C2 The JIT In HotSpot Cliff Click

Обзор доклада
Александр Божнюк 371

Немного историй: 1997 год

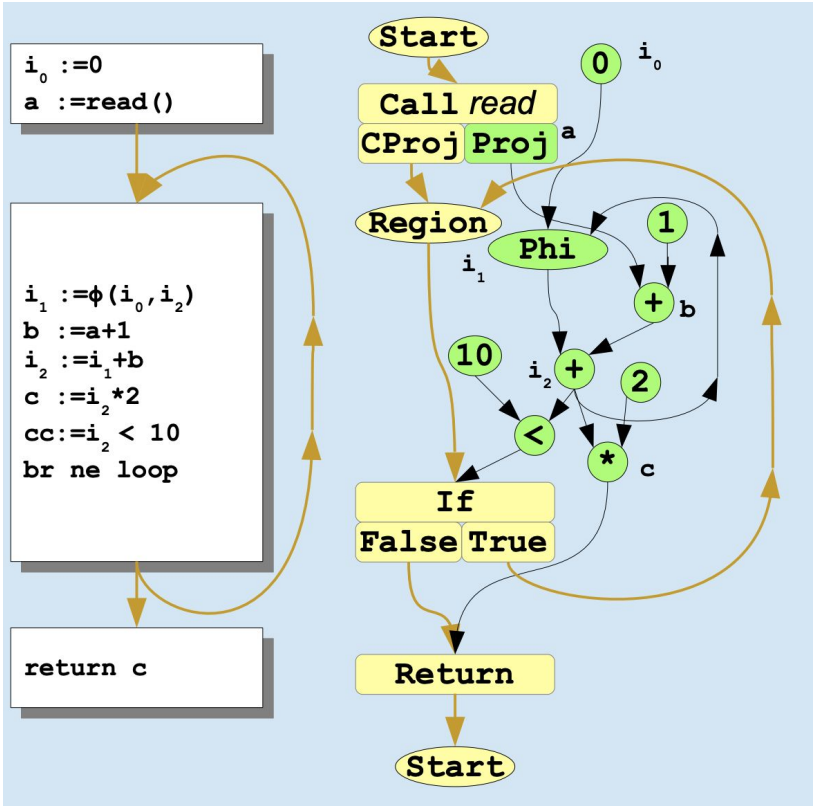
- Много статических AoT компиляторов
  - Медленная генерация кода, хорошее его качество
- Много интерпретируемых языков
- Много смешанных языков (Forth)
- Нужно сделать хороший и быстрый компилятор
- Упор на скорость с учетом памяти → маленький IR
- Но IR не простой
- Принципы спустя 20 лет, в большинстве своем, не изменились.

Sea of Nodes

- Всегда SSA
- Состоит из узлов и ребер
  - Не храним много данных в узлах (меньше == быстрее)
  - Ребра - это просто указатели, как в Си
  - Данные и управление используют тот-же граф
  - Данные отделены от управления, как-бы плавают вокруг.
- Строгая типизация, более точна, чем Java.

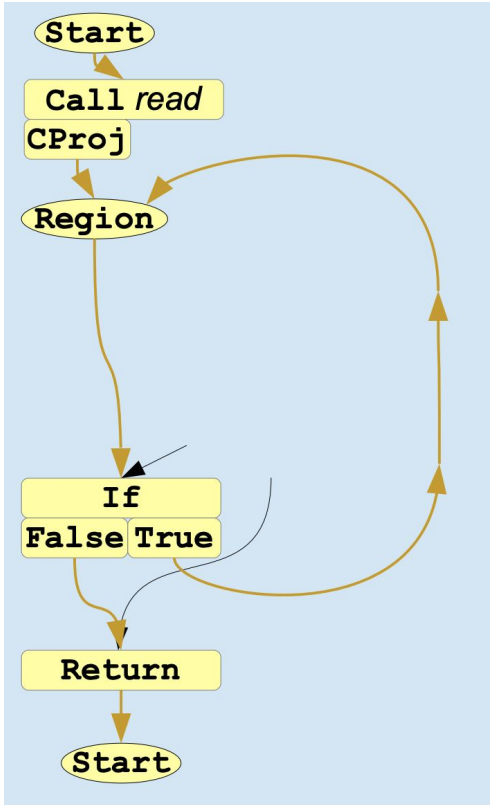
Sea of Nodes

- Слева обычный CFG с базовыми блоками
- Справа Sea of Nodes
- Желтым обозначаем потоки управления
- Зеленым обозначаем потоки данных



