QUIC Graphs: Relational Invariant Generation for Containers

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

University of Colorado Boulder
{arlen.cox, evan.chang, sriram.sankaranarayanan}@colorado.edu

Abstract. Programs written in modern languages perform intricate manipulations of containers such as arrays, lists, dictionaries, and sets. We present an abstract interpretation-based framework for automatically inferring *relations* between the set of values stored in these containers. Relations include inclusion relations over unions and intersections, as well as quantified relationships with scalar variables. We develop an abstract domain constructor that builds a container domain out of a *Quantified Union-Intersection Constraint* (QUIC) graph parameterized by an arbitrary *base domain*. We instantiate our domain with a polyhedral base domain and evaluate it on programs extracted from the Python test suite. Over traditional, non-relational domains, we find significant precision improvements with minimal performance cost.

1 Introduction

Container manipulating programs are ubiquitous. Essentially all high-level programming languages provide a standard library with container types, such as arrays, lists, dictionaries, and sets. In this paper, we investigate static analysis techniques for inferring assertions about the possible set of values that can be stored in containers at run time. Our analysis abstracts containers by the set of elements contained in them to infer facts about (a) the possible set of values in a container; and (b) how these values relate to values stored in other containers. In general, one may envision two main types of static analyses: (1) *content-centric* analyses that infer assertions for the possible sets of values in each container, *in isolation*; or (2) analyses that infer relations between the values stored in various containers, *as-a-whole*.

```
def extendClass(X):
  Y = set([x for x in X
       if x >= c])
  return Y
```

To illustrate this difference, consider the Python code function extendClass in the inset (the name of this function will become clearer below). This function takes a set X and returns a set Y where Y is the subset of elements from X such that each element is greater than or equal to some variable c. An important post-condition of extendClass is $Y \subseteq \{\nu \in X | \nu \geq c\}$, but neither the content-centric nor the as-a-whole analyses can produce this post-condition. The content-centric analysis, which represents each set X and Y as individual variables in a domain for reasoning about values, would produce $Y \subseteq \{\nu | \nu \geq c\}$ where ν ranges over the universe of values. Because

ECOOP 2013.

all values of X are not related to all values of Y in some way, a content-centric analysis cannot represent any relationship between X and Y. As-a-whole analyses, which reason only about the relationships between sets, can produce $Y \subseteq X$, but fail to infer anything about the individual elements of Y. By combining these two classes of analyses, our analysis finds the desired invariant: $Y \subseteq \{\nu \in X | \nu \geq c\}$.

```
def extendClass(D):
  E = {k:v for k,v in a.iteritems()
       if k >= "%"}
  return E
```

The extendClass function is abstracted from a function in Processing.js[1]. A simplified version of the original function is shown in the inset (in Python); the original set version models the key set of this dictionary version. This function copies a dictionary containing a number of values to another dictionary. It only copies those elements that start with letters higher than % in the ASCII table, specifically excluding keys starting with $. These dictionaries are used as objects, and in the context of this framework, $ is interpreted as private and thus should not be copied. Functions like this one are pervasive in programs written in dynamic languages because most run-time structures are implemented using dictionaries (or objects, maps, or tables) and those run-time structures are directly accessible by the developer and can be modified. As a result, previously simple operations such as inheriting a class become complex dictionary manipulations involving copy operations. To statically analyze programs written in dynamic languages, we require powerful new static analysis techniques that can reason about these kinds of functions.

Our analysis tracks (subset) inclusion relations between expressions involving set abstractions of containers through a special graph structure called a QUIC graph. A QUIC graph is a succinct encoding of *set expressions* and *inclusion relations* between them. The expressions represented by a QUIC graph are (1) basic, *atomic sets* that abstract the set of values stored in a container and singletons created by scalar expressions; (2) restricted unions and intersections of the atomic sets; and (3) comprehensions of set expressions through first-order predicates. The predicates are captured by an arbitrary base domain, which can reason about program variables and formal bound variables that represent the scalar-valued contents of a basic set. The QUIC graph is thus a compact structure for storing a conjunction of subset constraints between set expressions. In this paper, we define QUIC graphs and build abstract domain operations over these graphs. The QUIC graph domain is designed to yield a tight integration between the base domain and the QUIC graph domain so that the resulting analysis can transfer facts from one domain to another, quite seamlessly.

The content-centric analysis of containers is rather well understood (e.g., [7, 11, 13]). Such analyses focus on strategies for partitioning or splitting *summary variables* that smash the contents of the container into an essentially weak-updated scalar variable. These techniques are orthogonal and complementary to our work here. With summary variables, one might capture independent comprehensions, such as $X \subseteq \{\nu | p(\nu)\} \wedge Y \subseteq \{\nu | p'(\nu)\}$ for some predicates p and p'. If the predicates p and p' are the same or related, then these facts may indirectly imply

[1] `http://processingjs.org/`

a relation between X and Y but essentially only through their contents. On the flip side, the pure container-as-a-whole approach would track relations directly between X and Y without characterizing their contents. Some existing containers-as-a-whole approaches incorporate some fixed content reasoning (e.g., [23]). In this paper, we present a tight integration of these two approaches with domains for reasoning about scalar variables and their relations to the set elements. As a result, the QUIC graph domain promises to be a lot more powerful than a simple conjunction of both individual domains.

We have implemented the QUIC graph domain for a simple imperative programming language with integers and sets (of integers). The language captures basic arithmetic over integers and operations over sets such as union, intersections, difference, insertion/deletion of elements, and iteration over sets. We implemented analyzers using the QUIC graph domain, as well as two domains representing the content-centric and container-as-a-whole approaches. The evaluation was carried out by translating a variety of set manipulating programs from the Python test suite. The results are quite promising: the QUIC graph domain is more precise than the other domains, proving more properties than a simple combination of a content-centric approach and a container-as-a-whole approach.

Contributions: In this paper, we make the following contributions:

- We identify the need for simultaneous reasoning about containers as-a-whole *and* their contents to enable modular, precise reasoning of container-manipulating programs (Sect. 2).
- We describe QUIC graphs to represent universally-*Q*uantifed *U*nion and *I*ntersection set *C*onstraints in a canonical manner using a hypergraph data structure. We build an abstract domain (functor) based on QUIC graphs. A novel aspect of our domain is the use of predicate edge labels to capture set comprehensions (Sect. 3).
- We present a framework for *inference* using QUIC graphs. We show how to utilize the structure of QUIC graphs to compute all logical implications of a given QUIC graph. We present the inference procedure for strengthening base domain invariants within a QUIC graph. Finally, we show how laziness significantly improves the cost of inferring consequences of QUIC graphs, and describe an efficient implementation (Sect. 4).
- We define an abstract domain using QUIC graphs with inference and show how all domain operations and reductions are easily implemented using lazy inference (Sect. 5).
- We evaluate the effectiveness of our abstract domain on a set of benchmarks from the Python test suite. We find that for a reasonable performance overhead, our abstract domain is significantly more precise than either a content-centric or a container-as-a-whole approach and unlike the content-centric and container-as-a-whole approaches can automatically prove most properties specified in the Python test suite for set operations (Sect. 6).

$$
\begin{aligned}
\text{program} &::= \text{decl}^* \ \text{stmt}^* \\
\text{decl} &::= \textbf{int}\ \text{scalarVar} \mid \textbf{set}\ \text{setVar} \\
\text{stmt} &::= \text{scalarVar} \ \textbf{:=}\ \text{scalarExpr} \\
&\mid \text{setVar} \ \textbf{:=}\ \text{setExpr} \\
&\mid \textbf{loop}\ \text{stmt}^* \\
&\mid \textbf{branch}\ \text{stmt}^*\ \textbf{orelse}\ \text{stmt}^* \\
&\mid \textbf{havoc}\ \text{setVar} \\
&\mid \textbf{assume}\ \text{conditional} \mid \textbf{assert}\ \text{conditional} \\
\text{setExpr} &::= \emptyset \mid \{\text{scalarExpr}\} \mid \text{setVar} \mid \text{setExpr} \cup \text{setExpr} \\
&\mid \text{setExpr} \cap \text{setExpr} \mid \text{setExpr} \setminus \text{setExpr} \\
\text{scalarExpr} &::= \text{scalarVar} \mid \text{scalarConst} \mid \text{scalarUnary}(\text{scalarExpr}) \\
&\mid \text{scalarBinary}(\text{scalarExpr}, \text{scalarExpr}) \mid \textbf{choose}(\text{setExpr}) \\
\text{conditional} &::= \text{scalarConditionals} \mid \text{setExpr} \subseteq \text{setExpr} \mid \text{scalarVar}\ \textbf{in}\ \text{setVar} \\
\text{setVar} &::= \mathrm{X}, \mathrm{Y}, \mathrm{Z} \\
\text{scalarVar} &::= \mathrm{x}, \mathrm{y}, \mathrm{z} \\
\text{scalarConst} &::= \mathrm{c}
\end{aligned}
$$

Fig. 1. An imperative, set-manipulating programming language. A sequence of a symbol α is written as α^*.

