μZ– An Efficient Engine for Fixed Points with Constraints

Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura

Manchester University and Microsoft Research

Abstract. The μZ tool is a scalable, efficient engine for fixed points with constraints. It supports high-level declarative fixed point constraints over a combination of built-in and plugin domains. The built-in domains include formulas presented to the SMT solver Z3 and domains known from abstract interpretation. We present the interface to μZ, a number of the domains, and a set of examples illustrating the use of μZ.

1 Introduction

Classical first-order predicate and propositional logic are a useful foundation for many program analysis and verification tools. Efficient SAT and SMT solvers and first-order theorem provers have enabled a broad range of applications based on this premise. However, fixed points are ubiquitous in software analysis. Model-checkers compute a set of reachable states as a least fixed point, or dually a set of states satisfying an inductive invariant as a greatest fixed point. Abstract interpreters compute fixed points over an infinite lattice using approximations. An additional layer is required when using first-order engines in these contexts.

The μZ tool is a scalable, efficient engine for fixed points with constraints. At the core is a bottom-up Datalog engine. Such engines have found several applications for static program analysis. A distinguishing feature of μZ is a pluggable and composable API for adding alternative finite table implementations and abstract relations by supplying implementations of relational algebra operations join, projection, union, selection and renaming. Lattice join and widening can be supplied to use μZ in an abstract interpretation context. The μZ tool is part of Z3 [3] and is available from Microsoft Research since version 2.18[1].

2 Architecture

A sample program is in Fig. 1 and the main components of μZ are depicted on Fig. 2. As input μZ receives a set of relations, rules (Horn clauses) and ground facts (unit clauses). The last rule uses the

$$
\begin{array}{lll}
\ell_0 & : & [Int] \text{ using pentagon} \\
\ell_1 & : & [Int] \text{ using pentagon} \\
\ell_0(0). & & \\
\ell_0(x) & \leftarrow & \ell_0(x_0), x = x_0 + 1, x_0 < n. \\
\ell_1(x) & \leftarrow & \ell_0(x), n \leq x.
\end{array}
$$

Fig. 1. Sample μZ input

[1] `http://research.microsoft.com/en-us/um/redmond/projects/z3/`

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 457–462, 2011.

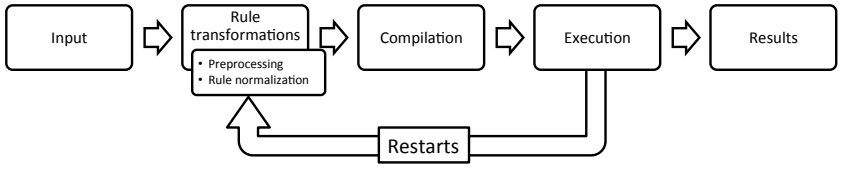

Fig. 2. μZ architecture

head predicate ℓ_1 and constraint $n \leq x$. The parameter n is symbolic, even during evaluation. A relation is specified by its domain and by its representation. The example from Fig. 1 uses the *pentagon* abstract domain. Representations may support approximation through widening that ensures convergence over infinite domains.

2.1 Rule Transformations

To allow for optimizations and/or additional features, we may perform various transformations on the input rules.

Free Variable Elimination. This transformation seeks to avoid large intermediary tables. It replaces rules of the form $p(\boldsymbol{x}, y) \leftarrow B[\boldsymbol{x}]$ by $p'(\boldsymbol{x}) \leftarrow B[\boldsymbol{x}]$. ($B[\boldsymbol{x}]$ stands for the body of the rule, and $\boldsymbol{x}$ denotes all the variables appearing in it.) Furthermore, each occurrence of p in a body of some rule requires a version with p'. E.g., for a rule $q(\boldsymbol{x}, y) \leftarrow p(\boldsymbol{x}, y), r(\boldsymbol{x}, \boldsymbol{z}), p(\boldsymbol{z}, y)$ we would need to introduce three rules $q(\boldsymbol{x}, y) \leftarrow p'(\boldsymbol{x}), r(\boldsymbol{x}, \boldsymbol{z}), p(\boldsymbol{z}, y)$, $q(\boldsymbol{x}, y) \leftarrow p(\boldsymbol{x}, y), r(\boldsymbol{x}, \boldsymbol{z}), p'(\boldsymbol{z})$, $q(\boldsymbol{x}, y) \leftarrow p'(\boldsymbol{x}), r(\boldsymbol{x}, \boldsymbol{z}), p'(\boldsymbol{z})$. The last rule introduces another free head variable, which can be eliminated using the same procedure. The transformation may increase the source program by an exponential factor, so μZ uses a limit on the number of such transformations.

