PROGRAM VERIFICATION

Types of verification that we will study:

Proof-based. We do not exhaustively check every state that the system can get in to, as one does with model checking; this would be impossible, given that program variables can have infinitely many interacting values. Instead, we construct a proof that the system satisfies the property at hand, using a proof calculus. This is analogous to the situation in Chapter 2, where using a suitable proof calculus avoided the problem of having to check infinitely many models of a set of predicate logic formulas in order to establish the validity of a sequent.

Semi-automatic. Although many of the steps involved in proving that a program satisfies its specification are mechanical, there are some steps that involve some intelligence and that cannot be carried out algorithmically by a computer. As we will see, there are often good heuristics to help the programmer complete these tasks. This contrasts with the situation of the last chapter, which was fully automatic.

Property-oriented. Just like in the previous chapter, we verify properties of a program rather than a full specification of its behaviour.

Application domain. The domain of application in this chapter is sequential transformational programs. ‘Sequential’ means that we assume the program runs on a single processor and that there are no concurrency issues. ‘Transformational’ means that the program takes an input and, after some computation, is expected to terminate with an output. For example, methods of objects in Java are often programmed in this style. This contrasts with the previous chapter which focuses on reactive systems that are not intended to terminate and that react continually with their environment.

Pre/post-development. The techniques of this chapter should be used during the coding process for small fragments of program that perform an identifiable (and hence, specifiable) task and hence should be used during the development process in order to avoid functional bugs.

Reading: Sec 4.1

Framework for software Verification

- Convert the informal description R of requirements for an application domain into an ‘equivalent’ formula ϕ_R of some symbolic logic;
- Write a program P which is meant to realise ϕ_R in the programming environment supplied by your company, or wanted by the particular customer;
- *Prove* that the program P satisfies the formula ϕ_R.

A first approximation:

Does not cover basic aspects talking to customers, evolving a specification etc

CoRE LANGUAGE

Setting is a C-like language without objects, procedures, threads, recursion/recursive data str

3 syntactic domains

- Integer Expressions
- Boolean expression
- commands

if & while statem

Grammar for integer expressions

$$E \ ::= \ n \mid x \mid (-E) \mid (E+E) \mid (E-E) \mid (E*E)$$

where n is any numeral in $\{\ldots,-2,-1,0,1,2,\ldots\}$ and x is any variable.

Convention 4.1 In the grammar above, ~~negation~~ unary minus $-$ binds more tightly than multiplication $*$, which binds more tightly than subtraction $-$ and addition $+$.

Grammar for Boolean Expressions

$$B \ ::= \ \texttt{true} \mid \texttt{false} \mid (!B) \mid (B \,\&\, B) \mid (B \,||\, B) \mid (E < E)$$

negation | disjunction — less than

conjunction

Task: How to make equality & other inequalities

Grammar for Commands

$$C \ ::= \ \texttt{x} = E \mid C;C \mid \texttt{if}\ B\ \{C\}\ \texttt{else}\ \{C\} \mid \texttt{while}\ B\ \{C\}$$

Intuitive Meaning

1. The atomic command `x` $= E$ is the usual assignment statement; it evaluates the integer expression E in the current state of the store and then overwrites the current value stored in x with the result of that evaluation.
2. The compound command $C_1; C_2$ is the sequential composition of the commands C_1 and C_2. It begins by executing C_1 in the current state of the store. If that execution terminates, then it executes C_2 in the storage state resulting from the execution of C_1. Otherwise – if the execution of C_1 does not terminate – the run of $C_1; C_2$ also does not terminate. Sequential composition is an example of a *control structure* since it implements a certain policy of flow of control in a computation.
3. Another control structure is `if` $B\ \{C_1\}$ `else` $\{C_2\}$. It first evaluates the boolean expression B in the current state of the store; if that result is true, then C_1 is executed; if B evaluated to false, then C_2 is executed.
4. The third control construct `while` $B\ \{C\}$ allows us to write statements which are executed repeatedly. Its meaning is that:

   a the boolean expression B is evaluated in the current state of the store;

   b if B evaluates to false, then the command terminates,

   c otherwise, the command C will be executed. If that execution terminates, then we resume at step (a) with a re-evaluation of B as the updated state of the store may have changed its value.

fac1 (x: input

```
y = 1;
z = 0;
while (z != x) {
    z = z + 1;
    y = y * z;
}
```

fac1 computes factorial of x & stores the result in y

Can you do this for any integer x? No!

Compute a number y whose square is less than the input x.

If the input x is a positive number, compute a number whose square is less than x.

$$(\!|\phi|\!)\ P\ (\!|\psi|\!) \tag{4.5}$$

which (roughly) means:

> If the program P is run in a state that satisfies ϕ, then the state resulting from P's execution will satisfy ψ.

