0.1 Verifying behavioral properties

The concern with program correctness stems from a period when projects were haunted by what was called the *software crisis*. Projects delivered software that contained numerous bugs and large programs seemed to become unmanageable, that is never error-free. One of the most radical ideas proposed to counteract the software crisis was to require that programs should formally be proven correct before acceptance. The charm of the idea, I find personally, is that programming in a way becomes imbued with the flavor of mathematics, which may in itself be one of the reasons that the method never became very popular.

0.1.1 State transformers

Proving the correctness of (imperative) programs is based on the notion of *states* and the interpretation of programs as *state transformers*. A *state*, in a mathematical sense, is simply a function that records a value for each variable in the program. For example, having a program S in which the (integer) variable i occurs, and a state ϕ, we may have $\phi(i) = 3$. States may be modified by actions that result from executing the program, such as by an assignment of a value to a variable. We employ *substitutions* to modify a state. As before, substitutions may be defined by an equation, as given in slide **??**.

Substitutions *0-1*

$$\phi[x := v](y) = \begin{cases} v & \text{if } x = y \\ \phi(y) & \text{otherwise} \end{cases}$$

Slide 0-1: Substitution

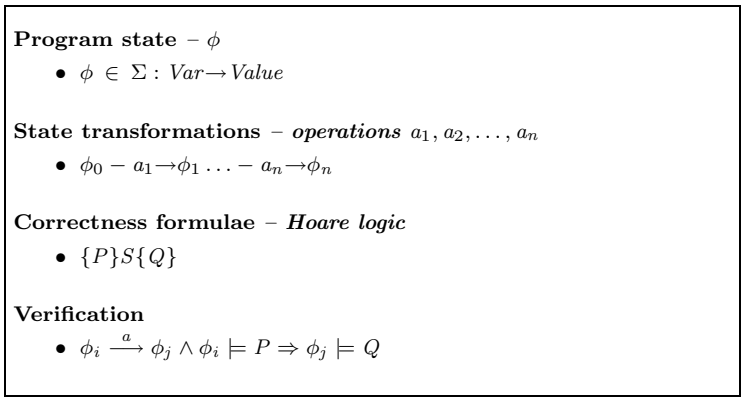
A substitution $\phi[x := v](y)$ states that modifying ϕ by assigning the value v to the variable x then, for a variable y, the state ϕ will deliver v whenever y is identical to x and $\phi(y)$ otherwise. When we have, for example, an assignment $i = 5$ then we have as the corresponding transition

$\phi - i = 5 \rightarrow \phi'$

where $\phi' = \phi[i := 5]$, that is ϕ' is like ϕ except for the variable i for which the value *5* will now be delivered.

Whenever we have a sequence of actions $a_1, \ldots, a_n$ then, starting from a state ϕ_0 we have corresponding state transformations resulting in states $\phi_1, \ldots, \phi_{n-1}$ as intermediary states and ϕ_n as the final state. Often the states ϕ_0 and ϕ_n are referred to as respectively the *input* and *output* state and the program that results in the actions $a_1, \ldots, a_n$ as the *state transformer* modifying ϕ_0 into ϕ_n.

To characterize the actions that result from executing a program, we need an operational semantics that relates the programming constructs to the dynamic behavior of a program. We will study such a semantics in section **??**.



Slide 0-2: The verification of state transformations

The requirements a program (fragment) has to meet may be expressed by using predicates characterizing certain properties of a program state. Then, all we need to do is check whether the final state of a computation satisfies these requirements.

Predicates characterizing the properties of a state before and after executing a program (fragment) may be conveniently stated by correctness formulae of the form

$\{P\}S\{Q\}$

where S denotes a program (fragment) and P and Q respectively the pre-condition and post-condition associated with S.

A formula of the form $\{P\}S\{Q\}$ is true if, for every initial state ϕ that satisfies P and for which the computation characterized by S terminates, the final state ϕ' satisfies Q. This interpretation of $\{P\}S\{Q\}$ characterizes *partial correctness*, partial since the truth of the formula is dependent on the termination of S (which may, for example, for a *while statement*, not always be guaranteed). When termination can be guaranteed, then we may use the stronger notion of *total correctness*, which makes the truth of $\{P\}S\{Q\}$ no longer dependent on the termination of S.

Pre- and post-conditions may also be used to check invariance properties. As an example, consider the following correctness formula:

$\{s = i * (i+1)/2\} i = i + 1;\ s = s + i;\ \{s = i * (i+1)/2\}$

It states that the begin and end state of the computation characterized by $i = i + 1;\ s = s + i$ is invariant with respect to the condition $s = i * (i+1)/2$. As an exercise, try to establish the correctness of this formula!

To verify whether for a particular program fragment S and (initial) state ϕ_i satisfying P the correctness formula $\{P\}S\{Q\}$ holds, we need to compute the (final) state ϕ_j and check that Q is true for ϕ_j. In general, for example in the case of non-deterministic programs, there may be multiple (final) states resulting from the execution of S. For each of these states we have to establish that it satisfies

(the post-condition) Q. We call the collection of possible computation sequences of a program fragment S the *traces* of S. Traces characterize the (operational) behavior of a program.

0.1.2 Assertion logic

Reasoning about program states based on the traces of a program may be quite cumbersome. Moreover, a disadvantage is that it relies to a great extent on our operational intuition of the effect of a program on a state. Instead, [Hoare69] has proposed using an axiomatic characterization of the correctness properties of programming constructs. An axiomatic definition allows us to prove the correctness of a program with respect to the conditions stating its requirements by applying the appropriate inference rules.

