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

Madhukar N Kedlaya Vineeth Kashyap

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.

1 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.

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

```
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

```
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

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 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`,

```
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

`fact > 0`.

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

2 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, $\{v \mid v = 0 \lor v = 1\}$ is not a valid logical qualifier.

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.

3 The Language $\lambda_{\texttt{imp}}$

Our tool works on programs written in $\lambda_{\texttt{imp}}$. Figure **??** describes the abstract syntax of the language. $\lambda_{\texttt{imp}}$ is an imperative language which has integers, variables, se-

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 1: Examples of a few logical qualifiers and what they describe

$$
\begin{aligned}
e \in Exp &::= n \mid x \mid e_1 \oplus e_2 \\
ce \in Cond &::= e_1 \odot e_2 \\
c \in Cmd &::= \vec{e} \mid \textbf{var}\ \vec{x}\ \textbf{in}\ e \mid \textbf{if}\ ce\ e_t\ \textbf{else}\ e_f \mid \textbf{while}\ ce\ e \mid x := e \\
\oplus \in BinOp &::= + \mid - \mid \times \\
\odot \in RelOp &::= > \mid < \mid = \mid >= \mid <= \mid !=
\end{aligned}
$$

Figure 4: Abstract syntax of $\lambda_{\texttt{imp}}$

quencing, `var` block for introducing new variables in a scope, binary operators (+, - and ×), assignment statements and `if` and `while` as control flow constructs. The language restricts the conditionals in the guards of `if` and `while` to conditional expressions (*Cond*), which use the relational operators ($\odot$).

4 Logical Qualifier Inference through Abstract Interpretation

We infer flow-sensitive logical qualifiers for each variable at each program point by using abstract interpretation. In this section, we describe the lattice for concrete collecting semantics, and the lattice for abstract semantics, provide a galois connection between the two, and then provide rules for abstract semantics informally.

4.1 Abstract domain

The lattice for concrete collecting semantics (which provides all possible values an integer variable in $\lambda_{\texttt{imp}}$ could be, at a given program point for a given program) is given by

Element A set of integers

$\top$ Set of all integers

$\bot$ Empty set

Partial Ordering Subset inclusion

Join Set union

Meet Set intersection

Equality Set equality

The lattice for abstract semantics (the elements of which represent the abstract values over which the abstract semantics computes) is given by:

Element $\{v \mid \psi(v)\}$ where $\psi(v)$ is a logical qualifier for v

$\top$ $\{v \mid \mathit{true}\}$

$\bot$ $\{v \mid \mathit{false}\}$

Partial Ordering $\{v \mid \psi_1(v)\} \sqsubseteq \{v \mid \psi_2(v)\}$ iff $\psi_1(v) \to \psi_2(v)$

Join $\{v \mid \psi_1(v)\} \sqcup \{v \mid \psi_2(v)\} = \{v \mid \psi_3(v)\}$ iff $\psi_1(v) \to \psi_3(v) \wedge \psi_2(v) \to \psi_3(v)$ and there is no $\psi_4(v) \neq \psi_3(v)$ such that $\psi_1(v) \to \psi_4(v) \wedge \psi_2(v) \to \psi_4(v) \wedge \psi_4(v) \to \psi_3(v)$

Meet $\{v \mid \psi_1(v)\} \sqcap \{v \mid \psi_2(v)\} = \{v \mid \psi_3(v)\}$ iff $\psi_3(v) \to \psi_1(v) \wedge \psi_3(v) \to \psi_2(v)$ and there is no $\psi_4(v) \neq \psi_3(v)$ such that $\psi_4(v) \to \psi_1(v) \wedge \psi_4(v) \to \psi_2(v) \wedge \psi_3(v) \to \psi_4(v)$

Equality $\{v \mid \psi_1(v)\} = \{v \mid \psi_2(v)\}$ iff $\psi_1(v) \leftrightarrow \psi_2(v)$

Observe that because the abstract values are logical qualifiers (which cannot have disjunctions), when $\{v \mid \psi_1(v)\} = \{v \mid \psi_2(v)\}$, they must be syntactically equivalent too.