The specification of the program P, to calculate a number whose square is less than x, now looks like this:

$$(\!|\underline{x > 0}|\!)\ P\ (\!|\underline{y \cdot y < x}|\!). \tag{4.6}$$

Definition 4.3 1. The form $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ of our specification is called a Hoare triple, after the computer scientist C. A. R. Hoare.

2. In (4.5), the formula ϕ is called the precondition of P and ψ is called the postcondition.
3. A store or state of core programs is a function l that assigns to each variable x an integer $l(x)$.
4. For a formula ϕ of predicate logic with function symbols $-$ (unary), $+$, $-$, and $*$ (binary); and a binary predicate symbols $<$ and $=$, we say that a state l satisfies ϕ or l is a ϕ-state – written $l \vDash \phi$ – iff $\mathcal{M} \vDash_l \phi$ from page 128 holds, where l is viewed as a look-up table and the model $\mathcal{M}$ has as set A all integers and interprets the function and predicate symbols in their standard manner.
5. For Hoare triples in (4.5), we demand that quantifiers in ϕ and ψ only bind variables that do not occur in the program P.

Example 4.4 For any state l for which $l(x) = -2$, $l(y) = 5$, and $l(z) = -1$, the relation

1. $l \vDash \neg(x + y < z)$ holds since $x + y$ evaluates to $-2 + 5 = 3$, z evaluates to $l(z) = -1$, and 3 is not strictly less than -1;

$(\!|\top|\!)\ P\ (\!|\psi|\!)$

We set precondition to T if there is no precondition

Definition 4.5 (Partial correctness) We say that the triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ is satisfied under partial correctness if, for all states which satisfy ϕ, the state resulting from P's execution satisfies the postcondition ψ, provided that P actually terminates. In this case, the relation $\vDash_{\mathsf{par}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds. We call $\vDash_{\mathsf{par}}$ the satisfaction relation for partial correctness.

$$(\!|\phi|\!)\ \underbrace{\texttt{while true \{ x = 0; \}}}_{P}\ (\!|\psi|\!)$$

Definition 4.6 (Total correctness) We say that the triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ is satisfied under total correctness if, for all states in which P is executed which satisfy the precondition ϕ, P is guaranteed to terminate and the resulting state satisfies the postcondition ψ. In this case, we say that $\vDash_{\mathsf{tot}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds and call $\vDash_{\mathsf{tot}}$ the satisfaction relation of total correctness.

EXAMPLE OF A TYPICAL SPEC

Input: x

1. Let Succ be the program

```
a = x + 1;
if (a - 1 == 0) {
    y = 1;
} else {
    y = a;
}
```

The program Succ satisfies the specification $\boxed{(\!|\top|\!)\ \mathtt{Succ}\ (\!|y = (x+1)|\!)}$ under partial and total correctness, so if we think of x as input and y as output, then Succ computes the successor function. Note that this code is far from optimal.

Definition 4.8 1. If the partial correctness of triples $\llparenthesis \phi \rrparenthesis \; P \; \llparenthesis \psi \rrparenthesis$ can be proved in the partial-correctness calculus we develop in this chapter, we say that the sequent $\vdash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis \; P \; \llparenthesis \psi \rrparenthesis$ is valid.

2. Similarly, if it can be proved in the total-correctness calculus to be developed in this chapter, we say that the sequent $\vdash_{\mathsf{tot}} \llparenthesis \phi \rrparenthesis \; P \; \llparenthesis \psi \rrparenthesis$ is valid.

Soundness & Completeness

$\vdash_{\mathsf{par}}$ is sound if

$$\vDash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis \; P \; \llparenthesis \psi \rrparenthesis \text{ holds whenever } \vdash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis \; P \; \llparenthesis \psi \rrparenthesis \text{ is valid}$$

for all ϕ, ψ and P; and, similarly, $\vdash_{\mathsf{tot}}$ is sound if

$$\vDash_{\mathsf{tot}} \llparenthesis \phi \rrparenthesis \; P \; \llparenthesis \psi \rrparenthesis \text{ holds whenever } \vdash_{\mathsf{tot}} \llparenthesis \phi \rrparenthesis \; P \; \llparenthesis \psi \rrparenthesis \text{ is valid}$$

for all ϕ, ψ and P. We say that a calculus is *complete* if it is able to prove everything that is true. Formally, $\vdash_{\mathsf{par}}$ is complete if

$$\vdash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis \; P \; \llparenthesis \psi \rrparenthesis \text{ is valid whenever } \vDash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis \; P \; \llparenthesis \psi \rrparenthesis \text{ holds}$$

for all ϕ, ψ and P; and similarly for $\vdash_{\mathsf{tot}}$ being complete.