Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

Л.В. Городняя

МЕТОДЫ ДЕКОМПОЗИЦИИ ПРОГРАММ

Препринт
182

НОВОСИБИРСК 2018

Рецензент - к. физ.-мат. наук, Ф.А. Мурзин

Препринт посвящен анализу основных методов декомпозиции программ, поддержанных современными языками и системами программирования. Представлены результаты анализа особенностей техники выделения и представления процедур, функций, макросов, фрагментов и других компонентов программ в рамках разных парадигм программирования и отдельных приёмов обработки программ при компиляции. В центре внимания методы разделения представления программы на схему и её наполнения с акцентом на извлечение наполнений и приведение их к форме автономно развиваемых компонентов. Для иллюстрации использованы фрагменты ряда языков функционального и императивного программирования, макро-техника и языки управления заданиями. Содержание препринта представляет интерес для системных программистов, студентов и аспирантов, специализирующихся в области системного и теоретического программирования, и для всех тех, кто интересуется проблемами современной информатики, программирования и информационных технологий.

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект № 18-07-01048-а.

Siberian Branch of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

L.V. Gorodnyaya

A decomposition paradigm in programming

Preprint
182

Novosibirsk 2018

Reviewer Dr. F. A. Murzin

The preprint is devoted to the analysis of the main methods of program decomposition supported by modern programming languages and systems. The authors analyze specific features of a technique of isolation and presenting procedures, functions, macros, fragments and other program components within the framework of different programming paradigms and individual methods of program processing during compilation. The focus is on methods of separating the presentation of the program into a scheme and its contents, with an emphasis on extracting the contents and transforming them into autonomously developed components. Fragments of a number of functional and imperative programming languages, macro-techniques and task management languages are provided as illustrations.

The preprint will be of interest to system programmers, undergraduate and graduate students specializing in system and theoretical programming, and for all those interested in the problems of modern computer science, programming and information technology.

The research was supported by the Russian Foundation for Basic Research (RFBR) of the Siberian Branch of the Russian Academy of Sciences № 18-07-01048-a.

Введение

Задача декомпозиции программ в общем случае нацелена на создание программного инструментария, поддерживающего для доступной работоспособной программы, являющейся реализацией усложнённого комплекса определённых функций, извлечение отдельной конкретной функции, существование которой можно обнаружить с помощью ряда тестов в рамках заданного сценария применения исходной программы. Имеющийся инструментарий программирования позволяет представлять декомпозированные формы программ и интегрировать программные комплексы из специально созданных компонентов. Трудоёмкость создания компонентов примерно в 3-9 раз превышает трудоёмкость разработки отдельных программ близкого назначения. Возможность извлечения реализованных функций из готовых, отлаженных программ может стать основой более конструктивной технологии производства типовых компонентов для успешно освоенных областей приложения ИТ.

Вопросы декомпозиции программ наиболее подробно изучены в области теории схем программ [1] применительно к решению задач оптимизирующей компиляции и кодогенерации программ [2]. Обычно привлекается терминология теории графов [3]. Представление программы сопровождается информационно-логическим графом, отражающим связи между значениями переменных и последовательностями выполнения команд [4]. Основные понятия — перемещаемые участки программы, преобразования программы, критерии применимости преобразований, гарантирующие эквивалентность преобразованной исходной программы[5]. Функциональный подход к программированию ввёл возможности использования функционалов, способных применять и конструировать функции [6]. Методика смешанных или частичных вычислений добавила в этот круг понятие разметки и остаточной программы [7]. Существенное расширение пространства понятий дала методика вертикального слоения программ, предлагающая сводить программы к ряду сосредоточенных представлений рассредоточенных действий [8]. Расширение спектра парадигм программирования [9], развитие методов верификации программ [10] и рост актуальности разных моделей параллельных вычислений для ранее решавшихся задач [11], идеи мета- и супер-программирования [12] обогащают свод знаний о программе преставлением примеров входных данных и допустимых сценариев применения программы, что позволяет полнее рассматривать задачи декомпозиции программ в современных условиях.

В данном препринте рассматриваются особенности решения задачи декомпозиции программ [13], полезные для конструктивной технологии программирования [14] на базе расширяющегося корпуса отлаженных программ [15].

1. Общее представление о декомпозиции программ

Созданию программы обычно предшествует и сопутствует значительная работа по изучению постановки задачи, решение которой предстоит выразить в виде программы. Решение новых сложных задач обычно выполняется сведением к более простым подзадачам или более общим надзадачам, имеющим готовые решения или более удобным для программирования их решений в виде компонентов, из которых определёнными методами строится работоспособная программа решения исходной задачи. Представленные в созданной программе решения могут быть несколько преобразованы такими методами из соображений эффективности или из-за особенностей используемых средств программирования и аппаратуры. В результате полученная программа может не сохранять представления границ компонентов и динамики их изменения при развитии постановки исходной задачи. Представление таких границ может утратить соответствие изменившимся условиям применения программы решения задачи.

