FixBag* : A Fixpoint Calculator for Quantified Bag Constraints

Tuan-Hung Pham[1], Minh-Thai Trinh[2], Anh-Hoang Truong[2], Wei-Ngan Chin[3]

[1] University of Minnesota, Twin Cities, hung@cs.umn.edu
[2] Vietnam National University, Hanoi, {thaitm_52, hoangta}@vnu.edu.vn
[3] National University of Singapore, chinwn@comp.nus.edu.sg

Abstract. Abstract interpretation techniques have played a major role in advancing the state-of-the-art in program analysis. Traditionally, stand-alone tools for these techniques have been developed for the numerical domains which may be sufficient for lower levels of program correctness. To analyze a wider range of programs, we have developed a tool to compute symbolic fixpoints for *quantified bag domain*. This domain is useful for programs that deal with collections of values. Our tool is able to derive both *loop invariants* and *method pre/post conditions* via fixpoint analysis of recursive bag constraints. To support better precision, we have allowed disjunctive formulae to be inferred, where appropriate. As a stand-alone tool, we have tested it on a range of small but challenging examples with acceptable precision and performance.

1 Introduction

Abstract interpretation [2] is a technique to infer program's properties. It requires the least fixed point of a monotone function over an abstract domain of the program's semantics to be computed. Let $\langle L, \prec \rangle$ be a complete lattice and $\bot$ be its least element. A function $f : L \to L$ is monotone if $f(u) \prec f(v)$ when $u \prec v$ with all u, v in L. One classical method to find the fixed point is Kleene iteration, which computes the ascending chain $f_0 = \bot$, $f_{i+1} = f(f_i)$ with $i > 0$ until we find i^* satisfying $f_{i^*+1} = f_{i^*}$. Widening operator [3] is used to guarantee that the ascending chain is finite.

Traditionally, stand-alone abstract interpretation (AI) tools have been developed for the numerical domains (such as Octagon [8] and Polyhedra [4]). Little attention has been paid to building such tools for richer pure domains, such as bags, maps and sequences. Stand-alone AI tools focus primarily on the logics of the abstract domains and use sound mechanisms for approximating recursion via fixed point computation. They have been widely adopted by program analysis systems that are customized to analyze properties from programs (e.g. [1, 9]).

Some recent works [10, 11] have proposed methods to automatically infer disjunctive numerical invariants for added precision. However, numerical invariants are often insufficient for higher levels of program correctness. For example, many programs are constructed to compute a collection of values whose correctness cannot be captured using only numerical properties. Instead, we require a *quantified bag domain* to provide more precise program analyzers for such programs.

* The tool is available at http://loris-7.ddns.comp.nus.edu.sg/~project/fixbag/.

To the best of our knowledge, there is no current published tool that can discover quantified bag invariants. We present FixBag, a stand-alone fixpoint calculator for quantified bag constraints. The tool has the following characteristics:

- FixBag can infer disjunctive fixed points of formulae with bag constraints. The maximum number of disjuncts is provided by end-users. Supported bag's operators are union ($\cup$), intersection ($\cap$), and subset ($S_1 \subseteq S_2$), where S_1 and S_2 denote bags.
- FixBag can find fixed points with quantified constraints. Specifically, the system supports the universal quantifier of the form $\forall x \in S : P(x)$ and the existential quantifier of the form $\exists x \in S : P(x)$, where x, S, and $P(x)$ are a variable, a bag, and a predicate concerning x, respectively.
- FixBag partially supports arithmetic constraints on size properties over bags. Currently, FixBag allows the following types of size properties to be inferred: $|S_1| = m \times |S_2|$, $m \leq |S|$, or $|S| \leq m$, where m denotes an integer.

Section 2 gives an overview via examples. Section 3 introduces the algorithm to infer fixed points, as used in our tool. Section 4 summarizes our experimental results. Section 5 concludes with a short discussion on related works.

