01.013.
2. DFAlib https://github.com/myurushkin/dfalib. — 2016. — URL: https://github.com/myurushkin/dfalib (дата обр. 17.01.2017).
3. *Hopcroft J.*, *Motwani R.*, *Ullman J.* Introduction to Automata Theory, Languages, and Computation. — Pearson Education, 2014. — (Always learning). — ISBN 9781292039053. — URL: https://books.google.ru/books?id=RLFJnwEACAAJ.

Переразмещение матриц к блочному виду компилятором языка Си с минимизацией использования дополнительной памяти

Юрушкин М. В.[1], m.yurushkin@gmail.com
Семионов С. Г.[1], stassemionov@mail.ru
[1]Институт математики, механики и компьютерных наук им. И. И. Воровича, Южный Федеральный Университет

Аннотация

В данной работе представлен метод преобразования размещения матриц между строчным и блочным представлениями. Строчное размещение используется в языке программирования Си в качестве стандартного. Блочное размещение подразумевает хранение элементов матрицы в памяти таким образом, что элементы блока располагаются в памяти последовательно. Такая модель размещения обеспечивает высокую эффективность работы кеш-памяти процессора для алгоритмов, выполняющих значительное число операций над каждым элементом матрицы. Особенностью представленного метода является то, что он требует небольшой объем дополнительной памяти для переразмещения матрицы, который равен равен длине строки переразмещаемого блока. Кроме того, результаты численных экспериментов показали, что такой подход позволяет в 4-10 раз быстрее переразмещать матрицу, чем наивный алгоритм переразмещения матрицы. Предлагаемый алгоритм можно использовать в блочных алгоритмах, использующих блочное размещение матриц. Программная реализация данного алгоритма была реализована в рамках проекта системы ОРС.

Ключевые слова: Блочное размещение матриц, двойное блочное размещение матриц, кеш-память, высокопроизводительные вычисления

1 Введение

В статье предложен алгоритм переразмещения матриц из стандартного размещения в блочное, использующий не более $O\left(max(B_1, B_2)\right)$ дополнительной памяти, где B_1, B_2 — размеры блока. Кроме того, разработан алгоритм перехода из стандартного в двойное блочное размещение. Предлагаемый алгоритм является развитием метода обхода циклов перестановки, представленного в [2] для решения задачи in-place транспонирования матрицы. В данной работе рассматривается применение этого метода к задаче приведения матрицы к блочному/двойному блочному размещению.

Реализация предлагаемого алгоритма представлена в виде библиотеки компилятора языка Си системы ОРС (Оптимизирующая Распараллеливающая Система) [3]. Численные эксперименты показали, что предлагаемый алгоритм быстрее, чем наивный. Эксперименты также были проведены на задачах умножения матриц, алгоритма Флойда, QR [1] разложения матрицы, в которых использовались стандартное, блочное и двойное блочное размещения.

2 Задача поиска порождающих адресов циклов перестановки

В основе предлагаемого алгоритма лежит понятие цикла перестановки. Отображение f одного размещения в другое порождает некоторую перестановку P_f. Для выполнения переразмещения требуется произвести "сдвиг по циклу", порожденному адресом i:

$$i \to f(i) \to f(f(i)) \to \ldots \to f(\ldots f(i) \ldots) \to i$$

Для задачи перехода из стандартного размещения к блочному таких циклов может быть несколько. Возникает задача обнаружения всех циклов перестановки. Предлагаемый алгоритм решает эту задачу, выдавая список порождающих адресов всех циклов перестановки.

2.1 Индексная функция

Отображение строчного представления в блочное рассматривается в форме дискретной функции $f : [0;\ N_1N_2 - 1] \to [0;\ N_1N_2 - 1]$ ($d_1 \backslash d_2$ - число строк\столбцов в текущем блоке):

$$f(i) = \frac{i}{B_1 N_2} B_1 N_2 + \frac{i \bmod N_2}{B_2} d_1 B_2 + \frac{i \bmod B_1 N_2}{N_2} d_2 + \\ + i \bmod N_2 \bmod B_2 \quad (1)$$

Особенность соответствующей перестановки состоит в том, что во всех блочных полосах кроме, может быть, последней циклы "распределены" одинаково. Это позволяет свести поиск порождающих адресов на интервале $[0;\ N_1 N_2 - 1]$ к поиску на интервале $[0;\ B_1 N_2 - 1]$.