Развитие постановки задачи - естественный процесс, особенно если решение задачи и программирование выбранного решения требуют заметного времени. Как правило можно выделить стабильные и модифицируемые на уровне задачи части, схемы её решения и программы решения. Заметная часть таких частей может быть сведена к готовым решениям, возможно доступным как встраиваемые компоненты. Некоторые части могут рассматриваться по аналогии с известными решениями в штатных программных инструментах, но обычно без возможности извлечения, другие требуют независимого решения и программирования. Во всех таких случаях полезна возможность использовать декомпозированное представление постановки задачи, доступного инструментария, выбираемых решений и отлаживаемых программ.

Современные языки и системы программирования предоставляют широкий спектр средств представления декомпозированных программ, включая средства включения готовых компонентов. Подразумевается, что схема декомпозиции программы в таком случае складывается интуитивно, до перехода к программированию и частично представляется средствами выбранного языка программирования по

возможности. Сложности возникают при развитии постановки задачи и вытекающей, труднопредсказуемой, необходимости изменения выбранных решений и схемы декомпозиции программ, возможно с пересмотром границы между стабильными и изменяемыми частями. Для стабильных частей важно гарантировать сохранение их свойств, для изменяемых -- отсутствие воздействий на смежные части.

Некоторый круг таких работ допускает автоматизацию, обычно встроенную на внутренних уровнях обработки программ в компиляторах. Что-то может быть выполнено с учётом рекомендаций по реинжинирингу и производственному программированию. Решение проблем такого рода затруднено отсутствием единого подхода, поддерживающего синхронизованную декомпозицию представлений постановок задач, методов их решения и отлаживаемых программ решения. Полезна возможность согласованной декомпозиции программ на уровне текста, структуры данных и исполнимого кода без заметного нарушения характеристик производительности программ. Техника выполнения таких работ примыкает к использованию средств отладки, тестирования и верификации программ. Эксплуатационные характеристики таких средств на современном оборудовании уже допускают их массовое применение, подкреплённое серьёзной теоретической базой.

2. Декомпозированные формы программ

Первое приближение к созданию общей методики декомпозиции и факторизации программ дают прецеденты оптимизирующих преобразований программ в компилирующих системах программирования, обоснованных исследованием критериев применимости и сохранением эквивалентности. Доминирующая техника декомпозиции программ в системах программирования основана на анализе функционально-модульной структуры, допускающей раздельную компиляцию модулей и отладку программы по отдельным функциям, дополненную средствами макрогенерации текстов, структур данных и кодов. Выбор модулей обычно подчинён минимизации контактов с другими модулями и характеризуется сравнительно ясным описанием целей применения модуля, порой имеющего достаточно сложную структуру. Представление в программе макросов, функций, типов данных и процедур обычно допускает взаимодействия по общей памяти с частичным контролем типов значений и переменных или соответствия определённым условиям или тестам. Такое представление может изначально обладать подобием

описанию решаемой задачи. По мере развития постановки задачи и отладки программы их подобие обычно утрачивается.

Переход к рассмотрению в качестве программ таких долгоживущих программ, как системы программирования, обычно приводит задачу декомпозиции к выделению внутренних и промежуточных форм представления обрабатываемых данных. Каждая форма допускает переход к другой форме и обладает сравнительно независимым, но не простым, комплектом средств её анализа и преобразования, включая конвертацию в другую форму. Обычно промежуточные формы не вполне эквивалентны, их выбор и реализация подчинены различным целям. Понимание таких форм обременено не только заметным квалификационным цензом, но и зависимостью от решений разработчика, не редко утрачиваемых наследниками и преемниками в обновлённых версиях программы, что придаёт технологии развития программ несколько мистический оттенок, страх исказить работающее непонятное.

3. Подходы к декомпозиции программ

Созданию программного инструментария, поддерживающего для доступной работоспособной программы, являющейся реализацией усложнённого комплекса определённых функций, извлечение отдельной конкретной функции, существование которой можно обнаружить с помощью ряда тестов в рамках заданного сценария применения исходной программы, следует предпослать обзор подходов к декомпозиции программ, показать взаимосвязь техники преобразования программ и структур данных, определяющих цель и результат их анализа и преобразования.

Схематология. В области теории схем программ вопросы декомпозиции программ сведены к анализу графового представления взаимосвязей между действиями и данными, выраженных содержательной разметкой вершин и дуг графа. Обычно графовая форма имеет символьный эквивалент, более удобный для публикаций и текстовой обработки. Одной графовой форме могут соответствовать разные символьные формы, что даёт возможность языконезависимого исследования многих проблем. При анализе графа преимущественно выделяются линейные участки и гамаки — подграфы с одной входной вершиной и одной выходной, возможно не входящей в гамак, допускающие сведение к одной вершине. Таким образом удаётся декомпозировать программу на схему программы и иерархию ряда фрагментов наполнения схемы [1].