2 Overview

In this section, we walk through inferring the desired post-condition for the extendClass example from Sect. 1 to highlight the main challenges in obtaining precise combined content-as-a-whole invariants that motivate our design of the QUIC graph domain. At a high-level, deriving the desired post-condition for the extendClass function requires the careful application of transitive closure of inclusion constraints, an effective reduction [6] strategy with base domain elements, and a non-trivial join operator.

2.1 Set Language

We assume an imperative programming language with scalar values and set values whose elements are scalars, shown in Figure 1. We assume scalar operations (e.g., addition, subtraction, multiplication, and division) are given as unary or binary operators (scalarUnary or scalarBinary, respectively). For convenience, we fix a single scalar type (integers) in our language. Unless otherwise mentioned, sets are assumed to range over this type (integers). However, our framework is quite general. Because we only assume the base domain is a sound abstract domain, we can handle a variety of types including integers, floats, and strings by using base domains designed to reason over scalar variables of those types. We do not address sets of sets or complex structures such as lists in this paper. However, our framework can be extended to handle these types by instantiating with more complex base domains such as another domain for sets.

For the purposes of analysis, we take an input program and lower the program to introduce additional instrumentation variables. The lowering converts

```
1  def extendClass(X) {
2   Y := ∅;
3   for (x in X) {
4     if (x > c) {
5       Y := Y ∪ {x};
6     }
7   }
8   return Y;
9  }
```

```
1  def extendClass(X) {
2   Xo := ∅; Xi := X; Y := ∅;
3   loop {
4     assume Xi ≠ ∅;
5     x := choose(Xi); Xi := Xi \ {x};
6     branch {
7       assume x > c; Y′ := Y ∪ {x};
8       Y := Y′;
9     }
10    orelse {
11    }
12    Xo := Xo ∪ {x};
13  }
14  assume Xi = ∅;
15  return Y;
16 }
```

Fig. 2. Left: the extendClass example that filters positive elements from a set X into a set Y. Right: its lowered version.

all loops (e.g., **for-in**) into a single non-deterministic **loop** construct and all conditional statements into a non-deterministic **branch** construct. The **havoc** statement is an arbitrary value assignment for modeling unknown effects, and the **assume** statement is used to encode the conditions in each branch. One key instrumentation transforms each **for-in** loop over a set X to introduce two sets Xo, Xi that are assumed to partition X (i.e., $\mathrm{X} = \mathrm{Xi} \uplus \mathrm{Xo}$). The set Xo represents all variables that have been iterated over thus far. Likewise, Xi represents the elements of X that remain to be iterated over. The iteration order is assumed to be non-deterministic. The loop exits when $\mathrm{Xi} = \emptyset$ or alternatively $\mathrm{Xo} = \mathrm{X}$. We assume that iterations over a set X do not modify X in the body of the loop (as is the standard semantics for container iteration).

Example 1. Fig. 2 (left) shows a translation of the Python extendClass example from Sect. 1 to an imperative, set-manipulating program. This program filters elements from an input set X greater than or equal to c into a set Y. The set Y is a variable introduced in the translation to name the set being constructed by the comprehension. The lowered version of this program is also shown alongside (right).

2.2 Motivating Example

In Fig. 3, we annotate the lowered version of the extendClass from Fig. 2. At program point 1, set Xi is initialized to X, while Xo and Y are initialized to the empty set $\emptyset$. The extendClass loop begins at point 2. An arbitrary element x is

```
    def extendClass(X) {
1     Xo := ∅; Xi := X; Y := ∅;
2     loop {
3           [ X = Xi ∪ Xo ∧ Y ⊆ {ν ∈ Xo | ν > c} ]
4       assume Xi ≠ ∅; x := choose(Xi); Xi := Xi \ {x};
5       branch {
6         assume x > c; Y′ := Y ∪ {x};
7.a             [ X = Xi ∪ {x} ∪ Xo ∧ Y ⊆ {ν ∈ Xo | ν > c}
                  ∧ Y′ = Y ∪ {x} ∧ x > c ]
7.b             [ X = Xi ∪ {x} ∪ Xo ∧ Y ⊆ {ν ∈ Xo | ν > c}
                  ∧ Y′ = Y ∪ {x} ∧ {x} = {ν ∈ {x} | ν > c} ∧ x > c ]
          Y := Y′;
8               [ X = Xi ∪ {x} ∪ Xo ∧ Y ⊆ {ν ∈ Xo ∪ {x} | ν > c} ∧ x > c ]
        }
        orelse {
9               [ X = Xi ∪ {x} ∪ Xo ∧ Y ⊆ {ν ∈ Xo | ν > c} ]
        }
10      Xo := Xo ∪ {x};
11    }
12    assume Xi = ∅;
13          [ Y ⊆ {ν ∈ X | ν > c} ]
      return Y;
    }
```

Fig. 3. Inferring QUIC graph invariants on the extendClass example.

chosen out of set X at point 4 with the **choose** statement and removed from set Xi. The element x is added to set Y in the first case (point 6) of the non-deterministic **branch**, while set Y is left unchanged in the other (point 9). The final assignment in the loop (at point 10) simply moves the element x into set Xo to continue the iteration.

The boxed formulas in Fig. 3 are program invariants that we infer (under the pre-condition true), that is, the fixed-point result of an abstract interpretation. Our goal is to be able to derive the post-condition $Y \subseteq \{\nu \in X | \nu > c\}$, that is, output set Y is a subset of the positive elements of the input set X, at program point 13. Here and in the rest of this paper, we use ν as the bound variable for all comprehensions. In this figure, we selectively show the key constraints needed to derive this post-condition. We first observe that although inclusion constraints plus comprehension expressions are sufficient to state the desired post-condition, the inferred loop invariant at point 3 requires a more expressive set expression language (i.e., union expressions). It is straightforward to see that

this loop invariant $\mathrm{X} = \mathrm{Xi} \cup \mathrm{Xo}$ along with the loop exit condition $\mathrm{Xi} = \emptyset$ implies the desired post-condition and that the initial state where $\mathrm{Xi} = \mathrm{X} \wedge \mathrm{Xo} = \mathrm{Y} = \emptyset$ implies the loop invariant.

Let us consider the fixed point iteration of the loop (i.e., showing that loop invariant is inductive and thus consecutes) and focus on the transition to invariant 7.a—the difference with respect to the loop invariant is shown shaded. This transition begins with the addition of element x to set Y. The **assume** is reflected in the invariant with a base domain constraint $\mathrm{x} > \mathrm{c}$ shown to the right in the box. It is necessary to transfer the relationship between Y and Xo to Y' and Xo to generate the desired function post-condition. Knowing when to transfer these relationships by transitivity is critical to both performance and precision. The QUIC graph representation allows us to limit the guesswork of when to apply the various transitivity rules to derive additional facts.

