Aalto University
School of Science

CS-C2160 Theory of Computation

Lecture 12: Elements of Program Verification

Pekka Orponen
Aalto University
Department of Computer Science

Spring 2021

Topics:

- Programs as state transformers
  - A state-transformation semantics
  - Semantics of some basic operations
- Specification and correctness of programs
  - Weak and strong correctness
  - Proving weak correctness via loop invariants
  - A general verification approach
  - Example: Euclid's algorithm
- Establishing strong correctness

Background

- By Rice's Theorem, one cannot algorithmically decide any nontrivial input/output properties of computer programs.
- Nevertheless, one can still aim to prove that a given *individual* program has the desired behaviour, i.e. behaves correctly w.r.t. a given specification.
- More generally, one can aim to advance software design principles that guarantee that systematically developed programs are "correct by design".
- These ideas lead to the very broad field of "program verification", "program correctness" or "formal methods in software".
- In this lecture, we however only introduce the basic idea of thinking about programming in terms of *state transformations* and *correctness assertions (predicates)*, and how these ideas can be used to establish program correctness in simple cases.

Programs as State Transformers

12.1 A state-transformation semantics

- Let us consider programs (here just simple sequences of statements) written in a simple programming language with integer variables, assigments, conditional **if-then-else** statements and **while-do** loops.
- The *state* of such a program (sequence of statements) S with variables $x_1, x_2, \ldots, x_n$ comprises the current values of the variables, i.e. it is a vector $\omega = (\omega_1, \omega_2, \ldots \omega_n) \in \mathbb{Z}^n$.
- A program S then induces a semi-computable relation[1] $[\![S]\!]$ between states:

$$\omega [\![S]\!] \omega' \iff \text{executing } S \text{ in initial state } \omega \text{ halts and results in final state } \omega'.$$

- The state-transformation relation $[\![S]\!]$ can be taken as the semantical "meaning" of program S.

[1] Actually, a partially computable function in the case of deterministic programs, which is what we are considering here.

12.2 Semantics of some basic operations

- Let us then determine the state-transformation effects of various programming language constructs. This can be also viewed as defining a formal "state-transformation semantics" of our programming language.

- Assignment:

$$\omega[\![x_k \leftarrow \mathit{expr}(x)]\!]\omega', \quad \text{where } \omega'_i = \begin{cases} \omega_i & \text{for } i \neq k, \\ \mathit{expr}(\omega) & \text{for } i = k. \end{cases}$$

- Here *expr(x)* is any elementary expression containing variables in the program.[2] Thus, for instance in a program with variables x and y:

$$\langle x = 1, y = 2 \rangle [\![x \leftarrow x + 2 * y]\!] \langle x = 5, y = 2 \rangle.$$

[2] We are here slightly abusing notation by not distinguishing the *syntactic* expression on the left and its *meaning* (semantics) on the right.

- The semantics of the composite statement is obtained by simple relation composition:

$$\omega[\![S_1; S_2; \ldots; S_k]\!]\omega' \iff \omega([\![S_1]\!] \circ [\![S_2]\!] \circ \cdots \circ [\![S_k]\!])\omega'$$

- The conditional statement is similarly quite straightforward:

$$\omega[\![\textbf{if } \mathit{bool}(x) \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}]\!]\omega' \iff \begin{cases} \mathit{bool}(\omega) = \mathsf{T} \;\&\; \omega[\![S_1]\!]\omega', \text{ or} \\ \mathit{bool}(\omega) = \mathsf{F} \;\&\; \omega[\![S_2]\!]\omega'. \end{cases}$$

- Here *bool(x)* is any Boolean expression containing variables in the program.[3]

[3] We are again using the same notation for the syntactic expression $\mathit{bool}(x)$ and its meaning $\mathit{bool}(\omega)$.

- The semantics of the **while-do** -loop is more subtle:

$$\omega[\![\textbf{while } \mathit{bool}(x) \textbf{ do } S \textbf{ od}]\!]\omega'$$
$$\iff \begin{cases} \mathit{bool}(\omega) = \mathsf{F} \;\&\; \omega = \omega', \text{ or} \\ \mathit{bool}(\omega) = \mathsf{T} \;\&\; \omega[\![S; \textbf{while } \mathit{bool}(x) \textbf{ do } S \textbf{ od}]\!]\omega'. \end{cases}$$

- As an exercise, one may try to establish using these definitions e.g. the following:

$$\langle i = 1, m = 3\rangle [\![\textbf{while } i > 0 \textbf{ do } m \leftarrow 2 * m; i \leftarrow i - 1 \textbf{ od}]\!] \langle i = 0, m = 6\rangle$$

or more generally for $n \geq 0$, $m_0 \in \mathbb{Z}$:

$$\langle i = n, m = m_0\rangle$$
$$[\![\textbf{while } i > 0 \textbf{ do } m \leftarrow 2 * m; i \leftarrow i - 1 \textbf{ od}]\!]$$
$$\langle i = 0, m = m_0 * 2^n\rangle$$

