Verifikation nebenläufiger Programme
Wintersemester 2004/05

Ulrich Hannemann Jan Bredereke

2 Specification and Correctness Statements

2.1 Specifications, Logical Variables and Program Variables

A *specification* for a program P is given by a pair of predicates $< \varphi, \psi >$. The predicate φ is called the *precondition* for P, and ψ is called the *postcondition* for P.

Intuitively speaking a program P is correct w.r.t. a specification $< \varphi, \psi >$ if, for all initial states σ_0 such that $\models \varphi(\sigma_0)$, and for all final states $\sigma \in \mathcal{M}\,[\![P]\!]\,\sigma_0$, one has that $\models \psi(\sigma)$. Observe, since $fail, \perp \notin \Sigma$, that this definition only applies to final states $\sigma \in \Sigma$, i.e., to the case that P terminates for σ_0. So it can be alternatively formulated by:

> For all $\sigma_0, \sigma \in \Sigma$, if, for initial state σ_0, P terminates in final state σ, then $\models \varphi(\sigma_0)$ implies $\models \psi(\sigma)$.

Thus, for example, a specification for a program for extracting roots of real numbers up to a given accuracy may be $< \varphi, \psi >$, where:

$$\varphi : \Sigma \to \ Bool, \varphi(\sigma) \overset{\text{def}}{=} tt \text{ if } \sigma(y_1) \geq 0,$$

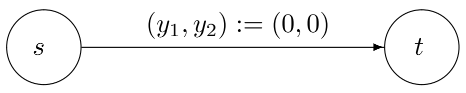
$$\psi : \Sigma \to \ Bool, \psi(\sigma) \overset{\text{def}}{=} tt \text{ if } |\sigma(y_2)^2 - \sigma(y_1)| \leq 10^{-7}$$

and Σ is instantiated as the set of mappings assigning reals to variables. In general we will make in the rest of this chapter and the following ones informal use of an assertion language to denote predicates. As an example, φ and ψ are denoted by the assertions $y_1 \geq 0$ and $|y_2^2 - y_1| \leq 10^{-7}$. Note that in our specification we freely use functions and relations specific to the domain of application, in this case the real numbers.

In this example, y_1 is the input variable and the initial value of y_2 is irrelevant. On the other hand on termination y_2 is the output variable and is expected to hold an approximation to the square root of y_1. The input predicate in this case restricts the range of inputs, for which the program is supposed to perform correctly, to nonnegative numbers. Given that a program P is correct w.r.t. $< \varphi, \psi >$, as above, then the *interpretation* of this fact (saying that P computes the square root of y_1) depends on the assumption that y_1 is not modified anywhere in the program. This assumption can be verified by examining the text of the program and ensuring that no assignment to y_1 is made.

Observe that program P' given by:

P' :



is also correct with respect to the specification above. But certainly this program *cannot be interpreted as computing in program variable y_2 the square root of the initial value of program variable y_1!*

In this example we conveniently had y_1 available at the last state in order to compare y_2^2 to it. In other cases we may assign new values to the input variables during the computation. In order to be able to relate the final state to the initial state and also to eliminate the need for independently verifying that input values are not modified, it is possible to use specifications parameterised by so-called *logical variables.* These variables are not allowed to occur in any program but are still considered to be part of the state. Thus we assume that the set of variables is partitioned into a set of program variables, namely those variables which are allowed to occur in programs, and a set of logical variables. Formally, $VAR = Pvar \cup Lvar$, where $Pvar$ denotes the set of program variables, $Lvar$ denotes the set of logical variables, and $Pvar \cap Lvar = \emptyset$. Since logical variables do not occur in any program, their values (1) are not subject to change under transitions, and, hence, (2) are not changed upon termination w.r.t. their values in the initial state. Thus they serve to remember the values of input variables in the initial state.

Consider again a program for computing the square root of a nonnegative real number. This time we assume that the program has a single program variable y that satisfies $y \geq 0$ on entry to the program, and is expected on termination to contain as output the square root of the input.

An appropriate parameterised specification in this case is given by

$$< \varphi(v, y), \psi(v, y) >$$

where:

$$\begin{array}{l} \varphi(v, y) : y = v \wedge v \geq 0, \\ \psi(v, y) : |y^2 - v| \leq 10^{-7}. \end{array}$$

Here v is assumed to be a logical variable. As was said above, logical variables do not occur in any program. But since we do not deal with any *syntactic* notion of program, we must still formulate a *semantic* criterion expressing this fact.