In invariant 7.b, we show a reduction step that transfers information from the base domain to the QUIC graph domain. In particular, we have that $\mathrm{x} > \mathrm{c}$, so it is also the case that $\forall \nu \in \{\mathrm{x}\} \,.\, \nu > \mathrm{c}$ (i.e., applying a $\forall$-introduction rule). In terms of QUIC graphs, we have that any constraint of the form $\{\mathrm{x}\} \subseteq \left\{\nu \in \bar{T} | B[\nu]\right\}$ can be strengthened to $\{\mathrm{x}\} \subseteq \left\{\nu \in \bar{T} | B[\nu] \wedge \nu > \mathrm{c}\right\}$ where B is a predicate described by the base domain and $\bar{T}$ is any basic set expression, including $\{\mathrm{x}\}$. For abstract interpretation, the conjunction $\wedge$ becomes a meet operator $\sqcap$ on base domain elements. Thus, we have that $\{\mathrm{x}\} \subseteq \{\nu \in \{\mathrm{x}\} | \nu > c\}$ as shown in invariant 7.b. This "seed" constraint is sufficient to derive other ones, such as $\{\mathrm{x}\} \subseteq \{\nu \in \mathrm{Y}' | \nu > c\}$, by transitivity on demand. The QUIC graph structure with singleton known-scalar sets enables an eager transfer of information from the base domain coupled with lazy propagation of this information (see Sect. 4).

This reduction step is used for deriving the invariant at point 8. At point 8, we show the invariant derived from 7.b by projecting out Y (and then renaming Y' to Y). From invariant 7.b, we can intuitively see that $\mathrm{Y}' \subseteq \{\nu \in \mathrm{Xo} | \nu > \mathrm{c}\} \cup \{\nu \in \{\mathrm{x}\} | \nu > \mathrm{c}\}$ by applying transitivity (and that union with any set is monotonic), so we have that $\mathrm{Y}' \subseteq \{\nu \in \mathrm{Xo} \cup \{\mathrm{x}\} | \nu > \mathrm{c}\}$, which gets to our desired result after projecting the old Y and renaming Y' to Y. It is not difficult to check this step; rather, the main challenge in an automated analysis is guessing that these are the appropriate steps to obtain the desired invariant. For example, both $\mathrm{Y}' \subseteq \{\nu \in \mathrm{X} \cup \{\mathrm{x}\} | \nu > \mathrm{c}\}$ and $\mathrm{Y}' \subseteq \mathrm{Xo} \cup \{\mathrm{x}\}$ are sound over-approximations of the projection that are syntactically close, but now we have lost too much precision to get our desired post-condition. From the QUIC graph perspective, this derivation is a propagation of facts across nodes and edges that can be done on demand by the *lazy closure* (see Sect. 4).

The invariant at point 9 in the unchanged case entails the invariant at which we just arrived at point 8 (except for the base domain constraint), so the result of the join at program point 10 is the invariant at point 8 without the base domain constraint $\mathrm{x} > \mathrm{c}$, and after the assignment, we get exactly the loop invariant at point 3.

In summary, it is difficult to derive enough constraints via transitivity and strong enough ones via reduction from the base domain. On the flip side, transitive

closure, even with restricted union and intersection constraints, is exponential (see Sect. 4). The QUIC graph representation eases this tension by representing inclusion constraints over unions, intersections, and comprehensions in a canonical manner that facilitates on-demand propagation of information.

3 QUIC Graphs

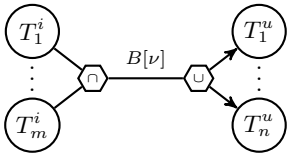
A *Quantified Union/Intersection Constraint graph* is a graph data structure that represents inclusions between set expressions. Throughout the rest of the paper we use the notation X, Y, Z with subscripts to represent set variables and x, y, z with subscripts to represent base domain variables. The special variable ν will be used as a formal bound variable for set comprehensions, as will be explained in this section. The symbol T represents *atomic set expressions* – one of three possible elements: the empty set $\emptyset$, a singleton set containing a base domain variable $\{x\}$ or a set variable X. The symbols $\bar{T}^i, \bar{T}^u$ represent a number of Ts in an intersection or a union respectively.

Definition 1 (QUIC edge). *Let* $T_1^i, \ldots, T_m^i = \bar{T}^i$ *and* $T_1^u, \ldots, T_n^u = \bar{T}^u$ *be symbols representing finite sets and* B *be a base domain abstract state involving a bound variable* ν*, acting as a predicate where* $\top$ *is* true *and* $\bot$ *is* false*. A* QUIC edge *is a constraint*

$$\bigcap_{i=1}^{m} T_i^i \subseteq \left\{ \nu \in \bigcup_{j=1}^{m} T_j^u \,\middle|\, B[\nu] \right\} \text{ represented using the notation } \dot{\bigcap} \bar{T}^i \,\dot{\subseteq}\, \dot{\bigcup} \bar{T}^u \Big|_B$$

which is an edge in an edge labeled hypergraph.

We use dots above the set operators simply to make clear that they are part of the syntax of a QUIC constraint or a QUIC edge. Graphically a QUIC edge is represented as a hyperedge:



For convenience, if there is only one T in the union (respectively intersection), we elide the union (respectively intersection) hex from the figure. Additionally, if the label $B[\nu]$ is top in the base domain, we elide the label from the edge.

Definition 2 (QUIC graph). *A QUIC graph* $G \in \widetilde{G}$ *is an edge labeled hypergraph constructed of QUIC edges. It represents a conjunction of constraints where each constraint corresponds to one QUIC edge in the graph. A QUIC graph has the following syntax:*

$$\begin{aligned} G ::= \;& G_1 \wedge G_2 \\ \mid \;& \dot{\bigcap} \bar{T}^i \,\dot{\subseteq}\, \dot{\bigcup} \bar{T}^u \Big|_B \end{aligned}$$

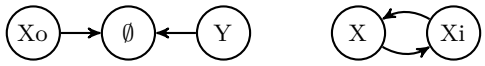
A QUIC graph is a canonical representation of the set of conjoined constraints. It is designed to be compact and to allow efficient inference operations (see Sect. 4).

We provide a series of examples to demonstrate the QUIC graph representation.

Example 2 (Basic QUIC graphs). Consider that would be produced after line 1 from the example in Fig. 3:

$$\mathrm{Xo} \subseteq \emptyset \ \wedge \ \mathrm{Xi} \subseteq \mathrm{X} \ \wedge \ \mathrm{X} \subseteq \mathrm{Xi} \ \wedge \ \mathrm{Y} \subseteq \emptyset$$

This is represented as a QUIC graph:



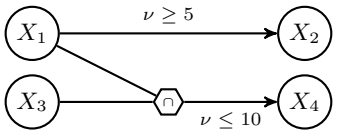
Unlike the constraint formula, the symbols X, Xi, and $\emptyset$ only occur once in the graph. This makes the relationships more clear and eliminates possible redundancy.

Conjoining multiple constraints produces a QUIC graph with multiple edges and including unions or intersections requires a hypergraph to show the relationships:

Example 3. We wish to encode the formula:

$$\dot{\bigcap} X_1 \dot{\subseteq} \dot{\bigcup} X_2 \Big|_{\nu \geq 5} \wedge \dot{\bigcap} X_1, X_3 \dot{\subseteq} \dot{\bigcup} X_4 \Big|_{\nu \leq 10}.$$

We draw this using the following hypergraph:



To be practical, a representation for set constraints cannot stand alone. There must be a way to represent relationships between sets and base domain variables as well. To do this we construct a combined domain where elements are pairs $(G, B) \in \widetilde{S} = \widetilde{G} \times \widetilde{B}$ where G is a QUIC graph domain instance and B is a base domain instance. Note that the base domain has two roles: (a) it labels edges in the QUIC graph and (b) it captures invariants on base domain variables.