Theorem 1. (Unique Join) *The join of any two elements* $\{v \mid \psi_1(v)\}$ *and* $\{v \mid \psi_2(v)\}$ *is unique.*

Proof. Let us assume that there are two distinct joins for elements $\{v \mid \psi_1(v)\}$ and $\{v \mid \psi_2(v)\}$. Let they be $\{v \mid \psi_3(v)\}$ and $\{v \mid \psi_4(v)\}$. By applying the definition of $\{v \mid \psi_1(v)\} \sqcup \{v \mid \psi_2(v)\}$, to each of these, and using the fact that they are distinct, we get, $\psi_1(v) \to \psi_3(v) \wedge \psi_2(v) \to \psi_3(v) \wedge \psi_4(v) \neq \psi_3(v) \wedge \psi_3(v) \nrightarrow \psi_4(v) \wedge \psi_1(v) \to \psi_4(v) \wedge \psi_2(v) \to \psi_4(v) \wedge \psi_4(v) \nrightarrow \psi_3(v)$. But this is unsatisfiable (can be verified by writing the truth table or feeding to a theorem solver). Thus, the assumption we made, that there are two distinct joins must be incorrect. Therefore, the join of any two elements $\{v \mid \psi_1(v)\}$ and $\{v \mid \psi_2(v)\}$ is unique. □

Using Theorem **??**, we can confirm that definition of the abstraction lattice we provide is indeed a lattice.

Theorem 2. (Galois connection) *There exists a galois connection between the concrete lattice and abstract lattice defined above.*

Proof. Let the abstraction function α be given by $\alpha(I) = \{v \mid \psi_1(v)\}$ where $I \subseteq \mathbb{Z}$ and $\forall i \in I, \psi_1(i)$ is true and there is no $\{v \mid \psi_2(v)\}$ such that $\psi_2(v) \to \psi_1(v)$ and $\forall i \in I, \psi_2(i)$ is true. Let the concretization function γ be given by $\gamma(\{v \mid \psi(v)\}) = \{i \mid i \in \mathbb{Z} \wedge \psi(i)\}$. Using these, we can see that there exists a galois connection between the concrete and the abstract lattice. □

4.2 Abstract semantics

Let block rule: Initialize the abstract values of all the variables introduced in the let block to the least precise value $\top$, i.e. $\{v \mid \textbf{true}\}$ and update the store with these mappings. Update $\mathbb{Q}^\star$ with the new set of logical qualifiers corresponding to each of the newly introduced variables.

Constant rule: Any constant integer i has an abstract value $\{v \mid v = i\}$.

Variable rule: Lookup most recent abstract value of the variable from the store.

Assign rule: For any assignment of the form $x := e$ first evaluate e to an abstract value $\{v \mid \psi(v)\}$. Then refine the abstract value of variable x using the following algorithm.

1. Assume α equals $(x_{new} = v) \wedge \psi(v) \wedge \Gamma$, where x_{new} is a unique variable name associated with x at the given program location and Γ is the assumption that is made in the environment.
2. For each logical qualifier β in $\mathbb{Q}^\star$, check whether $\alpha \implies \beta[v \mapsto x_{new}]$ is valid and filter out the ones that invalidate the formula into a set Σ. Filter out a set of logical qualifiers Λ which contain the relationship between old values of x, x_{old} with x_{new}.
3. Create a new abstract value $\{v_1 \mid \psi_1(v_1)\}$ for x, where $\psi_1(v_1)$ is the conjunction of all the logical qualifiers in $\Sigma - \Lambda$. Update the store with this new value.
4. Update the values of variables in the store which depended on x_{old} using the relations in Λ.
5. Check the validity of Γ using the constraints in Σ. If Γ is invalid, replace it with **true**.

Binary operator rule: For any binary operation of the form $e_1 \oplus e_2$, evaluate e_1 and e_2 to $\{v_1 \mid \psi_1(v_1)\}$ and $\{v_2 \mid \psi_2(v_2)\}$. Create a new abstract value $\{v_3 \mid \psi_3(v_3)\}$ using the following algorithm.

1. Assume α equals $\psi_1(v_1) \wedge \psi_2(v_2) \wedge (v_3 = v_1 \oplus v_2)$.
2. For each logical qualifier β in $\mathbb{Q}^\star$, check whether $\alpha \implies \beta[v \mapsto v_3]$ is valid and filter out the ones that invalidate the formula into a set Σ.
3. Create a new abstract value $\{v_3 \mid \psi_3(v_3)\}$, where v_3 is fresh variable name, $\psi_3(v_3)$ is the conjunction of all the logical qualifiers in Σ.

If rule: For an **if** expression of the format **if** e_1 e_2 **else** e_3, use the following algorithm to evaluate it. We assume that e_1 is of the form $lexp \odot rexp$.