2 Motivating Examples

Our tool is able to compute disjunctive fixpoints for constraint abstractions over the bag domain. To illustrate its capability, we shall analyze two list functions that are commonly used in functional languages by initially showing their respective constraint abstractions prior to fixpoint analysis. We stress that our tool is language-independent, as its inputs are logical formulae (with bag and size constraints) that may be applied to similar abstractions for programs from other programming languages too.

Our first example is a well-known `filter` function, which selects elements from a list that satisfy a predicate, `p`, as given below in Caml syntax:

```
filter p xs = match xs with
    | [] → []
    | x : xs → if (p x) then x : (filter p xs) else (filter p xs)
```

The corresponding constraint abstraction, named *filterB*, for this function has three parameters: p to denote the predicate `p` of `filter`, S to capture the elements of input list `xs`, and R to capture the elements of the method's output.

$$\begin{array}{rcl} \mathit{filterB}(p, S, R) & \equiv & S{=}\{\} \wedge R{=}\{\} \ \vee \ \exists x, S_1, R_1 \cdot S{=}\{x\}{\cup}S_1 \wedge \\ & & \quad (p(x) \wedge R{=}\{x\}{\cup}R_1 \wedge \mathit{filterB}(p, S_1, R_1)) \ \vee \\ & & \quad \neg p(x) \wedge R{=}R_1 \wedge \mathit{filterB}(p, S_1, R_1)) \end{array}$$

When this constraint abstraction is passed to our tool, we could infer the following fairly precise closed-form formula using universally quantified bags:

$$\mathit{filterB}(p, S, R) \ \equiv \ (\forall x \in R : p(x)) \ \wedge \ (\forall x \in S - R : \neg p(x)) \ \wedge \ R \subseteq S$$

Our next example is a membership function that determines if an element exists within an input list or not. Its Caml code is given below:

```
mem v xs = match xs with | [] → false
                         | x : xs → if x = v then true else (mem v xs)
```

The corresponding constraint abstraction has three parameters, as shown:

$$memB(v,S,r) \equiv S{=}\{\}\wedge\neg r \ \vee \ \exists x, S_1 \cdot S{=}\{x\}\cup S_1 \wedge (x{=}v\wedge r \ \vee \ x{\neq}v\wedge memB(v,S_1,r))$$

A precise closed-form formula for this function requires both disjunction and quantified bag formulae, as shown below, which our tool can derive:

$$memB(v,S,r) \ \equiv \ \ (\forall x \in S : x{\neq}v)\wedge\neg r \ \vee \ (\exists x \in S : x{=}v)\wedge r$$

These two examples show that a good treatment of quantified formulae and disjunctions is needed to support more precise analysis.

3 Algorithm

This section presents the algorithm behind FixBag. One of the necessary operators used in fixpoint analysis is hulling, which is well-developed for numerical domains. However, to the best of our knowledge, there is no work that calculates the hulling operations on bag/set domain to date. To realize this, we propose a rule-based approach that uses propagation and simplification rules to attain the hulling of formulae for the bag domain. Similar to CHR [5], our propagation rules $\mathcal{R}_p$ add new implied constraints to a formula while simplification rules $\mathcal{R}_s$ reduce the size of a formula by removing redundant constraints from it. Although these rules [4], by themselves, simply preserve the logical equivalences of the original formula, they can help create intermediate results that would play important roles in other operations. Let $\phi_1 \Cap \phi_2$ be $\bigwedge_{i=1}^{k} d_i$ where $d_1, ..., d_k$ are all shared conjuncts of two conjunctive formulae ϕ_1 and ϕ_2. Definition 1 shows how to find the hulling result of ϕ_1 and ϕ_2.