Компиляция программ. Применение схематологии программ к

решению задач оптимизирующей компиляции и кодогенерации программ обогащает представление логико-информационных зависимостей между действиями и данными оценкой частоты использования действий и времени существования данных [2]. Привлечение терминологии теории графов позволяет формулировать критерии преобразования программ в достаточно общей форме, без чрезмерной связи с понятиями используемых языков и машин [3]. Из представления программы выводится её информационно-логический граф, отражающий представленные в программе связи между значениями переменных и последовательностями выполнения команд [4]. Основные понятия — перемещаемые участки программы, эквивалентные преобразования программы, критерии применимости преобразований, гарантирующие частичную эквивалентность преобразований [5]. Выделяются участки с однократным именованием значений (SSA-формы, участки с однократным присваиванием), что делает достижимой экономию памяти благодаря возможности учёта независимости данных по времени. Любой участок программы в принципе можно представить как композицию SSA-форм, используя новые имена для повторных присваиваний или декомпозируя участок на иерархию вложенных участков, что можно рассматривать как неявное переименование. Классификация участков по частоте использования лежит в основе оптимизации времени исполнения программы, исключения константных вычислений, запроцедуривания совпадающих выражений, раскрытия процедур, перемещения действий на минимальные участки повторяемости, преобразования рекурсии в итерацию или обратно.

Функциональный подход к программированию позволяет манипулировать участками программы как данными, вводит возможность использования функционалов, способных применять и конструировать функции. Возникает декомпозиция программы на иерархию функций разного уровня, специализация функций по категориям, обладающих разными схемами организации вычислений, и классификация значений, представление которых допускает полный контроль их соответствия программируемой обработке. [6].

Методика смешанных или частичных вычислений добавила в этот круг понятие разметки и остаточной программы. Таким образом появляются рассредоточенные шаги выполнения программы. Способы разметки могут быть весьма различны: от входных или выходных данных до произвольной пометки отдельных действий в программе или маршрутов в её графовой форме. При исследовании потенциала смешанных вычислений было установлено, что они дают общую

методику конструирования частных или специализированных проекций программы, эквивалентных преобразований программ, их обобщения и в конце концов перехода от интерпретатора языка программирования к его компилятору. Узким местом развития такой общей методики оказалась так называемая теорема Суслова, согласно которой для поддержки универсальных смешанных вычислений нужны языки, допускающие факторизацию программ относительно произвольной разметки, т.е. любую композицию выделенных конструкций программы надо уметь делать параметром некоторой функции или процедуры [7].

Предложенная А.Л. Фуксманом методика **вертикального слоения программ**, предлагающая сводить программы к ряду сосредоточенных представлений рассредоточенных действий, существенно расширила пространства понятий для декомпозиции и факторизации программ. Вместо участков, выделяемых по принципу соседства, можно оперировать конструкциями, выделяемыми по общности смысла или понимания. Части таких конструкций могут быть синхронизованы по мере сборки из них исполнимого кода программы техникой макрогенерации или открытой постановки процедур. Допустима и реализация на многопроцессорных конфигурациях [8].

Происходящее по мере прогресса аппаратуры, информационных технологий и сфер их применения заметное расширение спектра **парадигм программирования** показало взаимосводимость разных систем понятий, что позволяет решать проблемы декомпозиции программ через использование конвертации на подходящие изобразительные средства, как фактически и происходит при многоуровневой компиляции программ. Императивно-процедурное программирование приспособлено к выделению фрагментов изменения значений отдельных переменных. Функциональное программирование побуждает к выделению отображений области определения функции в область её значений. Логическое программирование позволяет не придавать заметного значения порядку вычислений в пользу недетерминированному перебору ряда формально равноправных вариантов. Объектно-ориентированное программирование, наследуя почти все эти основные возможности декомпозиции программ, добавляет к ним классификацию фрагментов по классам объектов и пространствам имён. Всё это в разной мере востребовано на разных фазах жизненного цикла программ. Парадигмальная декомпозиция позволяет смягчать переходы на очередные фазы и этапы разработки программы [9].

Методы верификации программ работают с привлечением

логических условий, которые могут быть заданы независимо от программы или выведены из неё. Комплект логических условий и структура их представления могут выполнять роль параметров разметки программы для её декомпозиции по методике смешанных вычислений. Кроме того, техника верификации, отладки и тестирования программ даёт ряд вспомогательных подходов к организации вычислений и выделению позиций для управления выбором правила обработки программы [10].