1. Evaluate *lexp* and *rexp* to abstract values $\{v_1 \mid \psi_1(v_1)\}$ and $\{v_2 \mid \psi_2(v_2)\}$.
2. Let
   $cond := \psi_1(v_1) \wedge \psi_2(v_2) \wedge \Gamma$
   $isTrue := \textbf{sat?}((v_1 \odot v_2) \wedge cond)$
   $isTrue := \textbf{sat?}(!(v_1 \odot v_2) \wedge cond)$

   where **sat?** is a function that tests the satisfiability of a given expression.
3. Obtain new stores σ_t and σ_f by evaluating e_2 and e_3 as follows.
   $\sigma_t := \textbf{eval}(e_2, \sigma, \Gamma\ \textbf{extend}\ ((v_1 \odot v_2) \wedge cond))$
   $\sigma_f := \textbf{eval}(e_3, \sigma, \Gamma\ \textbf{extend}\ (!(v_1 \odot v_2) \wedge cond))$

4. Obtain σ_1 and σ_2 using the formula
   $\sigma_1 := \mathit{isTrue} \ ?\ \sigma_t \ :\ \sigma_f$
   $\sigma_2 := \mathit{isFalse} \ ?\ \sigma_f \ :\ \sigma_t$
5. Obtain the final store by merging σ_1 and σ_2
   $\sigma_{final} := \sigma_1 \bigsqcup \sigma_2$

While rule: For a **while** expression of the format **while** $e_1\ e_2$, use the following algorithm to evaluate it. We assume that e_1 is of the form $\mathit{lexp} \odot \mathit{rexp}$. Let $\sigma_s = \sigma_1 = \sigma_2$ be the current store.

1. $\sigma_s = \sigma_1 \bigsqcup \sigma_2$
2. Evaluate *lexp* and *rexp* to abstract values $\{v_1 \mid \psi_1(v_1)\}$ and $\{v_2 \mid \psi_2(v_2)\}$.
3. Let
   $\mathit{cond} := \psi_1(v_1) \wedge \psi_2(v_2) \wedge \Gamma$
   $\mathit{isTrue} := \textbf{sat?}((v_1 \odot v_2) \wedge \mathit{cond})$
   $\mathit{isTrue} := \textbf{sat?}(!(v_1 \odot v_2) \wedge \mathit{cond})$

   where **sat?** is a function that tests the satisfiability of a given expression.
4. Obtain new stores σ_b by evaluating e_2 as follows.
   $\sigma_b := \textbf{eval}(e_2, \sigma, \Gamma\ \textbf{extend}\ ((v_1 \odot v_2) \wedge \mathit{cond}))$
5. Obtain σ_1 and σ_2 using the formulae
   $\sigma_1 := \mathit{isTrue} \ ?\ \sigma_b \ :\ \sigma_s$
   $\sigma_2 := \mathit{isFalse} \ ?\ \sigma_s \ :\ \sigma_b$
6. If $(\sigma_1 \bigsqcup \sigma_2) \neq \sigma_s$; goto 1. Else we have reached a fixpoint.
7. Re-evaluate e_1 in σ_s and get the predicate $\mathit{fPred} = !(v_1 \odot v_2) \wedge \mathit{cond}$. Extend Γ with *fPred*.

5 A Simple Theory for Multiplication

Modern day SMT solvers do not handle non-linear arithmetic on variables well. To solve this problem, we decided to encode a theory for multiplication in our toolkit. Since the set of logical qualifiers we deal with is limited, we were able to come up with a fixed set of rules that produce a sound abstract value of a product. We calculate the product of two abstract values $\{v_1 \mid \psi_1(v_1)\}$ and $\{v_2 \mid \psi_2(v_2)\}$ using the following rules.