Definition 1 (Hulling). *Given two conjunctive formulae ϕ_1 and ϕ_2, we divide each of them into two parts: the first one (Γ) contains all conjuncts of the form $m_1 \leq |S| \leq m_2$ and the other one (Δ) has the remaining conjuncts. Thus, the two original formulae are represented as $\phi_1 = \Gamma_1 \wedge \Delta_1$ and $\phi_2 = \Gamma_2 \wedge \Delta_2$. The hulling operation is defined as $\phi_1 \boxtimes \phi_2 = \Gamma_1 \boxtimes \Gamma_2 \wedge \Delta_1 \boxtimes \Delta_2$ where*

- $\Gamma_1 \boxtimes \Gamma_2 =$
  $\underbrace{(\bigwedge m_{i1} \leq |S_i| \leq m_{i2})}_{(1)} \wedge \underbrace{(\bigwedge m_{j1} \leq |S_j| \leq m_{j2})}_{(2)} \wedge \underbrace{(\bigwedge min(m_{i'1}, m_{j'1}) \leq |S_{i'j'}| \leq max(m_{i'2}, m_{j'2}))}_{(3)}$
  where (1) contains all $(m_{i1} \leq |S_i| \leq m_{i2}) \in \Gamma_1$ that S_i is not in in Γ_2, (2) consists of all $(m_{j1} \leq |S_j| \leq m_{j2}) \in \Gamma_2$ that S_j is not in Γ_1, and (3) has all $S_{i'j'}$ that has $(m_{i'1} \leq |S_{i'j'}| \leq m_{i'2}) \in \Gamma_1$ and $(m_{j'1} \leq |S_{i'j'}| \leq m_{j'2}) \in \Gamma_2$.
- $\Delta_1 \boxtimes \Delta_2 = \underset{\mathcal{R}_s}{\texttt{simplify}}(\Delta_1 \widehat{\boxtimes} \Delta_2)$ *where* $\Delta_1 \widehat{\boxtimes} \Delta_2 = \underset{\mathcal{R}_p}{\texttt{propagate}}\Delta_1 \Cap \underset{\mathcal{R}_p}{\texttt{propagate}}\Delta_2$.

[4] $\mathcal{R}_p$ and $\mathcal{R}_s$ are available at the tool's website.

The latter is the harder operation. Intuitively, we find two closures (corresponding to Δ_1 and Δ_2) of conjunctive constraints that are closed under the set of propagation rules $\mathcal{R}_p$, compute the intersection of the closures, and simplify the result by the collection of simplification rules $\mathcal{R}_s$. We also define a version of the hulling operation without simplification as $\phi_1 \widehat{\boxtimes} \phi_2 = \Gamma_1 \boxtimes \Gamma_2 \wedge \Delta_1 \widehat{\boxtimes} \Delta_2$. It is needed when we want to check whether a particular conjunct contributes to the hulling result or not. This notion is used to measure the closeness of two conjunctive formulae in Definition 2. Given two conjunctive formula ϕ_1 and ϕ_2, $\phi_1 \oslash \phi_2$ quantifies their closeness as a rational number [5] in the range of 0..1. The larger the number is, the closer they are to each other. We denote $\lceil \phi \rceil$ ($\lfloor \phi \rfloor$) the number of conjuncts (disjuncts) in a conjunctive (disjunctive) formula ϕ.

Definition 2 (Affinity Measure). *Given two conjunctive formulae ϕ_1 and ϕ_2, the affinity measure $\oslash$ is defined as follows:* $\phi_1 \oslash \phi_2 = \frac{\lceil (\phi_1 \wedge \phi_2) \Cap (\phi_1 \widehat{\boxtimes} \phi_2) \rceil}{\lceil \phi_1 \wedge \phi_2 \rceil}$

While our hulling only works with conjunctive formulae, selective hulling [10] can deal with disjunctive ones. Our tool can increase the precision of the output fixpoints by allowing up to μ disjuncts to be present during the analysis process. The main idea is to repeatedly call hulling with a closest pair of disjuncts taken from the latest formula until there are at most μ disjuncts remaining. The function `normalize` converts a given formula into the disjunctive normal form.

