САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Серебряков Сергей Николаевич

Схема сборки проектов с агрессивным переиспользованием порождений

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
к. ф.-м. н. Булычев Д. Ю.

Рецензент:
к. ф.-м. н. Чашников Н. В.

Санкт-Петербург
2013

SAINT PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Chair of Software Engineering

Sergey Serebryakov

Project build technique with aggressive generation reuse

Graduation thesis

Admitted for defence.
Head of the chair:
Ph.D., Professor Andrey Terekhov

Scientific supervisor:
Ph.D. Dmitry Boulytchev

Reviewer:
Ph.D. Nikolay Chashnikov

Saint Petersburg
2013

Оглавление

Введение

Сборка проекта в интегрированных средах разработки (Integrated Development Environment, IDE) — это процесс генерации низкоуровневых артефактов из исходных файлов высокого уровня (программного кода, ресурсов и т.п.) по правилам, описанным в проекте.

Кэш компиляции — это некоторая структура данных, которая отражает мгновенное состояние проекта с точки зрения его реализации на низком уровне (например, для языка Java это совокупность класс-файлов плюс информация о зависимостях, соответствиях класс-файлов исходным файлам и т.д.). Такое хранилище может быть использовано (и используется в современных средах разработки) в процессе инкрементальной компиляции.

Инкрементальная компиляция представляет собой следующую оптимизацию процесса сборки проекта: вместо полной компиляции всего проекта с нуля перекомпилируются только изменённые со времени последней компиляции файлы, а также зависимые от них, на которые повлияли изменения (а также зависимые от них, и так далее — можно представить это как обход графа зависимостей в ширину). Например, если в одном из исходных Java-файлов разработчик изменил сигнатуру некоторого метода, то нужно перекомпилировать не только этот файл, но и все файлы, где присутствует вызов этого метода (т.е. зависимые). В целях экономии вычислительных ресурсов информация о зависимостях подобного рода сохраняется в кэше компиляции, чтобы при следующем запуске сборки проекта можно было эффективно вычислять набор файлов, которые подлежат перекомпиляции. В этот же кэш попадают и класс-файлы — в случае, когда ни исходный файл, ни те файлы, от которых он зависит, не изменились со времени последней компиляции, ясно, что можно переиспользовать результат его последней компиляции — взять сохранённый класс-файл вместо того, чтобы тратить вычислительные ресурсы на компиляцию.

Кэш компиляции подвергается обновлению при каждой компиляции, чтобы находиться в актуальном состоянии, пригодном к использованию при следующем сеансе компиляции. Если над проектом работает несколько человек, у каждого из которых есть локальная рабочая копия репозитория, то у каждого “нарастает” свой локальный кэш компиляции, при этом бóльшая его часть у всех одинакова. В процессе такой коллективной

разработки случаются ситуации, когда состояние проекта в локальной копии разработчика существенно отличается от состояния, для которого был посчитан его локальный кэш компиляции. Тогда инкрементальная компиляция теряет свои преимущества и мало чем отличается от процедуры полной сборки проекта с нуля. Примерами таких случаев могут служить "cold start" — сборка проекта в ситуации отсутствия кэша вообще — или переключение между ветками ("branches") — например, основной и релизной ветками. Однако можно воспользоваться тем, что разработчиков всё-таки несколько и предложить способ переиспользования кэшей одних разработчиков другими, чтобы в таких ситуациях не приходилось каждый раз строить их с нуля.

В данной работе предлагается такой способ переиспользования кэшей. Рассматривается главным образом процесс компиляции исходных файлов на языке Java, однако полученные результаты являются достаточно общими для того, чтобы данный подход можно было применить к другим языкам программирования. Заметим, что описываемый подход применим не только к компиляции, но также и к другим вычислительно сложным процессам, которые получают на вход некоторый набор исходных файлов и выдают в качестве результата некоторый набор порождений (примером такого процесса может служить построение индексов).

Постановка задачи

В рамках дипломной работы были поставлены следующие задачи:

- Построить формальную модель предметной области.
- Построить аксиоматику, описывающую свойства модели, позволяющую доказывать нетривиальные утверждения и вместе с тем отражающую ограничения реального мира.
- Описать задачу инкрементальной компиляции в терминах построенной формальной модели, сформулировать и доказать соответствующую теорему.
- Сформулировать и доказать теорему о переиспользовании порождений.

Обзор существующих подходов

Сборка программного проекта — неотъемлемая часть процесса промышленного программирования. Для автоматизации и оптимизации процесса сборки научным и профессиональным сообществами были разработаны различные подходы.

Утилита `make`

`make`[1] — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка (linking) в исполняемые файлы или библиотеки. `make` — одна из самых широко распространённых утилит для отслеживания зависимостей, в значительной степени благодаря её включению в ОС Unix и ОС семейства BSD. Программа была создана Стюартом Фельдманом (Stuart Feldman) в 1977 году и описана в его работе [3]. Существует несколько версий `make`, основанных на оригинальной `make` или написанных с нуля, использующих те же самые форматы файлов и базовые принципы и алгоритмы, а также содержащие некоторые улучшения и расширения.

Утилита использует специальные `make`-файлы, в которых указаны зависимости файлов друг от друга и правила, определяющие действия для удовлетворения этих зависимостей. Решение о том, нужно ли перегенерировать целевые файлы (targets), принимается на основе информации о времени последнего изменения каждого файла. Если время последнего изменения исходных файлов или файлов-зависимостей больше соответствующего времени изменения целевых файлов, то `make` определяет и запускает необходимые программы для перегенерации целевых файлов, иначе сгенерированные ранее целевые файлы считаются актуальными (up-to-date). Таким образом, за счёт того, что избыточные перегенерации не производятся, экономятся вычислительные ресурсы.

Однако в случаях, когда файл меняется, а его время изменения остаётся в прошлом (т.е. предшествует текущему моменту), такой подход даёт сбой, и пользователь должен принудительно запустить полную переком-

[1] `http://ru.wikipedia.org/wiki/Make`

пиляцию. Примерами таких случаев могут служить восстановление более старой версии исходного файла или ситуация, когда файлы находятся на сетевой файловой системе, чьё локальное время не синхронизировано с машиной, на которой запускается `make`. Соответственно, если время последнего изменения файла оказалось в будущем, то это вызовет избыточную перекомпиляцию. Кроме того, перекомпиляция (тоже, разумеется, избыточная) будет произведена, даже если изменение состояло, например, в модификации комментария внутри исходного файла.

В работе [4] предпринята попытка формализовать подход к сборке, реализуемый утилитой `make`. Анализируется проблема безопасности использования `make` для инкрементальной компиляции как альтернативы полной сборке (brute-force build). Безопасность (safeness) инкрементальной компиляции формулируется следующим образом: пусть была произведена полная сборка, затем исходные файлы (и, возможно, `make`-файл) были изменены; тогда при выполнении некоторых условий результат произведённой после этого основанной на `make` инкрементальной компиляции эквивалентен результату повторной полной сборки, если бы такая была произведена. С целью формулировки этих условий для `make` строится семантическая модель. Ключевым результатом работы является сформулированный набор критериев, которым должны удовлетворять правила `make`-файла, чтобы обеспечить вышеупомянутую безопасность. Помимо этого, установлены условия, при которых `make`-файл может быть модифицирован с сохранением свойства безопасности.

Хотя сформулированные критерии и являются довольно интуитивными, полученный результат представляет собой формальное обоснование существующей практики использования `make`. Кроме того, показано, что на основанную на `make` инкрементальную компиляцию можно полагаться и в некоторых неочевидных ситуациях, например, при определённых модификациях `make`-файла.

