Formal Methods and Specification (SS 2021) Lecture 12: Operational Program Semantics

Stefan Ratschan

Katedra číslicového návrhu
Fakulta informačních technologií
České vysoké učení technické v Praze

Evropský sociální fond Praha & EU: Investujeme do vaší budoucnosti

Motivation

Definition of "program correctness" in Lecture 4
depended on the notion of "program execution",
which we did not define precisely.

Instead:
tiny imperative programming language,
for which we all agree on its behavior

Now: precise definition of programming language behavior

still for a very small language

Example Program

1: $i \leftarrow 0$
2: **input** k
3: **if** $a[i] = k$ **then**
4:     **stop**
5: $i \leftarrow i + 1$
6: **goto** 2

Example execution for random initial values, and user input 0

pc	i	k	a
1	7	2	[9, 9, 9, ...]
2	0	2	[9, 9, 9, ...]
3	0	0	[9, 9, 9, ...]
5	0	0	[9, 9, 9, ...]
6	1	0	[9, 9, 9, ...]

Program State

Program counter + values of all program variables

We assume a program that has

- L lines, and
- variables from a set V s.t. every variable v has type T_v.

A *program state* is a function that assigns

- to the special variable pc an element from $\{1, \dots, L\}$, and
- to each variable $v \in V$ an element from the set T_v.

Example: $\{pc \mapsto 2, i \mapsto 1, a \mapsto [6, 2, 3, 4], k \mapsto 3\}$

Set of all states: S

State Evolution

Each step of the program

- takes a certain state $s \in S$, and
- computes another state $s' \in S$.

In this case we write $s \rightarrow_P s'$.

The relation $\rightarrow_P \subseteq S \times S$ is called *transition relation* of the program P

Why a relation, not a function?

Side Effects

- program may be influenced by environment
  (e.g., read from disk, ask for user input)
- program may influence environment
  (e.g., write to disk, display etc.)

Hence:

- more than one possible next state,
- more than one possible final output for one input

How to Define the Transition Relation?

First for individual commands (e.g., assignment, **goto**),
then for whole programs.

Based on logical formulas
(to allow usage of tools, demo)

We assume the necessary logical theories
(and will not write them explicitely).

Transition Relation: Assignments (Example)

P is a program that has at line 5 command $i \leftarrow i + 1$.

$$s = \{pc \mapsto 5, a \mapsto [4, 5, 6, 7, 8], i \mapsto 2, k \mapsto 7\}$$

$$s' = \{pc \mapsto 6, a \mapsto [4, 5, 6, 7, 8], i \mapsto 3, k \mapsto 4\}$$

$s \rightarrow_P s'$?

What does $i \leftarrow i + 1$ mean? $i = i + 1$?

Different variable! $i' = i + 1$

the other variables? they do not change: $a' = a$, $k' = k$

$$pc' = pc + 1 \wedge i' = i + 1 \wedge a' = a \wedge k' = k$$

Transition Relation: Assignments (Example)

$$s = \{pc \mapsto 5, a \mapsto [4,5,6,7,8], i \mapsto 2, k \mapsto 7\}$$
$$s' = \{pc \mapsto 6, a \mapsto [4,5,6,7,8], i \mapsto 3, k \mapsto 4\}$$

$$pc' = pc + 1 \wedge i' = i + 1 \wedge a' = a \wedge k' = k$$

s, s' should satisfy this, renaming of variables

$$\pi(s') = \{pc' \mapsto 6, a' \mapsto [4,5,6,7,8], i' \mapsto 3, k' \mapsto 4\}$$

$$s \sqcup \pi(s') = \left\{ \begin{array}{l} pc \mapsto 5, a \mapsto [4,5,6,7,8], i \mapsto 2, k \mapsto 7, \\ pc' \mapsto 6, a' \mapsto [4,5,6,7,8], i' \mapsto 3, k' \mapsto 4 \end{array} \right\}$$

$$s \sqcup \pi(s') \not\models pc' = pc + 1 \wedge i' = i + 1 \wedge a' = a \wedge k' = k$$