Модели параллельных вычислений вносят ещё большее разнообразие причин актуальности детализации и утончения конструкций, используемых для ранее решавшихся задач, особенно в сфере мелкозернистого параллелизма, трудно выражаемого на стандартных языках программирования [11].

Идеи **мета- и супер-программирования** [12] обогащают и пополняют информационную картину преставлением примеров входных данных и допустимых сценариев применения программы, что приводит к более полному рассмотрению задачи декомпозиции программ в современных условиях.

Полный спектр особенностей решения задачи декомпозиции программ приводит к целесообразности объединения в единой информационной среде возможностей разных теорий, языков и систем программирования и программных инструментов[13].

Инструментальная поддержка общих решений задачи декомпозиции программ полезна для формирования **конструктивной технологии** программирования, работающей подобно дифференциальному исчислению в математическом анализе. Конкретно, такая технология может выделять работоспособные компоненты из готовых программ [14]. Для таких экспериментов существует обширный полигон в виде расширяющегося **корпуса отлаженных программ** [15].

Общие методы приведения программ к декомпозированной форме подробно описаны в работе, рассматривающей уровень изученности задачи в качестве важнейшего критерия классификации программы решения задачи. Следующий уровень декомпозиции получается на основе анализа спектра видов семантических систем, образующих решение [16]. Отдельно изучена целесообразность парадигмальной декомпозиции представлений программ, гарантирующей ортогональность при комплексации программ из независимо изготовляемых модулей. В этом отношении ряд подзадач могут иметь несколько взаимозаменяемых решений для разных парадигм и понимания качества программ [17]. Вопросы выделения типовых

компонентов программ для повышения производительности программ, конструируемых из частей, обладающих удостоверенными свойствами, освещены в работе, рассматривающей в качестве компонентов наравне с текстом или исполнимым кодом любые спецификации и тесты. [18]. Практичность синтаксически ориентированных преобразований показана в работе, нацеленной на повышение уровня средств определения языков программирования [19]. Возможность автоматизации декомпозиции и границы факторизации программ изучены в работе, связанной с исследованием проблем усовершенствования систем параллельного программирования, ряд трудностей которого связан с неотделённостью мелко-зернистых аспектов понятий в языках программирования. Например, средства представления структур данных обычно сращены с последовательностью вычисления элементов структуры. Другая сложность связана с проблемой представления синхронизации процессов [20]. Более подробное изложение представлено в работах [21,22].

4. Декомпозиция символьных выражений

Графовые формы, используемые в разных подходах к декомпозиции программ при их анализе, интерпретации и компиляции, можно без принципиального нарушения общности рассматривать как эквиваленты символьных выражений, таких как в языке Lisp. Символьные выражения представляют собой бинарные деревья, узлы которого содержат левую и правую части, а листья представлены атомами. Выделяются списки -- бинарные деревья, движение по правым частям которых завершается специальным атомом, символизирующим пустой список. Виды конструкций, представляющих программы, распознаются по головным элементам списков. При необходимости в более общих графах можно добавить специальные распознаватели символьных моделей для нестандартных структур данных или конструкций.

Lisp: Символьное выражение	*Примечание*
(**defun** LENGTH (L) (**prog** (U V) (**setq** V 0) (**setq** U L) A (**cond** ((**null** U) (**return** V))) (**setq** U (**cdr** U)) (**setq** V (+ 1 V)) (**go** A))) (**LENGTH** '(A B C D)) (**LENGTH** '((X . Y) A CAR (N B) (X Y Z)))	Объявление функции LENGTH, реализуемой в императивном стиле с двумя рабочими переменными. Инициирование рабочих переменных. Меткой «A» помечена проверка завершения и выработка результата. Шаг продолжения функции. Переход на метку «A». Применение функции LENGTH даёт 4. Применение функции LENGTH даёт 5.

Пример 1. Символьное выражение для представления программы на языке Lisp

Любая позиция в символьном выражении может быть выделена селектором, являющимся композицией функций CAR-CDR, что можно рассматривать как поддержку произвольной разметки. Такие композиции можно кодировать как регулярные выражения вида "(A*D*)*" или "(AnDm)*", где "n" и "m" -- кратность вхождения "A" и "D" соответственно. Каждая композиция представляет определённый маршрут в дереве. Прохождение маршрута можно сопровождать выявлением и накоплением характеристик маршрута или элементов дерева. Маршрут завершается на некотором узле дерева, являющегося корнем выделенного таким образом поддерева. В примере 1 маршрут "AD" завершается выбором имени функции "LENGTH", маршрут "ADD" -- выбором списка её аргументов "(L)", маршрут "ADDD" -- телом определения функции с помощью формы "PROG", являющейся встроенной в язык Lisp моделью императивных вычислений, маршрут "DDD" — список, содержащий тело определения, а маршрут "DDDD"

— пустым списком.