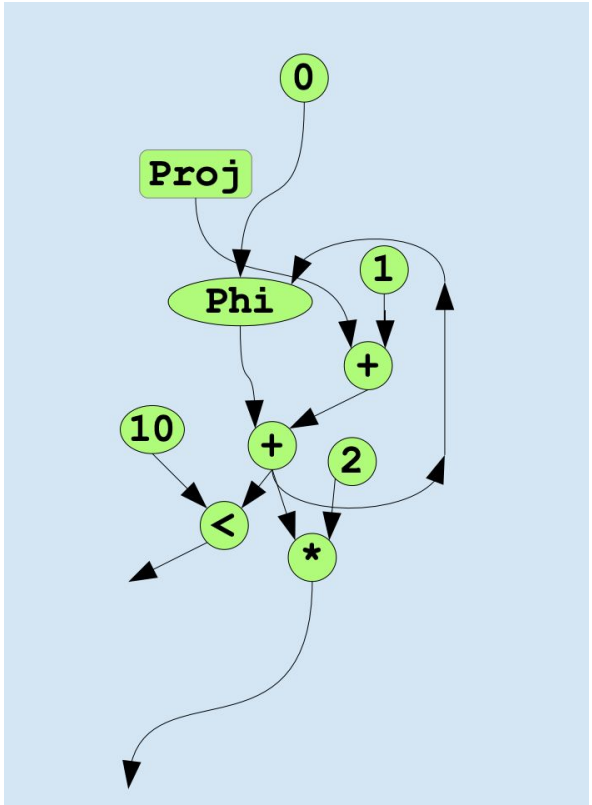
Sea of Nodes: Control Flow Graph

- Есть узлы:
  - Start - начало вычисления
  - Region - слияние потоков управления
  - If - разделение потоков управления
  - Call - вызов функции
- - Нет базовых блоков
- - В конце находится узел Start, по графу можно ходить в любом направлении.



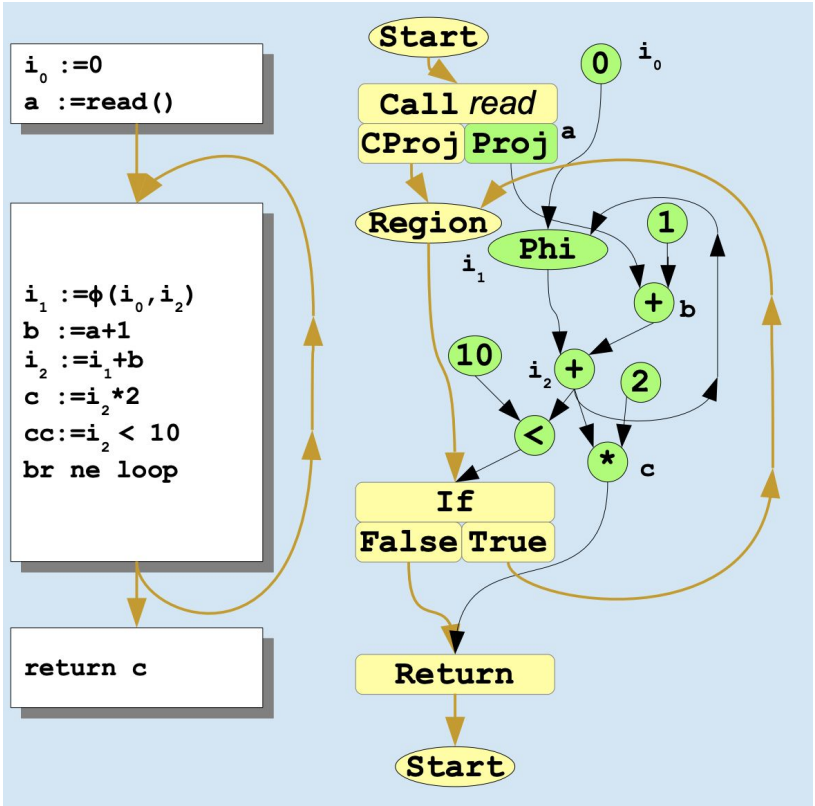
Sea of Nodes: Data Flow Graph

- SSA форма!
- Константы, арифметика, фи-узлы
- Нет имен переменных
- Нет базовых блоков



