Relational Solver for Java Generics Type System

Peter Lozov Dmitry Kosarev Dmitry Ivanov Dmitry Boulytchev

Saint Petersburg State University

STEP

December 15, 2023

JAVA Language

- One of the most popular high-level programming languages
- An active research topic is approaches and tools for JAVA code verification and testing
- One of the prominent method for software testing is ***symbolic execution***

JAVA Generics Type Solver

Our experience shows that **JAVA *generics type solver*** is a crucial part of symbolic execution engine

- Difficult to implement directly
  - $\alpha \prec\!\!\prec$ `List<AtomicBoolean>` $\wedge$ `Vector<T>` $\prec\!\!\prec \alpha$
- ***Subtyping relation*** in JAVA with generics is undecidable
- The verifier can be implemented according to the **JAVA Language Specification**[1] (JLS)
  - Relational programming will allow the verifier to be used as a solver

[1] `https://docs.oracle.com/javase/specs/jls/se20/jls20.pdf`

JAVA Generics Type System

JAVA type subsystem we are dealing with contains

- `class Object`
- `interface List<T>`
  `extends Collection<E>`
- `class TreeMap<K, V>`
  `extends AbstractMap<K, V>`
  `implements NavigableMap<K, V>, Cloneable`
- `interface Term<T extends Term<T>>`
- `void foo(Collection<? extends Destroyable> x)`
- `void bar(Collection<? super Integer> x)`

Why do We Need the JAVA Generics Type Solver?

```
static <T> Set<T> makeSingleton(T firstElement) {
    Set set;
    if (firstElement instanceof Integer) {
        set = new TreeSet<T>();
    }
    else if (firstElement instanceof String) {
        set = new HashSet<T>();
    }
    else {
        throw new IllegalArgumentException("Incorrect generic parameter");
    }
    set.add(firstElement);
    return set;
}
```

Why do We Need the JAVA Generics Type Solver?

```
static <T> Set<T> makeSingleton(T firstElement) {
  ...
}

public static void main(String[] args) {
        var set1 = makeSingleton(1);    // OK
        var set2 = makeSingleton("2"); // OK
        var set3 = makeSingleton(3.0); // Exception
}
```

Why do We Need the JAVA Generics Type Solver?

```
static <T> Set<T> makeSingleton(T firstElement) {
    ...
}

public static void main(String[] args) {
        var set1 = makeSingleton(1);    // OK
        var set2 = makeSingleton("2"); // OK
        var set3 = makeSingleton(3.0); // Exception
}
```

Runtime error instead of **compile time error**

Why do We Need the JAVA Generics Type Solver?

Automatic synthesis of test input data by type of arguments

```
void foo(Collection<? extends Destroyable> x) {
    ...
}
```

Why do We Need the JAVA Generics Type Solver?

Automatic synthesis of test input data by type of arguments

```
void foo(Collection<? extends Destroyable> x) {
    ...
}
```

Need to find an arbitrary instantiated subtype of

`Collection<? extends Destroyable>`

Relational Programming

- Approach based on the idea of describing programs as relations
  - Can be considered as a branch of ***logic programming***
  - Without non-relational constructs such as side effects or extra-logical features
  - Uses ***interleaving search strategy***, which is known to be complete
- Conventional relational language is MINIKANREN[2]
  - Initially embedded DSL for SCHEME/RACKET
  - Ported to many host languages such as SCALA, HASKELL, JAVA, etc.
  - We use a strongly-typed implementation for OCAML, called OCANREN

[2] `http://minikanren.org/`

Relational Reverse Computations

- miniKanren allows to express ***reverse computations***
- Some complicated programs considered as an inversions of simpler programs
  - List sorting $\Longleftrightarrow$ All permutations generating
  - Type inference $\Longleftrightarrow$ Type inhabitation problem
- In particular, ***solvers*** are inversions of ***verifiers***
  - Verifiers is often easier to implement

Functional vs. Relational Addition Implementation

```
let rec add x y =
  match x with
  | Z    → y
  | S xs → S (add xs y)
```

```
let rec addᵒ x y z =
  ocanren {
    x ≡ Z ∧ z ≡ y ∨
    fresh xs, zs in
      x ≡ S xs ∧
      z ≡ S zs ∧
      addᵒ xs y zs}
```

