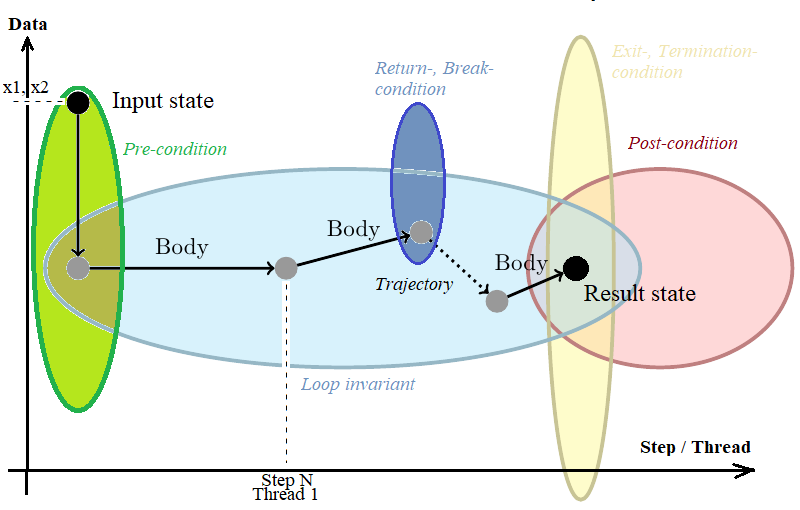
### Program semantic

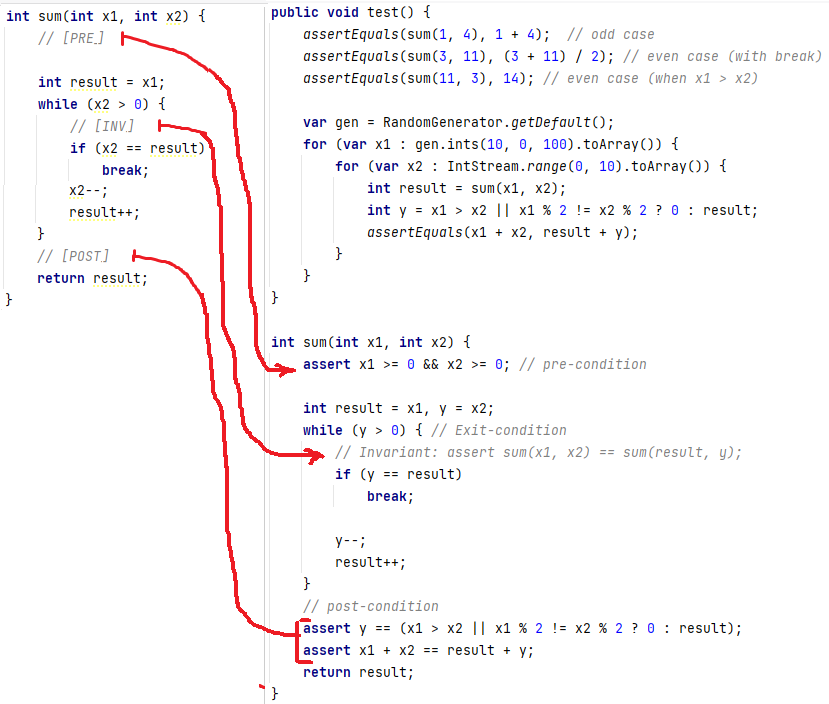
Исходя из классической семантики Флойда-Хора-Дейкстры каждая программа имеет предусловие (pre-condition), постусловие (post-condition), а каждый цикл имеет инвариант цикла (loop invariant) вместе с условием выхода (exit-condition) и окончанием (termination-condition) цикла. Инвариант цикла дает фундаментальную информацию о цикле, показывая чего он пытается достичь, и как он этого достигает. Поэтому можно утверждать, что понимание цикл означает понимание его инварианта, а вместе с тем и понимание всей программы!