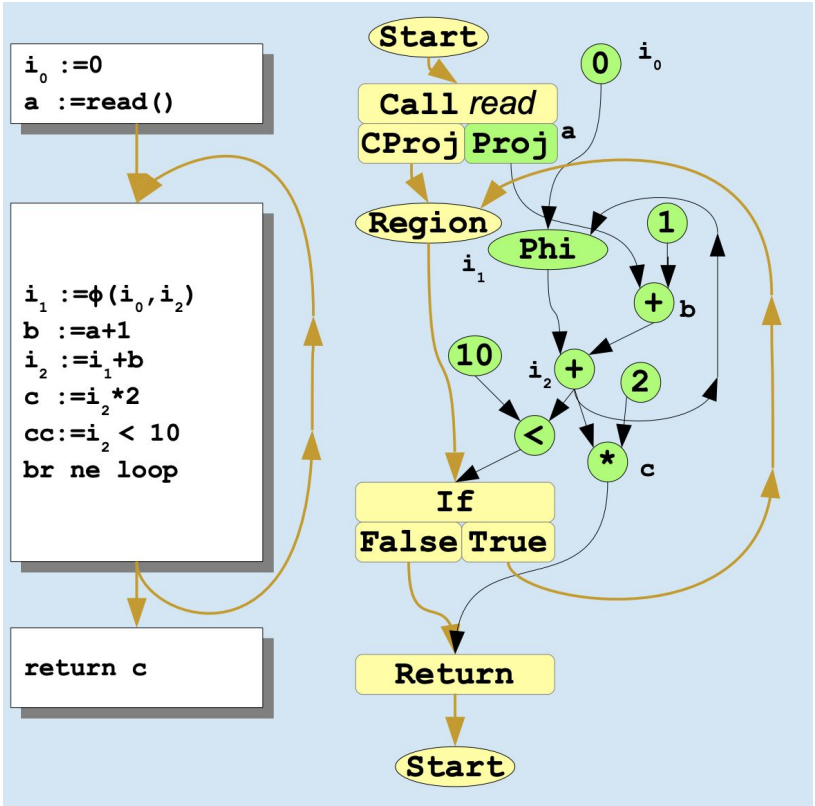
Sea of Nodes

- Узлы принимают и выдают потоки управления и данных
- **Call** - принимает управление и выдает управление и данные
- **If** - принимает управление и данные, а выдает 2 управления
- **Return** - принимает управление и данные, возвращает управление



Sea of Nodes

- Значение имеет только порядок ребер!
- Семантика формируется только через ребра!
- Не имеет значения “место” узлов, важно, как они связаны ребрами

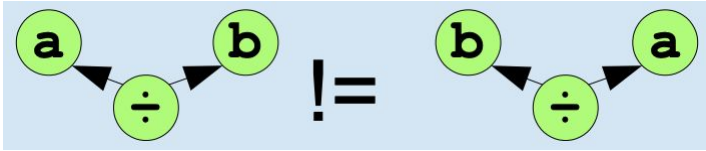


Основные проходы

- Построение **Sea of Nodes** из байткода
  - Включает в себя инлайнинг и оконные оптимизации (peephole opts)
- **Iter** - применение оконных оптимизаций, пока можем;
  - DCE, c-prop, CSE, ls/st opts и многое другое
  - Линейность
- GCP, Loop Opts
- Global Code Motion - граф становится “хрупким”
- Кодогенерация, аллокация регистра, установка в JVM, запуск

Sea of Nodes: Nodes

- Представляет из себя C++ v-table
  - ±35 виртуальных методов: имена, оптимизации и так далее
  - Оптимизируется скорость и размер. Small == Fast
- Есть Use→Def ребра - для операций
  - С указатели из узла Use в узел Def
  - Имеет значение порядок
  - Разрешен NULL (в _in[0]).
- Позже добавили Def → Use ребра
  - Не упорядочены, просто список, нет NULL.



Оконные оптимизации (Peepholes)

- Graph Rewrite Rules
  - Работаем в локальной области графа
  - Замена куска графа чем-то “получше”, но эквивалентным семантически
  - Нет изменений вне области, соседи области не знают об изменениях
- Многие области порождаются одним узлом
  - У узла есть виртуальный метод **Ideal**, чтобы проанализировать область и оптимизировать.
- Таких правил сотни. От самых простых до самых сложных.

Iter: повторение оконных оптимизаций

- Один из важнейших проходов оптимизации;
  - Берем (pull) узел из списка работ (worklist);
  - Оптимизируем, если можем;
  - Проверяем вещи, которые общие для всех узлов (c-prop, GVN, DCE);
  - Если есть изменения, кладем (push) соседей в список работ;
  - Так повторяем, пока список работ не будет пуст.
- Избавляемся от мусора в коде;
- Вызывается почти везде в компиляторе;
- **Быстрота:** вызов всех Iter выполняется за линейное время от размера программы.

