LOGICAL ENCODING OF PROGRAMS

AWS ALBARGHOUTHI

ABSTRACT. These notes describe the process of encoding program executions as a formula in first-order logic. This allows us to apply various forms of automated verification by checking satisfiability of the encodings.

1. Language

Recall the simple imperative language we used to present operational semantics in class. For now, we will use a subset of that language that does not involve while loops.

Definition 1.1 (Language Syntax)**.** A program P is defined as follows:

$$\begin{aligned} P ::= {} & x \leftarrow a && \text{assignment} \\ & \mid \textsf{if } b \textsf{ then } P_1 \textsf{ else } P_2 && \text{conditional} \\ & \mid P_1; P_2 && \text{sequential composition} \end{aligned}$$

where a is an arithmetic (integer) expression over program variables, and b is a Boolean expression over program variables. We shall use V to denote the set of all program variables. All variables $v \in V$ are assumed to be integer-typed ($\mathbb{Z}$).

Definition 1.2 (Language Semantics)**.** Recall that a *state* $s : V \to \mathbb{Z}$ of a program P is a map from variables V to integer values. Given a program P, the operational semantics define a relation $\to$ describing the execution of P beginning at some state s and ending in state s', denoted as follows:

$$\langle P, s \rangle \to s'$$

2. Transition Relations

We now define *transition relations*, which are relations defining the operational semantics of a program. We will later show how to define transition relations in first-order logic. Given a program P, we will define its transition relation $\mathit{trans}(V, V')$ over two sets of variables, the program variables V and a copy of program variables V'. The idea is that the variables V hold the initial state s of the program, and V' hold the final state s' after the program has completed execution. This is best seen through an example:

Example 2.1. Consider the simple program P defined as a single statement:

$$x \leftarrow x + 1$$

The set of variables V of P is the singleton set $\{x\}$. The set V' is $\{x'\}$. Therefore the transition relation $\mathit{trans}(x, x')$ is over two variables, x and x'. Concretely, trans is defined as follows:

$$\mathit{trans} = \{(n, n+1) \mid n \in \mathbb{Z}\}$$

That is, for any number n, the pair $(n, n+1)$ is in the transition relation, denoting that if we start executing the program with $x = n$, we end up with $x' = n + 1$.

Observe how there is a one-to-one correspondence between elements of the transition relation and pairs of states s, s' such that $\langle P, s\rangle \to s'$.

3. First-Order Theory

Recall the first-order theories we discussed in class. To encode the relation *trans*, we will use the theory of *linear integer arithmetic* (LIA). LIA allows formulas to contain integer-valued variables, equality, ineqality, addition, and *no multiplication* (multiplication will only be to replace addition, i.e., $x + x$ will be written as $2x$). Checking satisfiability of formulas in LIA is decidable.

Definition 3.1 (Linear Integer Arithmetic)**.** Formally, an LIA formula φ is of the following form:

$$\begin{aligned} \varphi ::= {} & a_1 = a_2 \\ & \mid a_1 \leqslant a_2 \\ & \mid a_1 < a_2 \\ & \mid \varphi \wedge \varphi \\ & \mid \varphi \vee \varphi \\ & \mid \neg\varphi \\ & \mid \exists x.\, \varphi \\ & \mid \forall x.\, \varphi \end{aligned}$$

where a_1, a_2 are arithmetic expressions of the form $c_1x_1 + \ldots + c_nx_n$, where $c_i \in \mathbb{Z}$ and x_i are first-order variables. For instance, $2x + 3y$. Note that $\Rightarrow$ and $\iff$ can be written using $\wedge, \neg$.

A model m for a formula φ, denoted $m \models \varphi$, is a mapping from the free variables of φ, denoted $fv(\varphi)$, to integers, that satisfies the formula. For example, consider $x + y > 0$. Let $m = \{x \mapsto 1, y \mapsto 0\}$. We have $m \models x + y > 0$

4. Encoding Transition Relations

Single assignments Let us now consider how we would encode a transition relation of a single assignment. Let the assignment be of the form

$$x \leftarrow a$$

We simply transform this into a formula φ defined as follows:

$$\varphi \equiv x' = a \wedge \bigwedge_{y \neq x, y \in V} y' = y$$

Notice that the variable x on the lhs of the assignment is encoded as x', denoting the final value of x after the assignment is performed. Notice also that the final state of every variable y other than x is the same as its initial value, since it is not modified by the assignment statement.