Magic Sets. The classical magic sets transformation [1] is an option in μZ that specializes a set of rules with respect to a query.

Coalescing Rules. This transformation moves constants into a new relation and a group of rules with a single rule:

$$p(\boldsymbol{x}, \boldsymbol{c_1}) \leftarrow B[\boldsymbol{x}, \boldsymbol{c_1}] \;..\; p(\boldsymbol{x}, \boldsymbol{c_n}) \leftarrow B[\boldsymbol{x}, \boldsymbol{c_n}] \quad \mapsto \quad p(\boldsymbol{x}, \boldsymbol{y}) \leftarrow B[\boldsymbol{x}, \boldsymbol{y}], r(\boldsymbol{y})$$

where r is a fresh relation containing tuples $\boldsymbol{c_1}, \ldots, \boldsymbol{c_n}$. The transformation trades n updates to p using unions by one update to p and a join between B and the new relation r. In the context of μZ the transformation is particularly useful after a Magic set transformation. We have not found it useful directly for the user input.

Join Planning. The join planner splits long rules so that each rule contains at most two positive relation predicates in its body. Number of negative relation predicates and non-relation predicates in rules is not limited, because these do

not lead to introduction of intermediate relations. The planner uses information on the expected size of relations in order to make the intermediate relations as small as possible. To this end the solver periodically restarts and reruns the planner to make use of better size estimates. The join planner also attempts to identify shared parts of rules in order to avoid their repeated evaluation.

2.2 Compilation to an Abstract Machine

The compiler transforms the bodies of rules into relational algebra operations. These operations are atomic instructions in an abstract machine, which also contains control and data-flow instructions to handle applications of the rules until a fixed point is reached.

We use a binary relation $r(x, y)$ to illustrate the compilation of rule bodies into relation algebra. Renaming is used to reorder the arguments to correspond to $r(x, y)$. Selection restricts r by fixing a column to a constant, equating columns, or constraining r with respect to an arbitrary predicate φ. Multiple relations are combined using joins, and projection removes variables that are not used in the head.

$r(y, x)$	$\rho_{\#1\mapsto\#2,\#2\mapsto\#1}(r)$
$r(a, y)$	$\sigma_{\#1=a}(r)$
$r(x, x)$	$\sigma_{\#1=\#2}(r)$
$r(x, y), \varphi[x, y]$	$\sigma_{\varphi[\#1,\#2]}(r)$
$r(x, y), q(z, x)$	$r \bowtie_{\#1=\#2} q$
$T[x, \boldsymbol{y}]$	$\pi_{\#1}(T[x, \boldsymbol{y}])$

The abstract machine furthermore contains instructions for conditional jumps, swap, copy, load, complementation (for stratified Datalog programs), and creating empty relations.

The effect of applying a rule is to update the relation in the head by taking a union with the relation computed in the body. The corresponding union operation comes in two flavors: if the head relation is used in a non-recursive context, the corresponding operation destructively updates the head relation. If the head relation is used in a recursive context, the union operation furthermore computes a *differenec* relation Δ that is used to detect termination and to minimize the number of records that need to be examined in subsequent joins. This contrasts using a containment relation to check for termination.

	content
$R_0 := \{0\}$	ℓ_0
$R_2 := R_0$	Δ_{ℓ_0}
while$(R_2 \neq \bot)$ {	
$R_3 := \top$	
$R_4 := R_3 \bowtie R_2$	x, x_0
$R_4 := \sigma_{\#1=\#2+1}(R_4)$	
$R_4 := \sigma_{\#2<n}(R_4)$	
$R_5 := \pi_{\#2}(R_4)$	x
$R_2, R_0 := R_5 \setminus R_0, R_0 \cup R_5$	
}	
$R_1 := \sigma_{\neg(\#1<n)}(R_0)$	ℓ_1
$\ell_0 := R_0$	
$\ell_1 := R_1$	