But for

$$s' = \{pc \mapsto 6, a \mapsto [4,5,6,7,8], i \mapsto 3, k \mapsto 7\}$$
$$s \sqcup \pi(s') \models pc' = pc + 1 \wedge i' = i + 1 \wedge a' = a \wedge k' = k$$

Transition Relation: Assignments

If $s(pc)$ points to a line with an assignment $v \leftarrow t$:

$s \rightarrow_P s'$ iff

$$s \sqcup \pi(s') \models pc' = pc + 1 \wedge v' = t \wedge \bigwedge_{u \in V, u \neq v} u' = u$$

where

- $\pi : S \rightarrow S'$, where S' is as S, but assigns values to primed variables,

$$\text{for all } r \in S, v \in \{pc\} \cup V, \pi(r)(v') := r(v)$$

- for functions r, r' with disjoint domains R and R',
  $r \sqcup r'$ is a function with domain $R \cup R'$ for all $v \in R \cup R'$,

$$(r \sqcup r')(v) = \begin{cases} r(v), \text{if } v \in R, \text{and} \\ r'(v), \text{if } v \in R'. \end{cases}$$

Transition Relation: Control Structures

- If $s(pc)$ points to a line **goto** r:

  $s \rightarrow_P s'$ iff
  $$s \sqcup \pi(s') \models pc' = r \wedge \bigwedge_{u \in V} u' = u$$

- If $s(pc)$ points to a line **if** P **then**:

  $s \rightarrow_P s'$ iff
  $$s \sqcup \pi(s') \models [P \Rightarrow pc' = pc + 1] \wedge [\neg P \Rightarrow pc' = l] \wedge \bigwedge_{u \in V} u' = u$$

  where l is the number of the program line after the end of the **if-then** block. (`if_then.cvc`)

- Further control structures: combination of **if-then** and **goto**
  see also structured programming theorem [Böhm and Jacopini, 1966]

Transition Relation: Assertions

Two kinds of assertions:

- `@`
- **assume**

Same run-time behavior:

If $s(pc)$ points to a line **assume** ϕ or `@` ϕ,

$s \rightarrow_P s'$ iff

$$s \sqcup \pi(s') \models pc' = pc + 1 \wedge \phi \wedge \bigwedge_{u \in V} u' = u$$

Transition Relation: Side Effects

input v ???

$v' = \text{input}()$???

input x; **input** y

$x' = \text{input}() \wedge y' = \text{input}()$???

Equality is transitive, hence: $x' = y'$!

User input does not only depend on the program state.

Side-effect: influence of something, or on something
not part of the program state.

A mathematical model of the user is too complicated.

Transition Relation: Side Effects

If $s(pc)$ points to a line of the form **input** v:

$s \rightarrow_P s'$ iff

$$s \sqcup \pi(s') \models pc' = pc + 1 \wedge \bigwedge_{u \in V, u \neq v} u' = u$$

if $u = v$?

Non-determinism, demo: `input.cvc`
(program variables x,y,z, command **input** y)

output v?

$s \rightarrow_P s'$ iff

$$s \sqcup \pi(s') \models pc' = pc + 1 \wedge \bigwedge_{u \in V} u' = u$$

We will usually not model the monitor, harddisk etc.,
still, it is possible to model them.

Non-determinism

Further examples:

- We do not know the value of sensor inputs
- We do not know the speed of program threads
- We do not want to model a random number generator
- We do not want to model the rounding of floating-point arithmetic
- We do not have or want to ignore the source code of a library

In general: The result of the fact that the program behavior

- is not precisely known to us, or
- we do not want to model it precisely.

see also "abstraction"
(does the user behave deterministically?)