Пояснение этой идеи представлено на рисунке (вверху) и заключается в следующем:

* Работу программы можно рассматривать как траекторию (control flow) из состояний данных в пространстве переменных программы и шагов выполнения для каждого потока вычисления. Основную трудность составляет представление циклического процесса, в большинстве случаев неизвестного, а иногда бесконечного, по числу шагов. Что не может быть представимо в виде фиксированного тензора для обучения.
* Переход между начальным (input state) и конечным (result state) состоянием состоит из большого числа шагов (body loop), состояний которые удовлетворяют инварианту, и приводят текущее состояние ближе (в каком-то смысле движение по стрелкам) к выходному состоянию. Чтобы решение достигло своей цели за конечное число шагов, необходимо понятие дискретного условия окончания цикла (termination-condition). Что позволяет представить в символьном (конечном) виде необходимые условия инвариантов для обучения, как способ обобщения и уменьшения вариативности огромного пространства состояний на рисунке вверху.
* Возможность найти и обобщить постусловие, как характеристику возможного решения, состоит из поиска различных инвариантов условий. В результате постусловие можно определить как пересечение инварианта и других возможных условий.

### Motivated example



В данном примере поданный на вход код (окно слева) – это контекст, который требуется «понять». Понимание будет состоять из подстановки вместо специального токена [INV], как маски для инварианта в виде условия (с определенными ограничениями) на языке программирования. Так как инвариант будет обучаться возвращать только true в любом контексте и при любых тестах, то это и будет считаться «смыслом» контекста, заданного пользователем. В данном случае (идеальный пример) для функции sum был найден post-condition в упрощенном виде result == x1 + x2, что является отражением смысла данного алгоритма.

В данном примере с алгоритмом sum минимальное добавление условия с break в цикл внесло значительное изменение в осмыслении данного решения, так как переменная **y**, входящая в инвариант, не всегда равна 0 и поэтому требует детальной интерпретации. Без **break** post-condition выглядел бы как sum(x1, x2) == result. С **break** post-condition – это sum(x1, x2) == result + y и возникает вопрос: что такое переменная **y**? Попробуем избавиться от **y** и выразить ее в терминах переменных с известной интерпретацией. Получается, что **y** всегда равна 0 либо самому result в зависимости от того, что x1 > x2 или от взаимной четности/нечетности x1 и x2.

### Model learning

Обучение будет состоять из тренировки двух нейронных сетей Generator и Discriminator по аналогии обучения GAN. Generator – сеть, которая будет учиться находить выражения инвариантов в случайно выбранных местах кода из набора данных. А Discriminator будет для этого кода с инвариантами генерировать тесты, которые будут стараться нарушить эти инварианты. После того, как готовы сгенерированные варианты кода с тестом и инвариантами они прогоняются на тестовой среде и определяется награда (или наказание) для Discriminator, который нашел противоречие в инвариантах, или для Generator, который нашел устойчивый инвариант.

Дисперсия инвариантов, представленных в символьном виде и имеющих множество ограничений (например, операции присвоения запрещены, своего рода специальный DSL), значительно меньше, чем пространство программ, представленных выше. Такого рода уменьшение дисперсии является одним из мотивов в направлении этого исследования.

### Invariant improvement

Как заставить Generator использовать переменные с известной интерпретацией? Таким образом, возникают следующие эвристические особенности при поиске инвариантов:

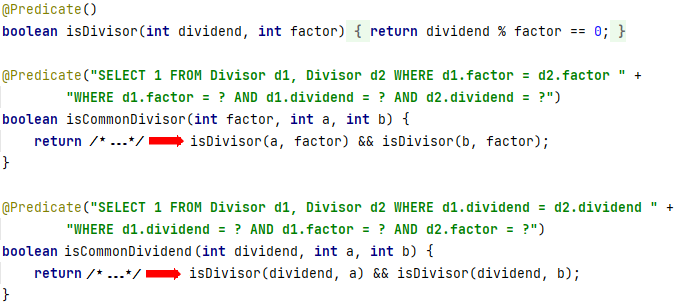
* Избегать тавтологий.
* Чем короче инвариант, тем лучше.
* Чем меньше неинтерпретируемых переменных, тем лучше.
* Чем выше уровень интерпретируемых переменных или функций, тем лучше.