Функция GENSYM вырабатывает уникальные атомы, которые можно связывать с выделенными поддеревьями функцией CONS или PAIRLIS и сохранять в ассоциативном списке. Деструктивные функции RPLACA и RPLACD позволяют заменить значение из левой или правой части узла дерева на заданное значение, например на атом, связанный с удаляемым поддеревом. Таким образом получается декомпозиция символьного выражения на схему и наполняющие её фрагменты. Связанные с фрагментами атомы могут выполнять роль специальных фрагментных переменных. Можно определить правило интерпретации схем, выполняющее вызов значений фрагментных переменных как составных операторов или макросов. Полученное декомпозированное представление программы можно рассматривать как расслоение на схему и её наполнение

.

Схема	*Фрагменты*	*Примечание*
(defun LENGTH (L) (prog (U V) (#**G0001)** A (cond ((null U) (return V))) (#**G0002)** (go A)))	(#**G0001** (setq V 0) (setq U L)) (#**G0002** (setq U (cdr U)) (setq V (+ 1 V)))	#G0001 и #G0002 – атомы, созданные функцией GENSYM Линейный участок до цикла: **Инициирование** рабочих переменных. Линейный участок внутри цикла: шаг **продолжения** функции.

Пример 2. Результат декомпозиции символьного выражения из Примера 1.

Нужный результат декомпозиции может быть получен и без деструктивных функций техникой копирования пройденного маршрута. Это позволяет манипулировать одновременно с исходным и расслоённым представлением. Можно предварительно сделать копию символьного выражения функцией COPY. Функция NSUBST позволяет в произвольном дереве заменять атомы на заданные фрагменты, что даёт возможность восстановления исходного символьного выражения

по мере необходимости. Этот же результат даёт функция SUBST без потери схемы символьного выражения. Значения атомов в ассоциативном списке всегда можно найти функцией ASSOC, которая по заданному атомы или фрагменту находит в ассоциативном списке соответствующий фрагмент. Эта функция может быть применена и для выполнения более сложных преобразований символьных выражений. Например, при размещении фрагментов в ассоциативном списке можно с помощью функции EQUAL проверить наличие уже размещённых копий фрагмента и вместо нового связывания использовать более ранний атом, что даёт эффект оптимизации повторных вычислений.

Расслоённое представление программы можно организовать как работу с комплексом ассоциативных списков над возможно общим списком атомов или просто списками фрагментов, расположенными в едином порядке для применения техники отображений с помощью функций вида MAP, MAPCAR, MAPFN, MAPLINE (см. Приложение 1). Отдельный ассоциативный список "сосредоточенно" представляет фрагменты общего по смыслу действия. Эти фрагменты "рассредотачиваются" при восстановлении или генерации символьного выражения. При генерации техника отображений позволяет для каждой позиции схемы символьного выражения можно задавать свои правила сборки результата.

Схема	***Слой** переменной **V***	***Слой** переменной **U***	*Примечание*
(defun LENGTH (prog (U V) **(#G0001)** A (cond ((null U) (return V))) **(#G0002)** (go A)))	(#**G0001** (setq V 0)) (#**G0002** (setq V (+ 1 V)))	(#**G0001** (setq U L)) (#**G0002** (setq U (cdr U)))	#G0001, #G0002 — атомы для **синхронизации слоёв** До цикла: **инициирование** переменных. Внутри цикла: шаг **продолжения** функции.

Пример 3. Расслоение фрагментов декомпозиции символьного выражения из Примера 2.

Рассматривая грамматику языков программирования (ЯП) как программу конструирования анализаторов программ или генераторов кода, можно префиксы правильных текстов программ использовать как параметры селекторов в символьном выражении, представляющем грамматику. Это даёт возможность смягчить зависимость работы с определением ЯП от наименований грамматических понятий.

Исходный текст на C	*Символьное выражение графовго представления при компиляции*

// Example.c
#include <stdio.h>
 	
int global;
 	
int main
(int argc,
char *argv[ ]) {
 	
int param = 1;
 	
myPrint (param);
 	
return 0;
}
	; Комментарии не имеют представления
	; Макросы не имеют представления
	
	(**VarDecl** global 'int)
	
	(**FunctionDecl** main (int, char *)
	(**ParmVarDecl** argc 'int)
	(**ParmVarDecl** argv 'char** 'char**)
	(**CompoundStat**
	(**DeclStat** (**VarDecl** param 'int)
	(**IntegerLiteral** 1 'int))
	(**CallExpr** 'void
	(**ImplicitCastExp**r 'void '(*) '())
	(**DeclRefExpr** 'param 'int))
	(**ReturnStmt** (**IntegerLiteral** 0 'int))
	)
	)

Пример 4. Символьное выражение для Clang-представления примера программы на Си.