To specify the concretization for both QUIC graphs and QUIC graphs combined with an external base domain, a concretization (where γ is overloaded for all concretizations) for the base domain is required:

$$\gamma : \widetilde{B} \to \mathcal{P}((\textsc{BaseVar} \to \textsc{BaseVal}) \times \mathcal{P}(\textsc{BaseVal}))$$

The symbol BaseVar is all base domain variables, BaseVal is all base domain values. This is a non-standard concretization because given some abstraction, it

returns a set of functions that map base domain variables to base domain values and for each function, there is a corresponding set that contains the base domain values to which the bound variable ν can be assigned. This is used to define concretization for QUIC graphs

Definition 3 (Concretization). *The concretization γ of a QUIC graph G has the following type, given that* SetVar *is all set domain variables:*

$$\gamma : \widetilde{G} \to \mathcal{P}((\textsc{SetVar} \to \mathcal{P}(\textsc{BaseVal})) \times (\textsc{BaseVar} \to \textsc{BaseVal}))$$

Where the result is a set of pairs of functions (η, η_B), where η maps set variables to sets of base domain values and η_B maps base domain variables to base domain values. These two functions mappings are valid with respect to constraints both on the sets and on the base domain.

The concretization function is then defined as such:

$$\gamma(G_1 \wedge G_2) \stackrel{def}{=} \{(\eta, \eta_B) | (\eta, \eta_B) \in \gamma(G_1) \text{ and } (\eta, \eta_B) \in \gamma(G_2)\}$$

$$\gamma\left(\dot{\bigcap} [T_1^i, \cdots, T_n^i] \dot{\subseteq} \dot{\bigcup} [T_1^u, \cdots, T_m^u]\Big|_B\right) \stackrel{def}{=}$$

$$\left\{ (\eta, \eta_B) \;\middle|\; \begin{array}{l} (\eta_B, \bar{b}) \in \gamma(B) \text{ and} \\ \text{for all } \nu.\ \left(\nu \in \eta(T_1^i) \text{ and } \cdots \text{ and } \nu \in \eta(T_n^i)\right) \\ \qquad \text{implies } \left(\nu \in \bar{b} \text{ and } (\nu \in \eta(T_1^u) \text{ or } \cdots \text{ or } \nu \in \eta(T_m^u))\right) \end{array} \right\}$$

The concretization for a combined domain S is the same set of pairs (η, η_B), so the type and concretization follow:

$$\gamma : \widetilde{G} \times \widetilde{B} \to \mathcal{P}((\textsc{SetVar} \to \mathcal{P}(\textsc{BaseVal})) \times (\textsc{BaseVar} \to \textsc{BaseVal}))$$

$$\gamma((G, B)) \stackrel{def}{=} \left\{(\eta, \eta_B) | (\eta, \eta_B) \in \gamma(G) \text{ and } (\eta_B, \bar{b}) \in \gamma(B)\right\}$$

Expressivity: We now discuss the expressivity limitations of QUIC graphs. As such, QUIC graphs allow unions, intersections and comprehensions of sets but in a restricted manner. We motivate some of our design choices here.

The first expressivity restriction arises from the manner in which comprehension is introduced in our language. For instance, we are able to express inclusions of the form $X \subseteq \{\nu \in Y \mid B[\nu]\}$ through a QUIC edge. However, QUIC graphs as presented here cannot express the reverse inclusions of the form $\{\nu \in X | B[\nu]\} \subseteq Y$. There are two main reasons for this restriction: (a) Representing reverse inclusions requires a new type of edge relation along with fresh reduction rules for this edge. Additionally, there are many interactions between this new type of relation and existing relations that need to be captured. (b) Reverse inclusions require an abstract domain that implements the underapproximate semantics whereas the inclusions used in QUIC graphs use the standard overapproximate abstract semantics. This ensures that existing abstract domains can be integrated with QUIC graphs without introducing new domain

Set Operation		Operation using Union/Intersection
$X \subseteq Y \uplus Z$	$\Leftrightarrow$	$X \subseteq Y \cup Z \ \wedge \ Y \cap Z \subseteq \emptyset$
$Y \uplus Z \subseteq X$	$\Leftrightarrow$	$Y \subseteq X \ \wedge \ Z \subseteq X \ \wedge \ Y \cap Z \subseteq \emptyset$
$X \subseteq Y \setminus Z$	$\Leftrightarrow$	$X \subseteq Y \ \wedge \ X \cap Z \subseteq \emptyset$
$Y \setminus Z \subseteq X$	$\Leftrightarrow$	$Y \subseteq X \cup Z$

Fig. 4. Encoding set difference and disjoint union in QUIC graphs.

operations. A full theory of QUIC graphs that captures both types of relations will be tackled in the future.

The other expressivity limitation arises from the introduction of union and intersection operations. Note that the relation $X \cup Y \subseteq Z$ can be equivalently expressed simply as $X \subseteq Z \ \wedge \ Y \subseteq Z$. Likewise the intersection $Z \subseteq X \cap Y \ \Leftrightarrow \ Z \subseteq X \ \wedge \ Z \subseteq Y$. This motivates the direction of the union and intersection hyperedges in QUIC graphs. We do not directly represent relations between nested unions and intersections unless special existentially quantified variables are permitted in the graph.

Example 4. For instance, the relation $(X_1 \cup X_2) \cap X_3 \subseteq X_4$ cannot be expressed unless a special existentially quantified set variable X_5 is introduced with the constraints

$$\dot{\bigcap} X_5 \dot{\subseteq} \dot{\bigcup} X_1, X_2\Big|_\top \wedge \dot{\bigcap} X_1 \dot{\subseteq} \dot{\bigcup} X_5\Big|_\top$$
$$\wedge \dot{\bigcap} X_2 \dot{\subseteq} \dot{\bigcup} X_5\Big|_\top \wedge \dot{\bigcap} X_5, X_3 \dot{\subseteq} \dot{\bigcup} X_4\Big|_\top$$

Finally, relations involving disjoint unions and set difference can also be represented directly using QUIC graph as shown in Figure 4.

Self Loops: Self-loops on QUIC graphs are quite useful to encode assertions that are true of the contents of X in relation to the scalar program variables $x_1, \ldots, x_n$.

Example 5. Let X be a set and x be a program variable. We wish to express that every element in X is between x and $x+10$. We do so in the QUIC graph domain using the self-loop from X to itself labeled by the assertion $\nu \geq x \ \wedge \ \nu \leq x+10$. In effect, the loop represents the containment relation written

$$X \subseteq \{\nu \in X \mid \nu \geq x \ \wedge \ \nu \leq x+10\} \quad \text{or} \quad \forall \nu \in X.\, \nu \geq x \ \wedge \ \nu \leq x+10\,.$$

QUIC graphs naturally represent relationships between set variables, singleton sets and the empty set. However, QUIC graphs do not necessarily represent all possible relationships. In the next section, we show how to derive other relationships from those already in a QUIC graph.

4 Closure

The *closure* of a QUIC graph adds all of the implied logical facts to both the QUIC graph and the base domain. Most of the domain operations of a QUIC graph are defined in terms of the closure by the application of inference rules to add edges to a QUIC graph and strengthen the existing edge labels.

Inference rules are shown in full in Fig. 5. We use three judgment forms. One states when given a combined domain of a QUIC graph and a base domain, $S = (G, B)$, a particular containment relationship is derivable. If the relationship is derivable, the inference judgment provides a predicate B_e that holds on that relationship. The judgment takes the form

$$(G, B) \vdash \dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_e} ,$$

where $\bar{T}^i$ is the set of intersected vertices and $\bar{T}^u$ is the set of unioned vertices. This judgment relies on an auxiliary judgment $B \vdash x = y$ where x and y are base domain variables. This judgment states when an equality between variables is derivable from a base domain element (and is supplied by the base domain). We also define a judgment $(G, B) \vdash x = y$ that states when an equality can be derived from set constraints.

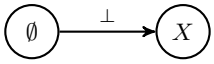
4.1 Inference Rules

We now explain the inference rules for QUIC graphs in detail. A brief explanation of the rules follow.

The (Emp) inference rule generates QUIC graph edges from the empty set to any node, labeled with the bottom base domain element $\bot$ (i.e., with the $\emptyset$ concretization or is equivalent to the predicate **false**).

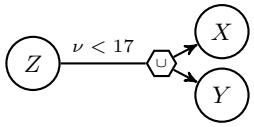
Example 6. Consider the QUIC graph G

By applying (Emp), we get the QUIC graph G':



The (Self-Loop) and (Self-Prop) inference rules generate and strengthen the labels present on self loops in QUIC graphs. The strengthening takes information from an outgoing edge and propagates it back to the self loop.