Инкрементальная компиляция в IntelliJ IDEA

IntelliJ IDEA[2] — интегрированная среда разработки программного обеспечения на многих языках программирования (в частности, Java), разработанная компанией JetBrains[3]. Помимо полной перекомпиляции проекта (пункт меню "Rebuild Project"), IDEA предоставляет возможность инкрементальной компиляции (пункт меню "Make Project"), при произведении которой компилируются только файлы, которые были изменены с момента последней компиляции (с учётом соответствующих зависимостей, вычисляемых автоматически)[4]. Начиная с 12-й версии поддерживается "external build" — выполнение задач компиляции в отдельном процессе, независимом от основного процесса IDE[5].

В текущей реализации инкрементальной компиляции в IntelliJ IDEA используется локальное кэширование, изменения файлов отслеживаются по временным меткам (timestamps) — для каждого файла запоминается время его самого позднего изменения на момент последней компиляции, и файл считается изменившимся, если фактическое время его самого позднего изменения не совпадает с запомненным. Отслеживание осуществляется с помощью виртуальной файловой системы (Virtual File System), поддерживающей моментальные снимки (snapshots) содержимого жёсткого диска пользователя[6]. В кэше хранятся абсолютные пути к файлам, таким образом, при перемещении проекта или его части в другую директорию накопленный кэш теряется.

Инкрементальная компиляция в Eclipse

Eclipse[7] — свободная интегрированная среда разработки, развиваемая и поддерживаемая Eclipse Foundation. На основе Eclipse Platform разрабатываются IDE для различных языков программирования, в том числе наиболее популярная из них — Eclipse Java IDE. Eclipse Platform различает полную (full build) и инкрементальную сборки (incremental build).

[2] http://www.jetbrains.com/idea/

[3] http://www.jetbrains.com

[4] http://www.jetbrains.com/idea/webhelp/compilation-types.html

[5] http://blogs.jetbrains.com/idea/2012/12/intellij-idea-12-compiler-twice-as-fast/

[6] http://confluence.jetbrains.com/display/IDEADEV/IntelliJ+IDEA+Virtual+File+System

[7] http://ru.wikipedia.org/wiki/Eclipse

Инкрементальные сборки проектов намного быстрее полных потому, что для компиляции рассматриваются только те исходные ресурсы, которые изменились со времени последней компиляции. Инкрементальный сборщик вызывается каждый раз, когда сохраняется файл. Отмечается, что во многих компиляторах непосредственно на саму компиляцию тратится довольно мало времени; большая часть времени тратится на разрешение контекста (context resolving). Для улучшения производительности последовательных сеансов компиляции рекомендуется сохранять информацию о контексте после компиляции[8].

Eclipse и IntelliJ IDEA по-разному подходят к процессу компиляции. В частности, Eclipse использует свой собственный компилятор Java (Eclipse Compiler for Java, ECJ), отличающийся от классического компилятора `javac`. Этот компилятор позволяет генерировать класс-файлы даже для исходных файлов, содержащих ошибки, то есть таких, на которых классический `javac` завершился бы аварийно и не сгенерировал бы ничего. Эти класс-файлы требуются среде Eclipse для построения в фоновом режиме синтаксического дерева (AST) и обеспечения соответствующей функциональности, требуемой от IDE. Кроме того, такой подход позволяет автоматически, без прямой команды пользователя, в фоновом режиме компилировать исходные файлы при их изменениях (“automake”), возможно, обнаруживая при этом ошибки и сообщая о них пользователю, а также (при использовании соответствующего режима) запускать класс-файлы, полученные при компиляции исходного кода, содержащего ошибки. Среде IDEA для построения синтаксического дерева не требуются скомпилированные классы, поэтому компилятор используется только для порождения исполняемых класс-файлов. Для этой цели может использоваться как классический компилятор `javac`, так и компилятор Eclipse. Функциональность “automake” также поддерживается, независимо от того, какой компилятор используется[9].

Как и IntelliJ IDEA, Eclipse отслеживает изменения файлов по временным меткам, и, следовательно, подвержен тем же проблемам, связанным с системами контроля версий и рассинхронизацией времени на разных ком-

[8] `http://wiki.eclipse.org/FAQ_How_do_I_make_my_compiler_incremental?`

[9] `http://blogs.jetbrains.com/idea/2012/06/brand-new-compiler-mode-in-intellij-idea-12-leda/#comment-272265`

пьютерах.

Инструмент ccache

ccache[10] — это инструментальное средство разработки программного обеспечения, которое кэширует порождения C/C++ компиляции, так чтобы в следующий раз компиляции с теми же самыми параметрами и входными данными можно было избежать и взять результаты из кэша. Это позволяет значительно ускорить время перекомпиляции. Обнаружение повторов производится с помощью хэширования различных видов информации, которая должна быть уникальной для сеанса компиляции, и последующего использования хэш-сумм для идентификации кэшированных порождений. Поддерживаются языки C, C++, Objective-C и Objective-C++. Гарантируется, что такой подход безопасен (safe): важнейший аспект кэша компиляции — всегда возвращать в точности те же самые порождения, которые вернул бы настоящий компилятор. Это включает в себя предоставление точно тех же объектных файлов и предупреждений компиляции, которые были бы выданы настоящим компилятором. Одной из возможностей ccache является совместное использование кэша. Группа разработчиков может увеличить hit rate кэша, совместно используя директорию, в которой он хранится. С этой целью возможно расположить кэш-директорию на сетевой файловой системе NFS (или другой файловой системе подобного рода).

Инструмент Amake

В работах [1, 2] предлагается модификация утилиты make под названием Amake, реализующая подход, при котором система сборки, основанная на make-файлах (Makefile-based build system), улучшается с помощью автоматического анализа зависимостей. Помимо стандартных зависимостей между файлами (а именно, зависимостей целевых файлов от исходных и других целевых), отслеживаются следующие зависимости: команды оболочки ОС (shell), используемые для генерации целевых файлов; программы, исполнение которых предписывается правилами; библиотеки

[10] http://ccache.samba.org/

общего пользования (shared libraries), используемые программами, упомянутыми в правилах; переменные окружения (environment variables). Вместо временных меток для определения изменившихся зависимостей используются подсчитываемые для файлов хэш-суммы. Также хэш-суммы подсчитываются для содержимого правил, а также для исполняемых файлов упоминаемых в них программ, с помощью чего отслеживаются их изменения. Для автоматического отслеживания изменившихся файловых зависимостей применяется перехват вызовов методов (таких как `open`) стандартной библиотеки C (Standard C Library); для этого используется переменная окружения `LD_PRELOAD`, что сужает область применения Amake до Unix-подобных ОС. В архитектуре выделяется кэш, в котором хранятся целевые файлы, порождённые предыдущими сеансами компиляции. Содержимое кэша предлагается хранить на сетевом NFS-диске, а индекс — в реляционной базе данных. Это допускает одновременное использование системы несколькими разработчиками. Отмечается, что использование хэш-сумм лишь незначительно медленнее сравнения временных меток.

Выводы

Для решения задачи сокращения времени сборки программного проекта разработчиками применяются различные методы. Среди них отказ от произведения компиляции, если исходный код не изменился с момента предыдущего сеанса компиляции; сохранение сгенерированных порождений в специальном хранилище-кэше с индексацией, основанной на временных метках или хэшах; оптимистический подход, который состоит в компиляции исходного кода не по команде пользователя, а по мере изменения файлов, в фоновом режиме.

В рассматриваемой в данной работе задаче фигурируют несколько разработчиков, а значит, несколько компьютеров. В такой ситуации временные метки в силу их относительности использовать нельзя. Предлагается вместо временных меток применять хэширование исходных файлов, а именно хэширующую структуру, способную считать контрольные суммы не только на уровне отдельных файлов, но и на уровне директорий и модулей

проекта. При этом пути файлов следует также представлять в переносимом, относительном виде, а именно отсчитывать от корневой директории проекта.