Fig. 3. Compiled version of Fig. 1

The requirement on Δ is the following, where p is the head relation and q is the relation from the body: $q \subseteq p \rightarrow \Delta = \bot$, and $q \not\subseteq p \rightarrow q \setminus p \subseteq \Delta \subseteq p \cup q$. Loops are terminated when Δ is empty. The natural value for Δ would be $q \setminus p$, but for some representations this might be expensive to evaluate.

We use the relation dependency graph to obtain loops of smaller size and evaluation order which leads to faster propagation of newly derived facts. We identify strongly connected components of recursive head predicates and saturate each of these components separately. We attempt to find an acyclic induced subgraph of each component, and use this subgraph to obtain the evaluation order inside the loop. The benefit of such order is that we do not have to carry the "differences" of relations which are in the acyclic subgraph across loop iterations.

The engine supports two compilation modes: In standard Datalog compilation mode, rules are compiled into instructions that perform a bottom-up saturation. In the presence of stratified negation, it performs the saturation per stratum. The compiler generates two phases in abstract interpretation mode: union on recursive predicates is replaced by widening, and the recursive predicates are reset (to the empty relations) between phases. Other compilation modes are possible in future versions of μZ. In particular a mode for bounded-model checking where fixed points are unrolled to a fixed depth, is of interest in applications.

2.3 Execution

Execution of the compiled code is performed by a register machine interpreter. Registers store relation objects that implement relational algebra methods. Compile time transformations suffice to avoid using counters or other data types in registers.

2.4 Tables and Relations

The abstract machine works at the level of relational algebra while the representation of relations is delegated to implementations.

Finite Collections. A basic representation of relations is as a finite collection of records. Finite sets admit *iterators* that can enumerate elements from the collection. Our default representation of finite collections is by hash tables with *on-demand indexing*. Thus, one hash table may be indexed by multiple columns at a time depending on which columns are used in different joins. An index is created or updated when the correponding column is used in a join or a selection by a constant. We found that hash tables offered a more efficient representation for our benchmarks when compared with BDDs, though it is possible to create examples where BDDs are significantly more compact [5]. We have plugged in BDDs over the external relation API using the BuDDy package[2].

Abstract Relations. The real utility of the relational algebra core is achieved by also admitting relation representations that are truly abstract.

A precise, but abstract representation is achieved by mapping relational algebra operations back to first-order formulas. We call this the *SMT relation* as it uses the SMT solver Z3 for quantifier-elimination during projection and checking for convergence. Translation from relational algebra into first-order logic is a simple transliteration. For example $\lceil \bot \rceil = \mathit{false}$, $\lceil \pi_x R \rceil = \exists x. \lceil R \rceil$

[2] `http://buddy.wiki.sourceforge.net`

and $\lceil R \bowtie_{\mathcal{E}} S \rceil = \lceil R \rceil \wedge \lceil S \rceil \wedge \mathcal{E}$. The set of satisfying assignments to the free variables in the formula correspond to records that are members of the relation. Computing Δ requires a satisfiability check of $\lceil R \rceil \wedge \neg \lceil S \rceil$, which introduces a formula with quantifier alternation. μZ relies on Z3's support for quantifier elimination for bit-vectors, Presburger arithmetic and algebraic data types to compute Δ.

μZ also contains two built-in abstract relations for conjunctions of integer intervals and bounds (relations of the form $x < y$). These domains are well-known from abstract interpretation [2]. They are closed under join, projection and selection, but they are not closed under union. Union is instead approximated by a convex hull operation. The domains also support widening operations.

