QUICr: A Reusable Library for Parametric Abstraction of Sets and Numbers

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

University of Colorado Boulder

Abstract. This paper introduces QUICr, an abstract domain combinator library that lifts any domain for numbers into an efficient domain for numbers and *sets* of numbers. As a library, it is useful for inferring relational data invariants in programs that manipulate data structures. As a testbed, it allows easy extension of widening and join heuristics, enabling adaptations to new and varied applications. In this paper we present the architecture of the library, guidelines on how to select heuristics, and an example instantiation of the library using the APRON library to verify set-manipulating programs.

1 Introduction

Programs do not consist entirely of scalar variables. In nearly all programming languages, collections are either implemented as a library or built-in as a first-class type. Therefore, when verifying programs, it is vital to support containers as well as scalar values. In the decision procedures community, this is widely recognized with support for arrays, sets, and maps [1–3], but when invariant generation is concerned, such as in abstract interpretation [4], only arrays have been carefully considered [5–11], leaving other containers rarely explored [12–14]. Given that there is a plethora of abstract domains for reasoning about scalars [15, 16], it is necessary to build abstract domains that not only reason about containers, but also interact efficiently and precisely with existing domains for scalars. The best way to ensure interaction is, rather than building abstract domains for containers, building domain combinators that construct abstract domains for containers from existing abstract domains for numbers. Recently, such a domain for arrays was created [5], and a domain for sets was created [12]. This paper describes an implementation of the domain for sets called QUICr[1].

The implementation of the domain for sets is an OCaml functor that takes a numeric abstract domain and builds a domain for simultaneous reasoning about numbers and sets of numbers. It constructs a relational abstract domain such that without knowing the specific contents of a set, that set can be related to another set by equality or subset relationships. This kind of reasoning is useful for a variety of applications:

---

[1] Library available at http://pl.cs.colorado.edu/projects/quicgraphs

- *Whole-program verification of container-manipulating programs* – The QUICr library can reason about constants and known sets as well as unknown sets. It does not conflate known parts with unknown parts unnecessarily and thus can keep track of sets that are partially known and partially unknown. Iteration over these sets as well as addition and removal from these sets is supported allowing the inference of a wide range of program properties.
- *Modular verification of container-manipulating programs* – If a whole program is not known, a precondition can be provided as a relational constraint between sets without knowing any contents. Inferred loop invariants and post-conditions will also be relational with respect to any existing sets. The QUICr library will automatically utilize any capabilities of the underlying numerical domain to express relationships between set variables.
- *Shape/data abstraction combinators* – Advanced domain combinators such as those provided by shape analyzers [17–19] typically cannot to incorporate collection data abstractions into their inductive definitions. To verify both shape and data properties of programs, current analyzers [20] rely on multi-set abstractions. By adding relational domains, data can be related between multiple inductive definitions and thus more precisely represented.
- *Parametric abstraction combinators* – Effective reasoning about sets enables the construction of new domain combinators that do not yet exist today without relational set abstractions. When combining abstract domains [21, 22], it is often useful to be able to express relationships between an unbounded number of elements in each abstract domain. With a relational domain for sets it is possible to express these relationships efficiently and effectively.

The goal of this paper is to both document how to use the QUICr library as well as to inspire thinking about applications for set domains. Towards this goal we provide an overview of the underlying representation as it is implemented in the library (Section 2). This knowledge is sufficient to understand the heuristics currently employed in the library and how the heuristics can be extended to handle additional and different situations that may arise from as-of-yet unknown applications (Section 3). To demonstrate that the QUICr library is precise and efficient with the built-in heuristics, we also give results from some modular verification of container-manipulating programs (Section 4).