Вычислительно ёмким этапом компиляции является разрешение зависимостей. В языке Java зависимости могут быть разного рода — использование имени класса, определённого в другом файле, вызов метода, использование поля, наследование от класса и пр. При современных масштабах проектов построение графа зависимостей вручную, как это предполагается, например, в `make`, не представляется возможным, граф должен строиться автоматически.

В рассмотренных работах прослеживаются как попытки формального описания предметной области и построения на этой базе критериев безопасности переиспользования порождений (т.е. намерения определить теоретические границы применимости подхода), так и технические решения, направленные на автоматизацию и ускорение процесса сборки, а также учитывающие переносимость и одновременное участие в проекте нескольких разработчиков. В силу того, что в теоретических построениях в качестве основной системы сборки рассматривается система, основанная на `make`-файлах, остаётся простор для дальнейших научных исследований: построения более общих формальных систем, а также базирующихся на них технических решений.

Основные определения и аксиоматика

Мы рассматриваем компиляцию как функцию, получающую на вход набор файлов, содержащих исходный код, и некоторый контекст. Контекст представляется в виде множества порождений. Возвращаемое значение функции, то есть результат компиляции — это тоже некоторое множество порождений. Здесь мы понимаем порождение не как класс-файл, а как некоторый неделимый атрибут, например, имя класса, поле класса, метод, элементарный тип и т.д. Это связано с тем, что в языках, подобных Java, наблюдаются зависимости разного рода; например, использование поля — совсем не то же самое, что вызов статического метода. При этом различные зависимости по-разному влияют на процесс инкрементальной компиляции при изменениях исходных файлов. К примеру, если разработчик изменил сигнатуру метода, то следует рассмотреть для перекомпиляции все файлы, где вызывался этот метод; однако если было изменено только тело метода, то такая модификация никак не могла затронуть другие файлы. Такой неоднородный граф зависимостей нельзя получить, сравнивая лишь содержимое файлов.

В полученных на вход исходных файлах могут содержаться упоминания-ссылки на такие атрибуты, определённые в других файлах, таким образом, для успешной компиляции соответствующие атрибуты должны быть представлены в контексте; в противном случае значение функции будет не определено (такую ситуацию назовём “недоопределённостью”). Также возможен случай, когда в исходном файле определяется тот же самый атрибут, который уже представлен в контексте; в таком случае возникает неоднозначность, и значение функции будет не определено (такую ситуацию назовём “переопределённостью”). Наконец, существуют вырожденные входы, значение функции на которых не определено ни в каком контексте.

Обозначим Σ — множество входов (исходных файлов), Ω — множество выходов (порождений); эти множества не более чем счётны. Тогда частичная функция порождения выходов по входам (функция компиляции) будет выглядеть так:

$$gen : 2^{\Omega} \times 2^{\Sigma} \to 2^{\Omega}$$

Если $gen(\omega, \sigma) = \omega'$, то будем говорить, что ω' — результат порождения входа σ в контексте ω. Это означает, что в результате компиляции набора исходных файлов в некотором состоящем из порождений-атрибутов контексте получается набор порождений-атрибутов. Успешной компиляции соответствует определённость функции на соответствующих входе и контексте, а неуспешной — неопределённость.

Для функции порождения был установлен следующий набор аксиом:

- Аксиома о переопределённости: если $gen(\omega, \sigma)$ — определена, то

$$\omega \cap gen(\omega, \sigma) = \varnothing.$$

  Эта аксиома означает, что в результате процесса компиляции исходных файлов в каком-то контексте не может получиться ничего, что содержалось в этом контексте. Таким образом, при успешной компиляции невозможна ситуация неоднозначности, когда в исходном файле определяются атрибуты, которые уже были определены в других файлах и предоставлены в качестве контекста для компиляции.

- Аксиома о дизъюнктном разбиении: если $gen(\omega, \sigma) = \omega'$, то существует единственное дизъюнктное разбиение $\omega' = \bigcup_{s \in \sigma}^{\varnothing} \omega'_s$, удовлетворяющее свойству

$$\forall s \in \sigma : gen(\omega \cup \omega' \setminus \omega'_s, \{s\}) = \omega'_s.$$

  Эта аксиома означает, что для любого атрибута-порождения, получившегося в результате успешной компиляции, можно единственным образом указать исходный файл, в котором он был определён; при этом если в соответствии с этим принципом разделить все получившиеся атрибуты на множества, то каждое такое множество, состоящее из всех определённых в некотором исходном файле атрибутов, будет соответствовать результату успешной компиляции этого исходного файла в контексте, состоящем из всех остальных атрибутов и изначального контекста. Таким образом, всегда можно установить однозначное соответствие между результатами компиляции и исходными файлами, предоставленными на входе, то есть разделить результат компиляции

на непересекающиеся множества, находящихся во взаимно однозначном соответствии с входными исходными файлами; также верно, что если выбрать один исходный файл и пополнить контекст порождениями всех остальных, то результат такой компиляции будет совпадать с подмножеством результата исходной компиляции, соответствующим этому исходному файлу.

- Аксиома о минимально необходимом контексте: если $gen(\omega, \{s\})$ — определена, то в ω существует единственное наименьшее по включению подмножество $d_\omega(s)$, такое, что $gen(d_\omega(s), \{s\})$ — определена. Эта аксиома означает, что в случае успешной компиляции в некотором контексте единственного исходного файла из этого контекста можно выделить такое подмножество, что компиляция будет успешной и при замене контекста на это подмножество; при этом если в качестве контекста взять любое собственное подмножество этого подмножества, компиляция будет неуспешной. Таким образом, для любого исходного файла существует минимально необходимый контекст, который, тем не менее, может быть разным в зависимости от изначального контекста.

- Аксиома о сужении контекста: если $gen(\omega, \{s\})$ — определено, то для произвольного $\omega' \subseteq \omega$, такого, что $d_\omega(s) \subseteq \omega'$, $gen(\omega', \{s\})$ тоже определено и равно $gen(\omega, \{s\})$. Эта аксиома означает, что если в качестве контекста взять любое подмножество изначального контекста, содержащее минимально необходимый контекст, то компиляция будет успешной, более того, её результат совпадёт с результатом изначальной компиляции. Таким образом, результат компиляции не меняется при сужении контекста до тех пор, пока компиляция успешна.

- Аксиома о недоопределённости: если $gen(\omega, \sigma) = \omega'$ и для $s \in \sigma$ существует $s_1 \in \sigma$, такой, что

$$d_{\omega \cup \omega' \setminus \omega'_s}(s) \cap \omega'_{s_1} \neq \varnothing,$$

то $gen(\omega, \sigma \setminus \{s_1\})$ не определена. Эта аксиома означает, что если при успешной компиляции в одном из входных исходных файлов используются атрибуты, определённые в другом входном исходном файле, то

компиляция в том же контексте того же набора входов без этого второго файла будет неуспешной. Таким образом, имеют место не только зависимости исходных файлов от контекста, но и зависимости их друг от друга; если такая зависимость не удовлетворена, возникает ситуация недоопределённости.

- Аксиома о случаях неопределённости: если $gen(\omega, \{s\})$ не определено, то верно одно из следующих утверждений: a) имеет место ситуация недоопределённости; b) имеет место ситуация переопределённости; c) вход $\{s\}$ вырожденный, т.е. $\nexists \omega' : gen(\omega', \{s\})$ определено. Эта аксиома означает, что если компиляция неуспешна, то причиной этому может быть либо то, что не все требуемые зависимости исходного файла удовлетворены, либо то, что в исходном файле определяются уже определённые в контексте атрибуты, либо то, что файл не компилируется ни в каком контексте (например, содержит синтаксические ошибки).