Sea of Nodes: Edges

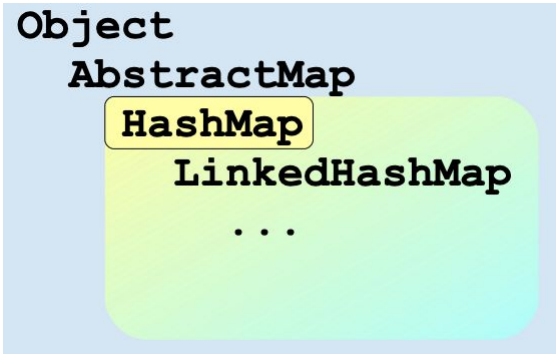
- Являются указателями: скорость!
  - Но мы не можем помечать ребра
- Для Use→Def ребер метки и не нужны;
  - Уже есть порядок, он несет за собой семантику;
- Некоторые узлы выдают множество результатов (outputs);
  - Нужно пометить на ребре, какой из результатов нужен.
- Используем проекции. Это срез кортежа результатов;
  - ProjNode, который идет следом за MultiNode (пример: Call);
  - Это помечает ветку.
- Это нужно только в 5% случаев.

Types

- О чем может рассуждать компилятор;
- Тип - множество значений;
- Все узлы имеют тип
  - Даже узел Region имеет тип “Control”
- Типы используются при различных оптимизациях;
  - c-prop, CHA, Inlining…
- Особенно полезны в CHA & Inlining

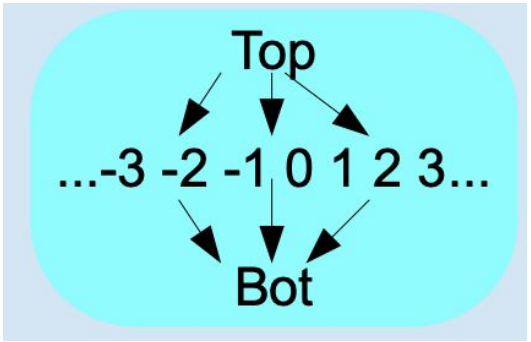
Types: какие они бывают

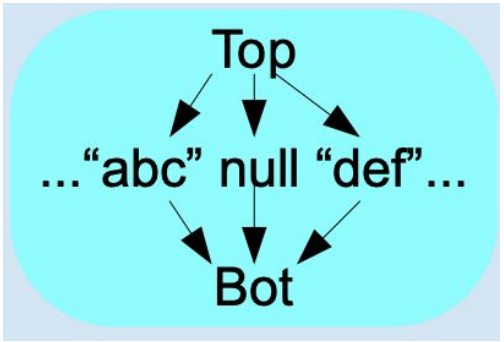
- Все целочисленные, а также определенные диапазоны (пр. 1-10)
- Float 32, 64; float константы;
- Кортежи, массивы, объекты (instances) - Memory State
- Указатели: raw, oop (instance, array, klass)
- Klass: точные (exact) и неточные (inexact)
  - HashMap vs HashMap-и-что-то-ниже



Types: алгебра

- Типы определяют Решетку
  - А также Булеву Алгебру: Complete, Complemented, Distributive, Bounded
- “Bot” - не константа, значение неизвестно, компилятора пытается посчитать его, как хочет этого пользователь;
- “Top” - все константы сразу. Компилятор выбирает ту, которую захочет.
- Эта модель распространяется на объекты, классы, и на вообще все типы.





Types

- Описания типов могут быть большими
- Типы иммутабельные, используется hash consing (интернирование)
  - Такие лучше сравнивать через “==”
    - equals() медленнее
    - equals() может впасть в бесконечный цикл
  - Большинство созданных типов попадает в таблицу

https://en.wikipedia.org/wiki/Hash_consing

Pessimistic vs Optimistic

- Iter и Peepholes - пессимистичные
  - Программа корректна до и после
  - Каждая оконная оптимизация локально корректна
  - Типы “поднимаются” по решетке вверх
- GCP - оптимистична
  - Все типы инициализированы в Top - программа НЕ корректна
  - Типы “падают” по решетке, пока не разрешатся конфликты
  - Когда оптимизация заканчивается, программа корректна
- GCP в определенных моментах лучше Iter

Pessimistic vs Optimistic

- Value() - вычисляет новый тип на основе типов на входе
  - И для Iter, и для GCP
  - Типы или строго “поднимаются”, или строго “падают
- Value() - требование к монотонности
  - Выходные типы падают IFF какой-то входной тип падает
  - Если все старые входные типы isa новые входные типы, то старый выходной тип isa новый выходной тип.