Example 4.1. Consider the assignment

$$x \leftarrow x + y$$

This is encoded as the transition relation

$$trans(x, y, x', y') \equiv x' = x + y \wedge y' = y$$

assuming $V = \{x, y\}$

Loop-free Programs Now that we have discussed how to construct *trans* for a simple program comprised of a single assignment, we can define the rules for full (loop-free) programs.

Definition 4.2 (Transition-Relation Encoding)**.**

$$\begin{aligned}
\mathsf{enc}(x \leftarrow a) &\triangleq x' = a \wedge \bigwedge_{y \neq x, y \in V} y' = y \\
\mathsf{enc}(\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2) &\triangleq (b \Rightarrow \mathsf{enc}(P_1)) \wedge (\neg b \Rightarrow \mathsf{enc}(P_2)) \\
\mathsf{enc}(P_1; P_2) &\triangleq \exists V''.\, trans_1(V, V'') \wedge trans_2(V'', V') \\
\\
&\quad \text{where } trans_1(V, V') \equiv \mathsf{enc}(P_1) \\
&\qquad\quad\ \ trans_2(V, V') \equiv \mathsf{enc}(P_2)
\end{aligned}$$

To illustrate the encoding, let us consider some examples:

Example 4.3. Consider the following program P

$$\mathsf{if}\ x > 0\ \mathsf{then}\ x \leftarrow x + 1\ \mathsf{else}\ x \leftarrow y$$

The encoding $\mathsf{enc}(P)$ is a formula over V and V' defined as follows:

$$(x > 0 \Rightarrow x' = x + 1 \wedge y' = y) \wedge (x \leqslant 0 \Rightarrow x' = y \wedge y' = y)$$

The left conjunct defines the transition relation of the *then* branch; the right conjunct defines the *else* branch.

Example 4.4. Now consider the following program P:

$$x \leftarrow x + 1; y \leftarrow y + 1$$

The function enc encodes this program one instruction at a time, and combines the results. First, we encode $x \leftarrow x + 1$ as follows:

$$trans_1(x, y, x', y') \equiv \mathsf{enc}(x \leftarrow x + 1) \equiv x' = x + 1 \wedge y' = y$$

Then, we encode $y \leftarrow y + 1$:

$$trans_2(x, y, x', y') \equiv \mathsf{enc}(y \leftarrow y + 1) \equiv y' = y + 1 \wedge x' = x$$

We now simply conjoin the two transition relations; however, we have to be careful about the variable names: the output variables of $trans_1$ need be the same as the input variables of $trans_2$. Therefore we conjoin them as follows:

$$trans_1(x, y, x'', y'') \wedge trans_2(x'', y'', x', y')$$

Notice that we have renamed the outputs of $trans_1$ and inputs of $trans_2$ so that they actually match; these variables are $V'' = \{x'', y''\}$. Intuitively, V'' defines the state of the program after executing the first instruction; but since we only care about the final state, we *quantify the variables* V'' *away.* Finally, we arrive at the following encoding

$$\exists x'', y''.\, (x'' = x + 1 \wedge y'' = y) \wedge (y' = y'' + 1 \wedge x' = x'')$$

The first conjunct is $trans_1(x, y, x'', y'')$, and the second is $trans_2(x'', y'', x', y')$. Notice that the free variables of this formula are x, y and x', y', defining the initial states and final states, respectively.

5. Soundness and Completeness

We need to show that our encoding is sound and complete. Soundness ensures that the transition relation does not over-approximate the operational semantics relation →. Completeness, on the other hand, ensures that every element of → is also in the encoding.

Theorem 5.1 (Soundness)**.** *Fix a program P with variables V. Let $m \models \mathsf{enc}(P)$. Let*

$$s = \{v \mapsto m(v) \mid v \in V\}$$
$$s' = \{v \mapsto m(v') \mid v' \in V'\}$$

Then, $\langle P, s\rangle \rightarrow s'$.