Example 7. Consider the QUIC graph G



$$\frac{}{(G \wedge \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}, B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}} \text{(In-Graph-R)} \qquad \frac{}{(G,B) \vdash \dot{\bigcap} \emptyset \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{\bot}} \text{(Emp)}$$

$$\frac{}{(G,B) \vdash \dot{\bigcap} T \dot{\subseteq} \dot{\bigcup} T\Big|_{\top}} \text{(Self-Loop)} \qquad \frac{(G,B) \vdash \dot{\bigcap} T \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap} T \dot{\subseteq} \dot{\bigcup} T\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap} T \dot{\subseteq} \dot{\bigcup} T\Big|_{B_a \sqcap B_b}} \text{(Self-Prop)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}}{(G,B) \vdash \dot{\bigcap} T, \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}} \text{(Add-Left)} \qquad \frac{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}}{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} T, \bar{T}^u\Big|_{B_e}} \text{(Add-Right)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} T^i \dot{\subseteq} \dot{\bigcup} T_1^u, \cdots, T_m^u\Big|_{B_0} \qquad (G,B) \vdash \dot{\bigcap} T_j^u \dot{\subseteq} \dot{\bigcup} T_j^u\Big|_{B_j}, \text{ for } j = 1 \cdots m}{(G,B) \vdash \dot{\bigcap} T^i \dot{\subseteq} \dot{\bigcup} T_1^u, \cdots, T_m^u\Big|_{B_0 \sqcap \left(\bigsqcup_{j=1}^m B_j\right)}} \text{(Union-Prop)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} T_j^i \dot{\subseteq} \dot{\bigcup} T_j^i\Big|_{B_j}, \text{ for } j = 1 \cdots m \qquad (G,B) \vdash \dot{\bigcap} T_1^i, \cdots, T_n^i \dot{\subseteq} \dot{\bigcup} T_u\Big|_{B_0}}{(G,B) \vdash \dot{\bigcap} T_1^i, \cdots, T_n^i \dot{\subseteq} \dot{\bigcup} T^u\Big|_{B_0 \sqcap \left(\prod_{j=1}^m B_j\right)}} \text{(Inter-Prop)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} \bar{T}_a^i \dot{\subseteq} \dot{\bigcup} T, \bar{T}_a^u\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap} T, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup} \bar{T}_b^u\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap} \bar{T}_a^i, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup} \bar{T}_a^u, \bar{T}_b^u\Big|_{B_a}} \text{(Union-Trans)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} \bar{T}_a^i \dot{\subseteq} \dot{\bigcup} T\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap} T, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap} \bar{T}_a^i, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_a \sqcap B_b}} \text{(Inter-Trans)}$$

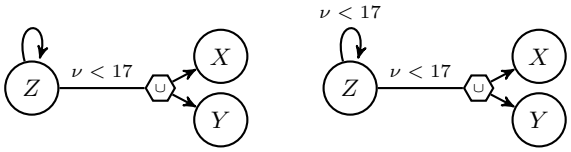
$$\frac{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e}}{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_e \sqcap B}} \text{(Base-Str)} \qquad \frac{B \vdash x = y}{(G,B) \vdash \dot{\bigcap} \{x\} \dot{\subseteq} \dot{\bigcup} \{y\}\Big|_{\nu = x}} \text{(Eq-Base)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u\Big|_{B_a \sqcap B_b}} \text{(Double-Edge)}$$

$$\frac{(G,B) \vdash \dot{\bigcap} \{x\} \dot{\subseteq} \dot{\bigcup} \{y\}\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap} \{y\} \dot{\subseteq} \dot{\bigcup} \{x\}\Big|_{B_b}}{(G,B) \vdash x = y} \text{(Eq-Set)}$$

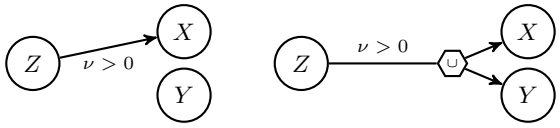
Fig. 5. Inference rules for closure of QUIC graphs. **Notation:** $\bar{T}^i, \bar{T}^u$ are sets of vertices, T are individual vertices of the graph, B, B_a, B_b are base abstract states and x, y are base domain variables.

Evaluating the (Self-Loop) rule on Z gives G' on the left. Evaluating the (Self-Prop) rule on Z and $X \cup Y$ pushes the predicate $\nu < 17$ onto the self loop at Z, giving G'' on the right:



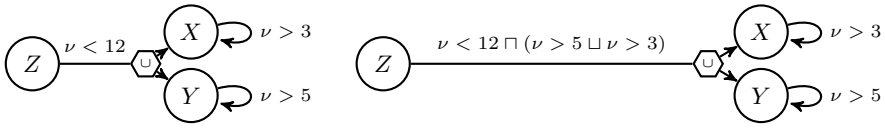
The (Add-Right) rules allows adding extra elements to the union on the right-hand side of an inclusion. (Add-Left) is the dual rule for intersection.

Example 8. Consider the QUIC graph G on the left. Applying (Add-Right) to Z and X, adding Y, gives the QUIC graph G' on the right:



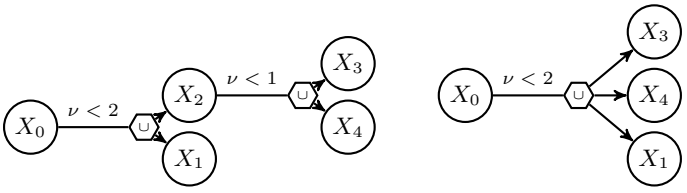
The (Union-Prop) rule pushes information from self loops backward onto edges. (Inter-Prop) performs the same operation on intersections.

Example 9. Consider the QUIC graph G on the left. Applying (Union-Prop) to Z, X and Y yields the graph shown to the right.



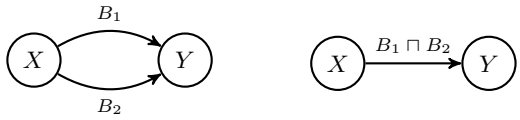
The (Union-Trans) rule combines two union edges to produce a single union edge. This rule loses information from one of the edges, but that information can be regained through the application of (Union-Prop). We write $\sqcap$ and $\sqcup$ for the meet and join operator in the base domain, respectively. (Inter-Trans) does the same for intersection without losing information.

Example 10. Consider the QUIC graph G on the left. The two union edges are combined to produce the union edge on the right. Even though $\nu < 1$ is a stronger constraint than $\nu < 2$, the resulting constraint is the weaker $\nu < 2$.



The (Double-Edge) rule merges two edges between the same vertices into a single edge. QUIC graphs do not track multiple edges between the same two vertices, so a duplicate edge must immediately be converted to a single edge with this rule.

Example 11. Consider the two edges on the left. Since QUIC graphs cannot represent these, they are combined into the single edge on the right.



The rule (Base-Str) strengthens any edge in the graph with the current facts from the base domain B. The rule (Eq-Base) strengthens relationships in the set domain by adding a constraint on the bound variable. The latter also uses equality in the base domain to infer equalities in the set domain. Oppositely, (Eq-Set) uses equalities in the set domain to infer base domain equalities.

Definition 4 (Closure). *Let (G, B) be a QUIC graph and a base domain predicate. The closure (G^*, B^*) is the conjunction of all*

$$\dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_e} \text{ such that } (G, B) \vdash \dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_e}$$

and the constraining of B with all equalities $x = y$ given by the judgment $(G, B) \vdash x = y$.

4.2 Soundness

We first define soundness for systems of inference rules. For a QUIC graph analysis to be sound, the underlying system of inference rules must be sound.

Definition 5 (Inference Soundness). *An inference is sound if the following two conditions hold:*

1. *if $(G, B) \vdash \dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_e}$, then $\gamma((G, B)) \subseteq \gamma\left(\dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_e}\right)$.*
2. *if $(G, B) \vdash x = y$, then for all $(\eta, \eta_B) \in \gamma((G, B))$, we have that $\eta_B(x) = \eta_B(y)$.*

Let us assume that $\widetilde{B}$, the base domain, is a sound abstract domain [5].