Global Code Motion / Global Value Numbering: Click

Combining Analysis, Combining Optimizations: Click, Cooper

Inlining

- Исполнение может триггерить компиляцию (10к)
- Смотрим стек для подходящего метода
  - Мы формируем контекст вокруг горячего метода
- Парсим байткод, строим SSA
  - Пока парсим, смотрим, в какой момент инлайнить
  - Используем CHA для уточнения типов
    - Он полагается на peepholes
  - Он получаем маленький и горячий метод
  - Эвристика: вставляем триггеры, встроенные, get/set, Unsafe
  - Исключаем больше методы и уже скомпилированные

Inlining

- Выбор делается на моменте парсинга байткода
  - Часто довольно рано
- Недостаток знаний у оптимизатора:
  - Анализ частот вызовов
  - Уточнение типов
  - Промахи при инлайнинге

Global Code Motion

- Строит “настоящий” CFG
- Распутываем Sea of Nodes и кладем узлы в Блоки
- Глобальный планировщик, обращает внимание на задержки и частоту
  - Код выходит из циклов
  - В ветви с низкой частотой, в пути деоптимизации
- IR теперь “хрупкий”: местоположение кода имеет значение.

Global Code Motion, Global Value Numbering: Click

Graph Coloring Register Allocator

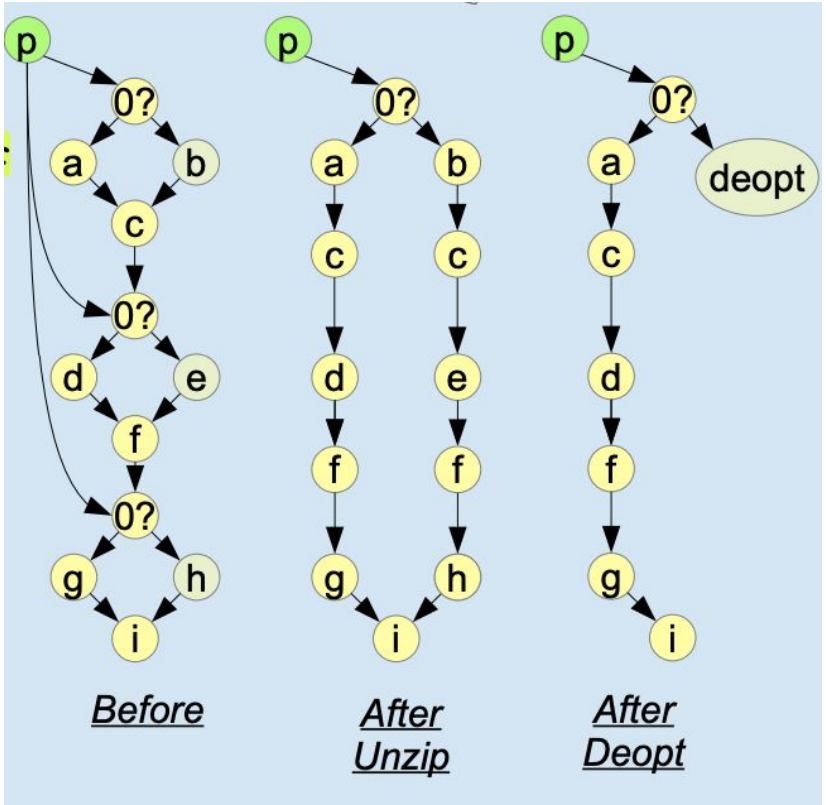
- Одна из самых сложные фаз в компиляторе
- Дает самое мощное ускорение
- Устойчиво к “over-inlining”

Некоторые Java-специфичные оптимизации

- Все проверки байткода производятся как нормальный IR
  - Нет разницы между пользовательской проверкой на null и встроенной проверкой на null
- Компилятор устойчив к проверкам на null
  - 90% устраняется при парсинге байткода
  - 5% как проверка через операции с памятью
  - 5% как отдельная ветка

Некоторые Java-специфичные оптимизации

- Избавляемся от повторяющихся проверок на null
- Проверка может почти всегда быть успешной
  - Деоптимизация, если провалились



Некоторые Java-специфичные оптимизации

- Вызовы: CHA 90% вызовов делает статическими
  - Позволяет делать inlining
- Оставшиеся 10% - Inline Cache
  - 90% (из 10%) - попадание
    - Мы смотрим на класс и делаем вызов
  - 10% (из 10%) - промах. Обычный виртуальный вызов.