Specification and Correctness of Programs

12.3 Weak and strong correctness

- In program verification, one is usually not concerned about the detailed state transformation effected by a program S, but only that S corresponds to some *specification*.
- A specification can be formulated as a pair of logical conditions or "predicates" over the program variables, a *precondition* $P = P(x)$ and a *postcondition* $Q = Q(x)$.
- A program S is *weakly correct* with respect to a specification $\langle P, Q \rangle$, denoted $\{P\}S\{Q\}$, if given an initial state ω where $P(\omega)$ holds, program S will transform it into a state ω' where $Q(\omega')$ holds, assuming S halts.
- As an example, one can easily verify the following:

$$\{x = x_0, y = y_0\}t \leftarrow x;\ x \leftarrow y;\ y \leftarrow t\{x = y_0, y = x_0\}$$

- A program S is *strongly* or *totally correct* with respect to specification $\langle P, Q \rangle$ if $\{P\}S\{Q\}$ holds, and S halts from any initial state ω satisfying $P(\omega)$.

12.4 Proving weak correctness via loop invariants

- An essential challenge in establishing the (weak) correctness of a program P is proving the correctness of the **while-do** loops.
- This can be done by means of *loop invariants*, which are also a useful way of thinking about loop design in everyday practical programming.
- A predicate I is an *invariant* for a program S if $\{I\}S\{I\}$.
- For example, one can easily verify that the predicate

$$I(m,i) : \{m * 2^i = N\},$$

for any constant $N \in \mathbb{Z}$, is an invariant for the body of the loop in our previous exponentiation program:

$$\{m * 2^i = N\} m \leftarrow 2 * m;\ i \leftarrow i - 1 \{m * 2^i = N\}$$

- If an invariant is preserved in each iteration of the body of a loop, then it is preserved throughout the whole loop. Thus, for any constant $N \in \mathbb{Z}$, the following holds:

$$\{m * 2^i = N\}$$
while *i > 0* **do** $m \leftarrow 2 * m;\ i \leftarrow i - 1$ **od**
$$\{m * 2^i = N\}$$

- We are free to set the value of the loop variable i before the loop, and we know that $i = 0$ at its end. Let us take advantage of this:

$$\{m = m_0, n \geq 0\}$$
$$i \leftarrow n;$$
$$\{m * 2^i = m_0 * 2^n\}$$
while *i > 0* **do** $m \leftarrow 2 * m;\ i \leftarrow i - 1$ **od**
$$\{m * 2^i = m_0 * 2^n \ \&\ i = 0\}$$
$$\Rightarrow \{m = m_0 * 2^n\}$$

- This establishes the correctness of our exponentiation program:

$\{m = m_0, n \geq 0\} i \leftarrow n;$ **while** *i > 0* **do** $m \leftarrow 2 * m;\ i \leftarrow i - 1$ **od** $\{m = m_0 * 2^n\}$

12.5 A general verification approach

- These ideas lead to a general approach to proving the weak correctness of any simple iterative program S such as discussed here, with respect to a specification $\{P\}S\{Q\}$:

1. Associate an appropriate invariant I to each loop in program S.
2. Verify the invariance of the predicates I.
3. Verify that according to the program flow, for each loop the respective invariant I holds when the loop is entered. The initial boundary condition is given by the program precondition P.
4. Verify that the loop invariant(s) of the outermost loop(s) guarantee that the program postcondition Q holds.

- The only items in this recipe that require insight into how the program operates are items (1) and (2), the rest are mechanical.
- These are also the items to which one should pay particular attention in practical program design.

12.6 Example: Euclid's algorithm (1/3)

- As another example, let us consider the classical Euclid's algorithm for computing the greatest common divisor $\gcd(m,n)$ of two nonnegative integers m and n. As a boundary case, we define $\gcd(m,0) = \gcd(0,m) = m$.
- For brevity, we use here the abbreviation "$m \leftrightarrow n$" for the operation of exchanging values of variables m and n, and also allow **if-then** staments without an **else** part.
- Here's the algorithm and a specification for its (weak) correctness:

$$
\begin{aligned}
&P : \{m_0 \geq 0, n_0 \geq 0\} \\
&m \leftarrow m_0;\ n \leftarrow n_0; \\
&\textbf{if } m < n \textbf{ then } m \leftrightarrow n; \\
&\textbf{while } n > 0 \textbf{ do} \\
&\quad m \leftarrow m - n; \\
&\quad \textbf{if } m < n \textbf{ then } m \leftrightarrow n; \\
&\textbf{od} \\
&Q : \{m = \gcd(m_0, n_0)\}
\end{aligned}
$$

