Идея состоит в том, что на вход подаются простые функции, которые возвращают boolean (as Atomic relation) или Iterable<Entity> (as Atomic function). Затем к ним применяется Operator в виде SQL-подобного языка и на выходе ожидается Ideal-code, который соответствует семантике композиции Atomic relations/functions и Operator, в том числе оптимизированный в плане производительности. Например:

# Greatest Common Divisor

// Input with atom relation unit

boolean isDivisor(int dividend, int factor) {  
 return dividend % factor == 0;  
}

|  |  |  |
| --- | --- | --- |
| **#** | **Operator** | **Output** |
|  | SELECT factor, d1.diveden as a, d2.diveden as b FROM Divisor d1, Divisor d2 WHERE d1.factor = d2.factor | Iterable<Triplet<Integer, Integer, Integer>>  getCommonDivisor() {  throw new  UnsupportedOperationException("Because of limitless");  } |
|  | SELECT factor FROM Divisor WHERE dividend = 0 | Iterable<Integer> getIfDividendZero() {  throw new  UnsupportedOperationException("Because of limitless");  } |
|  | SELECT factor FROM  (SELECT factor, a, b FROM Divisor d1, Divisor d2 WHERE d1.factor = d2.factor) as CommonDivisor WHERE a = :a AND b = :b | public int gcd(int a, int b) {  return b == 0 ? a : gcd(b, a % b); }  List<Integer> getCommonDivisors(int a, int b) {  int greatest = gcd(a, b);  var list = new ArrayList<Integer>();  list.add(greatest);  for (int i = 2; i \* i <= greatest; i++) {  if (greatest % i != 0)  continue;  if (i \* i != greatest)  list.add(greatest / i);  list.add(i);  }  list.add(1);   return list; }  @Test public void testCommonDivisors() {  for (var factor : getCommonDivisors(12, 18)) {  assert isCommonDivisor(factor, 12, 18);  } } |
|  | SELECT MAX(factor) FROM (SELECT factor FROM Divisor d1, Divisor d2 WHERE d1.factor = d2.factor) as CommonDivisor WHERE a = ? AND b = ? | int getMaxDivisor(int a, int b) {  return b == 0 ? a : getMaxDivisor(b, a % b); } |
|  | SELECT MIN(dividend) FROM  (SELECT dividend, a, b FROM Divisor d1, Divisor d2 WHERE d1.dividend = d2.dividend) as CommonDividend WHERE a = ? AND b = ? | public int lcm(int a, int b) {  return a / gcd(a, b) \* b; }  int getMinDividend(int a, int b) {  return lcm(a, b); } |

# Knapsack

// Atom of scalar product  
int dot(int[] a, int[] b) {  
 int s = 0;  
 for (int i = 0; i < Math.*min*(a.length, b.length); i++)  
 s += a[i] \* b[i];  
 return s;  
}  
// Atom of vector [x1, .., xn] where xi ∈ {0, 1} & n = length  
boolean isZeroOrOne(int[] vector, int length) {  
 if (length <= 0 || vector == null || vector.length != length)  
 return false;  
  
 for (int e : vector)  
 if (!(e == 0 || e == 1))  
 return false;  
 return true;  
}

|  |  |  |
| --- | --- | --- |
| **#** | **Operator** | **Output** |
|  | SELECT MAX(dot(:v, N.vector)) FROM ZeroOrOne AS N WHERE N.length = :n AND dot(:w, N.vector) <= :W | int maxDotVector(int[] w, int[] v, int n, int W) {  int[] m = new int[W + 1];   for (int ni = 0; ni < n; ni++) {  for (int wi = W; wi >= 0; wi--) {  if (w[ni] <= wi) {  var prevMax = m[wi - w[ni]];  if (m[wi] < prevMax + v[ni]) {  m[wi] = prevMax + v[ni];  }  }  }  }  return m[W]; } |