Некоторые Java-специфичные оптимизации

- Range Check Elimination
  - (1) Находим ограниченные циклы и переменные индукции
  - (2) Peel. Очищаем цикл от ненужных проверок
  - (3) Вставляем пред- и пост- циклы для крайних случаев
  - (4) Удаляем проверки из внутреннего цикла
  - Делаем Iter на этом цикле
  - (5) Для маленьких тел цикла, делаем unroll по степеням двойки
    - Смотрим на размер цикла, чтобы не выйти за кеш
    - Опять делаем iter на развернутом теле, и повторяем (5)

Некоторые Java-специфичные оптимизации

- Checkcast, instanceof, arraystore
- Быстрая проверка на тип через таблицы и иерархии

Fast subtype checking in the HotSpot JVM: Click, Rose

Heroic Optimizations

- Много проверок редко проваливаются
  - Array-store, range checks, most null checks…
  - Проверка могла еще ни разу не провалиться
- Предполагаем, что проверка и не провалится (и готовимся к тому, что все же провалиться может)
- Оптимизируем для наилучшего случая
  - Но при этом делаем приготовления для восстановления в случае провала.

Heroic Optimizations

- Общий случай: получаем ускорение
- Случай провала: деоптимизируемся
  - Меняем кадр стека на кадр интерпретатора
  - Ставим флаг провала
  - Делаем заново профилирование
  - Делаем снова JIT-компиляцию, но учитываем флаг. Второй раз уже не делаем дерзких оптимизаций
  - Трудности: нужно правильно отслеживать флаги через множество слоев встраивания

Deoptimization

- Это бэкап
  - Опция восстановления для каждого провала
  - Возвращение к интерпретатору
  - Принцип - оптимизируй горячие методы, и что можно откомпилировать
- Следим за состоянием JVM
  - Следим через “Safepoints”
  - Мапинг регистров, стека, констант на стек JVM
  - Не сильно относится к GC safepoints

Deoptimization

- С точки зрения компилятора…
  - Safepoint - это Call
  - Читает все состояние JVM с эффектами память
- Safepoint - это просто узел
  - Use→Def в каждое состояние JVM
  - Очень малая частота выполнения
  - Эффект: биты, нужные для интерпретатора, но не для компилятора отправляются на стек и перемещаются в ветвь с малой частотой.

Debugging

- Та же проблема, что и с Heroic opts:
  - Общий быстрый случай, редкий медленный случай
  - Как найти баг в медленном случае?
- Множество флагов
  - Редкий случай может стать общим
  - Готовься к 10x замедлению
- Множество выборов для компиляции
  - Бинарный поиск с -XX:CIStart/CIStop
  - Множество других опций для компиляции

Summary

- Sea of Nodes
  - Вся семантика в ребрах и узлах
  - Узлы маленькие и ребра это сырые указатели - скорость
- Graph Rewrite Rules
  - Легкие оптимизации через Peepholes
- Типы: Быстры и точны
  - Определяют то, о чем компилятор может говорить
  - Теория: булева алгебра: Complete, Distributive, Complemented, Bounded (Ranked) Lattice
  - Hash-cons, интернирование

Summary

- Множество Java-специфичных оптимизаций
  - Быстрый/медленный случай
    - Деоптимизация вместо медленного случая
  - Null check unzipping
  - Range Check Elimination
  - Subtype checks, CHA
- Aggressive Inlining
  - Возможно делается слишком рано
- Graph-Coloring Register Allocator
  - Терпим к Aggressive Inlining
- Все это прошло проверку временем
  - Много могло измениться, но костяк остался

Источники

- Modern Compiler Textbook:
  - Engineering: A Compiler: Keith Cooper, Linda Torczon
- Optimistic vs Pessimistic; Type theory
  - Combining Analysis, Combining Optimizations: Click, Cooper
  - https://en.wikipedia.org/wiki/Lattice_(order)
- Design of the C2 IR:
  - A Simple Graph-Based IR: Click, Paleczny
  - From Quads to Graphs: An IR’s Journey: Click
  - Global Code Motion, Global Value Numbering: Click
  - The Java HotSpot Server Compiler: Click, Paleczny, Vick
  - Fast subtype checking in the HotSpot JVM: Click, Rose