Logical Qualifier Inference for ${}_0JS$

Madhukar N Kedlaya
Vineeth Kashyap

1 Introduction

In our project, we infer *Logical Qualifiers* for variables in a program written in an imperative language called ${}_0JS$. Our work is inspired by the *Liquid Types* proposed by Rondon et al[3] for OCAML language. We use abstract interpretation and *Predicate Abstraction* for automatically inferring logical qualifiers for program variables. These logical qualifiers can be used to prove certain safety properties for the input program. We use *Z3* theorem prover[2] for solving constraints.

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 *Hindley-Milner* Type Inference to infer types of the subexpressions. Second, we implement our logical qualifier inference system for ${}_0JS$ which unlike OCAML, ${}_0JS$ 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.

2 ${}_0JS$

${}_0JS$ is an imperative language which has integers, variables, sequencing, *let* block for introducing new variables in a scope, binary operators (+, - and ×), assignment statements and *if* and *while* as control flow constructs. Figure 1 describes the abstract syntax of the language.

$$\begin{aligned} e \in \mathit{Exp} ::=\ & n \mid x \mid \vec{e} \mid \textbf{var}\ \vec{x}\ \textbf{in}\ e \mid e_1 \oplus e_2 \\ & \mid \textbf{if}\ e_1\ e_2\ \textbf{else}\ e_3 \mid \textbf{while}\ e_1\ e_2 \\ & \mid x := e \\ \oplus \in \mathit{BinOp} ::=\ & + \mid - \mid \times \end{aligned}$$

Figure 1. Abstract syntax of ${}_0JS$

3 Setup

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*[1], 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.

4 Methodology

We define *logical qualifier* as a predicate over special value variable v and a placeholder variable $\star$ which can be replaced by any program variable in the current scope. We use $\mathbb{Q} = \{v = 0, v > 0, v \geq 0, v < 0, v \leq 0, v \geq \star, \star \geq v\}$ as the set of *logical qualifiers* that we use to describe the abstract value of a variable. This set can be expanded to improve precision. We now describe our abstract interpreter based on the following lattice structure.

Elements Each element in the lattice is of the form $\{v \mid e\}$ where v is a *value variable* and e is a *refinement predicate* over the value variable. A refinement predicate is a conjunction over set of *logical qualifiers*.

$\top$ Top element of the lattice is the most precise abstract value possible which is $\{v \mid False\}$.

$\bot$ Bottom element of the lattice is the least precise abstract value possible which is $\{v \mid True\}$.

$\sqsubseteq$ The ordering relation between the lattice elements is reverse implication, $\Leftarrow$.

We define abstract value as a tuple $\{v \mid \psi(v)\}$, where v is an unique variable name for a particular program location and position and $\psi(v)$ is constraint associated with the variable. We define $\mathbb{Q}^\star$, a set of logical qualifiers where every $\star$ in $\mathbb{Q}$ is replaced by program variables in current scope. We refine the abstract value associated with the variable using the following rules.

Let block rule: Initialize the abstract values of all the variables introduced in the let block to $\bot$, i.e. $\{v \mid True\}$ 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 $(x_1 = v) \wedge \psi(v)$, where x_1 is a unique variable name associated with x at the given program location.
2. For each logical qualifier β in $\mathbb{Q}^\star$, check whether $\alpha \implies \beta[v \mapsto x_1]$ is valid and filter out the ones that invalidate the formula into a set Σ.
3. Create a new abstract value $\{x_1 \mid \psi(x_1)\}$ for x, where $\psi(x_1)$ is the conjunction of all the logical qualifiers in Σ. Update the store with this new value.

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

1. Assume α equals $\psi(v_1) \wedge \psi(v_2) \wedge (v_3 = v_1 \oplus v_2)$.
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 $\{v_3 \mid \psi(v_3)\}$, where v_3 is fresh variable name, $\psi(v_3)$ is the conjunction of all the logical qualifiers in Σ.

If rule: First, define a helper function *join* as follows.

$$join(\{v_i \mid \psi(v_i)\},\ \{v_j \mid \psi(v_j)\},\ \mathbb{Q}^\star) = \{v_k \mid \psi(v_k)\}$$

where

- v_k is a fresh variable.
- $\psi(v_k)$ is conjunction of all the logical qualifiers from $\mathbb{Q}^\star$ which satisfy $f(\beta) \Rightarrow ((v_k = v_i)\ \wedge\ (v_k = v_j)\ \wedge\ \psi(v_i)\ \wedge\ \psi(v_j) \implies \beta[v \mapsto v_k])$.

For an if expression of the format $if\ e_1\ then\ e_2\ else\ e_3$, use the following algorithm to evaluate it to an abstract value $\{v_0 \mid \psi(v_0)\}$.

1. Evaluate e_1 to an abstract value $\{v_1 \mid \psi(v_1)\}$.
2. Evaluate e_2 in an environment extended with $\psi(v_1)$ as an additional constraint to get back $\{v_2 \mid \psi(v_2)\}$.
3. Evaluate e_3 in an environment extended with $!\psi(v_1)$ as an additional constraint to get back $\{v_3 \mid \psi(v_3)\}$.
4. Perform $join(\{v_2 \mid \psi(v_2)\}, \{v_3 \mid \psi(v_3)\}, \mathbb{Q}^\star)$ to get back $\{v_0 \mid \psi(v_0)\}$.
5. Merge the two stores from e_2 and e_3 to get logically qualified values for each of the variables using the *join* function.

5 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 2. 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.

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

Figure 2. Abstract syntax of CL

6 Progress

We have currently implemented an abstract interpreter that can infer logical qualifiers for all the constructs in ${}_0\mathit{JS}$ with the exception of *while*. We are squashing out bugs and testing the analysis for the remaining constructs. We still need to work out the abstract semantics for *while*, create a good test benchmark, conduct experiments to get insights on performance and precision of our analysis, and then as a nice to have feature, add integer arrays into our language, so that we can infer and verify more interesting properties.

We believe that the abstract interpretation formulation of inferring logical qualifiers is novel, and is a different take on Liquid Types[3]. We plan to formalize the abstract interpretation rules and prove soundness and termination.

References

[1] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of Z3: Integrating SMT and programming. In *Computer-Aideded Deduction (CADE) Tool Demo*, 2011.

[2] Microsoft. Z3: Theorem prover. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`, 2012. [Online; accessed 10-May-2012].

[3] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.