Введём разбиение результата $gen(\omega, \sigma) = \omega'$ на классы следующим образом. Класс $B^0_{\omega'}$ — это те порождения или части порождений, которые зависят только от входа σ. Класс $B^1_{\omega'}$ — те порождения или части порождений, которые зависят только от входа и порождений класса B^0_ω из контекста. По индукции, класс $B^i_{\omega'}$ — те порождения или части порождений, которые зависят только от входа и порождений классов B^j_ω, где $j < i$.

Например, в языке Java к порождениям нулевого класса принадлежат имена классов и интерфейсов; объявления полей и сигнатуры методов принадлежат к порождениям первого класса, поскольку они могут использовать классы и интерфейсы в качестве типов (а эти объявления принадлежат к нулевому классу).

- Аксиома об эквивалентных контекстах: $\forall s \in \Sigma,\ \forall \omega,\ \omega' \subseteq \Omega$: если $gen(\omega, \{s\})$ определено, то
  a) если $gen(\omega', \{s\})$ определено, то

$$B^0_{gen(\omega,\{s\})} = B^0_{gen(\omega',\{s\})};$$

  b) если для $k \in \mathbb{N}\ \forall i \in [0; k-1]\ B^i_\omega = B^i_{\omega'}$, то $gen(\omega', \{s\})$ определено и

$$B^k_{gen(\omega,\{s\})} = B^k_{gen(\omega',\{s\})}.$$

Эта аксиома означает, что при совпадении контекстов с точностью до $(k-1)$-го класса результат компиляции произвольного входа в этих контекстах совпадает с точностью до k-го класса. В дальнейшем для обозначения равенства с точностью до k-го класса будем использовать обозначение $=^k$: для $\omega_1, \omega_2 \subset \Omega$ запись $\omega_1 =^k \omega_2$ означает, что $\forall i \in [0; k]$ верно $B^i_{\omega_1} = B^i_{\omega_2}$.

Как видно, все аксиомы довольно естественны в том смысле, что каждая из них находит своё отражение в реалиях предметной области.

Приведём полезное следствие из аксиомы о дизъюнктном разбиении:

Следствие. Если $gen(\omega, \sigma) = \omega'$, то для любого $\tau \subseteq \sigma$ можно определить $\omega'_\tau := \bigcup\limits_{s \in \tau} \omega'_s$, при этом

$$\tau_1 \cap \tau_2 = \varnothing \Rightarrow \omega'_{\tau_1} \cap \omega'_{\tau_2} = \varnothing$$

Доказательство. Пусть $\tau_1 \cap \tau_2 = \varnothing$, $\omega'_{\tau_1} \cap \omega'_{\tau_2} \neq \varnothing$. Тогда рассмотрим $x \in \omega'_{\tau_1} \cap \omega'_{\tau_2}$. По аксиоме о дизъюнктном разбиении $\exists! s_x : x \in \omega'_{s_x}$. Поскольку $x \in \omega'_{\tau_1} = \bigcup\limits_{s \in \tau_1} \omega'_s$, то $s_x \in \tau_1$. Аналогично $s_x \in \tau_2$. Следовательно, $\tau_1 \cap \tau_2 \neq \varnothing$. $\square$

Когда один и тот же набор исходных файлов компилируется в разных контекстах, результаты могут отличаться. Тем не менее, в силу того, что различные входные файлы зависят от предоставляемого при компиляции контекста по-разному, множества порождений, соответствующие отдельным файлам, могут совпадать, то есть не влиять напрямую на результат компиляции. Будем различать файлы, влияющие и не влияющие на результат компиляции. Для этого введём понятие дифференциала, которое поможет сформулировать теоремы об инкрементальной компиляции и о переиспользовании порождений. Дифференциал — это подмножество входа, содержащее только те файлы, которые влияют на результат компиляции.

Определение. Пусть ω, $\tilde{\omega}$ — множества порождений, σ — множество входов. Известно, что определены $gen(\omega, \sigma) = \omega'$, $gen(\tilde{\omega}, \sigma) = \tilde{\omega}'$. Тогда дифференциал $\partial \frac{\omega}{\tilde{\omega}} \sigma = \partial$ — это наименьшее подмножество σ, удовлетворяющее свойству: $gen(\omega \cup \omega'_\partial, \sigma \setminus \partial)$ и $gen(\tilde{\omega} \cup \tilde{\omega}'_\partial, \sigma \setminus \partial)$ либо одновременно

не определены, либо одновременно определены и равны.

Свойство 1: ∂ всегда определён и в худшем случае равен σ. Это означает, что влияющее на результат компиляции подмножество можно выделить всегда; при этом, когда изменение контекста затронуло все файлы входного набора, это подмножество будет равно входному набору.

Свойство 2: $\partial\frac{\omega}{\omega}\sigma = \varnothing$. Это означает, что если контексты совпадают, то и результат компиляции не меняется, поэтому дифференциалу не принадлежит ни один файл.

Лемма об эквивалентном дифференциале. Пусть для некоторых $\sigma, \omega, \tilde{\omega}$ таких, что $gen(\omega, \sigma) = \omega'$, $gen(\tilde{\omega}, \sigma) = \tilde{\omega}'$, определён дифференциал $\partial = \partial\frac{\omega}{\tilde{\omega}}\sigma$. Пусть для некоторого $k \in \mathbb{N}$ и некоторого $\hat{\omega} \subset \Omega$ такого, что $gen(\hat{\omega}, \sigma) = \hat{\omega}'$, верно $\hat{\omega} =^{k-1} \tilde{\omega}$. Тогда

$$gen(\omega \cup \omega'_\partial, \sigma \setminus \partial) =^k gen(\hat{\omega} \cup \hat{\omega}'_\partial, \sigma \setminus \partial)$$

Доказательство.

$$\tilde{\omega} =^{k-1} \hat{\omega} \Rightarrow$$

$$\tilde{\omega}' =^{k} \hat{\omega}' \Rightarrow$$

$$\tilde{\omega}'_\partial =^{k} \hat{\omega}'_\partial \Rightarrow$$

$$\tilde{\omega} \cup \tilde{\omega}'_\partial =^{k-1} \hat{\omega} \cup \hat{\omega}'_\partial \Rightarrow$$

$$gen(\tilde{\omega} \cup \tilde{\omega}'_\partial, \sigma \setminus \partial) =^k gen(\hat{\omega} \cup \hat{\omega}'_\partial, \sigma \setminus \partial)$$

Но по определению дифференциала $gen(\tilde{\omega} \cup \tilde{\omega}'_\partial, \sigma \setminus \partial) = gen(\omega \cup \omega'_\partial, \sigma \setminus \partial)$. $\square$

Теорема об инкрементальной компиляции

Рассмотрим ситуацию, когда для некоторого состояния проекта, представляемого набором исходных файлов, была проведена успешная полная компиляция в пустом контексте, то есть не использующая ничего, кроме самих исходных файлов; результат этой компиляции был сохранён в некотором кэше; затем разработчик удалил, добавил или изменил некоторые файлы, модифицировав таким образом состояние проекта; затем он собирается скомпилировать это новое состояние. Разумеется, он может провести повторную полную компиляцию нового состояния в пустом контексте. Однако можно воспользоваться тем, что результат повторной компиляции файлов, которые не подверглись изменению и не зависели существенным образом от изменившихся, будет совпадать с результатом первой компиляции. Это значит, что для этих файлов можно взять результат компиляции из кэша, а остальные подвергнуть перекомпиляции, сэкономив таким образом вычислительные ресурсы. Это интуитивное умозаключение формализуется в теореме об инкрементальной компиляции.

Теорема 1 (об инкрементальной компиляции).