Definition 3 (Selective Hulling). *Given a disjunctive formula $\phi = \bigvee_{i=1}^{k} d_i$ and a maximum number of disjuncts μ, the selective hulling operation $\oplus_\mu$ is defined as* $\oplus_\mu \phi = \texttt{normalize}(\widehat{\oplus}_\mu \phi)$ *where $\widehat{\oplus}_\mu \phi$ is recursively defined as follows:*

$$\widehat{\oplus}_\mu \phi = \begin{cases} \phi \text{ if } k \leq \mu \\ \widehat{\oplus}_\mu \big((d_{i'} \boxtimes d_{i''}) \vee \bigvee_{i \in \{1..k\} \setminus \{i', i''\}} d_i \big) \text{ if } k > \mu \\ \qquad \text{where } (i', i'') = \texttt{argmax}(d_{j'} \oslash d_{j''}) \text{ with } 1 \leq j', j'' \leq k \text{ and } j' \neq j'' \end{cases}$$

Widening [3] is used to ensure that the fixpoint analysis terminates. To maintain disjunctions in the widening process, selective widening [10] is required.

Definition 4 (Selective Widening). *Given two disjunctive formulae $\phi_1 = \bigvee_{i=1}^{k} d_i$ and $\phi_2 = \bigvee_{j=1}^{k} e_j$, the selective widening operation $\phi_1 \triangledown \phi_2$ is defined as* $\phi_1 \triangledown \phi_2 = \texttt{normalize}(\phi_1 \widehat{\triangledown} \phi_2)$ *where $\phi_1 \widehat{\triangledown} \phi_2$ is recursively defined as follows:*

$$\phi_1 \widehat{\triangledown} \phi_2 = \begin{cases} \phi_1 \boxtimes \phi_2 \text{ if } k = 1 \\ (d_{i'} \boxtimes e_{j'}) \vee \big(\bigvee_{i \in \{1..k\} \setminus \{i'\}} d_i \,\widehat{\triangledown}\, \bigvee_{j \in \{1..k\} \setminus \{j'\}} e_j \big) \text{ if } k > 1 \\ \qquad \text{where } (i', j') = \texttt{argmax}(d_{i''} \oslash e_{j''}) \text{ with } 1 \leq i'', j'' \leq k \end{cases}$$

The algorithm to find the fixpoint for formulae with bag constraints shares the same ideas with the one used in [10] to infer disjunctive postconditions. The main challenges here are how we support affinity measure, selective widening, and selective hulling to work with the bag domain. Given a recursive function f, we start with $f_0 = \bot$ (which is `false` in the bag domain) and then compute an

[5] For simplicity, we may also use an integer-based affinity measure [10] defined as $\phi_1 \oslash \phi_2 = \texttt{round}(\frac{\lceil (\phi_1 \wedge \phi_2) \Cap (\phi_1 \widehat{\boxtimes} \phi_2) \rceil}{\lceil \phi_1 \wedge \phi_2 \rceil} \times 98) + 1$.

ascending chain $f_1, f_2, ...$, until we find two equal consecutive elements $f_{i^*+1} = f_{i^*}$. If the current value in the chain is f_i, the next item f_{i+1} will be calculated as $bottomup(f, f_i, \mu)$, which is either $f(f_i)$ or its sound approximation with the help of selective hulling and widening operations. The first kind of operations helps us achieve a disjunctive fixpoint while the second one ensures the chain will converge. Algorithm 1 shows how we can obtain f_{i+1} from f, f_i, and μ.

Algorithm 1: Calculating $f_{i+1} = bottomup(f, f_i, \mu)$

```
1 f_fi ← normalize(f(f_i))
2 if ⌊f_fi⌋ < μ then
3 |  return f_fi
4 else if f_i = ⊥ or ⌊⊕_μ f_fi⌋ < μ or ⌊f_i⌋ < μ then
5 |  return ⊕_μ f_fi
6 else
7 |  return f_i ▽ (⊕_μ f_fi)
```

