QUIC Graphs: Relational Invariant Generation for Containers - Demo

Arlen Cox, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan

QUIC Graphs offer an efficient means for generating abstract domains representing sets and set properties of containers. To demonstrate the analyzer, we present two demo programs using two different underlying scalar domains. These will demonstrate the results of the analysis and the flexibility of the analysis.

Preparing to Run Demos

The analyzer is provided as a VirtualBox virtual machine. Install the virtual machine and run it. It will boot to the Ubuntu desktop. Start the terminal and enter the following commands:

```
$ cd set_analysis/demo
```

Selecting Scalar Domains

Because QUIC graphs build abstract domains from scalar abstract domains we can select from two different abstract domains. The default scalar abstract domain is the APRON polyhedron-based domain, but there are a variety of domains from which to choose:

Flag	**Domain**	
-eq	Equivalence Class	Represents equivalence classes of variables and constants. It uses a naïve implementation of union find and partition refinement to implement basic domain operations. It is implemented in the following file: `~/set_analysis/src/EqClassDomain.ml`
-oct	Octagon	This is the APRON abstract domain for octagons. Octagons represent constraints of the form $\pm x \pm y \pm c \leq 0$ where x and y are variables and c is a constant. It is implemented in the following file: `~/set_analysis/src/ApronDomain.ml`

-poly	Polyhedron	This is the APRON abstract domain for polyhedral. They represent arbitrary convex linear relationships between constants and variables. It is implemented in the following file: `~/set_analysis/src/ApronDomain.ml`

Note that all available abstract domains are relational. This implementation of QUIC graphs relies upon relations in the scalar domain to perform reductions. The domain is sound without a relational scalar domain, but it loses most precision.

Analyzing Copy Using a Polyhedron-based Scalar Domain

Run the this basic analysis using the following command:

```
$ ../Main -poly copy.js0
```

This will run the analysis on the copy program, attempting to prove assertions present in the program. It will also produce auxiliary files showing the generated invariants at many program points in the intermediate language. They are most easily viewed in the generated HTML file:

```
$ firefox copy.html
```

This shows the set and numeric constraints generated throughout the program in red.

Analyzing Copy Using an Equivalence Class Scalar Domain

Identical to the polyhedron-based analysis, run the analysis using an equivalence class domain:

```
$ ../Main -eq copy.js0
```

Because the QUIC graph domain is constructed on the fly, it can easily be switched to this simpler domain that is not part of the APRON library. Similarly this domain is sufficient to prove all properties in this program, so the output is empty.

Analyzing Filter

The filter program is like the copy program except that it does not copy every element. It only copies the elements greater than 10 and less than or equal to 11. This may seem like a trivial program, but it allows us to state an invariant at the end of the program that is expressible in both the polyhedron domain and in the equivalence class domain. However, because we used inequalities in the specification of the filter, the equivalence class domain loses too much information about what is in the set to prove the second property. Note that it can still prove that the result set r is a subset of the input set s:

```
$ ../Main -eq filter.js0
```

Now we see that a property could not be proven. This is due to the imprecision of the equivalence class abstract domain. This is because elements that are included are those greater than 10 and less than or equal to 11, which is not representable using equivalence classes. However polyhedrons can represent this fact:

```
$ ../Main -poly filter.js0
```

If we view the generate invariants, we see that they are quite complex:

```
$ firefox filter.html
```

This complexity comes from history variables and variables introduced by converting the program to A-normal form. If we look for critical facts, such as $r \subseteq \{\omega \in s \mid \omega > 10 \land \omega \leq 11\}$ are represented in the constraints where the verification happens.

Running Benchmarks

The benchmarks and many other sample programs are contained in a different directory:

```
$ cd ~/set_analysis/test.js0
```

They are written in a special language for encoding set constraints. Some of the programs have been translated from functions in the python test suite with the matching name. Others are adapted from the python test suite:

`./factored_original/*`	Programs adapted directly from the Python test suite. They are typically finite state.
`./factored_nondet/*`	Modified programs from the Python test suite. They have been modified to make them more generic. They are typically infinite state.
`./other_tests/*`	Simple sanity check tests

`./*`	A variety of tests developed over the course of the project; some before and some after publication.

The tests from the paper can be run using the `./run_tests.sh` script. The results are in the various log files generated for each benchmark run.