We express this semantic criterion in terms of the concept of the variables *involved* in a state transformation and a predicate, respectively.

Formally a function $f : \Sigma \rightarrow \Sigma$ involves the variables $\bar{x}$ if

- $\forall \sigma, \sigma' \in \Sigma.\ \sigma(\bar{x}) = \sigma'(\bar{x}) \Rightarrow f(\sigma)(\bar{x}) = f(\sigma')(\bar{x})$
- $\forall \sigma \in \Sigma, y \in Pvar \setminus \bar{x}.\ f(\sigma)(y) = \sigma(y)$.

The first condition expresses that if two states σ and σ' agree with respect to the variables $\bar{x}$, then so do their images under f. The second condition expresses that any other variable is not changed by f.

A predicate $\varphi : \Sigma \rightarrow Bool$ involves at most the variables $\bar{x}$ if

- $\forall \sigma, \sigma' \in \Sigma.\ \sigma(\bar{x}) = \sigma'(\bar{x}) \Rightarrow \varphi(\sigma) = \varphi(\sigma')$.

This condition expresses that the outcome of φ depends at most on the variables $\bar{x}$.

For $\bar{x}$ we will use the notations $f(\bar{x})$ and $\varphi(\bar{x})$ to indicate that f and φ involve the variables $\bar{x}$. Note that for any function f which involves $\bar{x}$ we also have that f involves $\bar{y}$, for any $\bar{x} \subseteq \bar{y}$ (and a similar remark applies to predicates). Moreover we have that if f involves the variables $\bar{x}$ and $\bar{y}$ then f involves $\bar{x} \cap \bar{y}$ (and similarly for predicates). This we can prove as follows: Let σ and σ' be such that $\sigma(\bar{x} \cap \bar{y}) = \sigma'(\bar{x} \cap \bar{y})$. Let $\bar{z} = \bar{y} \setminus \bar{x}$ and $\sigma'' = (\sigma : \bar{z} \mapsto \sigma'(\bar{z}))$. So we have that $\sigma(\bar{x}) = \sigma''(\bar{x})$ and $\sigma''(\bar{y}) = \sigma'(\bar{y})$. Since f involve the variables $\bar{x}$ and $\bar{y}$ it then follows that $f(\sigma)(u) = f(\sigma'')(u) = f(\sigma')(u)$, for $u \in \bar{x} \cap \bar{y}$. In other words, $f(\sigma)(\bar{x} \cap \bar{y}) = f(\sigma')(\bar{x} \cap \bar{y})$. Next let $u \notin \bar{x} \cap \bar{y}$, that is, $u \notin \bar{x}$ or $u \notin \bar{y}$. Since f involves the variables $\bar{x}$ and $\bar{y}$ it thus follows that $f(\sigma)(u) = \sigma(u)$. A similar argument applies to predicates. Although this proves that sets of involved variables of f and φ are closed under finite intersection, it does not necessarily imply that they are closed under infinite intersection.

Consequently we restrict ourselves to functions f and predicates φ for which there exists a *finite* set of variables which are involved in f and φ. Since any intersection with a finite set can be reduced to a finite intersection, the *smallest* sets of variables involved in f and φ are well-defined. From now on we will call these smallest sets the sets of variables involved in f and φ, denoted by $var(f)$ and $var(\varphi)$ respectively.

Note that $var(f)$ will contain both the so-called *read* and *write* variables of f, that is, those variables which are read by f and those variables which can be changed by f. The *read-only* variables of a state transformation f, denoted by $read(f)$, are those variables $\bar{x} \subseteq var(f)$ which are unchanged by f. Formally,

$$\forall \sigma \in \Sigma.\ f(\sigma)(\bar{x}) = \sigma(\bar{x}).$$

The write variables of f, denoted by $write(f)$, can then be formally defined as the remaining variables of $var(f)$.

For every program P we require that every state transformation f and boolean condition c of P satisfies that $var(f) \subseteq Pvar$ and $var(c) \subseteq Pvar$. This requirement then formalises the condition that logical variables do not occur in any program. We will use the phrase 'the variable x occurs in the state transformation f (condition c)' for $x \in var(f)$ ($x \in var(c)$). By $var(P)$, for a program P, we denote the variables occurring in its state transformations and boolean conditions. For programs we then use the phrase 'the variable x occurs in P' for $x \in var(P)$. Observe that this notion of occurrence of a variable in a transition system (or diagram) P is different from that in a syntactic representation of P.

