Logical Qualifier Inference for $\lambda_{\texttt{imp}}$

Madhukar N. Kedlaya
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, USA
mkedlaya@cs.ucsb.edu

Vineeth Kashyap
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, USA
vineeth@cs.ucsb.edu

***Abstract*—Inferring program invariants that hold at particular program points is useful proposition, and has applications in automated assertion proving, program optimizations etc. In this project, we are interested in inferring a particular kind of program invariant, which we call *logical qualifiers*, a term inspired from Liquid Types [?]. Restricting ourselves to inferring logical qualifiers allows us to discover interesting program invariants, yet have a terminating algorithm, using the theory of abstract interpretation and monotone framework.**

In this work, we describe the underpinnings of logical qualifier inference for a simple imperative language. We implemented a flow-sensitive logical qualifier inference engine for this language, and we show how we leverage the automated theorem prover Z3 [?] in order to implement our tool. We also show how we can handle certain theories (in particular, multiplication) not handled by Z3, when we restrict ourselves to a particular set of logical qualifiers. Using a crafted example, we show how and what program invariants are discovered by our tool.

***Keywords*-abstract interpretation; predicate abstraction; verification;**

I. Introduction

Program invariants are logical predicates on program variables and constants, that hold on all possible runs of the program, irrespective of what inputs are provided to the program. Finding program invariants can be very useful for a variety of purposes. For example, a number of developers use program assertions to make sure certain properties are not violated at certain program points. This allows them to reason about code in a modular fashion, that is, one can assume that an assertion holds at a particular point in the program, and write the rest of the program with the assumption in mind.

Finding program invariants could also have potential applications in program optimizations. Say that we are able to prove that an integer variable is definitely greater than 0. Then it can be allocated in an `unsigned int` by the compiler. Or if we are able to prove that an integer variable is definitely between 0 and 255, then it can be allocated in an `unsigned char` by the compiler. Certain program invariants could also help eliminate dead code in a number of places. For example, proving the invariant `p! = 0` at a branch conditional which checks for `p! = 0` can help ascertain that the `else` branch will definitely not be taken. This would allow the compiler to generate straight line code instead of a branch.

```
var sum, p, final in
  sum := 0;
  p := 10;
  while (p > 0) {
    sum := sum + (p * p);
    p := p - 1;
  }
  final := sum - p;
  assert(p = 0)
```

Figure 1. A program in $\lambda_{\texttt{imp}}$ that calculates the sum of first 10 squares

We are interested in statically inferring program invariants. With such an exercise, intractability lurks at every step, and therefore, such an analysis should be designed very carefully. In general, proving all kinds of assertions automatically is not possible without programmer help (which is what interactive theorem provers like Coq [?] are for). Thus, we limit ourselves to inferring program invariants of a certain kind, called logical qualifiers, which is a term we are using from Liquid Types [?].

Let us dive into an example that can show what our tool can do. Consider the following program in the $\lambda_{\texttt{imp}}$ shown in Figure ??. By running our tool, we can infer that the following invariants hold at the end of the program: `sum > 0`, `sum > p`, `final >= sum`, `final > 0`, `final > p`. We can also prove the programmer assertion `p = 0`. If we manually reason about the code, we can see for ourselves that each of these invariants do in fact hold, and the assertion will pass.

In order to test our tool, we came up with a test suite with various programs in $\lambda_{\texttt{imp}}$, that we could infer interesting invariants from (some of these were taken from DART [?] and CUTE [?]). While doing this, we wrote code shown in Figure ??, for calculating factorial of a number. Our tool however, invalidated the program assertion!

Looking carefully, we found a bug in this code (which any reasonable unit test suite would have found too, but we can find it without running any test cases). The correct version of the program calculating factorial of a number is given

```
var n, fact in
  if (n > 0) {
    fact := n;
    while (n > 0) {
      fact := fact * (n - 1);
      n := n - 1;
    }
  } else {
    fact := 1
  };
  assert(fact > 0)
```

Figure 2. A program in $\lambda_{\texttt{imp}}$ that calculates factorial of a number, but has a bug

```
var n, fact in
  if (n > 0) {
    fact := n;
    while (n > 1) {
      fact := fact * (n - 1);
      n := n - 1;
    }
  } else {
    fact := 1
  };
  assert(fact > 0)
```

Figure 3. A program in $\lambda_{\texttt{imp}}$ that calculates factorial of a number, with the bug in Figure **??** fixed

in Figure **??**. The fix was to change the condition in the while loop from `n > 0` to `n > 1` (the bug caused `fact` to be always 0). After making the fix, our tool was able to prove the assertion. Thus, we can say that for any integer input value for `n`, `fact > 0`.