### Invariant equality

Как указывалось выше, можно рассматривать постусловие как комбинацию предусловия и всех инвариантов. Для упрощения инвариантов придётся использовать, кроме хорошо известных арифметических операций -, +, \*, %, другие «известные» функции типа max/min, sorted, perm, is\_in. Их идеализированный смысл достаточно однозначен, хотя существует множество различных реализаций каждой из них, тем более в разных языках программирования.

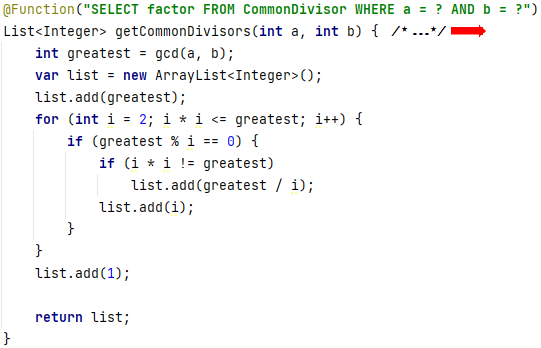
## Примеры для обсуждения применения инвариантов:

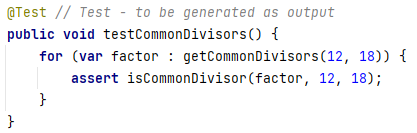
1. На вход NN подается код с аннотациями @Predicate или @Function, включающие выражения с SQL-подобным синтаксисом. В ответ ожидается сгенерированное решение на Java и сгенерированный test, который и проверяет правильность решения. Например:



В этом примере первый предикат рассматривается как аксиома (или таблица в БД) с имплементацией, данной заранее, и ее применение будет проверять (тестировать) отношение переменных dividend и factor. Второй и третий предикат генерируется, путем замены комментариев /\* … \*/ на корректный код, который тестируется отдельно.

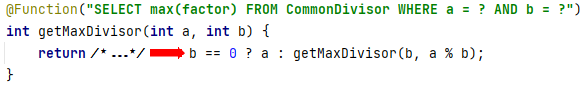
1. Теперь, тем же образом, выразим новую функцию на базе предыдущих предикатов:





Стоит заметить, что NN нашла оптимальный алгоритм поиска всех общих делителей!

1. В этом примере используется агрегирующая функция max, но NN находит оптимальный алгоритм, игнорируя предыдущий способ получения списка всех общих делителей и без итерирования по списку, использует более быстрый алгоритм!



## Обсуждение:

1. Грамматика PSQL:

statement ::= select

select ::= **SELECT** [**DISTINCT**] param **AS** alias [, param **AS** alias]\* **FROM** predicate [**AS** alias]

[[**INNER**] | (**LEFT** | **RIGHT** | **FULL**) [**OUTER**] **JOIN** predicate **ON** condition | **TRUE**]\*

[**WHERE** condition]

[**GROUP BY** param\* **HAVING** folding\_condition\*]

[**ORDER BY** param\*]

[**UNION** | **INTERSECT** | **EXCEPT** select]

[**LIMIT** numeric]

condition ::= [alias.]param = ([alias.]param | const) [**AND** [alias.]param = [param | const]]\*

const ::= **NULL** | (numeric | string) | ? | [const [, const]\*] | {param: const [, param: const]\*}

folding ::= **MIN** | **MAX** | **SUM** | **COUNT**

1. На основании вышеуказанной грамматики, как комбинатора, пространства поиска всех программ может быть кластеризованно по следующим свойствам:

* Joining (Union/Intersect/Except)
* Grouping with Folding (relation to constant)
* Limiting and Ordering
* Unfolding (set to constant, set to another set, set to relation)

Эти свойства позволят сформировать таксономию всех программ и искать способы разделения программ на атомарные составляющие, а с другой стороны из простых компонент строить более сложные программы.