Compositionality. Explanations can be tracked by adding a column to each relation and track rules by accumulating a term for the rules that are applied:

$$rl : p(\boldsymbol{x}) \leftarrow q(\boldsymbol{x}, y), r(\boldsymbol{x}, y). \quad \mapsto \quad rl' : p'(\boldsymbol{x}, rl(u, v, y)) \leftarrow q'(\boldsymbol{x}, y, u), r'(\boldsymbol{x}, y, v).$$

There can be an unbounded number of explanations for a derived fact, but it suffices to consider just one representative. We can encode this in a special *relational algebra of explanations*, where unions of two sets of explanations selects a suitable (in the case of μZ, oldest) representative. The remaining columns of p' do not belong to the algebra of explanations, but may be stored in a finite table or an abstract relation. To support such joint representations, μZ allows composing arbitrary tables and relations. The composition of a finite table with another finite table or relation is obtained by adding an additional column to the finite table to point to a table (relation) that contains values corresponding to a row. The usual relational algebra operations are extended directly for this representation. For example, the joint relation $r(x, y, z) : \{(1, 0, a), (1, 0, b), (1, 1, c)\}$ is represented as the map $[(1, 0) \mapsto \{a, b\}, (1, 1) \mapsto \{c\}]$, and projecting the second column produces $\pi_y r(x, y, z) : [1 \mapsto \{a, b, c\}]$. The product and intersection of two abstract relations is also available, but in this case projection and union are no longer precise because the normalized representation is as a vector of relations. Some precision is retained by supporting *reduced products* that lets domains communicate constraints. For example, we obtain the *Pentagon* domain by taking the product of the Interval i and Bound b domain subject to having restriction $(b \cup \{x < y\}) \wedge (z = x - y)$ contribute the interval constraint $(z \in [-\infty, -1])$; and extending unions on bounds to also accept intervals: $b \cup i := \{x < y \in b \mid sup_i(x) < inf_i(y)\}$.

3 Usage

A diagram showing the integration of options for μZ is in Fig. 4.

Interface. User can interact with μZ either by the means of the Z3 API (managed or C) or pass the problem specification in a file from the command line.

Input files can be in one of the following formats: SMT2 format extended by commands *rule* and *query* to add rules and start the fix-point search, in the *Bddbddb* [5] format, or in the *tuple* format which allows fast reading of large amounts of facts.

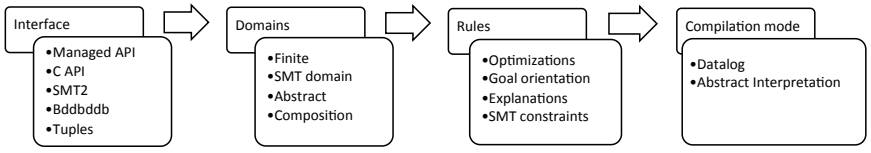

Fig. 4. Possible configurations of μZ

Applications. Although μZ is a new tool, it has been already used in several contexts[3]. We have run μZ on moderate size (2- 25K lines of pure Datalog code) benchmarks extracted from the Javascript security analyzer Gatekeeper [4]. It suffices to configure μZ using hash tables for storing relations. It spends in the order of 100ms on these benchmarks due to transformations such as the *free variable elimination* that eliminated many rules in favor of ground facts. The Bddbddb tool [5], in contrast relies on the existence of good variable orderings to avoid running out of physical memory. Using finite domains, we have also loaded a representation of the Windows base kernel, and ran various queries on it. The resulting data-base contained in the order of 10^6 facts. The efficient tuples front-end loads the 2 GB data-base within 20 seconds, but stand-alone saturation is infeasible. Queries can still be answered within a second on a standard dekstop PC after the Magic sets transformation. A demonstration of μZ for solving Traffic Jam puzzles is available. It illustrates the use of explanations.

4 Conclusion

μZ is a new efficient engine for fixed points with logical constraints. It integrates and is available with Z3. This tool paper explained the main architecture of μZ and provided background on pluggable and composable relations. We hope this tool will enable several future applications that rely on efficient fixed point core with special needs on domain representations.

References

1. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs. In: PODS, pp. 1–15. ACM, New York (1986)
2. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
3. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Guarnieri, S., Livshits, V.B.: Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In: USENIX, pp. 151–168 (2009)
5. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI, pp. 131–144 (2004)

[3] `http://research.microsoft.com/projects/z3/fixedpoints-index.html`