Functional vs. Relational Addition Implementation

```
let rec add x y =
  match x with
  | Z    → y
  | S xs → S (add xs y)
```

$$
\begin{array}{l}
\textbf{let rec}\ \texttt{add}^o\ \texttt{x y z} = \\
\quad \textbf{ocanren}\ \{ \\
\qquad \texttt{x} \equiv \texttt{Z} \wedge \texttt{z} \equiv \texttt{y} \vee \\
\qquad \textbf{fresh}\ \texttt{xs, zs}\ \textbf{in} \\
\qquad\quad \texttt{x} \equiv \texttt{S xs} \wedge \\
\qquad\quad \texttt{z} \equiv \texttt{S zs} \wedge \\
\qquad\quad \texttt{add}^o\ \texttt{xs y zs}\}
\end{array}
$$

$$\texttt{add}^o\ \texttt{(S Z) (S Z)}\ \alpha \Longrightarrow [\alpha = \texttt{S S Z}]$$

Functional vs. Relational Addition Implementation

```
let rec add x y =
  match x with
  | Z     → y
  | S xs  → S (add xs y)
```

```
let rec addᵒ x y z =
  ocanren {
    x ≡ Z ∧ z ≡ y ∨
    fresh xs, zs in
      x ≡ S xs ∧
      z ≡ S zs ∧
      addᵒ xs y zs}
```

$$\texttt{add}^o\ (\texttt{S Z})\ (\texttt{S Z})\ \alpha \Longrightarrow [\alpha = \texttt{S S Z}]$$

$$\texttt{add}^o\ \alpha\ (\texttt{S Z})\ (\texttt{S S Z}) \Longrightarrow [\alpha = \texttt{S Z}]$$

Functional vs. Relational Addition Implementation

```
let rec add x y =
  match x with
  | Z    → y
  | S xs → S (add xs y)
```

```
let rec addᵒ x y z =
  ocanren {
    x ≡ Z ∧ z ≡ y ∨
    fresh xs, zs in
      x ≡ S xs ∧
      z ≡ S zs ∧
      addᵒ xs y zs}
```

$$\texttt{add}^o\ \alpha\ \beta\ (\texttt{S S Z}) \Longrightarrow \begin{bmatrix} \alpha = \texttt{Z}, & \beta = \texttt{S S Z}; \\ \alpha = \texttt{S Z}, & \beta = \texttt{S Z}; \\ \alpha = \texttt{S S Z}, & \beta = \texttt{Z} \end{bmatrix}$$

Relational Conversion

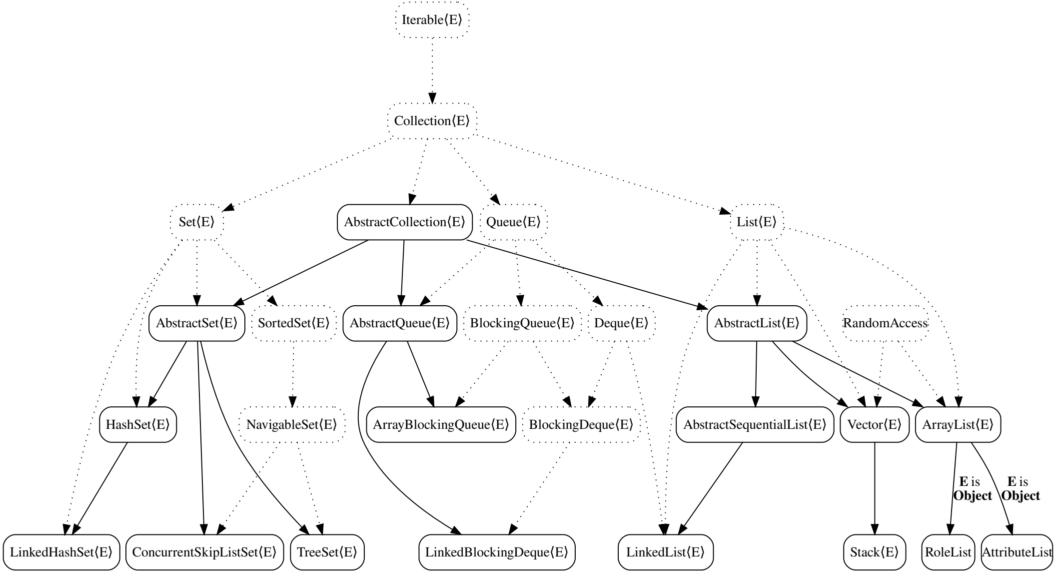
- In many cases it is easier to obtain relational program from functional one
- We use ***typed relational conversion*** tool, called NOCANREN
- In practice we mix hand-written and converted relational code