2.2 Алгоритм поиска порождающих адресов циклов

В данной работе предлагается следующее определение цикла, порожденного адресом i:

$$C(i) = \{i_{j+1 \bmod L} = f(i_j)\}_{j=0}^{L-1}, i_0 = i, L \in \mathbb{N} - \text{длина цикла}$$

Основная идея предлагаемого алгоритма заключается в последовательном обходе всего диапазона адресов $[0;\ B_1 N_2 - 1]$ и проверке для каждого адреса, порождает ли он новый цикл. Для различения новых циклов от пройденных будем использовать критерий ($\mathbb{P}(i)$ — множество адресов, пройденных к моменту достижения адреса i):

Цикл, порожденный адресом i, является пройденным

$$\Leftrightarrow \exists\, p \in \mathbb{N}\ \ \exists\, k \in \mathbb{P}(i) : f^p(i) = k \quad (2)$$

Переходя по циклу $C(i)$ к каждому последующему адресу, будем проверять, принадлежит ли этот адрес пройденной части интервала, и, если принадлежит, то текущий цикл определяется как пройденный, его обход прекращается, и производится переход к следующему адресу диапазона. Если при обходе был встречен исходный адрес i, то этот адрес помечается как порождающий, а цикл — как пройденный.

2.3 Оптимизация поиска порождающих адресов циклов

Особенностью предлагаемого метода является неоптимальное использование кеш-памяти процессора. При выполнении переразмещения, а именно при обращении к адресу $f(i)$, всегда будут происходить

кеш-промахи. Как правило, адреса i и $f(i)$ указывают на участки памяти, расположенные на достаточно большом расстоянии, из-за чего они никогда не будут загружены в кеш-память одновременно. Более того, т.к. данные загружаются в кеш-память т.н. кеш-линиями, то из всей загруженной порции используется лишь одна ячейка. Остальная ее часть останется неиспользованной, после чего нужно будет загружать другую кеш-линию.

Для повышения эффективности работы с кеш-памятью предлагается использовать особенности блочного размещения. В общем случае циклы перестановки являются "векторными", и обладают такой характеристикой как ширина:

$$w(C(i)) = \{\, T = max_{t \in \mathbb{N}}(t) + 1 : \forall\, p \in \mathbb{N} : f^p(i+t) = f^p(i) + t\}$$

Использование этой особенности дает значительные преимущества. Во-первых, можно перемещать несколько элементов за один шаг прохода по циклу — кеш-линии используются эффективнее. Во-вторых, зная ширину цикла, можем после его обхода перейти не на $i+1$-ый адрес, а сразу на $i+w$-ый, что ускоряет поиск не менее, чем в w раз.

Было сделано полезное наблюдение о минимальной ширине циклов:

$$w_{min} = \text{НОД}(N_2, B_2)$$

Более широкие циклы можно обнаруживать при поиске порождающих адресов, проверяя условия $f(i + w_{min}) = f(i) + w_{min}$ и $f^{-1}(i + w_{min}) = f^{-1}(i) + w_{min}$ при обнаружении каждого нового цикла.

Важный практический вывод, основанный на изучении свойств циклов: чем больше будет НОД у параметров N_2 и B_2, тем выше будет производительность алгоритма.

2.4 Оценка сложности алгоритма поиска

Основной операцией алгоритма поиска порождающих адресов является вычисление следующего адреса по циклу. Очевидно, ее сложность оценивается $O\,(1)$. Определим, сколько раз нужно произвести эту операцию для нахождения порождающих адресов всех циклов.