Theorem 1. *The inference rules in Figure 5 are sound according to Definition 5.*

4.3 Complexity

Closure of a QUIC graph is potentially expensive since the number of edges in the closure can be exponential in the worst case.

Theorem 2. *There are $O(2^n)$ possible hyperedges in a QUIC graph with n vertices.*

Without "tactics" to apply the rules cleverly in an implementation, the inference over QUIC graphs is intractable.

4.4 Lazy Inference Implementation

We now discuss how the inference operation is implemented in our approach. The goal of the implementation is to avoid a blowup in the number of graph edges and running time each time a closure is to be computed. Lazy inference is a tactic that computes an effective closure on demand. It is composed of many strategies. We describe the most important concepts used in our implementation. (a) *Simplification:* We apply many simplification passes to keep the QUIC graph in a canonical form. This automatically takes into account many of the inference rules from Fig. 5. (b) *Lazy inference:* Instead of computing the closure eagerly and adding a set of extra edges to the graph, we do so lazily whenever edge membership queries are issued by the abstract domain. (c) *Partial closure:* We note that many of the edges generated by a closure are not necessarily useful as invariants for proving properties. Therefore, we have implemented heuristics that choose edges to query. We call this process *candidate generation* since it affects which invariant candidates are considered by our analyzer at each step.

Simplification: Simplification consists of many different parts. The first simplification deals with edges from the empty set $\emptyset$. As such, they do not contribute to the inference. We assume that these edges implicitly exist but do not represent them.

Next, we consider *equivalence classes* of set variables. Two sets X, Y are equivalent if $X \subseteq Y \wedge Y \subseteq X$. Equivalence classes are identified using a maximal strongly connected component algorithm on the QUIC graph. Equivalence classes of sets can be compacted and one representative is chosen using a pre-defined variable ordering. All membership queries involving members of equivalence classes are first rewritten in terms of the representative members of the classes.

The (Double-Edge) rule is implicitly implemented by our data structure whenever we attempt to add two edges between the same set of nodes. Finally, we use (Self-Loop), (Self-Prop), (Union-Prop) and (Inter-Prop) to propagate labels and add new edges between representatives of equivalence classes. These rules also strengthen the labels on edges.

Lazy Inference: Next, we implement inference on demand by applying the inference rules to decide if a queried edge is present in the graph. This is performed by iterating the (Union-Trans) and (Inter-Trans) rules to compute transitive closures.

Candidate Generation: The computation of a lazy inference is driven by the choice of candidate query edges that we wish to add to the graph. To this end, a candidate generation heuristic is used in our implementation to choose candidate invariant facts. There are many possible heuristics for generating candidate query edges. We use set expressions that appear in the program including properties to be proved as a source of edges to keep in the partial closure. Another choice includes edges that are generated through transfer functions such as assignments. Once generated, we keep an edge as a candidate edge for future inference computations.

5 Domain Operations

In this section, we will discuss the abstract domain operations over the reduced product domain of QUIC graphs and the base domain $\widetilde{B}$ for base domain variables.

Notation: Let G be a QUIC graph. We will write $G[X \leftarrow X_0]$ to denote the graph obtained by changing the label of vertex X to X_0. We extend the notation to set expressions so that $T[X \leftarrow X_0]$ denotes the substitution of X by X_0 for each occurrence in the expression T.

We define abstract domain transition functions using semantic functions:

$$[\![\text{stmt}]\!]^{\text{S}} : \widetilde{S} \to \widetilde{S}$$

These functions are parameterized by stmt, which is a command in the language of sets and base domain operations. It takes an abstract state $S = (G, B) \in \widetilde{S}$ composed of a graph G and a base domain element B and returns an abstract state S' that represents the state after having executed command stmt on S.

Simple Transfer Functions: The transfer functions for some basic assignment states are represented below. In each case, the result may not be closed. Therefore, we may apply inference on the result, if necessary.

$$[\![\textbf{havoc } \text{X}]\!]^{\text{S}}(G, B) \stackrel{\text{def}}{=} (G[X \leftarrow X_0], B) \qquad X_0 \text{ is fresh}$$

$$[\![\text{X}\textbf{:=}\emptyset]\!]^{\text{S}}(G, B) \stackrel{\text{def}}{=} \left(G[X \leftarrow X_0] \ \wedge \ \begin{array}{c} X \end{array} \xrightarrow{\ \bot\ } \begin{array}{c} \emptyset \end{array} , B \right) \qquad X_0 \text{ is fresh}$$

$$[\![\text{X}\textbf{:=}\text{T}]\!]^{\text{S}}(G, B) \stackrel{\text{def}}{=} \left(G[X \leftarrow X_0] \ \wedge \ \begin{array}{c} X \end{array} \rightleftarrows \begin{array}{c} T \end{array} , B \right) \qquad X_0 \text{ is fresh}$$

The command **havoc** X assigns X to a non-deterministic value. Rather than projecting the vertex X from the graph, we rename the existing vertex to a fresh variable X_0. The vertex X_0 remains in the graph as a *history variable*. Operations such as join and widening will eliminate the necessary history variables, ensuring that they do not propagate out of scope. However history variables will exist for as long as possible as this may allow additional relationships to be inferred.

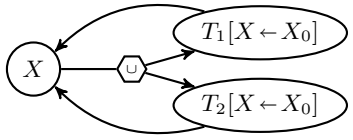
The command X**:=**$\emptyset$ assigns X to the empty set. Because it is performing a destructive update to X, X is renamed to a history variable X_0 as is standard when performing a destructive update. This leaves the symbol X completely unconstrained so that when constraints are added to X, those are the only constraints on X. The added constraint here is $X \subseteq \emptyset$, labeling it with the strongest possible predicate $\bot$.

The command X**:=**T assigns a set element T to X. This creates the two edges representing both $X \subseteq T$ and $T \subseteq X$. The edge labels are set to $\top$ and thus not shown as all the information from B can be added to these edges through the inference procedure.

Meet (Intersection): The meet of two abstract states $(G_1, B_1) \sqcap (G_2, B_2)$ is the conjunction of the set constraints and meet in the base domain (i.e., $(G_1 \wedge G_2, B_1 \sqcap B_2)$ where we overload the $\sqcap$ notation for both the QUIC graph and the base domain). Note that viewing the set constraints as graphs, meet is the union of two graphs.

Set Assignment Rule: We now complete domain operations for assignments of the form $X{:=}T_1 \cup T_2 \cdots \cup T_n$ (similarly for $X{:=}T_1 \cap \cdots \cap T_n$). The basic idea is to replace X by a history variable X_0 and introduce hyper-edges to capture the new relations formed. For simplicity, we consider the case $n = 2$

$$[\![\mathtt{X{:=}T_1 \cup T_2}]\!]^{\mathrm{S}}(G, B) \stackrel{\text{def}}{=} \left(G[X \leftarrow X_0] \ \wedge \ \right.$$



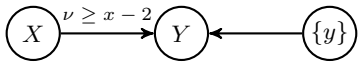
$$\left. , \ B \right)$$

X_0 is fresh

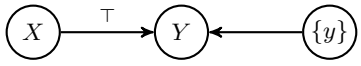
Intersection, disjoint union and set difference operations are similar. They rename X to a fresh variable X_0 and rename T_1 and T_2 similarly, if appropriate. Then a constraint that represents the appropriate equality is added to the QUIC graph.

Base Domain Assignment: An assignment to the base domain variables $x := e$ will result in three changes: (a) applying the assignment to the base domain element B, (b) applying the assignment to each edge label in the QUIC graph G and (c) any singleton node in the graph that involves x needs to be updated either by computing its post-condition w.r.t to the assignment, if invertible or renamed to a fresh set variable X_0 for a destructive assignment. All applications invoke the base domain transfer function and thus rely on the base domain for introduction (or not) of history variables. We illustrate this through a simple example.

