Software Verification

2024-2025

TP2: Semantics

Vincent Penelle

Content

- Implementation of the semantics of a control-flow automaton.

The goal of this lab session is for you to implement the module CommandSemantics.ml of our tool Simple Program Analyser. More precisely, you simply have to provide the traduction of each command in a formula describing its effect on a configuration of the control-flow automaton.

Rappel:

A *control-flow automaton* is a tuple $(Q, q_i, q_{bad}, \Delta)$ where Q is a finite set of states, $q_i \in Q$ is the initial state, q_{bad} is the final state (or "bad"), and $\Delta \subseteq Q \times \text{Op} \times Q$ is the set of transitions, where Op is the set of possible operation of the automaton. This set is defined hereafter. Said otherwise, it is an automaton labelled with operations (over variables)

We consider a set of variables X. An *expression* is an arithmetical expression over $\mathbb{Z}$ containing or not variables from X:

$$\texttt{exp} ::= n \in \mathbb{Z} | x \in X | \texttt{exp} + \texttt{exp} | \texttt{exp} - \texttt{exp} | \texttt{exp} \times \texttt{exp} | \texttt{exp} / \texttt{exp}$$

A *guard* is a comparison between two expressions:

$$\texttt{guard} ::= \texttt{exp} = \texttt{exp} | \texttt{exp} < \texttt{exp} | \texttt{exp} \leq \texttt{exp} | \texttt{exp} > \texttt{exp} | \texttt{exp} \geq \texttt{exp}$$

An operation is either `skip`, or a guard, or an affectation of the form $x := exp$, for $x \in X$ and exp an arithmetical expression:

$$op ::= \texttt{skip} | \texttt{guard} | x := \texttt{exp}$$

Point technique:

A *configuration* of a control-flow automaton $\mathcal{A}$ is a pair (q, σ), where q is a state, and σ a function from X to $\mathbb{Z}$ called *valuation*.

The *semantic* of a operation op is a relation containing valuations (σ_1, σ_2) such that σ_2 can be obtained by applying op to σ_1. Formally, we have $[\![\texttt{skip}]\!] = \text{id}$, $[\![\text{guard}]\!]$ is the set containing all (σ, σ) such that σ satisfies the guard, and $[\![\texttt{x} := \texttt{exp}]\!] = \{(\sigma, \sigma[x := exp])\}$.

More precisely, this semantics will be described as a first-order formula over $\mathbb{Z}$, which uses two copies of X, X_b and X_a representing respectively valuations before and after the operation.

The semantics of operations are the following:

$$[\![\texttt{skip}]\!] = \bigwedge_{x \in X} x_b == x_a$$

$$[\![\texttt{guard}]\!] = \texttt{guard}_b \wedge \texttt{side}_b(\texttt{exp}_1) \wedge \texttt{side}_b(\texttt{exp}_2) \wedge \bigwedge_{x \in X} x_b == x_a$$

$$[\![x := \texttt{exp}]\!] = x_a == \texttt{exp}_b \wedge \texttt{side}_b(\texttt{exp}) \wedge \bigwedge_{y \in X, x \neq y} y_b == y_a$$

where $\texttt{exp}_b$, $\texttt{side}_b$ and $\texttt{guard}_b$ denote these expressions on the copy X_b of X, $\texttt{exp}_1$ and $\texttt{exp}_2$ are the two members of `guard` and `side`(`exp`) is a formula ensuring that the expression is well-defined, i.e., no division by 0 occurs. This is necessary as Z3 considers that division by 0 is defined, but with no value specified (thus, if you don't put this restriction, the value of a division by 0 will exists, but can be anything).

Exercice 1: CommandSemantics.ml

Download the archive containing the code of this lab session. Complete the module CommandSemantics.ml, with the help of the following indications, and of those in CommandSemantics.mli which indicate the syntax used in the project and precise the role of functions.

Important point: you have to return a result of the form $(f, list)$, where *list* is the list of variables *modified* by the operation (i.e., empty for `skip` and `guard` and equal to $[x]$ for $x :=$ `exp`), and f is the formula described earlier, *without the right part (in red) indicating equalities between unmodified variables* (this allows to abstract this part and to give you a more efficient variant of this part of the formula when we'll treat the executions).

1. Start with formula_of_skip.
2. Define recursively `side` (on paper). This formula must be true if and only if the values of the variables ensure the expression does not contain any division by 0.
3. Implement expr_to_z3_expr. This function returns a formula representing the expression received as an argument, and a list of formula representing the different part of `side` (this allows to perform a unique AND at the end).
4. End by implementing formula_of_guard, fwd_formula_of_assign and bwd_formula_of_assign. Note that in those formula, we only use symmetrical operations (that should help you deducing the backward formula from the forward one).

At any moment in the lab, you can launch `maketest` to test your code (it will tell you if your formula correspond to what is expected, and will display potential differences – note that some of them are not so problematic if it is simply about the order). Think to replace failures with simple formula for the program to execute (e.g., you can put everywhere the same result as formula_of_skip as a placeholder).