2 Overview of QUIC Graphs

The QUICr library provides an implementation of the QUIC graphs [12] abstract domain combinator for numeric domains. It is used as part of an abstract interpretation system (see inset) that not only proves properties of programs, but performs necessary invariant generation. QUICr represents, accumulates and manipulates set constraints, while sharing equality information with the numeric domain in a way similar to
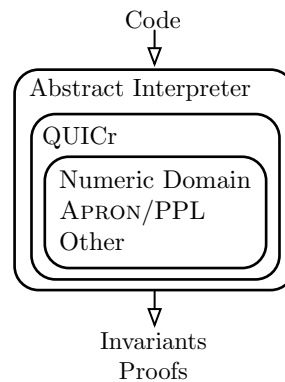
Code

Abstract Interpreter

QUICr

Numeric Domain
APRON/PPL
Other

Invariants
Proofs

Fig. 1. Example QUIC graph and accompanying numeric domain instances, showing a sequence of inferences: (a) is derived by pushing the external numeric domain fact $x \leq 11$ and $\nu < x$; (b) is derived by inferring a new self-loop with $\top$ and pushing (a); (c) is derived by pushing (b) $\sqcup \nu < 10$ and $3 < \nu$; and (d) is derived by inference/transitive closure of the graph.

Nelson Oppen [23]. This combination allows representing and inferring constraints that consist of both set constraints involving union, intersection, subset, and equality, along with numeric constraints that are dictated by the representations used by the underlying numeric domain.

For straight-line code, QUICr can be used to implement a decision procedure and can compute the validity of Hoare triples for set manipulating programs. For example, the inset shows a Hoare triple that can be validated using QUICr instantiated with an equivalence classes-based numeric domain. The QUICr library manipulates the formulas to learn that

$[x = y]$
$A := C \cup \{x\};$
$B := C \cup \{y\}$
$\begin{bmatrix} x = y \wedge A = B \\ \wedge A = C \cup \{x\} \end{bmatrix}$

$A = B$ because they are both equal to C unioned with the same value. The problem that QUICr solves is not only how to efficiently represent and manipulate these formulas to construct a decision procedure, but how to do so to automatically infer loop invariants.

QUICr represents set constraints using a constraint hypergraph where each vertex represents either the empty set, a singleton set, or a set variable, while each hyperedge represents a constraint between the linked vertexes. Each constraint is constructed of three parts: (1) a union of each edge target; (2) an intersection of each edge source; and (3) a numeric domain instance that labels the edge and represents a fact that holds on each element ν in the intersection. Each edge then represents the constraint that each element in the intersection must satisfy the label constraint and must also be in the union.

Figure 1 shows an instance of a QUIC graph. On the left is a graph representing the set constraints and on the right is a numerical domain constraint. This particular graph represents the following three basic facts:

$$A \subseteq \{\nu \in B \cup C \mid 3 < \nu\} \qquad B \subseteq \{\nu \in D \cup E \mid \nu < x\} \qquad x \leq 11$$

There are several other edges shown using lighter lines that represent facts derived from the basic facts. These derived facts come from a three-part lazy inference process that (1) *pushes* facts from one edge to another, strengthening numeric domain instances along the way, (2) *infers* new edges from the existing edges

Table 1. Heuristics refine the candidate generation strategy to both improve performance and increase precision over a naive candidate generation by providing hints [24] for join and widen operations

Name	Description	Effect
Empty Remove	Replace edges with empty set with equivalent edges without that empty set.	Improves performance by removing redundant edges.
Min Rewrite	Assign a total order to all vertexes in the graph and for each edge and add an edge using the identical edge using the minimum equivalent vertex for each vertex.	Increases use of empty sets and thus of Empty Remove; increases use of singleton sets and reductions associated with singletons; creates likely-common edges.
Patt Match	Pattern matching identifies certain subgraphs and eagerly adds derived facts based on those subgraphs.	Improves precision by adding additional candidates that correspond to likely edges in the join; often arise from specific code patterns (iteration over sets).

using a lazy form of transitive closure, and (3) *cycles* equalities found in the set constraints back to the numeric constraints as necessary. These three inference operations rely solely on basic domain operators provided by numeric domains, such as built-in meet, join, and widen operators. These inferred strengthenings are used to implement domain operations such as join, widen, and containment.