1. Данные подход пытается связать реляционный и процедурный подходы для более наглядного выражения того, что требуется от алгоритма и что процедурный код делает.
2. Обратной задачей связи между процедурным кодом и реляционным, является построение SQL-подобного выражения по данному на вход процедурному коду. Что создает предпосылки для оптимизации и поиска скрытых дефектов в оригинальном коде.
3. Для выражения тестов в виде assert будут использованы разнообразные, но дуальные формы предикатов и функций, которые позволят упростить и повысить наглядность заложенного в коде смысла.

## General overview

### Термины

**Axiom** – короткий код на базовом языке программирования, рассматриваемый как базис (например арифметические операции) для построения и проверки более сложных программ. Аксиомы разделяются на предикаты (Predicate) и функции (Function).

**Spec (PSQL)** – лаконичная формулировка постановки задачи на псевдо-коде DSL. В данном случае выбран SQL-подобный язык (PSQL).

**Explanation** – процесс шаг за шагом трансформирующий программу от простой версии на базе аксиом до идеальной версии, включающей аксиомы только как assert.

**Ideal-code** – код на базовом языке программирования, который достаточно точно и эффективно решает поставленную задачу.

**Real-code** – код на базовом языке программирования, который рассматривается как шаблон или скетч для решения задачи (может содержать ошибки).

**Test-case** – код на базовом языке программирования,

**EvalTool** -

**DesignTool** – Z3 Solver

**NN** –

## Cloud computing - Free multimedia iconsHuman Generic Detailed Outline iconSchema overview

**#1 (Dual to #2)**

**Axiom 2**

**Axiom 1**

**Axiom N**

**assert   
invariant**

NN

**Spec  
 (PSQL)**

**assert   
invariant**

**Ideal-code   
(Java)**

**EvalTool**

DesignTool

**Test-case  
(Java)**

**#2 (Dual to #1)**

**Axiom N**

**Axiom 1**

NN

**Spec  
 (PSQL)**

**EvalLoop/ DesignTool**

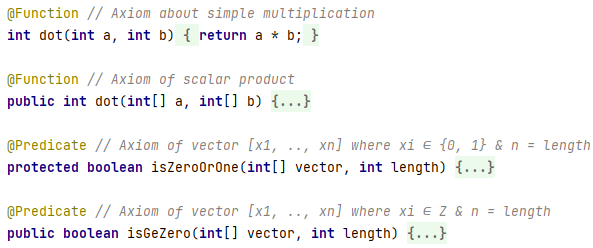
**Real-code   
(Java)**

**Ideal-code + Test-case**

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Tasks** |
| 1 | Axioms + Pseudo Code (as specification) | Code and Test-case | Code and unit-tests synthesis, invariant/pre/post condition specification inferring |
| 2 | Real-code | Axioms + Spec and Ideal-code + Test-case | Bug fixing and defect detection, code development estimation, code duplication (plagiarism), Category of the code, idioms mining |
| 3 | Natural Language | Code and Test-case |  |