Пусть дано: $\sigma \subset \Sigma$, $gen(\varnothing, \sigma) = \omega^\sigma$. Пусть $\rho, \alpha \subset \Sigma$, при этом $\rho \subseteq \sigma$, $\sigma \cap \alpha = \varnothing$; $\Delta = \Delta^\rho_\alpha \sigma = \sigma \setminus \rho \cup \alpha$. Известно, что определено $gen(\omega^\sigma_{\sigma\setminus\rho}, \alpha) = \omega_\alpha$ и $gen(\varnothing, \Delta) = \omega^\Delta$. Обозначим

$$\partial_1 = \partial\frac{\omega^\sigma_\rho}{\omega_\alpha}(\sigma \setminus \rho),$$

$$\omega^1_{\partial_1} = gen(\omega^\sigma_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_\alpha, \partial_1),$$

$$\partial_2 = \partial\frac{\omega^\sigma_{\sigma\setminus\rho}}{\omega^\sigma_{\sigma\setminus\rho\setminus\partial_1} \cup \omega^1_{\partial_1}}(\alpha),$$

$$\omega^2_{\partial_2} = gen(\omega^\sigma_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_{\alpha\setminus\partial_2} \cup \omega^1_{\partial_1}, \partial_2).$$

Тогда:

$$gen(\varnothing, \Delta) =^2 \omega^\sigma_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_{\alpha\setminus\partial_2} \cup \omega^1_{\partial_1} \cup \omega^2_{\partial_2}$$

Здесь состояниям проекта соответствуют наборы входов функции ком-

пиляции; изменению файлов, содержащихся в проекте, соответствует вычитание из набора входов одного множества и добавление другого. Исходное состояние проекта обозначено σ, из него удаляются файлы множества ρ и добавляются файлы множества α. Тогда получившееся новое состояние являет собой $\sigma \setminus \rho \cup \alpha$. Внутри множества $\sigma \setminus \rho$ выделяется дифференциал ∂_1, а внутри множества α — дифференциал ∂_2. Файлы, соответствующие множествам дифференциалов, перекомпилируются в указанном в теореме контексте. Проиллюстрируем происходящее. На первом изображении показаны рассматриваемые множества входов и их подмножества, на втором — соответствующие им порождения.

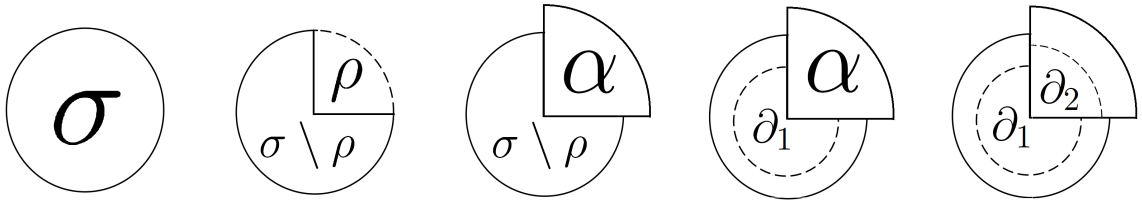

Рис. 1: Рассматриваемые в теореме 1 множества входов и их подмножества

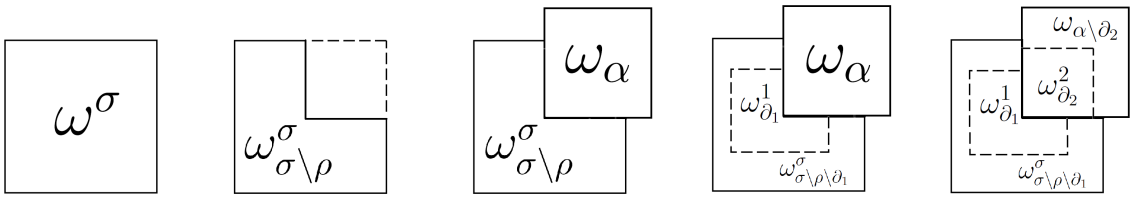

Рис. 2: Порождения, соответствующие множествам и подмножествам входов

Практическое применение данной теоремы таково: в ситуации, когда сохранён кэш для некоторого состояния проекта, которое впоследствии было изменено, предлагается обоснование подхода, состоящего в переиспользовании кэша и перекомпилировании меньшего количества файлов и, таким образом, позволяющего избежать полной компиляции этого состояния; предъявляется множество файлов, для которых следует переиспользовать кэш, множество файлов, которые следует перекомпилировать, а также контекст, в котором нужно проводить компиляцию, чтобы получившийся результат совпал с результатом полной компиляции нового состояния в пустом контексте.

Доказательство: Докажем сначала, что $gen(\varnothing, \Delta) =^0 \omega^\sigma_{\sigma\setminus\rho} \cup \omega_\alpha$. По аксиоме о дизъюнктном разбиении $gen(\varnothing, \Delta) = \omega^\Delta_{\sigma\setminus\rho} \cup \omega^\Delta_\alpha$. При этом по аксиоме об эквивалентных контекстах $\omega^\Delta_{\sigma\setminus\rho} =^0 \omega^\sigma_{\sigma\setminus\rho}$. Рассмотрим

$$\omega^\Delta_\alpha = gen(\omega^\Delta_{\sigma\setminus\rho}, \alpha)$$

и

$$\omega_\alpha = gen(\omega^\sigma_{\sigma\setminus\rho}, \alpha).$$

Поскольку

$$\omega^\Delta_{\sigma\setminus\rho} =^0 \omega^\sigma_{\sigma\setminus\rho},$$

то по аксиоме об эквивалентных контекстах

$$\omega^\Delta_\alpha =^1 \omega_\alpha.$$

Тогда, действительно,

$$gen(\varnothing, \Delta) =^0 \omega^\sigma_{\sigma\setminus\rho} \cup \omega_\alpha.$$

Докажем, что дифференциал $\partial_1 = \partial\dfrac{\omega^\sigma_\rho}{\omega_\alpha}(\sigma \setminus \rho)$ имеет смысл. Для этого проверим, что определены $gen(\omega^\sigma_\rho, \sigma \setminus \rho)$ и $gen(\omega_\alpha, \sigma \setminus \rho)$. Первое выражение — это $\omega^\sigma_{\sigma\setminus\rho}$ и определено по аксиоме о дизъюнктном разбиении. Рассмотрим второе выражение: поскольку $\omega^\Delta_\alpha =^1 \omega_\alpha$ и $gen(\omega^\Delta_\alpha, \sigma \setminus \rho)$ определено, то и $gen(\omega_\alpha, \sigma \setminus \rho)$ определено. Тогда дифференциал, действительно, имеет смысл.

Докажем, что определено $\omega^1_{\partial_1}$. Поскольку определено $gen(\omega^\Delta_{\sigma\setminus\rho\setminus\partial_1\cup\alpha}, \partial_1)$ и $\omega^\sigma_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_\alpha =^0 \omega^\Delta_{\sigma\setminus\rho\setminus\partial_1\cup\alpha}$, то по аксиоме об эквивалентных контекстах $\omega^1_{\partial_1} = gen(\omega^\sigma_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_\alpha, \partial_1)$ определено.