Because the graph representation can be exponential in number of variables, the QUICr implementation lazily derives facts, on demand. To implement this, QUICr uses a rewrite rules approach, which has two benefits: (1) it allows easy viewing of progress – each rule can print out status information to indicate where it is being applied and why; (2) it allows easy extension of the rules – adding new rules or reordering rule application is simply a matter adding new calls or reordering calls in the rule application function. This architecture can thus be easily extended to improve precision by identifying application-specific reductions or to increase performance by eliminating application of some rules.

To implement join and widen operations, QUICr employs a generator/checker strategy. A list of candidate edges is generated with unknown numeric domain constraints. The analyzer attempts to derive each of these edges in both inputs to join or widen. When the edge can be derived in both, it is added to the result using a computed numeric domain constraint. Otherwise, the candidate is ignored. Initially, the candidates are chosen to be all of the edges from both sides of the join. Unfortunately, this strategy suffers from a variety of problems. Edges that are derivable by lazy inference in both graphs, but are not directly in either graph will be lost. Additionally, there can be many edges that are redundant or nearly redundant, causing many extra checker invocations. To improve precision and performance, QUICr employs heuristics to refine the initial list of candidate edges. The implemented heuristics and their effects are detailed in Table 1.

3   QUICr Usage and Extension

QUICr is implemented as a functor in the OCaml language. A *functor* is a module constructor that takes a module as a parameter and produces another module. In this case, it takes a numeric abstract domain as a parameter and produces an abstract domain for sets of numbers. The current requirements on the numeric abstract domain are that (1) it must be possible to add and remove fresh variables from the domain – this is used to add or remove the bound variable; (2) it must provide sound top, bottom, meet, join, widen, and containment operators – these are used when pushing facts; and (3) it must provide an interface to retrieve and add equality constraints in the numeric domain. If these conditions are met, any numerical abstract domain can be substituted.

Unlike the implementation in [12], QUICr is more general and comes with four numerical domains provided: (1) a simple equivalence class abstract domain, where variables can be equal to each other and numeric constants; and domains provided by Apron: (2) boxes; (3) octagons; and (4) polyhedra. Additionally, it can be instantiated with any numeric domain that meets the Apron interface (such as PPL [16]). Adapting other existing domains only requires developing a functor that behaves as an interface adapter.

While built-in heuristics that are described in Table 1 are sufficient for many applications (see Section 4), they might not be sufficient for every application. If this is the case, additional heuristics can be added. The easiest form of heuristic to add is additional pattern matches. QUICr provides a graph matching system that can match arbitrary subgraphs against a template pattern. These matches can trigger rewrites and refine candidate generation. This is especially useful if a particular pattern is not being discovered by the built-in heuristics. For example, patterns are provided to identify nested unions and to then unnest the unions. This situation arises frequently when iteratively copying and manipulating sets.

Once instantiated, QUICr provides pretty printers for set constraints. It can output to both the console as well as to HTML and LaTeX. The provided example application uses the HTML output to produce programs annotated with the standard mathematical representation of sets.

4   Instantiation

Provided with the library is an example analyzer that allows selecting from the command line among the three of the four included numeric domains. It uses a simple input language reminiscent of JavaScript to analyze set and number manipulating programs. To evaluate the effectiveness of the QUICr, it is instantiated with the Apron-provided polyhedra numeric domain [15]. Then, by hand translation of Python programs, the resulting abstract domain is applied to all of the set-manipulating programs in the Python test suite, attempting to validate the assertions specified in the test suite. For example, the copy test iteratively copies one set to another, producing a set identical to the original. The results are shown in Table 2.

Table 2. Results on a set of small benchmarks. **#N**: # of numeric domain variables, **#S**: # of set variables, **#A**: # of assertions to be proved, **#P**: # of assertions automatically proved, **T(s)**: Time taken (seconds), **#I**: number of iterations of abstract interpreter before convergence. – represents a time out (600 seconds). Heuristics were selected based on the first four tests and validated on remaining tests.

	#N	#S	#A	#P	T(s)	#I
copy	1	6	2	2	0.2	2
filter	4	6	2	2	0.6	3
merge	2	14	2	1	0.6	4
partition	4	8	4	4	1.1	3
generic_max	3	8	6	3	0.9	6
b_filter	6	6	2	2	0.7	3
b_map	9	7	2	2	0.2	5
b_max_min	3	4	1	1	0.4	3

. . .