Конструирование анализаторов программ по символьному представлению грамматики используемого ЯП допускает автоматизацию. Реализацию генератора кода можно рассматривать как перевод программы на другой язык и автоматизировать, используя грамматики перевода, причём переводов может быть более одного, каждый из которых можно рассматривать как отдельный слой. Взаимосвязь слоёв может быть выражена при автоматизации на уровне мета-определения.

По префиксу "int main" в примере 4 можно выделить конструкцию

"(FunctionDecl main (int, char *)", а по префиксу "int main (int argc," -- конструкцию "(ParmVarDecl argc 'int)" без претензий на владение терминологией компилятора. При таком методе использование префиксов правильных текстов в качестве селекторов позволяет из определения ЯП и его реализации выделять востребованную часть.

Методика смешанных и частичных вычислений позволяет из общей конструкции интерпретатора или компилятора ЯП выделять проблемно-ориентированную проекцию. Функциональный подход позволяет формировать замыкания выделяемых функций и оперировать временем фактического выполнения действий в стиле отложенных или опережающих вычислений.

5. Требования к инструментальной среде

Поддержка полного свода методов декомпозиции программ предъявляет особые требования к организации инструментальной среды. Кроме комплексного применения систем программирования, отладки, тестирования и верификации программ возникает необходимость пересмотра состава доступных форм представления программ и средств манипулирования выбранными представлениями. Прежде всего, представление программы должно быть обладать триадностью и многосортностью. Программа одновременно представлена текстом, структурой данной и исполнимым кодом, причём структура данных имеет графовый вид наряду с формульным эквивалентом. Средства обработки программ должны обладать полисемантическим определением, причём определения могут быть не только встроенными, но и программируемыми. Технически в инструментальной среде следует поддержать следующие средства обработки программ:

1) идентификация синтаксически, семантически или прагматически различимых позиций;

2) доступ к любой различимой позиции;

3) выделение фрагмента определённой категории (элемент, поддерево, путь доступа или др.) по любой различимой позиции;

4) проверка достижимости заданной позиции;

5) выбор позиций, идентифицирующих фрагменты, удовлетворяющие заданному условию;

6) формирование маршрута продвижения по программе от одной позиции до другой;

7) преобразование фрагмента в определение функции или параметр заданной функции или контекст вызова функции;

8) преобразование программы в схему с вызовами функций, эквивалентных выделенным фрагментам;

9) проверка сохранения функциональных характеристик программы после преобразования;

10) выполнение программы с помощью разных методов интерпретации и/или компиляции.

Этот список имеет предварительный характер. Предполагается его продолжение с учетом методов вертикального слоения программ и преобразования программ, ориентированных на параллельные вычисления.

Заключение

Приходится делать вывод, что при наличии многообразия средств представления декомпозированных программ, не существует простенького решения задачи декомпозиции готовых программ в общем случае. Предварительные эксперименты можно выполнить на материале символьных выражений с использованием систем производственного функционального программирования, таких к Clisp или Cmucl, поддерживающих средства основных и фундаментальных парадигм программирования.

Создание программного инструментария, поддерживающего для произвольной доступной работоспособной программы, являющейся реализацией сложного комплекса известных функций, извлечение из неё отдельной конкретной функции, может быть выполнено на базе дополнительных экспериментальных исследований с использованием комплекса из ряда парадигмально различных систем программирования, систем отладки, тестирования и верификации программ над корпусом свободно распространяемых программных средств.

Литература

1. Котов В. Е., Сабельфельд В. К. Теория схем программ.— М.: Наука. Гл. ред. физ.-мат. лит., 1991.— 248 с.— ISBN 5-02-013974-2.
2. Ершов А.П., Кожухин Г.И., Поттосин И.В. Руководство к пользованию системой АЛЬФА. – Новосибирск: Наука, 1968. – 179 с.
3. Касьянов В.Н., Евстигнеев В.А. Графы в программировании: обработка, визуализация и применение. − С-Пб.: ЕХП-Петербург, 2003, − 1104 с.