In the rest of the paper, we describe how our tool works.

II. Brief Overview of Our Tool

Logical Qualifiers: A logical qualifier is used to describe a value using some predicates. We write a logical qualifier in the form $\{v \mid \psi(v)\}$, and it is read as the set of values v such that the predicate $\psi(v)$ holds. For the rest of the paper, we assume that the domain of discourse for values is the set of all integers. The Table **??** shows some examples of logical qualifiers and the values they describe. Note that we only allow the conjunction of simple logical formulae, and no disjunctions. That is, the simple logical formulae should be of the form $v \odot e$ (or *true* or *false*), where v is the integer value on which the predicate is being defined, $\odot$ is a relational operator, and e is any expression in the language $\lambda_{\texttt{imp}}$. A logical qualifier is the conjunction of such simple logical formulae. Thus, for example, $\{v \mid v = 0 \ \vee \ v = 1\}$ is not a valid logical qualifier.

Logical Qualifier	What it means
$\{v \mid true\}$	Set of all integers
$\{v \mid false\}$	Empty set
$\{v \mid v > 0\}$	Set of all positive integers
$\{v \mid v > 0 \ \wedge \ v < 10\}$	Set of all integers between 0 and 10
$\{v \mid v > x\}$	Set of all integers greater than the current value of x

Table I
Examples of a few logical qualifiers and what they describe

$$
\begin{aligned}
a \in AExp &::= n \mid x \mid a_1 \oplus a_2 \\
b \in BExp &::= \textbf{true} \mid \textbf{false} \mid b_1 \otimes b_2 \mid a_1 \ominus a_2 \mid !b \\
\oplus \in ABinOp &::= + \mid - \mid \times \\
\otimes \in BBinOp &::= \wedge \mid \vee \mid \Longrightarrow \\
\ominus \in ACompOp &::= < \mid \leq \mid > \mid \geq \mid \equiv \mid \neq
\end{aligned}
$$

Figure 4. Abstract syntax of CL

Input and Output to the Tool: Our tool takes as input, a program written in $\lambda_{\texttt{imp}}$, as well a finite set of logical qualifiers, and infers at each program point and for each variable, the set of logical qualifiers that hold. In the current implementation, we have fixed the set of simple logical formulae that make up the logical qualifiers that are inferred. In particular, we use { $v >= 0$, $v <= 0$, $v = 0$, $v > *$, $v < *$, $v >= *$ }, although this can be easily changed. Here we use $*$ as a place holder for all the variables that are in scope. One can also write `assert` statements, and we print whether we were able to prove it. In case we fail to prove it, either the assertion does not hold, or logical qualifier set is insufficient to prove it.

III. Implementation

We implement our abstract interpreter in *Scala* programming language. Scala is a JVM based multi-paradigm language which compiles down to Java class files. Therefore, our tool can run on any machine that has JVM and Z3 installed in it. We use *ScalaZ3*[**?**], Scala binding for the *Z3* theorem prover, in our tool for solving constraints. We also require *GMP*, GNU Multiple Precision Arithmetic Library installed in our system as an added dependency for Z3.

Constraint Language: CL: We came up with a constraint language, CL that can easily interface with Z3 theorem prover. Our constraint language abstract syntax is described in Figure **??**. To prove $\alpha \implies \beta$, we first see whether $\neg(\alpha \implies \beta)$ is satisfiable for any set of assignments of value to variables in α or β. If it is, we infer that $\alpha \implies \beta$ is false, if not it is true.

IV. Conclusion

We have presented a static analysis that infers logical qualifiers for programs written in $\lambda_{\texttt{imp}}$, that helps us prove interesting assertions in a crafted benchmark. This involved

framing the abstract domain and computable abstract semantics for the abstract interpreter. We implemented the tool in Scala, and used Z3 for discharging implications required for the different abstract semantics operations. Since Z3 does not handle multiplication efficiently, we implemented a simple theory of multiplication for the specific set of logical qualifiers we used in our benchmark programs.

For future work, we would like to extend the language with various interesting features, like procedures, pointers, dynamic memory allocation etc., and experiment with how this affects the analysis. We would also like to add new types to the language, like strings, booleans, objects etc.

Acknowledgment

The authors would like to thank... more thanks here

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.