First, we normalize the result of $f(f_i)$ to achieve f_{f_i}, which is a candidate for f_{i+1}. If $\lfloor f_{f_i} \rfloor < \mu$, we return f_{f_i}, otherwise we need to find a suitable approximation of f_{f_i} that has no more than μ disjuncts. If $f_i = \bot$ or $\lfloor \oplus_\mu f_{f_i} \rfloor < \mu$, the approximation is $\oplus_\mu f_{f_i}$. If the two previous conditions fail, we have $f_i \neq \bot$ and $\lfloor \oplus_\mu f_{f_i} \rfloor = \mu$. At this point, the approximation will depend on $\lfloor f_i \rfloor$ because selective widening only works with two formulae that have the same number of disjuncts. Therefore, if $\lfloor f_i \rfloor < \mu$, we still return $\oplus_\mu f_{f_i}$. Finally, when $\lfloor \oplus_\mu f_{f_i} \rfloor = \lfloor f_i \rfloor = \mu$ holds, the best approximation is $f_i \triangledown (\oplus_\mu f_{f_i})$, which not only maintains up to μ disjuncts but also contributes to the convergence of the chain.

Soundness. The soundness of our tool is critically dependent on the soundness of the closure process used in the hulling operation. This process is guaranteed to be sound, as long as each simplification and propagation rule (from $\mathcal{R}_s$ and $\mathcal{R}_p$, respectively) preserves logical implication. This property helps to ensure that selective widening and selective hulling will always generate a sound over-approximation of f_{f_i}.

4 Experimental results

We tested our tool on a set of methods from the OCaml's List library. To support higher-order functions, FixBag handles uninterpreted functions and multi-argument predicates. It takes the abstraction of each function and a number μ, denoting the maximal number of disjuncts allowed during the fixpoint inference, as its arguments and returns two closed-form (fixpoint) formulae, one to denote the function's post-condition and the other to derive its pre-condition. We also measured the running-time of the analysis process. These results and the inference rules are detailed at our tool's website.

We have encountered several examples where disjunctive analysis can obtain more precise fixpoints than conjunctive analysis. Conjunctive analysis can be simulated using $\mu = 1$. In general, each analyzed function has an upper bound of μ; increasing μ over this bound does not help achieve more precise fixpoints and does not affect the analysis time.

5 Related Works and Conclusion

Libraries to support abstract interpretation are popular for program analysis systems, but they are focused mostly on the numeric domains [8, 1]. In the non-numeric domains, abstract interpretation tools have been developed for shape analysis [7, 12] and for constraint-based analysis [6]. The former is for discovering data shapes of heap-manipulating programs rather than their pure properties; and are thus focused on program codes rather than logical formulae. The latter is meant as a scalable tool for flow-based constraints, rather than for analyzing collections. Both systems do not automatically handle quantified formulae and have restricted use of disjunctive formulae.

We have built a stand-alone abstract interpretation tool for quantified bag domain. Our use of simplification and propagation techniques is inspired from CHR [5], while the use of affinity-based hulling and widening is targeted at more precise disjunctive fixpoints. Our experiments have shown that our tool is capable of efficiently analyzing the collection properties for non-trivial functions.

Acknowledgement. We thank the reviewers for insightful feedback, and gratefully acknowledge the support of MoE research grant R-252-000-411-112.

References

1. Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *SCP*, 72:3–21, June 2008.
2. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, pages 238–252, 1977.
3. Patrick Cousot and Radhia Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *PLILP*, pages 269–295, 1992.
4. Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL'78*, pages 84–96, 1978.
5. Thom W. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
6. John Kodumal and Alexander Aiken. Banshee: A Scalable Constraint-Based Analysis Toolkit. In *SAS'05*, pages 218–234, 2005.
7. Tal Lev-Ami and Shmuel Sagiv. TVLA: A System for Implementing Static Analyses. In *SAS'00*, pages 280–301, 2000.
8. Antoine Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19:31–100, March 2006.
9. Bertrand net and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV'09*, pages 661–667, 2009.
10. Corneliu Popeea and Wei-Ngan Chin. Inferring Disjunctive Postconditions. In *ASIAN'06*, pages 331–345, 2007.
11. Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static Analysis in Disjunctive Numerical Domains. In *SAS'06*, pages 3–17, 2006.
12. Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, pages 335–348, 2009.