Докажем теперь, что

$$gen(\varnothing, \Delta) =^1 \omega^\sigma_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_\alpha \cup \omega^1_{\partial_1}.$$

По аксиоме о дизъюнктном разбиении

$$gen(\varnothing, \Delta) = \omega^\Delta_{\sigma\setminus\rho\setminus\partial_1} \cup \omega^\Delta_\alpha \cup \omega^\Delta_{\partial_1}.$$

Уже доказано, что $\omega^{\Delta}_{\alpha} =^1 \omega_{\alpha}$. По лемме об эквивалентном дифференциале

$$gen(\omega^{\sigma}_{\rho} \cup \omega^{\sigma}_{\partial_1}, \sigma \setminus \rho \setminus \partial_1) =^2 gen(\omega^{\Delta}_{\alpha} \cup \omega^{\Delta}_{\partial_1}, \sigma \setminus \rho \setminus \partial_1),$$

то есть $\omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} =^2 \omega^{\Delta}_{\sigma\setminus\rho\setminus\partial_1}$. Таким образом,

$$\omega^{\Delta}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega^{\Delta}_{\alpha} =^1 \omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_{\alpha}.$$

Поэтому

$$\omega^{\Delta}_{\partial_1} = gen(\omega^{\Delta}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega^{\Delta}_{\alpha}, \partial_1) =^2 gen(\omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_{\alpha}, \partial_1) = \omega^{1}_{\partial_1}.$$

Тогда, действительно, выполняется равенство с точностью до первого класса.

Докажем, что дифференциал

$$\partial_2 = \partial \frac{\omega^{\sigma}_{\sigma\setminus\rho}}{\omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega^{1}_{\partial_1}}(\alpha)$$

имеет смысл. Для этого проверим, что определены $gen(\omega^{\sigma}_{\sigma\setminus\rho}, \alpha)$ и $gen(\omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega^{1}_{\partial_1}, \alpha)$. Первое выражение — это ω_{α}, оно определено по условию. Рассмотрим второе выражение: поскольку

$$\omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega^{1}_{\partial_1} =^0 \omega^{\Delta}_{\sigma\setminus\rho}$$

и $gen(\omega^{\Delta}_{\sigma\setminus\rho}, \alpha)$ определено, то и $gen(\omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega^{1}_{\partial_1}, \alpha)$ определено. Тогда дифференциал, действительно, имеет смысл.

Докажем, что определено $\omega^{2}_{\partial_2}$. Поскольку определено $gen(\omega^{\Delta}_{\sigma\setminus\rho\cup\alpha\setminus\partial_2}, \partial_2)$ и

$$\omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_{\alpha\setminus\partial_2} \cup \omega^{1}_{\partial_1} =^0 \omega^{\Delta}_{\sigma\setminus\rho\cup\alpha\setminus\partial_2},$$

то по аксиоме об эквивалентных контекстах

$$\omega^{2}_{\partial_2} = gen(\omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_{\alpha\setminus\partial_2} \cup \omega^{1}_{\partial_1}, \partial_2)$$

определено.

Докажем теперь, что

$$gen(\varnothing, \Delta) =^2 \omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_{\alpha\setminus\partial_2} \cup \omega^{1}_{\partial_1} \cup \omega^{2}_{\partial_2}.$$

По аксиоме о дизъюнктном разбиении

$$gen(\varnothing, \Delta) = \omega^{\Delta}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega^{\Delta}_{\alpha\setminus\partial_2} \cup \omega^{\Delta}_{\partial_1} \cup \omega^{\Delta}_{\partial_2}.$$

Уже доказано, что $\omega^{\Delta}_{\sigma\setminus\rho\setminus\partial_1} =^2 \omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1}$, $\omega^{\Delta}_{\partial_1} =^2 \omega^1_{\partial_1}$. Из этого следует, что $\omega^{\Delta}_{\sigma\setminus\rho} =^2 \omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega^1_{\partial_1}$. По лемме об эквивалентном дифференциале

$$gen(\omega^{\sigma}_{\sigma\setminus\rho} \cup \omega^{\alpha}_{\partial_2}, \alpha \setminus \partial_2) =^3 gen(\omega^{\Delta}_{\sigma\setminus\rho} \cup \omega^{\Delta}_{\partial_2}, \alpha \setminus \partial_2),$$

то есть $\omega_{\alpha\setminus\partial_2} =^3 \omega^{\Delta}_{\alpha\setminus\partial_2}$. Таким образом,

$$\omega^{\Delta}_{\sigma\setminus\rho} \cup \omega^{\Delta}_{\alpha\setminus\partial_2} =^2 \omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_{\alpha\setminus\partial_2} \cup \omega^1_{\partial_1}.$$

Поэтому

$$\omega^{\Delta}_{\partial_2} = gen(\omega^{\Delta}_{\sigma\setminus\rho} \cup \omega^{\Delta}_{\alpha\setminus\partial_2}, \partial_2) =^3 gen(\omega^{\sigma}_{\sigma\setminus\rho\setminus\partial_1} \cup \omega_{\alpha\setminus\partial_2} \cup \omega^1_{\partial_1}, \partial_2) = \omega^2_{\partial_2}.$$

Тогда, действительно, выполняется равенство с точностью до второго класса. $\square$

Теорема о переиспользовании порождений

Рассмотрим ситуацию, когда несколько разработчиков одновременно работают над одним и тем же проектом. Каждый разработчик редактирует файлы, составляющие его локальную копию проекта, и время от времени инкрементально собирает свою копию на своём же компьютере; таким образом, у каждого разработчика есть локальный кэш компиляции, в котором хранятся результаты последнего сеанса сборки. При условии регулярного обмена изменениями файлов, например, обновления локальных копий проекта из общего репозитория, можно предполагать, что эти локальные копии отличаются друг от друга незначительно. В процессе такой коллективной разработки случаются ситуации, когда состояние проекта в локальной копии разработчика существенно отличается от состояния, для которого был посчитан его локальный кэш компиляции. Тогда инкрементальная компиляция теряет свои преимущества и мало чем отличается от процедуры полной сборки проекта с нуля. Примерами таких случаев могут служить "cold start" — сборка проекта в ситуации отсутствия кэша вообще — или переключение между ветками ("branches") — например, основной и релизной ветками. В таких ситуациях может быть разумным при компиляции использовать вместо локального кэша компиляции кэши других разработчиков. Эта задача формализуется и решается в теореме о переиспользовании порождений.

Теорема 2 (о переиспользовании порождений).

Пусть $\forall i \in [1:n]$ дано: σ_i, $\omega_i = gen(\varnothing, \sigma_i)$, $\sigma'_i \subseteq \sigma_i$ ($\sigma'_i \cap \sigma'_j = \varnothing$ при $i \neq j$). Обозначим $\omega'_i = gen_i(\sigma'_i)$. Обозначим

$$\partial_i = \partial \frac{\omega_i \setminus \omega'_i}{\bigcup\limits_{j \neq i} \omega'_j} \sigma'_i,$$

$$\omega_{\bigcup\limits_k \partial_k} = gen(\bigcup_k \omega_{\sigma'_k \setminus \partial_k}, \bigcup_k \partial_k),$$

$$\xi_i = \partial \frac{\omega_{\sigma_i \setminus \sigma'_i \cup \partial_i}}{\bigcup\limits_{j \neq i} \omega_{\sigma'_j \setminus \partial_j} \cup \omega_{\bigcup\limits_k \partial_k}} \sigma'_i \setminus \partial_i.$$

Тогда:

$$gen(\varnothing, \bigcup_k \sigma'_k) =^2 \bigcup_k \omega_{\sigma'_k \setminus \partial_k \setminus \xi_k} \cup \omega_{\bigcup\limits_k \partial_k} \cup gen(\bigcup_k \omega_{\sigma'_k \setminus \partial_k \setminus \xi_k} \cup \omega_{\bigcup\limits_k \partial_k}, \bigcup_k \xi_k)$$

Как и в предыдущей теореме, состояниям проекта соответствуют наборы входов функции компиляции. Рассматривается n состояний, обозначенных σ_i, для каждого из которых известен (кэширован) результат его компиляции ω_i. Новое состояние представляется как объединение непересекающихся частей имеющихся состояний, обозначенных σ'_i. Для нахождения результата компиляции этого нового состояния переиспользуются части известных кэшей, обозначенные ω'_i, а также докомпилируется некоторое подмножество этого состояния, выраженное с помощью дифференциалов. Проиллюстрируем происходящее с помощью примера для $n = 4$. На первом изображении показаны известные состояния и выделенные в них непересекающиеся части, на втором — соответствующие им порождения, на третьем — новое состояние и выделяемые в нём подмножества, на четвёртом — соответствующие им порождения.