Example 12. Consider the QUIC graph G



and let $B : y \geq x$. Consider the destructive assignment $x := y + 1$. The transfer function yields the QUIC graph G':



with the assertion $B' : \ x = y + 1$. We compute a partial closure on the result, which effectively pushes the constraint $x = y + 1$ on the edges of the graph G'.

Choose: The **choose** command selects an element from a set and assigns it to a base domain variable. It takes quantified information from the set domain and applies it to the resulting base domain variable. The strategy to handle $\texttt{x} := \texttt{choose}(T)$ for an abstract state (G, B) is the following:

1. Perform an inference operation on (G, B) giving (G^*, B^*).
2. Extract the base domain constraint B_e from a self-loop on T:

$$G^* = G' \ \wedge \ \dot{\bigcap} T \dot{\subseteq} \dot{\bigcup} T \Big|_{B_e} \ .$$

3. Replace the bound variable ν in B_e with a fresh base domain variable y giving B_y. This process transfers all the facts that apply to elements in set T and applies them to the variable y.
4. Compute the meet $B' = B^* \sqcap B_y$. This transfers those facts about y to the base domain.
5. Perform the destructive update $\texttt{x} := y$ on (G^*, B') to get the result of choose.

Projection: The projection of a base-domain variable x from (G, B) is performed by (a) projecting x from B and (b) projecting x from each label in G. These are performed by calling the projection defined in the base domain $\widetilde{B}$.

The projection of a vertex T from the QUIC graph G first computes its partial closure (G^*, B^*). Next, we remove all conjuncts involving the vertex T from G^* to obtain the projection.

Join: Let (G_1, B_1) and (G_2, B_2) be the arguments for the join operation. We first compute the partial closure of (G_1^*, B_1^*) of G_1 and likewise the partial closure (G_2^*, B_2^*) of (G_2, B). The join (G, B) is then defined where $B = B_1^* \sqcup B_2^*$ and G is all conjuncts

$$\dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u \Big|_{B_1 \sqcup B_2}$$

where there exists some G_1' and G_2' such that

$$G_1^* = G_1' \ \wedge \ \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u \Big|_{B_1} \quad \text{and} \quad G_2^* = G_2' \ \wedge \ \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u \Big|_{B_2} \ .$$

Widening: As such the QUIC graph domain is a product of a finite graph domain and an abstract base domain. Widening is required iff the base domain does not satisfy the ascending chain condition. The basic widening algorithm is precisely the same as the join operation with the modification that the base domain widening operation is applied for each QUIC edge instead of widening.

6 Evaluation

We now present a preliminary evaluation of our prototype analyzer. The QUIC graphs domain introduced in this paper has two main aspects: (a) it enables relational reasoning between sets to prove that one set (expression) is contained

in another; and (b) it allows us to qualify relations between sets using base domain predicates, in effect allowing us to reason with *set comprehension*. The evaluation in this section is intended to answer the following questions:

1. How much does each of the two ingredients (relations between sets + set comprehensions) add to the ability of the analysis to prove properties of commonly encountered use cases?
2. What is the added cost due to each of the two ingredients to the overall domain?

To carry out the evaluation, we introduce two simplified versions of the QUIC graphs domains namely the 'set' and 'elem' domains. (A) The 'set' domain allows relations between sets but no comprehensions. This is a realization of a container-as-a-whole approach. We create this domain by using the trivial two element $(\bot, \top)$ base domain. (B) The 'elem' domain disallows relations between sets but allows us to reason about the contents of the set using a *summary variable*. This is a realization of a content-centric domain. To simulate this domain, we modify the original QUIC graphs domain to just allow self loops on nodes as the only possible edge. In effect, the predicate on such an edge must be true of every element in the set. Furthermore, the process is exactly equivalent to introducing a *summary variable* for each set variable and performing a base-domain analysis using this summary variable.

Benchmarks: The next step is to choose a series of benchmarks that represent common motifs for set (container) usage in dynamic languages. To evaluate our approach we used two sets of benchmarks. We designed our analysis using the first set of benchmarks, which exercise four commonly occurring operations on containers 'copy', 'filter', 'partition' and 'merge'. We then ran our analysis, *unmodified*, on translated versions of all of the programs from the Python test suite [24] for dictionaries and sets. We removed extraneous parts of these tests and simply translated the core part of each program to an equivalent program in our input language. Each test has a set of pre-defined assertions to be established by our analyzer.

Results: Figure 1 summarizes the results of our analysis run on these benchmarks on an Apple MacBook Pro, on a 2.2GHz Intel Core i7 with 8GB RAM running Mac OS X 10.8.2. We now discuss the comparison of precision and running time. The memory required by most analysis runs was under 150 MB. It is quite clear from the results table that the combination of relational reasoning and comprehension using base domain predicates is quite powerful. Whereas the QUIC graphs domain can prove a majority of the properties, restricting it either by removing the comprehensions (set) or removing the relations between sets (elem) are both able to prove much fewer properties. Furthermore, every property proved by these domains is also proved by the QUIC graphs domain.

The comparison of costs indicates that the QUIC graphs domain is 1.2× slower than the set domain. However, it is 9× slower than the elem domain. The difference in performance is entirely expected since the QUIC graphs domain has

Table 1. Results on a set of small benchmarks. **Base Vars**: # of base domain (numerical) variables, **Set Vars**: # of set variables, **Num Prp**: # of assertions to be proved, **T**: Time taken (seconds), **#I**: number of iterations of abstract interpreter before convergence. – represents a time out (600 seconds)

	Base	Set	Num	# Proved			Time Taken (Iterations)					
ID	Vars	Vars	Prp	QG	set	elem	QG_T	(#I)	set_T	(#I)	$elem_T$	(#I)
copy	1	6	2	2	2	0	0.2	(2)	0.2	(2)	0	(2)
filter	4	6	2	2	1	0	0.6	(3)	0.5	(3)	0.1	(2)
generic_max	3	8	6	3	0	0	0.9	(6)	0.6	(6)	0.2	(4)
merge	2	14	2	1	1	0	0.6	(4)	0.6	(4)	0.1	(4)
partition	4	8	4	4	2	0	1.1	(3)	0.9	(3)	0.2	(2)
b_filter	6	6	2	2	0	0	0.7	(3)	0.6	(3)	0.1	(2)
b_map	9	7	2	2	2	2	0.2	(5)	0.3	(5)	0.1	(4)
b_max_min	3	4	1	1	1	1	0.4	(3)	0.3	(3)	0.1	(2)
b_reduce	7	4	1	0	0	0	0.4	(3)	0.3	(3)	0.1	(2)
iter_ind	20	12	1	1	0	0	84.4	(39)	67.9	(39)	6.8	(14)
mul_ret	9	2	2	2	0	0	0.2	(6)	0.1	(6)	0.1	(6)
nest_dep	5	7	1	0	0	0	2.2	(12)	2.2	(12)	0.4	(6)
resize1	15	5	5	4	0	0	1.7	(18)	1.1	(18)	1	(18)
simp_cond	11	5	4	3	0	0	4.6	(12)	1.6	(12)	1.3	(12)
simp_nest	9	10	2	0	0	0	–	(1399)	–	(1612)	0.7	(6)
srange	6	2	2	2	0	0	0.1	(6)	0.1	(6)	0.1	(6)
Total			37	29	9	3	98.3	(125)	77.3	(125)	11.4	(92)

to perform a lot more reasoning steps. We also find that one example times out (after 600 seconds).

Limitations. While QUIC graphs are an effective abstract domain, but some properties were not proven due to imprecision in the analysis. There are four sources of this imprecision: (1) incomplete candidate generation, (2) imprecise base domain, (3) no cardinality reasoning, and (4) syntactic restrictions within QUIC graphs.

To reduce needless inference in many examples, we use candidate generation (Sect. 4) to reduce the number of rule applications. Because candidate generation reduces the potential edges that can result from a join, it can cause the join to lose more information than is strictly necessary. This is this cause for many of the failures in Table 1, including the failure to prove one of the properties in 'merge'. Further work on candidate generation is quite important.