Program Termination

Command **stop**

$s \rightarrow_P s'$ iff $\perp$

Summary: Example

1:	$i \leftarrow 1$	$[pc = 1 \wedge pc' = pc + 1 \wedge i' = 1 \wedge a' = a] \vee$
2:	$a[i] \leftarrow 0$	$[pc = 2 \wedge pc' = pc + 1 \wedge a' = \mathrm{write}(a, i, 0) \wedge i' = i] \vee$
3:	$i \leftarrow i + 1$	$[pc = 3 \wedge pc' = pc + 1 \wedge i' = i + 1 \wedge a' = a] \vee$
4:	**goto** 1	$[pc = 4 \wedge pc' = 1 \wedge i' = i \wedge a' = a]$

For whole program?

demo (`program.cvc`)

Transition Relation: Summary

$s \rightarrow_P s'$ iff

$$s \sqcup \pi(s') \models \Phi_P$$

where Φ_P (*transition constraint*) is a formula of the form

$$\bigvee_{l \in \{1,\dots,l^{\max}\}} pc = l \wedge \Phi_{P,l}$$

where $\Phi_{P,l}$ is the formula corresponding to line l of the program P.

Hence: We have a predicate-logical formula
that describes single program steps.

In addition:
transition relation $\rightarrow_P$ + set of initial states: transition system.

So we can use tools from MI-TES
(temporal logic, invariant etc.)

Program Execution

A program can do an arbitrary number of steps according to $\rightarrow_P$:

$r \rightarrow_P^* r'$ iff

there is a sequence $s_1, \ldots, s_n$ s.t. $r = s_1 \rightarrow_P \cdots \rightarrow_P s_n = r'$

Example

For an arbitrary relation $\rightarrow$,

we call $\rightarrow^*$ the *reflexive-transitive closure* of the relation $\rightarrow$

If we want to exclude zero steps (i.e., $n = 1$):

transitive closure $\rightarrow^+$

$r \rightarrow_P^* r'$ does not mean that r' is the final program state

Operational Program Semantics

Semantics of program P:

relation $[\![P]\!] \subseteq S \times S$ s.t. $[\![P]\!](s, s')$ iff

- $s \to_P^* s'$,
- there is no s'', s.t. $s' \to_P s''$

Intuition: $[\![P]\!](s, s')$ iff

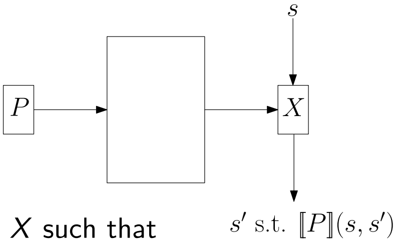
for an initial state s, the state s' is a corresponding final state

Usually, for the initial state s, $s(pc) = 1$.

But: $[\![P]\!]$ is defined for arbitrary initial states

Why operational? constraint-based variant

Usage of Semantics



A program with the specification

- Input: source code of program P,
- Output: executable machine code X such that
  for every state $s \in S$,
  the execution of X with initial state s results in s' with

$$[\![P]\!](s, s')$$

is called compiler.

A program with the specification

- Input: program P, state $s \in S$
- Output: s' such that $[\![P]\!](s, s')$

is called interpreter.

An interpreter can internally use a compiler!

Usage of Semantics

Using the definition of transition relation we precisely defined,
what an interpreter, compiler has to do.

Therefore, for an arbitrary program, on an arbitrary computer,

- a compiler writer knows exactly how to compile the program, and
- a programmer knows exactly, how the program will behave

For some programming languages this works like this (e.g., Standard ML)

Usage of Semantics

Unfortunately, for many programming languages

- there is only an informal description of their behavior, and
- the only precise definition of their behavior is a certain interpreter/compiler:

  $[\![P]\!](s, s')$ iff s' is the result of executing P for the input s
  using the given interpreter/compiler.

Problem: that many different possible behaviors as different compilers.

Sometimes a certain compiler is designated as
the compiler defining the standard behavior (*reference compiler*).

Very often only a subset of the language has a formal semantics

Alternatives

Semantics of programming language:
an area of computer science on its own.

Alternatives to our approach:

- Different variants of operational semantics
- Denotational semantics
- Axiomatic semantics (e.g., Hoare calculus)
- Algebraic semantics

Advantages of our approach:

- Operational semantics fits programming in imperative languages
- Constraints can be directly used by tools.

Literature I

Corrado Böhm and Giuseppe Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.