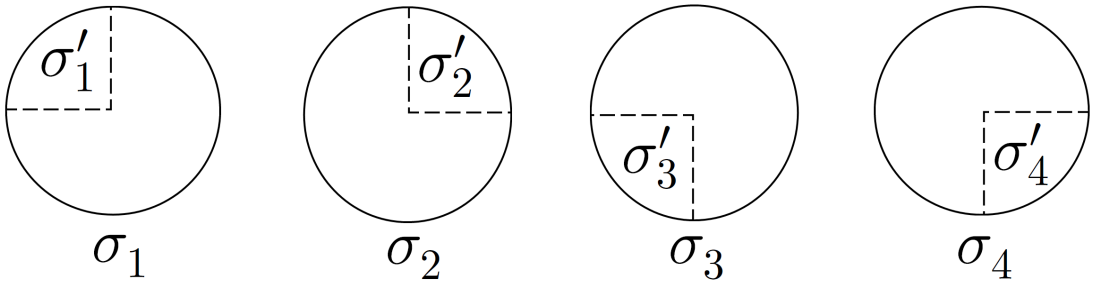

Рис. 3: Рассматриваемые в теореме 2 известные состояния и выделенные в них части

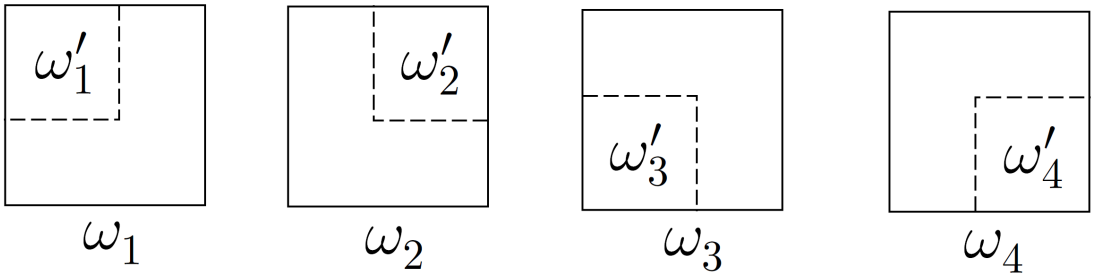

Рис. 4: Порождения, соответствующие известным состояниям и их частям

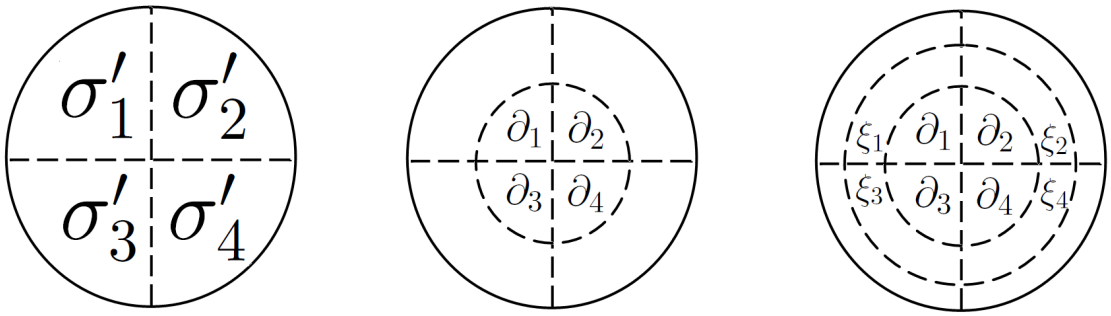

Рис. 5: Рассматриваемое в теореме 2 новое состояние и выделяемые в нём подмножества

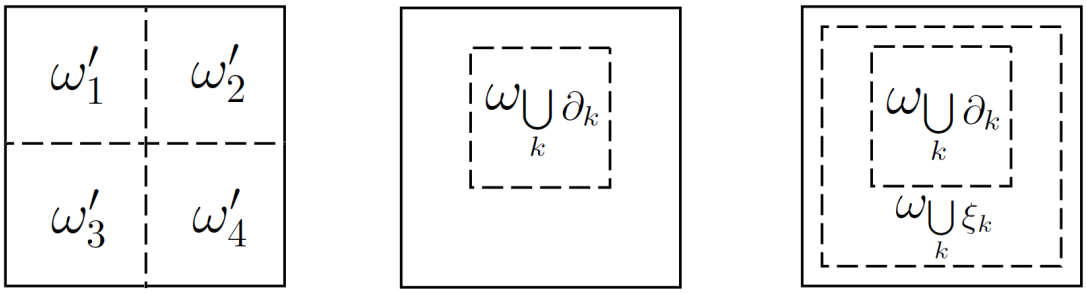

Рис. 6: Порождения, соответствующие новому состоянию и его подмножествам

Практическое применение данной теоремы таково: в ситуации, когда для некоторого состояния проекта неизвестен результат компиляции, но оно представимо в виде частей состояний с известными кэшированными результатами, предлагается подход, состоящий в переиспользовании части кэшей и перекомпилировании меньшего количества файлов и, таким образом, позволяющий избежать полной компиляции этого состояния; предъявляется множество файлов, для которых следует переиспользовать кэши, множество файлов, которые следует перекомпилировать, а также контекст, в котором нужно проводить компиляцию, чтобы получившийся результат совпал с результатом полной компиляции этого состояния в пустом контексте.

Доказательство: Обозначим $\Delta = \bigcup\limits_k \sigma'_k$.

Докажем, что

$$gen(\varnothing, \bigcup_k \sigma'_k) =^0 \bigcup_k \omega_{\sigma'_k}.$$

Для этого достаточно заметить, что

$$gen(\varnothing, \bigcup_k \sigma'_k) = \bigcup_k \omega^{\Delta}_{\sigma'_k} =^0 \bigcup_k \omega_{\sigma'_k}.$$

Докажем, что дифференциал

$$\partial_i = \partial \frac{\omega_i \setminus \omega'_i}{\bigcup\limits_{j \neq i} \omega'_j} \sigma'_i$$

имеет смысл. Для этого проверим, что определены $gen(\omega_{\sigma_i} \setminus \omega_{\sigma'_i}, \sigma'_i)$ и $gen(\bigcup\limits_{j \neq i} \omega'_j, \sigma'_i)$. Поскольку определено $gen(\varnothing, \sigma_i)$, то по аксиоме о дизъюнктном разбиении определено и $gen(\omega_{\sigma_i} \setminus \omega_{\sigma'_i}, \sigma'_i)$. Далее, из определённости $gen(\varnothing, \bigcup\limits_k \sigma'_k)$ по аксиоме о дизъюнктном разбиении следует определённость $gen(\bigcup\limits_{j \neq i} \omega^{\Delta}_{\sigma'_j}, \sigma'_i)$; поскольку $\bigcup\limits_{j \neq i} \omega^{\Delta}_{\sigma'_j} =^0 \bigcup\limits_{j \neq i} \omega'_j$, то отсюда по аксиоме об эквивалентных контекстах определено $gen(\bigcup\limits_{j \neq i} \omega'_j, \sigma'_i)$.