0-3

Axioms

- assignment – $\{Q[x := e]\}x = e\{Q\}$
- composition – $\{P\}S1\{R\} \wedge \{R\}S2\{Q\} \Rightarrow \{P\}S1;\ S2\{Q\}$
- conditional – $\{P \wedge b\}S\{Q\} \Rightarrow \{P\}if(b)S\{Q\}$
- iteration – $\{I \wedge b\}S\{I\} \Rightarrow \{I\}while(b)S\{I \wedge \neg b\}$

Consequence rules

- $P \rightarrow R \wedge \{R\}S\{Q\} \Rightarrow \{P\}S\{Q\}$
- $R \rightarrow Q \wedge \{P\}S\{R\} \Rightarrow \{P\}S\{Q\}$

Procedural abstraction

- $m(x) \mapsto S(x) \wedge \{P\}S(e)\{Q\} \Rightarrow \{P\}m(e)\{Q\}$

Slide 0-3: The correctness calculus

In slide **??** correctness axioms have been given for *assignment*, *sequential composition*, *conditional statements* and *iteration*. These axioms rely on the side-effect free nature of expressions in the programming language. Also, they assume convertibility between programming language expressions and the expressions used in the assertion language.

The *assignment* axiom states that for any post-condition Q we can derive the (weakest) pre-condition by substituting the value e assigned to the variable x for x in Q. This axiom is related to the weakest pre-condition calculus introduced by [Dijkstra76]. It is perhaps the most basic axiom in the correctness calculus for imperative programs. As an example, consider the assignment $x = 3$ and the requirement $\{P\}\ x = 3\ \{y = x\}$. Applying the assignment axiom we have $\{y = 3\}\ x = 3\ \{y = x\}$. Consequently, when we are able to prove that P implies $y = 3$, we have, by virtue of the first *consequence* rule, proved that $\{P\}\ x = 3\ \{x = y\}$.

The next rule, for *sequential composition*, allows us to break a program (fragment) into parts. For convenience, the correctness formulae for multiple program

fragments that are composed in sequential order are often organized as a so-called proof outline of the form $\{P\}\ S1\ \{R\}\ S2\ \{Q\}$. When sufficiently detailed, proof outlines may be regarded as a proof. For example, the proof outline

$\{s = i*(i+1)/2\}\, i = i+1;\ \{s+i = i*(i+1)/2\}\, s = s+i;\ \{s = i*(i+1)/2\}$

constitutes a proof for the invariance property discussed earlier. Clearly, the correctness formula for the two individual components can be proved by applying the assignment axiom. Using the sequential composition rule, these components can now be easily glued together.

As a third rule, we have a rule for conditional statements of the form *if*$(b)\ S$. As an example, consider the correctness formula

$\{true\}\ if(x{>}y)\ z = x;\ \{z{>}y\}$.

All we need to prove, by virtue of the inference rule for conditional statements, is that $\{x{>}y\}z = x\{z{>}y\}$ which (again) immediately follows from the assignment axiom.

As the last rule for proving correctness, we present here the inference rule for iterative (*while*) statements. The rule states that whenever we can prove that a certain invariant I is maintained when executing the body of the *while* statement (provided that the condition b is satisfied) then, when terminating the loop, we know that both I and $\neg b$ hold. As an example, the formula

$\{true\}$ `while (`i`>0) i--;` $\{i{\leqslant}0\}$

trivially follows from the *while* rule by taking I to be *true*.

Actually, the *while* rule plays a crucial role in constructing verifiable algorithms in a structured way. The central idea, advocated among others by [Gries], is to develop the algorithm around a well-chosen invariant. Several heuristics may be applied to find the proper invariant starting from the requirements expressed in the (output) predicate stating the post-condition.

In addition to the assignment axiom and the basic inference rules related to the major constructs of imperative programming languages, we may use so-called *structural* rules to facilitate the actual proof of a correctness formula. The first structural (*consequence*) rule states that we may replace a particular pre-condition for which we can prove a correctness formula (pertaining to a program fragment S) by any pre-condition of which the original pre-condition is a consequence, in other words which is stronger than the original pre-condition. Similarly, we may replace a post-condition for which we know a correctness formula to hold by any post-condition that is weaker than the original post-condition. As an example, suppose that we have proved the formula

$\{x \geqslant 0\}\ S\ \{x{<}0\}$

then we may, by simultaneously applying the two consequence rules, derive the formula

$\{x{>}0\}\ S\ \{x{\leqslant}0\}$

which amounts to strengthening the pre-condition and weakening the post-condition. The intuition justifying this derivation is that we can safely *promise less and require more*, as it were.

Finally, the rule most important to us in the present context is the inference rule characterizing correctness under *procedural abstraction*. Assuming that we have a function m with formal parameter x (for convenience we assume we have

only one parameter, but this can easily be generalized to multiple parameters), of which the (function) body consists of $S(x)$. Now, moreover, assume that we can prove for an arbitrary expression e the correctness formula $\{P\}\ S(e)\ \{Q\}$, with e substituted for the formal parameter x in both the conditions and the function body, then we also have that $\{P\}\, m(e)\{Q\}$, provided that P and Q do not contain references to local variables of the function m.

In other words, we may abstract from a complex program fragment by defining a function or procedure and use the original (local) correctness proof by properly substituting actual parameters for formal parameters. The *procedural abstraction* rule, which allows us to employ functions to perform correct operations, may be regarded as the basic construct needed to verify that an object embodies a (client/server) *contract*.