4. Поттосин И.В. Система СОКРАТ: Окружение программирования для встроенных систем. − Новосибирск, 1992. − 20с. (Препр./РАН. Сиб. отд-ние. ИСИ; N11)
5. Иткин В.Э. Вопросы конструирования устойчивых алгоритмов на базе многовыходных модулей //Актуальные вопросы технологии программирования. Ленинград, 1989 (совм. с Котляровым В.П.)
6. *mcCarthy J.* LISP 1.5 Programming manual / J. mcCarthy. The mIT Press, Cambridge, 1963. 106 p
7. Ершов А.П. Смешанные вычисления: потенциальные приложения и проблемы исследования. // Тезисы докладов и сообщений. Всесоюзная конференция "Методы математической логики в проблемах искусственного интеллекта и систематическое программирование", ч.2. − Вильнюс, 1980. − с. 26-55.
8. Фуксман А.Л. Технологические аспекты создания программных систем - М. : Статистика, 1979. – 183 с.
9. Городняя, Л.В. Парадигмы программирования: анализ и сравнение / Из-во СО РАН, РФФИ 17-11-00042. 216 с.
10. Савенков К. Верификация программ на моделях. / Курс ВМК МГУ имени М.В.Ломоносова. http://savenkov.lvk.cs.msu.su/mc/lect02.pdf
11. *Воеводин В. В.* Параллельные вычисления. / В. В. Воеводин, Вл. В. Воеводин. – СПб.: БХВ-Петербург, 2002. 608 с.
12. Климов А.В, Романенко С.А. Введение в суперкомпиляцию и Краткая история суперкомпиляции в России
13. Городняя Л.В. Подход к декомпозиции систем программирования для пошагового их определения. // Трансляция и оптимизация программ. − Новосибирск, 1984.
14. Уоткинс Д., Хаммонд М., Эйбрамз Б. – Программирвоание на платформе .Net. − М. Вильямс, 2003. − С. 367.
15. Лавров С.С. Расширяемость языков. Подходы и практика. // Прикладная информатика. 1984. Вып. 2. − 17-23.
16. Городняя Л.В. Подход к ограничению новизны в программистских экспериментах. // Препринт N 357. Надзаг ВЦ СО АН СССР, Новосибирск, 1982. − с.29.
17. Городняя Л.В. Парадигмальная декомпозиция определения языка программирования // Научный сервис в сети Интернет: труды XVIII Всероссийской научной конференции (19-24 сентября 2016 г., г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2016. — С. 115-127. : http://keldysh.ru/abrau/2016/21
18. Городняя Л.В. Банк улучшаемых компонентов информационных систем. //Новосибирск, ИСИ СО РАН. 2005. Препринт 130. 31 с.

19. Городняя Л.В. Резервы синтаксически ориентированного конструирования систем программирования. //Научный сервис в сети Интернет: труды XIX Всероссийской научной конференции (18-23 сентября 2017 г., г. Новороссийск). − М.: ИПМ им. М.В.Келдыша, 2017. С. 120-129. : http://keldysh.ru/abrau/2017/proc.pdf
20. Городняя Л.В. О проблеме автоматизации параллельного программирования // В сборнике Международной суперкомпьютерной конференции «Научный сервис в сети Интернет: многообразие суперкомпьютерных миров » http://agora.guru.ru/abrau2014 с. 191-196.
21. Городняя Л.В. Резервы синтаксического конструирования систем программирования. Электронные библиотеки. 2018. Т. 21. No 1
22. Городняя Л.В. Парадигмальный подход к факторизации определений языков и систем программирования. :https://system-informatics.ru/ru/articles

Приложение 1

Описание упоминавшихся функций языка Lisp.

ASSOC -- поиск узла, содержащего в ассоциативном списке связь атома с его определением.

Определение	*Применение*
(DEFUN assoc (x al) (COND ((equal x (CAAR al)) (CAR al)) ((QUOTE T) (assoc x (CDR al))))) ; Частичная функция — ; рассчитана на наличие ; ассоциации.	(assoc 'B '((A . (m n)) (B . (CAR x)) (C . w) (B . (QUOTE T)))) ; = (B . (CAR x)) (assoc '(B A) '((A . (m n)) (B . (CAR x)) (C . w) ((B A) . (QUOTE T)))) ; = ((B A) . (QUOTE T)))

CAR -- выбор левой части узла бинарного дерева.

CDR -- выбор правой части узла бинарного дерева.

CONS -- создание узла бинарного дерева из заданных левой и правой частей.

COPY -- создание копии символьного выражения, независимо размещённой в памяти.

EQUAL -- сравнение двух символьных выражений.

Определение	*Применение*
(DEFUN equal (x y) (COND ((ATOM x) (COND ((ATOM y) (EQ x y)) ((QUOTE T) (QUOTE NIL)))) ((equal (CAR x)(CAR y)) (equal (CDR x)(CDR y))) ((QUOTE T) (QUOTE NIL))))	(equal '(A (B)) '(A (B))) ; = T (equal '(A B) '(A . B)) ; = NIL

GENSYM -- создание уникального атома, отличного от всех других.

MAP -- отображение списков с помощью заданной функции и определённого конструктора результата.

Определение списка	*Применение*
(DEFUN map (cr fn xl) ; Поэлементное преобразование XL ; с помощью функции FN (COND ; Пока XL не пуст (xl (FUNCALL cr (FUNCALL fn (CAR xl)) ; применяем FN к голове XL (map cr fn (CDR xl))))))	(DEFUN lens (xl) (map #'CONS #'length xl)) ; Длины элементов xl.

MAPCAR -- отображение ряда списков с помощью заданной функции и с формированием списка результатов.