Relational Solver for JAVA Generics Type System

Solving ***a system of subtyping inequations*** for JAVA generic types with free variables

- Using ***relational programming*** techniques and verifier-to-solver approach
- Applying a number of problem-specific optimizations for boosting the performance

Java Type Table



Direct Subtyping Relation

$$
\begin{array}{rcll}
C\left\langle \alpha_1^{U_1} \ldots \alpha_k^{U_k} \right\rangle & \prec & S\langle T_1 \ldots T_n\rangle & , \quad S \text{ is a direct supertype of } C \\
C\langle T_1 \ldots T_k\rangle & \prec & C\langle S_1 \ldots S_k\rangle & , \quad \forall i\,.\, S_i \supseteq T_i \\
C\langle T_1 \ldots T_k\rangle & \prec & S & , \quad C\langle \lfloor T_1 \rfloor \ldots \lfloor T_k \rfloor\rangle \prec S \\
\bigcap T_i & \prec & T_i & \\
\alpha^{\bigcap T_i} & \prec & T_i & \\
T & \prec & \alpha_T & \\
\textbf{null} & \prec & T & \\
I & \prec & \texttt{Object} & , \quad I \text{ is an interface with no direct superinterface} \\
T[\,] & \prec & S[\,] & , \quad T \prec S \\
\texttt{Object}[\,] & \prec & \texttt{Object} & \\
\texttt{Object}[\,] & \prec & \texttt{Cloneabe} & \\
\texttt{Object}[\,] & \prec & \texttt{Serializable} &
\end{array}
$$

Functional Verifier of Subtyping Relation

	`verify(x, y) =`
If I is an interface then I $\prec$ `Object`	**if** `x` is an interface && `y = Object` **then true**
if T $\prec$ S then T `[]` $\prec$ S `[]`	**elif** `x = t[] && y = s[] && verify(t, s)` **then true**
`Object[]` $\prec$ `Object`	**elif** `x = Object[] && y = Object` **then true**
`Object[]` $\prec$ `Cloneabe`	**elif** `x = Object[] && y = Cloneable` **then true**
`Object[]` $\prec$ `Serializable`	**elif** `x = Object[] && y = Serializable` **then true**
	else false

Relational Direct Subtyping Solver

- Functional verifier was implemented in OCaml
  - Straightforward implementation according to JLS
  - Verifier tests if two given ground types are in the subtyping relation
- Relational verifier was generated using noCanren
  - Verifier searches for all substitutions for free variables in incomplete types to make them subtype of each other
  - We excluded some JLS components from the implementation to ensure only instantiable classes to appear in answers
    - Capture conversion
    - `contains` relation

Unsoundness of Direct Subtyping

By JLS definition of direct subtyping

$$\begin{cases} A \prec \alpha_A^B \\ \\ \alpha_A^B \prec B \end{cases}$$

Unsoundness of Direct Subtyping

By JLS definition of direct subtyping

$$\left\{ \begin{array}{l} A \prec \alpha_A^B \\ \\ \alpha_A^B \prec B \end{array} \right. \quad \Rightarrow \quad A \prec\!\!\prec B$$

Unsoundness of Direct Subtyping

By JLS definition of direct subtyping

$$\begin{cases} A \prec \alpha_A^B \\ \\ \alpha_A^B \prec B \end{cases} \Rightarrow \quad A \prec\!\!\prec B$$

We found no explicitly requirement in JLS that a type variable α_A^B have consistent bounds

Reflexivity of Direct Subtyping

Type $C\langle T_1, \ldots, T_n \rangle$ is a direct subtype of itself by JLS definition of direct subtyping

Reflexivity of Direct Subtyping

Type $C\langle T_1, \ldots, T_n \rangle$ is a direct subtype of itself by JLS definition of direct subtyping

As a result, there are an infinite number of ways to prove that type T_1 is a subtype of T_2