Теорема 1 *Математическое ожидание числа шагов алгоритма, которое необходимо сделать для обнаружения порождающих адресов всех циклов, не превышает:*

$$\frac{B_1 N_2}{w_{min}} \left(\ln \left(\frac{B_1 N_2}{w_{min}} \right) + \lambda \right), \lambda \approx 0.57$$

Отсюда — оценка для временной сложности алгоритма поиска порождающих адресов:

$$O\left(\frac{B_1 N_2}{w_{min}} \ln\left(\frac{B_1 N_2}{w_{min}}\right)\right)$$

3 Эксперименты

Эксперименты проводились на компьютере с ОС Windows 7 x64, процессором Intel Core i5 3470, оперативной памятью объемом 8 Гб.

Алгоритм	Конфигурация запуска	Наивный алгоритм (с)	Предл. алгоритм (с)	Уск.
Переразмещение из стандартного размещения в блочное	N=5000, B=512	0.624	0.140	4.4
	N=5000, B=128	0.633	0.062	10.2
	N=7500, B=512	1.418	0.370	3.8
Переразмещение из стандартного размещения в двойное блочное	N=5000, B=512, D=64	0.623	0.178	3.5
	N=5000, B=128, D=64	0.623	0.100	6.2
	N=7500, B=512, D=64	1.410	0.457	3.1

Таблица 1: Сравнение производительности программных реализаций предлагаемого и наивного алгоритмов переразмещения матриц

В таблице 1 производится сравнение времени работы наивного (копирование в буферный массив) и предлагаемого алгоритма переразмещения матриц. Данные показывают, что предлагаемый алгоритм позволяет переразместить матрицу значительно быстрее. Также были произведены эксперименты для ряда блочных программ (умножение матриц, алгоритм Флойда и QR-разложение матрицы). В каждом из них сравниваются две версии одного алгоритма, одна из которых использует стандартное размещение матриц, а другая - блочное размещение. Переразмещение матриц из стандартного в блочное и наоборот производилось с помощью реализованных функций переразмещения матриц, использующих предлагаемый алгоритм. Во всех экспериментах время, занимаемое переразмещением, было включено в общее время расчетов. Программы с блочным размещением показали время на 20-80 процентов меньшее, чем у аналогов со стандартным размещением.

4 Заключение

Приведенные численные эксперименты доказывают, что предлагаемый алгоритм переразмещения матриц является более эффективным чем наивный алгоритм переразмещения матриц и позволяет получить ускорение в 4-10 раз в зависимости от размера матрицы и блока. Реализованная в рамках системы OPC высокопроизводительная программная реализация данного алгоритма может использоваться разработчиками блочных алгоритмов, которые применяют блочное или двойное блочное размещение матриц.

Список литературы

1. *Bischof C.*, *Loan C. V.* The WY Representation for Products of Householder Matrices // SIAM Journal on Scientific and Statistical Computing. — 1987. — Т. 8, № 1. — s2—s13. — DOI: 10.1137/0908009. — eprint: http://dx.doi.org/10.1137/0908009. — URL: http://dx.doi.org/10.1137/0908009.
2. *Fred G. Gustavson T. S.* In-Place Transposition of Rectangular Matrices // Applied Parallel Computing. State of the Art in Scientific Computing. —, 2007. — Июнь. — Т. 10. — С. 560—569. — ISSN 0302-9743. — DOI: 10.1007/978-3-540-75755-9_68.
3. OPC Оптимизирующая распараллеливающая система. — 2017. — URL: www.ops.rsu.ru (дата обр. 25.01.2017).

Научное издание

Языки программирования и компиляторы — 2017

Редактор Д. В. Дубров
Компьютерная вёрстка Д. В. Дуброва

Подписано в печать 29.03.2017 г. Заказ № 5711.
Бумага офсетная. Печать офсетная. Формат 60×84 1/ 16.
Усл. печ. лист. 16,39. Уч. изд. л. 16,0. Тираж 300 экз.

Отпечатано в отделе полиграфической, корпоративной и сувенирной продукции
Издательско-полиграфического комплекса КИБИ МЕДИА ЦЕНТРА ЮФУ.
344090, г. Ростов-на-Дону, пр. Стачки, 200/1, тел (863) 247-80-51.