# Summands combinations

// Input with atom relation unit

boolean isSum(int a, int b, int sum) {  
 return a + b == sum;  
}

|  |  |  |
| --- | --- | --- |
| **#** | **Operator** | **Output** |
|  | SELECT a FROM Sum WHERE b = :b AND sum = :sum | int getA(int b, int sum) {  return sum - b; } |
|  | SELECT a, b FROM Sum WHERE sum = :sum AND a >= 0 AND b >= 0 | Iterable<Pair<Integer, Integer>> getComponents(int sum) {  var list = new ArrayList<Pair<Integer, Integer>>();  for (var i = 0; i <= sum; i++)  list.add(new Pair<>(i, sum - i));  return list; } |
|  | SELECT summands FROM Sum WHERE sum = :sum | // Summands (positive) combinations with repetition (order doesn't matter) private void generateSummands(int sum, List<Integer> current, List<int[]> summands) {  if (sum == 0) {  summands.add(current.stream().mapToInt(i -> i).toArray());  return;  }  for (int i = Math.*min*(sum, current.isEmpty() ? sum : current.get(current.size() - 1)); i >= 1; i--) {  current.add(i);  generateSummands(sum - i, current, summands);  current.remove(current.size() - 1);  } }  Iterable<int[]> getSummands(int sum) {  var s = new ArrayList<int[]>();  generateSummands(sum, new ArrayList<>(), s);  return s; } |
|  | SELECT COUNT(\*) FROM Summands(:sum) | int getCountSummands(int sum) {  int[][] dp = new int[sum + 1][sum + 1];   // Base case: There's one way to make sum 0  for (int n = 0; n <= sum; n++)  dp[0][n] = 1;   // Fill the DP table  for (int n = 1; n <= sum; n++)  for (int s = 1; s <= sum; s++) {  // Exclude the current number  dp[s][n] = dp[s][n - 1];  // Include the current number if it's valid  if (s >= n)  dp[s][n] += dp[s - n][n];  }  return dp[sum][sum]; }  @Test public void test() {  int sum = 10;   var l = new ArrayList<Integer>();  var s = new ArrayList<int[]>();  generateSummands(sum, l, s);  assert s.size() == getCountSummands(sum); } |
|  | int[] getSortedArray(int[] elements) {  return IntStream.*of*(elements).sorted().toArray(); }  SELECT DISTINCT getSortedArray(summands) FROM Sum WHERE sum = :sum  SELECT DISTINCT (SELECT v1 FROM Sum.summands ORDER BY v1) FROM Sum WHERE sum = :sum | // Summands (positive) combinations without  // repetition (order does matter)  Iterable<int[]> getOrderedSummands(int sum) {  var s = new ArrayList<int[]>();  generateOrderedSummands(sum, new ArrayList<>(), s);  return s; } |

**Atom 1**

**Atom N**

**Operator  
 (PSQL)**

# Product overview

**Test-case (SMT-LIB)**

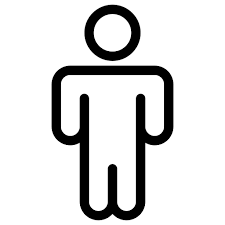
**Shema #1 (Dual to #2)**

**+**

**Ideal-code**

**EvalLoop**

**Run-time**



**LLM**

**Text**

**(NL)**

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

**Atom 1**

**Atom N**

**Operator  
 (PSQL)**

**EvalLoop**

**Run-time**

**Ideal-code**

**+**

**+**

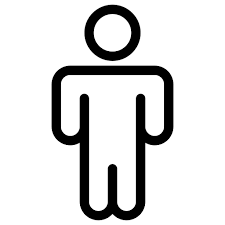
**Text**

**(NL)**

**Test-case (SMT-LIB)**

**LLM**

**Real-code**



|  |  |  |
| --- | --- | --- |
| **#** | **Representation** | **Interpretation** |
|  | Real-code | Programming patterns |
|  | Natural Language Text | Comments, Docs |
|  | Functional Dependencies (atomic code) |  |
|  | Relational Decomposition (PSQL) | SQL: join, group\_by, order\_by, min/max/count aggregation |
|  | First-Ordered Logic Models (SMT-LIB) | Arithmetic, Induction, ADT, Bit-vector, etc |

Reasoning:

* Dynamic Symbolic Execution / Program Model Checking
* Trajectories of optimization