- An appropriate invariant for the loop in Euclid's algorithm is simply that even if the values of m and n change, their gcd stays the same, i.e. the predicate "$\gcd(m,n) = \gcd(m_0,n_0)$". (Because of the way the program is written, we need to add to this the condition "$m \geq n$".)
- It is easy to validate that this predicate is preserved by the body of the loop, based on the known facts that (i) $\gcd(m,n) = \gcd(n,m)$ for all $m,n \geq 0$ and (ii) $\gcd(m-n,n) = \gcd(m,n)$ if $m \geq n$.
- In fact, for any value g the following holds:

$$
\begin{array}{l}
I : \{m \geq n \;\&\; gcd(m,n) = g\} \\
m \leftarrow m - n; \\
\{gcd(m,n) = g\} \\
\textbf{if } m < n \textbf{ then } m \leftrightarrow n \\
I : \{m \geq n \;\&\; gcd(m,n) = g\}
\end{array}
$$

- We thus get the following proof sketch for the weak correctness of Euclid's algorithm. All the steps between the indicated correctness assertions are straightforward to check, once the invariance of the loop predicate $I : \{m \geq n \;\&\; gcd(m,n) = g\}$ has been validated as we just did.

$$
\begin{array}{l}
P : \{m_0 \geq 0, n_0 \geq 0\} \\
m \leftarrow m_0;\; n \leftarrow n_0; \\
\textbf{if } m < n \textbf{ then } m \leftrightarrow n; \\
I : \{m \geq n \;\&\; gcd(m,n) = \gcd(m_0,n_0)\} \\
\textbf{while } n > 0 \textbf{ do} \\
\quad m \leftarrow m - n; \\
\quad \textbf{if } m < n \textbf{ then } m \leftrightarrow n; \\
\textbf{od} \\
I' : \{m \geq n \;\&\; gcd(m,n) = \gcd(m_0,n_0) \;\&\; n = 0\} \\
Q : \{m = \gcd(m_0,n_0)\}
\end{array}
$$

12.7 Establishing strong correctness

- The remaining part of the story is establishing the strong correctness of a program S that has already been validated as weakly correct w.r.t. a specification $\{P\}S\{Q\}$.
- This entails proving that all the loops in program S terminate, given an initial state ω that satisfies precondition $P(\omega)$.
- This is the theoretically most challenging part of the verification task, because it is essentially Turing's Halting Problem. In practice it is however seldom difficult.
- The usual approach is to determine, for each loop in the program, a *ranking function* $r(\omega)$ that maps admissible (given the precondition) states of the program to the nonnegative integers or some other well-founded ordered set,[4] and prove that in each iteration of the loop the value of $r(\omega)$ decreases.

[4] A partially-ordered set is *well-founded* if the ordering has no infinite descending sequences.

Exponentiation:

$\{n \geq 0\}$
$m \leftarrow 1;\ i \leftarrow n;$
while *i > 0* **do**
  $m \leftarrow 2 * m;$
  $i \leftarrow i - 1$
od
$\{m = 2^n\}$

In the case of the exponentiation program, one may clearly choose rank function $r(n, m, i) = i$.

Euclid's algorithm:

$\{m_0 \geq 0, n_0 \geq 0\}$
$m \leftarrow m_0;\ n \leftarrow n_0;$
if $m < n$ **then** $m \leftrightarrow n;$
while $n > 0$ **do**
  $m \leftarrow m - n;$
  if $m < n$ **then** $m \leftrightarrow n;$
od
$\{m = \gcd(m_0, n_0)\}$

For Euclid's algorithm, one can choose rank function $r(m_0, n_0, m, n) = m + n$.

Another option would be to rank-order the pairs (m, n) according to: $(m, n) \prec (m', n')$ if $m < n$ or ($m = n$ & $n < n'$).

Summary

- Programming languages can (and should) be given a precise *semantics*.
- One possibility is by giving rules for the *state transformations* effected by programs.[5]
- Given a semantics for the language, programs can be formally verified as correct with respect to a given *specification*.
- A specification can be formulated as a *pre-/post-condition pair* $\langle P, Q \rangle$, where P and Q are conditions on the program variables.
- Program S is *weakly correct* w.r.t. specification $\langle P, Q \rangle$, if precondition P holding before the execution of S guarantees postcondition Q holding after it.
- A program is *strongly correct* if in addition to weak correctness it always halts when the precondition holds.
- Key proof techniques: loop invariants, ranking functions.

[5] Cf. Turing machines as configuration-transformers.

Thanks for the Spring and success for the exam!