Определение - для одного списка	*Применение*
(DEFUN mapcar1 (fn1 xl) ; Поэлементное преобразование XL ; с помощью унарной функции FN (COND ; Пока XL не пуст (xl (CONS (FUNCALL fn1 (CAR xl)) ; применяем FN к голове XL (mapcar1 fn1 (CDR xl)) ; и переходим к остальным,)))) ; собирая результаты в список	(DEFUN next (xl) (mapcar1 #'1+ xl)) ; Очередные числа из xl (DEFUN 1st (xl) (mapcar1 #'CAR xl)) ; "Головы" элементов xl (DEFUN lens (xl) (mapcar1 #'length xl)) ; Длины элементов xl
Определение - для двух списков	*Применение*
(DEFUN mapcar2 (fn2 al vl) ; fn покомпонентно применить ; к соответственным элементам AL и VL (COND (xl (CONS (FUNCALL fn2 (CAR al) (CAR vl)) ; Вызов данного FN как бинарной функции (mapcar2 fn2 (CDR al) (CDR vl))))))	(mapcar2 #'+ '(1 2 3) '(4 6 9)) ; = (5 8 12) (mapcar2 #'* '(1 2 3) '(4 6 9)) ; = (4 12 27) (mapcar2 #'CONS '(1 2 3) '(4 6 9)) ; = ((1 . 4) (2 . 6) (3 . 9)) (mapcar2 #'EQ '(4 2 3) '(4 6 9)) ; = (T NIL ( ))

MAPFN -- обработка символьного выражения с помощью отображения списка заданных функций, результаты которых собираются в список.

Определение - для списка функций и их общего аргумента	*Применение*
(DEFUN mapfn (fl arg) (COND ; Пока список функций не пуст, (fl (CONS (FUNCALL (CAR fl) arg) ; применяем очередную функцию ; ко второму аргументу (mapfn (CDR fl) arg) ; и переходим к остальным функциям,)))) ; собирая их результаты в общий список	(mapfn '(length CAR CDR) '(a b c d)) ; = (4 a (b c d))

MAPLINE -- отображение списка функций и списка значений их аргументов в список результатов применения функции к соответствующему аргументу.

Определение - для потока функций и потока их аргументов	*Применение*
(DEFUN mapline (fl el) (COND (fl (CONS (FUNCALL (CAR fl) (CAR el)) (mapline (CDR fl) (CDR el))))))	(mapline '(length CAR CDR) '((a b c d) (c d) (3 4 5))) ; = (4 c (4 5))

NSUBST -- деструктивная подстановка значений атома вместо его вхождений в символьное выражение.

PAIRLIS -- связывание элементов двух списков в пары/узлы, размещаемые в ассоциативном списке для накопления будущих связей.

Определение	*Применение*
(DEFUN pairlis (x y al) (COND ((null x) al) ((QUOTE T) (CONS (CONS (CAR x) (CAR Y)) (pairlis (CDR x) (CDR y) al)))))	(pairlis '(A B C) '(u t v) '((D . y) (E . y))) ; = ((A . u) (B . t) (C . v) ; = (D . y) (E . y)) (pairlis '((A B) C) '(u (t v) z) '((D . y) (E . y))) ; = ((A B) . u) (C t v) ; = (D . y) (E . y))

SUBST -- создание преобразованной копии символьного выражения Z, в которой вхождения одного фрагмента Y заменены на другой фрагмент X.

Определение	*Применение*
(DEFUN subst (x y z) (COND ((equal y z) x) ((ATOM z) z) ((QUOTE T)(CONS (subst x y (CAR z)) (subst x y (CDR z))))))	(subst '(x . A) 'B '((A . B) . C)) ; = ((A . (x . A)) . C) (subst '(x . A) '(B C) '((A . B) . (B C))) ; = ((A . B) . (x . A))

RPLACA -- деструктивное изменение узла, заменяющее левую его часть на заданное значение. (rplaca x y) = (CONS y (CDR x)) без выделения памяти под результат.

RPLACD -- деструктивное изменение узла, заменяющее правую его часть на заданное значение. (rplacd x y) = (CONS (CAR x) y) без выделения памяти под результат.

Л.В. Городняя

МЕТОДЫ ДЕКОМПОЗИЦИИ ПРОГРАММ

Препринт
182

Рукопись поступила в редакцию 17.11.2018
Редактор Т. М. Бульонкова
Рецензент Ф.А. Мурзин

Подписано в печать 29.11.2018
Формат бумаги 60 × 84 1/16 Объем 1.71 уч.-изд.л., 1.88 п.л.
Тираж 50 экз.

Типография Оригинал-2, г. Бердск, ул. Олега Кошевого, 6, оф. 2
тел./факс: 8 (383) 328-32-38, (38341) 2-12-42, сот.: 8 913 987 77 67