Because the base domain is also an abstract domain, it is imprecise and may not be able to represent some necessary relationship. This is especially the case when there is a transformation applied to all elements of a set. The base domain must be able to represent that transformation that occurs to each element as a relation. In this test suite there is only one test that exercises this ability and the relations are all representable as linear relationships, so this imprecision does

not affect the results. However, if this were a problem, a new base domain could be selected because QUIC graphs are agnostic to the base domain.

The QUIC graphs domain does not track the cardinality of sets. As has been previously shown [18], cardinality can strengthen relationships, and therefore in QUIC graphs, cardinality constraints would create additional closure rules. For example, if for some set X we have that $\{1, 3, 7\} \subseteq X$ and that $|X| = 3$, then we can infer that $X \subseteq \{1, 3, 7\}$. It is possible that cardinality information could provide sufficient information to prove properties that failed in this test suite, but this information could likely be inferred in another way (such as better candidate generation) because most sets in the test suite have unknown cardinality.

QUIC graphs are syntactically restricted to allow comprehensions only on one side of a subset relationship. Reverse inclusions (Sect. 3) are not supported. We hypothesize that the ability to know that an element exists in a set will be beneficial when abstracting other containers using sets.

7 Related Work

There exists a large number of container analyses, mostly focused on arrays. Although there are many different approaches, the problem is fundamentally the same: partitioning an array in order to summarize different segments. Gopan et al. [13], Halbwachs et al. [15] and Cousot et al. [7] use an abstract interpretation framework with materialization and summarization. Therein, the partitions are inferred from the structure of the program. Seghir et al. [25] perform this in the context of predicate abstraction, similarly to abstract interpretation. Jhala et al [16], McMillan [22], Kovacs et al. [17], and Dillig et al. [9, 10] use theorem provers to perform this partitioning. Our approach does not use a partitioning scheme except for the special case of loops that iterate over sets. Furthermore, these approaches do not, in general, reason about comprehensions or relate the contents of different arrays.

There are several alternative approaches to reasoning about container manipulations. Marron et al. [20, 21] used a shape analysis to emulate data storage of containers. They used appropriate inductive predicates with carefully tuned, simplified implementations of the containers to get an automatic analysis. Dillig et al. [11] extended their previous work on arrays to more generic containers. Their approach uses base domain predicates as constraints on the sets of keys for maps. This is a highly tuned example of what we have been calling a content-centric domain. Their approach does not directly infer relationships between containers. However, they can indirectly infer relations through data invariants that relate their contents. Finally, Pham et al. [23] introduced a relational domain for sets. Their domain is similar to ours in that it is designed to directly represent relations between sets. Their approach represents what we term an as-a-whole approach for the most part. It does support a base domain of uninterpreted functions and can be precise for a restricted class of programs. Because they support only uninterpreted functions for the base domain, they have been able to implement some under-approximations required to infer equalities with predicate

comprehensions, but this base domain does not support any manipulation or reductions and thus is weaker than domains that we support.

The invariant generation procedure of [14] could infer many of the invariants that we infer given a sufficiently expressive list of predicate templates. They select from templates to use for quantified facts. As a result, their analysis requires user input and guidance for success, but the approach does offer some additional generality. Bouajjani et al. [3] present a similar, more automatic approach to dealing with quantified invariants, by pre-selecting appropriate templates for many applications. They apply their work to linked list structures and support multiple bound variables to be able to maintain sortedness properties. Like the work of [20], they use a shape analysis framework to approximate the shape and data of lists, while maintaining quantified side conditions on an integer base domain.

The QUIC graph data structure is similar to a formalization of constraint graphs [2, 12] use to prove complexity of satisfaction of constraints [1]. While the encoding is similar, there is no need for base domain labels since constraint graphs are unable to place quantified restrictions on the contents of the sets they constrain. In general, constraint graphs do represent sets, but they are intended to use sets to analyze programs rather than analyzing set-manipulating programs.

The decision procedures community has largely solved this problem of relational containers, but only for the problem of entailment checking. Decision procedures do not perform invariant generation. Bradley et al. [4] demonstrated decision procedures for arrays and other containers. The Z3 SMT solver implements an optimized version [8] of these decision procedures to speed up these problems. Also, Lam et al. [19] and Kuncak [18] developed a system that simultaneously reasons about sets and their cardinalities relationally. Since these tools solve the decision problem rather than the inference problem, they are incomparable, however the optimizations used in [8] are similar to operations that we define in our closure because they are Boolean algebra-like operations.

8 Conclusion

We have demonstrated a relational abstract domain for sets that combines a content-centric analysis with a container-as-a-whole approach. This is achieved through a new representation for set constraints called QUIC graphs that simplifies the representation of set expressions and inclusion relations that use comprehensions. Our evaluation of this domain shows that a combined approach using QUIC graphs is quite effective in practice. It outperforms weaker alternatives such as a content-centric approach and a container-as-a-whole approach.

Going forward, we are developing tighter integration of our domain to analyze a range of data structures in dynamic languages such as Python and JavaScript. Our future work will complete the QUIC graph structure to encode *reverse inclusion relations* (see Section 3), track set cardinalities more effectively and enable the tracking of auxiliary data that can help extend this analysis to specific structures such as arrays, lists and dictionaries.

Acknowledgments. We would like to thank Xavier Rival and the CUPLV group for insightful discussions on this work, as well as the anonymous reviewers for the helpful comments. This work is supported in part by the National Science Foundation through grants CCF-1055066 and CCF-1218208.

References

[1] Alexander Aiken, Dexter Kozen, Moshe Y. Vardi, and Edward L. Wimmers. The complexity of set constraints. In *Computer Science Logic (CSL)*, 1994.

[2] Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhendong Su. A toolkit for constructing type- and constraint-based program analyses. In *Types in Compilation (TIC)*. 1998.

[3] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2012.

[4] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2006.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, 1977.

[6] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*, 1979.

[7] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Principles of Programming Languages (POPL)*, 2011.

[8] Leonardo de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer Aided Design (FMCAD)*, 2009.

[9] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: beyond strong vs. weak updates. In *European Symposium on Programming (ESOP)*, 2010.

[10] Isil Dillig, Thomas Dillig, and Alex Aiken. Symbolic heap abstraction with demand-driven axiomatization of memory invariants. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2010.

[11] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *Principles of Programming Languages (POPL)*, 2011.

[12] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, 1997.

[13] Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. In *Principles of Programming Languages (POPL)*, 2005.

[14] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *Principles of Programming Languages (POPL)*, 2008.

[15] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *Programming Language Design and Implementation (PLDI)*, 2008.

[16] Ranjit Jhala and Kenneth McMillan. Array abstractions from proofs. In *Computer-Aided Verification (CAV)*. 2007.

[17] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering (FASE)*, 2009.

[18] Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, 2007.

[19] Patrick Lam, Viktor Kuncak, and Martin Rinard. Hob: a tool for verifying data structure consistency. In *Compiler Construction (CC)*, 2005.

[20] Mark Marron, Darko Stefanovic, Manuel Hermenegildo, and Deepak Kapur. Heap analysis in the presence of collection libraries. In *Program Analysis for Software Tools and Engineering (PASTE)*, 2007.

[21] Mark Marron, Mario Méndez-Lojo, Manuel Hermenegildo, Darko Stefanovic, and Deepak Kapur. Sharing analysis of arrays, collections, and recursive structures. In *Program Analysis for Software Tools and Engineering (PASTE)*, 2008.

[22] Kenneth McMillan. Quantified invariant generation using an interpolating saturation prover. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[23] Tuan-Hung Pham, Minh-Thai Trinh, Anh-Hoang Truong, and Wei-Ngan Chin. FixBag: A fixpoint calculator for quantified bag constraints. In *Computer-Aided Verification (CAV)*. 2011.

[24] Python. Python 2.7.3 test suite. `http://www.python.org`, 2012.

[25] Mohamed Nassim Seghir, Andreas Podelski, and Thomas Wies. Abstraction refinement for quantified array assertions. In *Static Analysis (SAS)*, 2009.