Example 2.1 First consider the state transformation f expressed by $(x, y) := (x + 3, y + 7)$. Clearly f involves $\{x, y\}$.

Secondly, consider $id : \Sigma \mapsto \Sigma$, $id(\sigma) = \sigma$, describing the usual meaning of **skip**. The set of program variables involved in id is the empty set, since for all $y \in Pvar$ one has that:

$$\forall \sigma \in \Sigma.\ id(\sigma)(y) = \sigma(y).$$

Now the objective for introducing such definitions is to approximate at a semantic level the syntactic notion of (program) variables *occurring* in the syntactic definition of some construct. This example therefore demonstrates why the notion "f involves $\bar{x}$" does not fully characterise that syntactic concept. This becomes clear when considering some syntactic representation of the program P manipulating, e.g., the program variables $\{x, y, z\}$. Within such a syntactic representation of P **skip** can be characterised by the assignment $(x, y, z) := (x, y, z)$, in which the variables x, y and z occur. The meaning of **skip** continues to be id, involving the empty set of variables. $\square$

2.2 Correctness Statements

Given a precondition φ, a computation whose initial state satisfies φ is called a *φ-computation.*

The main verification questions we would ask about a program P and a specification $< \varphi, \psi >$ are the following:

- **Partial Correctness.** A program P is *partially correct* with respect to a specification $< \varphi, \psi >$ if every terminating φ-computation also terminates in a state satisfying ψ, i.e., for all σ and σ' in Σ,

  $$\text{if } \models \varphi(\sigma) \text{ and } \sigma' \in \mathcal{M}\,[\![P]\!]\,\sigma \text{ then } \models \psi(\sigma').$$

  As notation for this property, the triple $\models \{\varphi\}\ P\ \{\psi\}$ will be used.

- **Success.** A program P is successful under φ (φ-successful) if there are no failing φ-computations. That is, for all σ in Σ,

  $$\models \varphi(\sigma) \text{ implies } fail \notin \mathcal{M}\,[\![P]\!]\,\sigma.$$

- **Convergence.** A program P is *convergent* under φ (φ-convergent) if there are no divergent φ-computations. That is, for all σ in Σ,

  $$\models \varphi(\sigma) \text{ implies } \perp \notin \mathcal{M}\,[\![P]\!]\,\sigma.$$

- **Total Correctness.** A program P is *totally correct* w.r.t. a specification $< \varphi, \psi >$ if every φ-computation terminates in a state satisfying ψ. This is equivalent to P being both φ-successful, φ-convergent and partially correct w.r.t. $< \varphi, \psi >$, i.e., the following properties hold for all σ and σ' in Σ:
  - $\models \varphi(\sigma)$ implies $\{\perp, fail\} \cap \mathcal{M}\,[\![P]\!]\,\sigma = \emptyset$,
  - $\models \varphi(\sigma)$ and $\sigma' \in \mathcal{M}\,[\![P]\!]\,\sigma$ imply $\models \psi(\sigma')$.

  Notation: $\models [\varphi]\ P\ [\psi]$.

Even though it seems that total correctness is the natural concept to be proved for sequential programs, it is established in separate steps that prove first partial correctness, and then success and convergence. Since different methods are used to establish each of these properties, their introduction as independent concepts is justified.

Note that partial correctness relative to $< \varphi, \psi >$ allows divergent and failing φ-computations, but no φ-computations which terminate in a state falsifying ψ.

Partial correctness and success of a program are instances of so-called *safety* properties. The term "safety", refers to the fact that during execution of a program *"no bad things happen"*, e.g., the property that *fail* never occurs as an intermediate state in any of its computations (an indication of the fact that a successor state is always defined) is a safety property. That is, a safety property of a program refers to the fact that this program maintains some invariant property which excludes such "bad" things from happening.

Convergence of a program is an instance of a so-called *liveness* property. The term "liveness", refers to the fact that *"good things eventually happen",* e.g., in every computation of a given program a certain location (such as, e.g., its end location, or exit label) is eventually reached, or a given program terminates provided some kind of so-called *fairness property* is satisfied. Fairness properties are also examples of liveness properties.