## Knapsack use-case

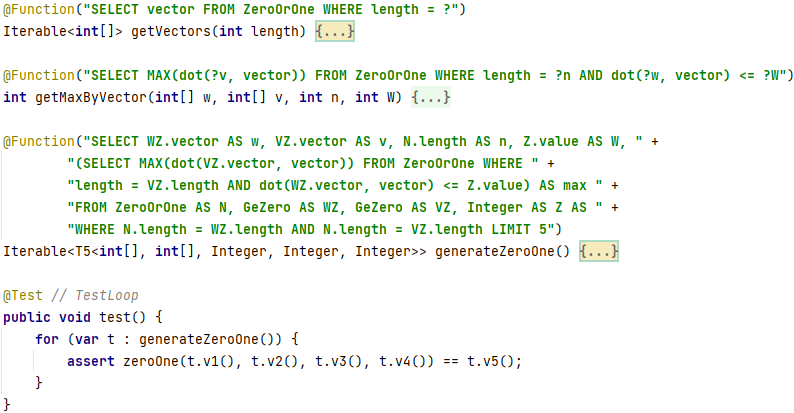
1. Аксиомы



1. Main Function

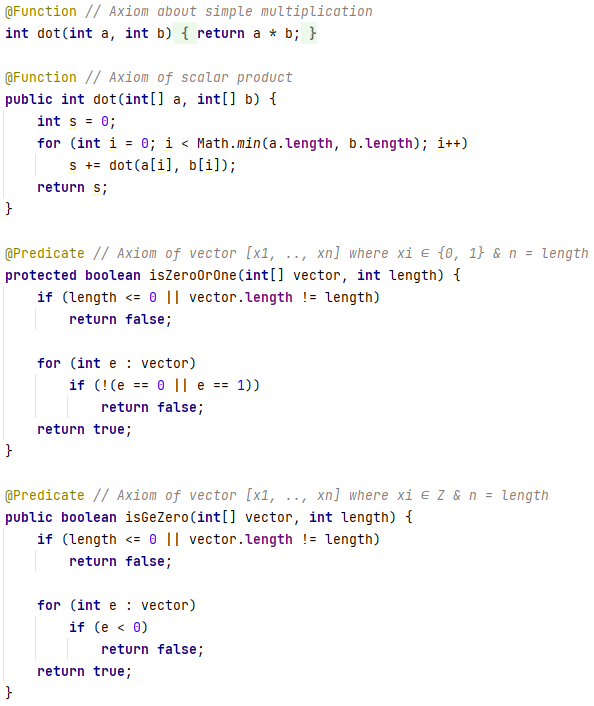


1. Test Function

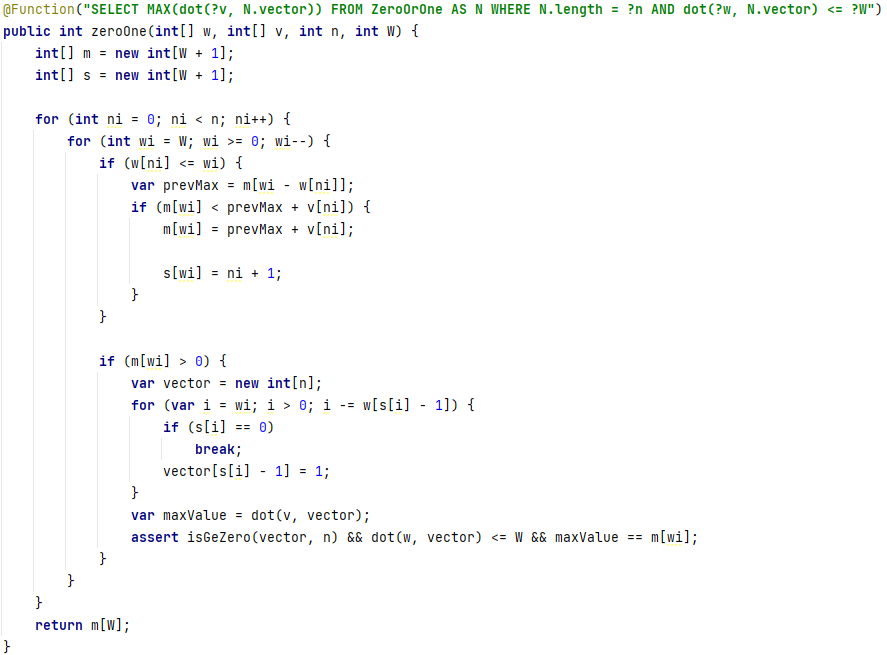


### Приложение

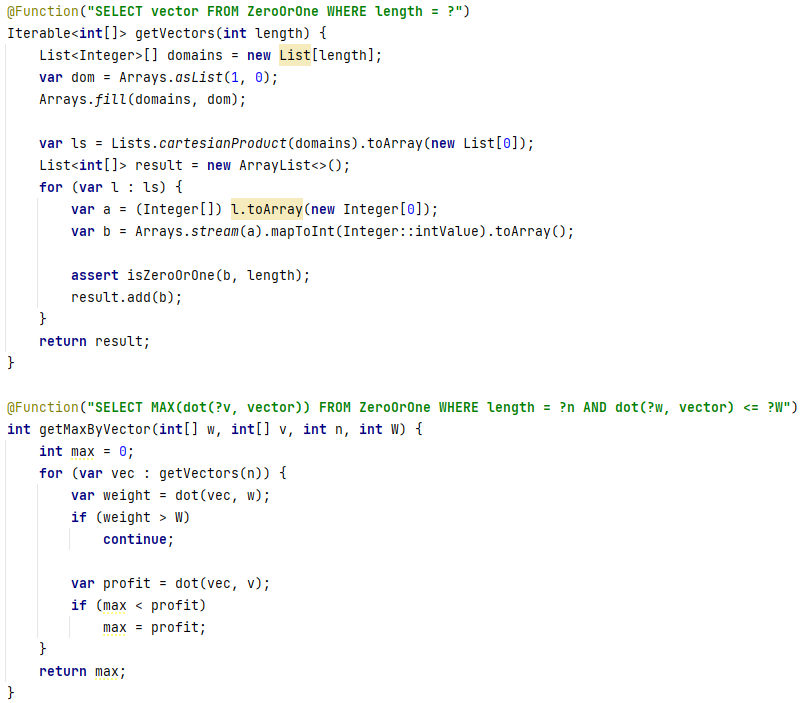
1. Axioms

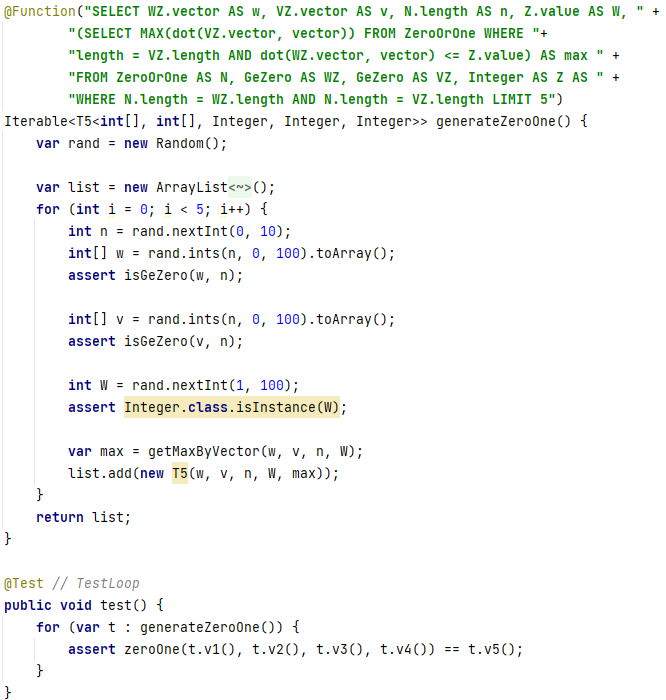


1. Main Function



1. Генерация теста путем получения закрытой (без аргументов) функции generateZeroOne(), которая формирует тестовые данные полным перебором домена из функции getVector() для получения максимального значения max.





## Выражение res.isEmpty() - всегда истинно, а на cur выдается, что присваивание бессмысленно.

List<Integer> process(Map<String, Integer> options, List<String> inputs) {

List<Integer> res = new ArrayList<>();

int cur = -1;

for (String str : inputs) {

if (str.startsWith("-"))

if (options.containsKey(str))

if (**cur == -1**) **cur** = options.get(str); // **A**

else if (options.containsKey("+" + str)) {

if (**cur == -1**) **cur** = **res**.isEmpty() ? -1 : // **B**

**res**.remove(**res**.size() - 1); // **C**

if (**cur != -1**) **res**.add(**cur** + str.length()); // **D**

}

}

return res;

}