Theorem 5.2 (Completeness)**.** *Fix a program P with variables V. Let s, s' be such that $\langle P, s\rangle \rightarrow s'$. Let*

$$m = \{v \mapsto s(v) \mid v \in V\} \cup \{v' \mapsto s'(v) \mid v \in V\}$$

Then, $m \models \mathsf{enc}(P)$.

Proof. Both proofs proceed by structural induction on the program. □

6. Verification

Now that we have defined the encoding, we can use it to check if a Hoare triple holds. Suppose we are given a Hoare triple $\{\phi\}P\{\psi\}$, where ϕ and ψ are LIA formulas over program variables. We would like to check if the Hoare triple is valid; informally, for any state s satisfying ϕ, for any state s' such that $\langle P, s\rangle \rightarrow s'$, we want to make sure that s' satisfies ψ.

To automatically check if the Hoare triple is valid, we encode *all* executions starting at ϕ and check whether they end up in ψ. Formally, we check validity of the following formula:

$$(\phi \wedge \mathsf{enc}(P)) \Rightarrow \psi' \text{ is VALID}$$
$$\textit{iff}$$
$$\{\phi\}P\{\psi\} \text{ is VALID}$$

Intuitively, $\phi \wedge \mathsf{enc}(P)$ defines the set executions of P constrained to the ones starting at ϕ. The implication ensures that all final states are contained in ψ'. Notice that we use a primed version of ψ, since final states in the encoding are over primed variables.

Example 6.1. Let us consider a simple example. To check validity of

$$\{x > 0\}x \leftarrow x + 1\{x > 1\}$$

we check validity of the following formula:

$$(x > 0 \wedge x' = x + 1) \Rightarrow x' > 1$$

which is valid.

Example 6.2. Here is another example, but this time the Hoare triple is invalid:

$$\{x > 0\} x \leftarrow x + y \{x > 1\}$$

The formula Ψ

$$\Psi \equiv (x > 0 \wedge x' = x + y \wedge y' = y) \Rightarrow x' > 1$$

is not valid. Here is a counterexample; i.e., a model m for the negation of the formula, $\neg\Psi$:

$$m = \{x \mapsto 0, y \mapsto 0, x' \mapsto 0, y' \mapsto 0\}$$

In other words, m says that if $x = y = 0$ in the initial state, the final state will be such that $x = 0$, therefore invalidating the postcondition, which specifies the $x > 1$ when the program terminates.

7. Bounded Verification

We will now enrich our programming language to include while loops. For simplicity, we shall assume a program P contains a single while loop; that is, P is of the form:

$$P_{pre}; \mathsf{while}\ b\ \mathsf{do}\ P_{body}$$

where P_{pre} and P_{body} are loop-free programs.

Suppose we want to check validity of a Hoare triple $\{\phi\}P\{\psi\}$ where P contains a while loop. We will now show how to check that the Hoare triple is valid assuming P can only take a bounded number of loop iterations, e.g., 3 loop iterations. In the literature, this process is known as *bounded verification*, *bounded model checking*, or *symbolic execution*.

The first step is to encode executions that take n loop steps in P. We define this using the function $\mathsf{enc}_n(P)$ as follows, where $n \in [0, \infty)$.

$$\mathsf{enc}_n(P) \triangleq \mathit{trans}_{pre}(V, V^1) \wedge \left(\bigwedge_{i=1}^{n} b^i \wedge \mathit{trans}_{body}(V^i, V^{i+1}) \right) \wedge \neg b^{n+1}$$

where $\mathit{trans}_{pre} \equiv \mathsf{enc}(P_{pre})$ and $\mathit{trans}_{body} \equiv \mathsf{enc}(P_{body})$. Notice how we rename the variables such that we have n copies of the transition relation of the loop body, where iteration i starts with variables V^i and ends with variables V^{i+1}. Also, note that at the end we ensure that the loop-exit condition is true: $\neg b^{n+1}$.

Now that we know how to encode n loop iterations, we can check if a Hoare triple is valid for loop $\leqslant n$ loop iterations by checking validity of the following formula:

$$\bigwedge_{i=1}^{n} \left(\phi \wedge \mathsf{enc}_i(P) \Rightarrow \psi^{i+1} \right)$$

Note that each conjunct i ensures that executions that start in ϕ and run for i loop iterations end in a state satisfying ψ.