Докажем, что

$$gen(\varnothing, \bigcup_k \sigma'_k) =^1 \bigcup_k \omega_{\sigma'_k \setminus \partial_k} \cup gen(\bigcup_k \omega_{\sigma'_k \setminus \partial_k}, \bigcup_k \partial_k).$$

По лемме об эквивалентном дифференциале для каждого i из равенства $\bigcup\limits_{j \neq i} \omega^{\Delta}_{\sigma'_j} =^0 \bigcup\limits_{j \neq i} \omega_{\sigma'_j}$ следует

$$gen(\omega_{\sigma_i \setminus \sigma'_i} \cup \omega_{\partial_i}, \sigma'_i \setminus \partial_i) =^1 gen(\bigcup_{j \neq i} \omega^{\Delta}_{\sigma'_j} \cup \omega^{\Delta}_{\partial_i}, \sigma'_i \setminus \partial_i),$$

то есть $\omega_{\sigma'_i \setminus \partial_i} =^1 \omega^{\Delta}_{\sigma'_i \setminus \partial_i}$. Значит,

$$\omega^{\Delta}_{\bigcup\limits_k \partial_k} = gen(\bigcup_k \omega^{\Delta}_{\sigma'_k \setminus \partial_k}, \bigcup_k \partial_k) =^2 gen(\bigcup_k \omega_{\sigma'_k \setminus \partial_k}, \bigcup_k \partial_k) = \omega_{\bigcup\limits_k \partial_k}.$$

Тогда, действительно, выполняется равенство с точностью до первого класса.

Докажем, что дифференциал

$$\xi_i = \partial \frac{\omega_{\sigma_i \setminus \sigma'_i \cup \partial_i}}{\bigcup\limits_{j \neq i} \omega_{\sigma'_j \setminus \partial_j} \cup \omega_{\bigcup\limits_k \partial_k}} \sigma'_i \setminus \partial_i$$

имеет смысл. Для этого проверим, что определены $gen(\omega_{\sigma_i\setminus\sigma'_i\cup\partial_i}, \sigma'_i\setminus\partial_i)$ и $gen(\bigcup\limits_{j\neq i}\omega_{\sigma'_j\setminus\partial_j}\cup\omega_{\bigcup\limits_k\partial_k}, \sigma'_i\setminus\partial_i)$. Первое выражение — это $\omega_{\sigma'_i\setminus\partial_i}$ и определено по аксиоме о дизъюнктном разбиении. Второе выражение определено по аксиоме об эквивалентных контекстах, поскольку определено

$$\omega^{\Delta}_{\sigma'_i\setminus\partial_i} = gen(\bigcup_{j\neq i}\sigma'_j\cup\partial_i, \sigma'_i\setminus\partial_i),$$

а $\bigcup\limits_{j\neq i}\sigma'_j\cup\partial_i =^0 \bigcup\limits_{j\neq i}\omega_{\sigma'_j\setminus\partial_j}\cup\omega_{\bigcup\limits_k\partial_k}$.

Докажем, что

$$gen(\varnothing, \bigcup_k\sigma'_k) =^2 \bigcup_k\omega_{\sigma'_k\setminus\partial_k\setminus\xi_k}\cup\omega_{\bigcup\limits_k\partial_k}\cup gen(\bigcup_k\omega_{\sigma'_k\setminus\partial_k\setminus\xi_k}\cup\omega_{\bigcup\limits_k\partial_k}, \bigcup_k\xi_k).$$

Уже доказано, что

$$\omega_{\bigcup\limits_k\partial_k} =^2 \omega^{\Delta}_{\bigcup\limits_k\partial_k},$$

а также, что для любого i

$$\bigcup_{j\neq i}\omega_{\sigma'_j\setminus\partial_j} =^1 \bigcup_{j\neq i}\omega^{\Delta}_{\sigma'_j\setminus\partial_j}.$$

Значит, для любого i

$$\bigcup_{j\neq i}\omega_{\sigma'_j\setminus\partial_j}\cup\omega_{\bigcup\limits_k\partial_k} =^1 \bigcup_{j\neq i}\omega^{\Delta}_{\sigma'_j\setminus\partial_j}\cup\omega^{\Delta}_{\bigcup\limits_k\partial_k}.$$

По лемме об эквивалентных дифференциалах

$$gen(\omega_{\sigma_i\setminus\sigma'_i\cup\partial_i\cup\xi_i}, \sigma'_i\setminus\partial_i\setminus\xi_i) =^2 gen(\bigcup_{j\neq i}\omega^{\Delta}_{\sigma'_j\setminus\partial_j}\cup\omega^{\Delta}_{\bigcup\limits_k\partial_k}\cup\omega^{\Delta}_{\xi_i}, \sigma'_i\setminus\partial_i\setminus\xi_i),$$

то есть $\omega_{\sigma'_i\setminus\partial_i\setminus\xi_i} =^2 \omega^{\Delta}_{\sigma'_i\setminus\partial_i\setminus\xi_i}$. Значит,

$$\omega^{\Delta}_{\bigcup\limits_k\xi_k} = gen(\bigcup_k\omega^{\Delta}_{\sigma'_k\setminus\partial_k\setminus\xi_k}\cup\omega^{\Delta}_{\bigcup\limits_k\partial_k}, \bigcup_k\xi_k) =^3 gen(\bigcup_k\omega_{\sigma'_k\setminus\partial_k\setminus\xi_k}\cup\omega_{\bigcup\limits_k\partial_k}, \bigcup_k\xi_k).$$

Тогда, действительно, выполняется равенство с точностью до второго класса. □

Заключение

В рамках дипломной работы были получены следующие результаты:

- Построена формальная модель, описывающая предметную область.
- Построена аксиоматика, отражающая свойства и ограничения реального мира и позволяющая доказывать нетривиальные утверждения.
- В терминах построенной формальной модели описана задача инкрементальной компиляции, сформулирована и доказана соответствующая теорема.
- Сформулирована и доказана теорема о переиспользовании порождений.

В качестве возможного продолжения работы планируется реализовать прототип, демонстрирующий работоспособность подхода на базе доказанных теорем. В качестве основы для реализации прототипа предполагается использовать среду разработки IntelliJ IDEA[11], разрабатываемой компанией JetBrains. Чтобы проверить работоспособность и эффективность предложенного подхода, предлагается провести на реализованном прототипе эксперименты, в ходе которых дать ответы на следующие вопросы: В каких случаях эффективно кэширование? В каких выгоднее компилировать с нуля? Что быстрее — передача кэшированных класс-файлов по сети или локальная перекомпиляция? В ходе экспериментов требуется рассмотреть различные операционные системы, различные файловые системы, различные дисковые носители (HDD vs. SSD), а также варьировать размеры проектов и размеры данных, пересылаемых по сети.

Полученные результаты являются достаточно общими для того, чтобы подход можно было применить и к другим языкам программирования. Заметим, что подход применим не только к компиляции, но также и к другим вычислительно сложным процессам, которые получают на вход некоторый набор исходных файлов и выдают в качестве результата некоторый набор порождений. Примером может служить задача переиспользования индексов, используемых средами разработки для реализации интеллектуальных

[11] `http://www.jetbrains.com/idea/`

функций вроде “Find Usages”, “Find Implementations”, рефакторинга и т.п. Построение таких индексов с нуля для крупных проектов занимает достаточно много времени, поскольку требует обхода и чтения всех исходных файлов проекта. Используя идеи, сходные с упомянутыми для задачи переиспользования порождений, можно добиться существенного сокращения времени построения этих индексов.

Список литературы

[1] Buffenbarger J. Amake: GNU Make with Automatic Dependency Analysis and Target Caching // British Computing Society — Configuration Management Specialist Group Conference. — 2012.

[2] Buffenbarger J. Adding Automatic Dependency Processing to Makefile-Based Build Systems with Amake // International Workshop on Release Engineering. — 2013.

[3] Feldman S. Make — A Program for Maintaining Computer Programs // Software — Practice and Experience. — 1979. — Т. 9. — С. 255–265.

[4] Jorgensen N. Safeness of Make-Based Incremental Recompilation // FME 2002: Formal Methods — Getting IT Right. — 2002. — С. 126–145.