	#N	#S	#A	#P	T(s)	#I
b_reduce	7	4	1	0	0.4	3
iter_ind	20	12	1	1	84.4	39
mul_ret	9	2	2	2	0.2	6
nest_dep	5	7	1	0	2.2	12
resize1	15	5	5	4	1.7	18
simp_cond	11	5	4	3	4.6	12
simp_nest	9	10	2	0	–	1399
srange	6	2	2	2	0.1	6
Total			37	29	98.3	125

The shown benchmarks are those that include loops over the contents of a set or multiple sets and therefore must infer loop invariants. The inference of these loop invariants is based on the widening operation and thus not guaranteed to be precise. However, it is nearly always fast and is able to prove a significant number of real properties automatically.

5   Ongoing Development

Not only is QUICr useful today, but it is useful as a platform for development of future set and multi-set domain combinators. The graph structure can be extended by adding additional edge types. For example, we are evaluating an edge type that utilizes an underapproximating numeric domain. With such an edge type, it is possible to infer equalities with set comprehensions and to support set complement operations.

Additionally, the graph structure can be extended to add additional bound variables. Currently there is one bound variable per edge, but this may be an unnecessary restriction. With multiple bound variables, more complex relationships can be represented in the base domain. For example, with multiple bound variables we could represent the constraint that a set consists of elements that are the sum of elements from two other sets. We want to exploit this to be able to infer functions that map one set onto another.

Despite our use of QUICr to perform verification of container-manipulating programs, we believe that sets can be used as part of many other analyses. It is our desire for the QUICr library to be integrated into new, innovative domain combinators that effectively use set relations to represent unbounded numbers of connections between domains. We also hope to see additional extensions beyond our own to support more advanced set operations including cardinality queries, Cartesian products, and multi-set operations.

References

1. de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD. (2009) 45–52
2. Kuncak, V.: Modular Data Structure Verification. PhD thesis, EECS Department, Massachusetts Institute of Technology (2007)
3. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI. (2006) 427–442
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 238–252
5. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL. (2011) 105–118
6. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. In: ESOP. (2010) 246–266
7. Seghir, M.N., Podelski, A., Wies, T.: Abstraction refinement for quantified array assertions. In: SAS. (2009) 3–18
8. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: FASE. (2009) 470–485
9. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI. (2008) 339–348
10. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: CAV. (2007) 193–206
11. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: POPL. (2005) 338–350
12. Cox, A., Chang, B.Y.E., Sankaranarayanan, S.: QUIC graphs: Relational invariant generation for containers. In: ECOOP. (2013) 401–425
13. Pham, T.H., Trinh, M.T., Truong, A.H., Chin, W.N.: Fixbag: A fixpoint calculator for quantified bag constraints. In: CAV. (2011) 656–662
14. Dillig, I., Dillig, T., Aiken, A.: Precise reasoning for programs using containers. In: POPL. (2011) 187–200
15. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV. (2009) 661–667
16. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1-2) (2008) 3–21
17. Berdine, J., Cook, B., Ishtiaq, S.: Slayer: Memory safety for systems-level code. In: CAV. (2011) 178–183
18. Chang, B.Y.E., Rival, X.: Modular construction of shape-numeric analyzers. In: Festschrift for Dave Schmidt. (2013) 161–185
19. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL. (1999) 105–118
20. Bouajjani, A., Dragoi, C., Enea, C., Rezine, A., Sighireanu, M.: Invariant synthesis for programs manipulating lists with unbounded data. In: CAV. (2010) 72–88
21. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. (1979) 269–282
22. Toubhans, A., Chang, B.Y.E., Rival, X.: Reduced product combination of abstract domains for shapes. In: VMCAI. (2013) 375–395
23. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2) (1979) 245–257
24. Laviron, V., Logozzo, F.: Refining abstract interpretation-based static analyses with hints. In: APLAS. (2009) 343–358