8. Induction using Bounded Verification

We have thus far discussed how to check correctness of a program up to an unrolling bound n. We will now show how to use bounded verification to construct an inductive argument that proves correctness of a program *for all* n.

Induction Principle Suppose we have some predicate $\mathit{prop}(i)$ that is parameterized by some number $i \in \mathbb{N}$, and we want to show that $\mathit{prop}(i)$ is true for all $i \in \mathbb{N}$. A proof by induction proceeds in two steps:

(1) **Base case:** Show that $\mathit{prop}(0)$ is true.
(2) **Inductive step:** Show that for any i, if $\mathit{prop}(i)$ is true, then $\mathit{prop}(i+1)$ is also true.

These two conditions, as we all know from kindergarten, imply that $\forall i \in \mathbb{N}.\, \mathit{prop}(i)$ is true.

Hoare proofs via induction We can try to apply induction to the proof of validity of a Hoare triple of the form:

$$\{\phi\} \text{ while } b \text{ do } P_{body} \ \{\psi\}$$

as follows:

(1) **Base case:** Show that

$$\phi \Rightarrow \psi$$

is valid. In other words, show that the program satisfies the Hoare triple if it takes 0 loop iterations.

(2) **Inductive step:** Show that

$$(\psi \wedge b \wedge \mathit{trans}_{body}(V, V')) \implies \psi'$$

is valid. In other words, assuming the program starts in a state satisfying the postcondition if the loop has terminated, then executing the loop once should maintain that the program still satisfies the postcondition.

Together, these conditions imply that the Hoare triple $\{\phi\}P\{\psi\}$ is valid.

Example 8.1. Consider the following Hoare triple:

$$\begin{aligned} &\{x \geqslant 0\} \\ &\text{while } x > 0 \text{ do} \\ &\quad x \leftarrow x - 1 \\ &\{x \geqslant 0\} \end{aligned}$$

(1) **Base case:** We get the following formula:

$$x \geqslant 0 \Rightarrow x \geqslant 0$$

This formula is trivially true.

(2) **Inductive step:** For the inductive step, we check that, assuming the program starts at the loop head in a state satisfying the postcondition, then executes one loop iteration, it ends in a state in the postcondition (assuming the loop terminates). This is encoded as follows:

$$(x \geqslant 0 \wedge x > 0 \wedge x' = x - 1) \Rightarrow x' \geqslant 0$$

This formula is valid. *Checking validity of this formula, effectively, checks that the postcondition is an inductive loop invariant.*

k-induction Induction, in its simple form above, will generally not work. The problem is that the postcondition may be too weak for an inductive loop invariant, and we instead require a stronger argument.

We can instead use k-induction, where consider k base cases and k steps of iteration. k-induction is a generalization of $1-$induction above:

(1) **Base case:** Show that $\mathit{prop}(0), \ldots, \mathit{prop}(k-1)$ is true.
(2) **Inductive step:** Show that if $\mathit{prop}(n), \ldots, \mathit{prop}(n+k-1)$ are true, then $\mathit{prop}(n+k)$ is true.

These two conditions together imply that $prop(i)$ is true for all $i \in \mathbb{N}$.

We can apply this idea to the proof of programs:

(1) **Base case:** Show that the following formulas are valid:

$$\begin{aligned} \phi &\Rightarrow \psi \\ \phi \wedge b \wedge trans(V, V^1) &\Rightarrow \psi^1 \\ &\dots \\ \phi \wedge b \wedge trans(V, V^1) \wedge \dots \wedge b^{k-2} \wedge trans(V^{k-2}, V^{k-1}) &\Rightarrow \psi^{k-1} \end{aligned}$$

This shows that the program is correct for $0, \dots, k$ loop iterations.

(2) **Inductive step:** Then, we show that the following formula is valid:

$$\left[\bigwedge_{i=0}^{k-1} (\psi^i \wedge b^i \wedge trans_{body}(V^i, V^{i+1})) \right] \Longrightarrow \psi^k$$

Together these two cases ensure that the program is correct for any number of loop iterations.

University of Wisconsin–Madison
E-mail address: `aws@cs.wisc.edu`