Relational Subtyping Solver

- Reflexive-transitive R^* closure for given relation R can be expressed in MINIKANREN directly by

$$R^*(x, y) \; = \; x \equiv y \vee \exists z \, . \, R(x, z) \wedge R^*(z, y)$$

- Hand-written relation

Relational Subtyping Solver Optimizations

- Simplifying a representation of class and interface identifiers
  - Peano numbers was replaced with integers manually
- Dynamic transitive closure evaluation
  - Two closure implementations for finding subtypes and supertypes
  - Dynamic selection of the optimal implementation depending on the arguments groundness
- Dynamic class table specialization
  - Direct supertypes statically evaluated from the class table for each class and interface
  - Dynamically generated relation depending on the class whose supertypes need to be found
- Removing duplicate answers

Dynamic Transitive Closure Evaluation

$$\overrightarrow{R}^*(x, y) = x \equiv y \vee \exists z \,.\, R(x, z) \wedge \overrightarrow{R}^*(z, y)$$
$$\overleftarrow{R}^*(x, y) = x \equiv y \vee \exists z \,.\, R(z, y) \wedge \overleftarrow{R}^*(x, z)$$

Dynamic Transitive Closure Evaluation

$$\overrightarrow{R}^*(x, y) = x \equiv y \vee \exists z \,.\, R(x, z) \wedge \overrightarrow{R}^*(z, y)$$

$$\overleftarrow{R}^*(x, y) = x \equiv y \vee \exists z \,.\, R(z, y) \wedge \overleftarrow{R}^*(x, z)$$

$$R^*(x, y) = \begin{cases} \overrightarrow{R}^*(x, y), \text{ if } x \text{ is ground} \\ \overleftarrow{R}^*(x, y), \text{ otherwise} \end{cases}$$

Dynamic Class Table Specialization

$$\texttt{sub\_id} = \alpha_1 \quad \texttt{super\_id} = \alpha_2$$

```
let get_superclass_id sub_id super_id =
  (sub_id ≡ 4  ∧  super_id ≡ 1)  ∨

  (sub_id ≡ 5  ∧  super_id ≡ 4)  ∨
  (sub_id ≡ 6  ∧  super_id ≡ 4)  ∨

  (sub_id ≡ 7  ∧  super_id ≡ 5)  ∨
  (sub_id ≡ 8  ∧  super_id ≡ 5)  ∨
  (sub_id ≡ 9  ∧  super_id ≡ 5)
```

Dynamic Class Table Specialization

sub_id = α_1 super_id = 4

```
let get_superclass_id sub_id super_id =
  (sub_id ≡ 4  ∧  super_id ≡ 1)  ∨

  (sub_id ≡ 5  ∧  super_id ≡ 4)  ∨
  (sub_id ≡ 6  ∧  super_id ≡ 4)  ∨

  (sub_id ≡ 7  ∧  super_id ≡ 5)  ∨
  (sub_id ≡ 8  ∧  super_id ≡ 5)  ∨
  (sub_id ≡ 9  ∧  super_id ≡ 5)
```

Dynamic Class Table Specialization

`sub_id` = 8   `super_id` = 5

```
let get_superclass_id sub_id super_id =
  (sub_id ≡ 4  ∧  super_id ≡ 1)  ∨

  (sub_id ≡ 5  ∧  super_id ≡ 4)  ∨
  (sub_id ≡ 6  ∧  super_id ≡ 4)  ∨

  (sub_id ≡ 7  ∧  super_id ≡ 5)  ∨
  (sub_id ≡ 8  ∧  super_id ≡ 5)  ∨
  (sub_id ≡ 9  ∧  super_id ≡ 5)
```

Removing Duplicate Answers

- Transitive closure builds all possible paths between a subtype and a supertype
  - Due to multiple inheritance of interfaces, several paths are possible
  - One path corresponds one answer

Removing Duplicate Answers

- Transitive closure builds all possible paths between a subtype and a supertype
  - Due to multiple inheritance of interfaces, several paths are possible
  - One path corresponds one answer
- Truncation of duplicate branches
  - Let's memorize the already calculated answers
  - In unfinished search branches, we monitor the query variable
  - If query variable corresponds to one of the answers found, we fail this branch early

Removing Duplicate Answers

- If there is only one subtyping inequation in the system, we will simply remove duplicates
  - Transitive closure finds the answer only in the last step
  - Until the answer is found, we cannot determine if it is a duplicate