$\mathit{case}\ (v_1 = 0 \vee v_2 = 0) :\ \{v_3 \mid v_3 = 0\}$
$\mathit{case}\ \{v_1 \mid \psi_1(v_1)\} = \{v_2 \mid \psi_2(v_2)\} :\ \{v_3 \mid (v_3 > 0) \wedge (v_3 \geq v_1)\}$
$\mathit{case}\ (v_1 > 0 \wedge v_2 > 0) :\ \{v_3 \mid (v_3 > 0) \wedge (v_3 \geq v_1) \wedge (v_3 \geq v_2)\}$
$\mathit{case}\ (v_1 > 0 \wedge v_2 < 0) :\ \{v_3 \mid (v_3 < 0) \wedge (v_3 < v_1) \wedge (v_3 \leq v_2)\}$
$\mathit{case}\ (v_1 < 0 \wedge v_2 > 0) :\ \{v_3 \mid (v_3 < 0) \wedge (v_3 \leq v_1) \wedge (v_3 < v_2)\}$
$\mathit{case}\ (v_1 > 0 \wedge v_2 \geq 0) :\ \{v_3 \mid (v_3 \geq 0) \wedge (v_3 \geq v_2)\}$
$\mathit{case}\ (v_1 > 0 \wedge v_2 \leq 0) :\ \{v_3 \mid (v_3 \leq 0) \wedge (v_3 < v_1) \wedge (v_3 \leq v_2)\}$
$\mathit{case}\ (v_1 < 0 \wedge v_2 \geq 0) :\ \{v_3 \mid (v_3 \leq 0) \wedge (v_3 \leq v_2)\}$
$\mathit{case}\ (v_1 < 0 \wedge v_2 \leq 0) :\ \{v_3 \mid (v_3 \geq 0) \wedge (v_3 > v_1) \wedge (v_3 \geq v_2)\}$
$\mathit{case}\ (v_1 \geq 0 \wedge v_2 > 0) :\ \{v_3 \mid (v_3 \geq 0) \wedge (v_3 \geq v_1)\}$
$\mathit{case}\ (v_1 \geq 0 \wedge v_2 < 0) :\ \{v_3 \mid (v_3 \leq 0) \wedge (v_3 \leq v_1)\}$
$\mathit{case}\ (v_1 \leq 0 \wedge v_2 > 0) :\ \{v_3 \mid (v_3 \leq 0) \wedge (v_3 \leq v_1) \wedge (v_3 < v_2)\}$

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

Figure 5: Abstract syntax of CL

case $(v_1 \leq 0 \wedge v_2 < 0)$: $\{v_3 \mid (v_3 \geq 0) \wedge (v_3 \geq v_1) \wedge (v_3 > v_2)\}$
case $(v_1 \geq 0 \wedge v_2 \geq 0)$: $\{v_3 \mid (v_3 \geq 0) \wedge (v_3 \geq v_1) \wedge (v_3 \geq v_2)\}$
case $(v_1 \geq 0 \wedge v_2 \leq 0)$: $\{v_3 \mid (v_3 \leq 0) \wedge (v_3 \leq v_1)\}$
case $(v_1 \leq 0 \wedge v_2 \geq 0)$: $\{v_3 \mid (v_3 \leq 0) \wedge (v_3 \leq v_2)\}$
case $(v_1 \leq 0 \wedge v_2 \leq 0)$: $\{v_3 \mid (v_3 \geq 0) \wedge (v_3 \geq v_1) \wedge (v_3 \geq v_2)\}$
default : $\{v_3 \mid \textsf{\textbf{true}}\}$

6 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.

7 Results

We tested our tool extensively using our own test cases and a few from the slides on CUTE and DART. We were able to prove all the assertions that the other tools claimed to prove. Some of the tests that were run are described in Table **??**.

8 Comparison with Liquid Types

Our work is inspired by the *Liquid Types* proposed by Rondon et al[**?**] for OCAML language. Our implementation differs from the original implementation for OCAML in three ways. First, we use abstract interpretation for performing logical qualifier inference, whereas the original implementation uses a type system formulation to

Test	Description	Result	Time taken (s)
fact.0js	Calculates factorial of a given number n.	factorial $>$ 0 $\wedge$ factorial $\geq$ input assertions passed.	0.92
fibo.0js	Calculates first n fibonacci numbers.	fibo $>$ 0 assertions passed.	1.34
max.0js	Gives the maximum of two numbers n1 & n2.	max $\geq$ n1 $\wedge$ max $\geq$ n2 assertions passed.	0.79
dart1.0js	First simple C example given in DART slides.	Proved that we can reach a bad state.	0.65
dart2.0js	C program with non-linear arithmetic from DART slides.	Due to our theory for multiplication we were able to analyse the whole program without getting stuck and proved that we can reach a bad state.	0.71
cute.0js	C program with non-linear arithmetic from CUTE slides.	Due to our theory for multiplication we were able to analyse the whole program without getting stuck and proved that we can reach a bad state.	0.7

Table 2: Tests and results.

infer the dependent types which have logical qualifiers. Second, we implement our logical qualifier inference system for $\lambda_{\texttt{imp}}$ which unlike OCAML, $\lambda_{\texttt{imp}}$ allows mutability and therefore offers a different set of challenges that need to be considered. Third, unlike the original implementation which was flow insensitive, our implementation is flow sensitive and we infer logical qualifiers at every point in the program.

9 Conclusion and Future Work

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.