Removing Duplicate Answers

- If there is only one subtyping inequation in the system, we will simply remove duplicates
  - Transitive closure finds the answer only in the last step
  - Until the answer is found, we cannot determine if it is a duplicate
- If there is more than one inequation, some branches will fail early
  - For the first inequation we are looking for all the answers
  - For the second and subsequent inequations, we verify the answers for the first inequality

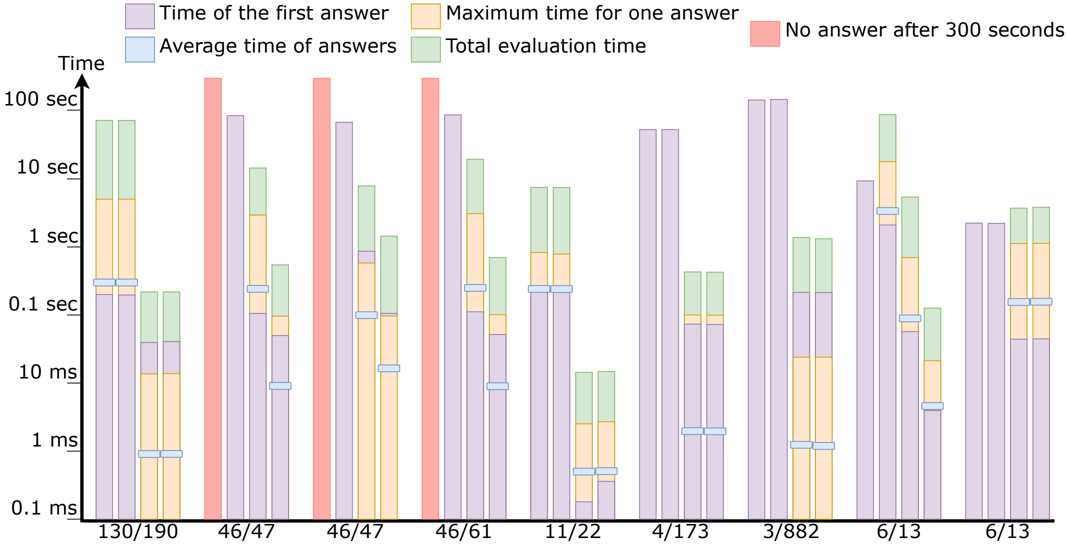
Evaluation

- Real class table containing more than 40000 classes and interfaces
- 9 benchmark queries of various shapes
  - $\alpha \prec\!\!\prec \texttt{java.util.List<Object>}$
  - $\alpha \prec\!\!\prec \texttt{java.util.AbstractCollection<Object>} \wedge$
    $\alpha \prec\!\!\prec \texttt{java.util.RandomAccess} \wedge$
    $\alpha \prec\!\!\prec \texttt{java.util.List<Object>}$
  - $\texttt{javax.management.AttributeList} \prec\!\!\prec \alpha$
  - $\texttt{kotlinx.collections.PersistentVector<Object>} \prec\!\!\prec \alpha \wedge$
    $\texttt{javax.management.AttributeList} \prec\!\!\prec \alpha \wedge$
    $\texttt{com.google.common.collect.ImmutableSortedSet<Object>} \prec\!\!\prec \alpha$
  - $\texttt{kotlinx.collections.PersistentVector<Object>} \prec\!\!\prec \alpha \wedge$
    $\alpha \prec\!\!\prec \texttt{java.util.List<Object>}$

Evaluation

- 4 versions of the solver
  - With no optimizations
  - With dynamic transitive closure evaluation only
  - With dynamic class table specialization only
  - With both optimizations
- 2 quantitative measures
  - Overall number of answers
  - Number of unique answers
- 4 time measures
  - Time of calculating the first answer
  - Maximal time for one answer
  - Average time taken over all answers
  - Total evaluation time

Evaluation Results



Conclusion

- Developed JAVA generic type solver using relational conversion and verifier-to-solver techniques
- Optimized the solver to improve the performance
- Evaluated the solver performance using real world JAVA class table and realistic benchmarks

Future Work

- Sorting the inequalities to improve performance
- Integrating the solver into our symbolic execution engine
- Supporting negative inequalities
  - Type α is **not** subtype of β
  - Type α is **not** supertype of β