Lecture Notes Program Verification

V2014, last upate Feb. 2016

I.S.W.B. Prasetya
Dept. of Information and Computing Sciences
Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands

email: `S.W.B.Prasetya@uu.nl`
URL: `www.cs.uu.nl\~wishnu.html`

Contents

Chapter 1

Program Semantics

A programming language is defined by its syntax, and a document describing the meaning, or semantic, of its every construct. For example each major version of Java is defined in the corresponding Java Language Specification. This document will tell us, for example, how the try...catch...finally construct in Java works. Similarly, Haskell is defined by Haskell Language Report. Documents like these serve as reference both for programmers and language implementators, so they indeed try to be as precise and as complete as possible. However, we should keep in mind that these are still documents expressed in natural languages. So some degree of ambiguity is inevitable. In contrast to this, a formal semantic, if one can be defined, would offer an unambiguous alternative. Although this may seem to be a huge benefit, it should also be pointed out that formulating a formal semantic that would completely cover the whole informal semantic of a programming language is not a small feat. It takes huge effort; many programming languages in practice do not actually have formal semantics, at least not one which is acknowledged as the standard formal semantic of the language.

Having said that, it is still useful to present a formal semantic of for example a subset of a language, and perhaps even limited to focus on certain aspects of its behavior. This is still useful for research, namely to study and gain insight on those language constructs or behavior aspects we focused on. In this chapter we will discuss several common ways to describe formal semantics: so-called 'operational semantic', 'denotational semantic', and 'axiomatic semantic'.

As the running example we will consider a simple language LAsg of assignments with scope:

$$\begin{array}{lcl}
\mathit{program} & \to & \mathit{block} \\
\mathit{block} & \to & \text{"\{"}\ \mathit{Decl}\ \text{";"}\ \mathit{statements}\ \text{"\}"} \\
\mathit{decl} & \to & \mathit{identifier}\ \text{"="}\ \mathit{expr} \\
\mathit{statements} & \to & \mathit{statement} \mid \mathit{statement}\ \text{";"}\ \mathit{statements} \\
\mathit{statement} & \to & \mathit{assignment} \mid \mathit{block} \\
\mathit{assignement} & \to & \mathit{identifier}\ \text{":="}\ \mathit{expr} \\
\mathit{expr} & \to & 0 \mid 1 \mid 2 \mid ... \\
 & \mid & \mathit{identifer} \mid \mathit{expr}\ \text{"+"}\ \mathit{expr}
\end{array} \tag{1.1}$$

A block declares a single local variable, and in its body have a sequence of statements, each is either an assignment or a block. Below are several examples:

$$\{\ x{=}10\ ;\ x := x{+}1\ \} \tag{1.2}$$

$$\{\ x{=}10\ ;\ \{\ y{=}x{+}10\ ;\ x{:=}y\}\} \tag{1.3}$$

$$\{\ x{=}10\ ;\ \{\ y{=}x{+}10\ ;\ \{x{=}y\ ; y := x{+}1\ \}\ ;\ x{:=}y\ \}\} \tag{1.4}$$

The first one will end with $x = 11$ before it leaves its top level (and only) block. The second one ends with $x = 20$ before it leaves its top level block. The third one has two nested blocks, where the innermost block declares its own x which would shadow the x from the top level block. The third example ends with $x = 21$ before it leaves its top level block.

A program is just a block. But a program also produces a result (a block does not), and let's define this result to be the final value of the single variable declared by this block.

Note: notice that the above syntax has a syntactical category named *expr*, which stands for 'expressions'. The syntax defines what an expression is, as opposed to for example a 'statement'. In the rest of the chapter we will however overload the term 'expression' to also mean any valid fragment of a program; so it can be an expression in the above sense, or a statement. This is just a matter of naming; we need a name to generically refer to program fragments, and we simply choose the term 'expression' for that.

1.1 Operational Semantics

An operational semantic defines the meaning of a program in terms of how it is executed. People further distinguish between so-called *structural* operational semantic (also called *small steps* semantic) [22] that describes each step of an execution, and *natural* operational semantic [16] (also called *big steps* semantic) that is more concerned about the end result of the execution. The concept of "execution" does not necessarily correspond to the actual execution as it happens on the hardware or some virtual machine. This depends on the purpose of the semantic. If it is intended to study how a program really executes on a virtual machine, then the corresponding concept of execution should be chosen. At such level of details, one can expect that the resulting semantic to be quite large and complex. If we only intend to conduct some study at a more abstract level, defining a more abstract concept of execution is reasonable. In our presentation here, and also when presenting examples of other types of semantics, we will restrict ourselves to abstract semantics.

1.1.1 Natural Operational Semantics

We will first take a look at natural operational semantic. The meaning of an expression e is defined by a formula of the form $\langle e, s\rangle \rightarrow m$. Here, s represents the current state of the program on which e is to be executed; m is the result of executing e. If e is an *expr* in LAsg it makes sense to define m to be a numeric value (since *LAsg* only has numeric expressions). If e is a statement, the result of its computation is a new state; so, m is this new state.

How do we represent states? Well, there are many options. It makes sense to say that the state of a program is described by the value of all its variables, that are currently in scope at that moment. You may also want to extend the concept of state with additional information. For example if you are interested in counting the number of 'steps' executed so far, you may choose to extend 'state' with a counter that is used to keep track of that information. For now, we will not add any extension.

We will represent a state as a list of pairs, each maps a variable name to a value. For example, [] denote a state that does not contain any variable (it is a valid state of a program that does not declare any variable). $[x \mapsto 0]$ is a singleton list, denoting a state that maps a variable named "x" to the value 0. Other variable names are not included, so they do not make part of the state. In other words; these other variables are undefined on the state. Similarly, $[x \mapsto 0\ ,\ y \mapsto 9]$ is a state where a variable named "x" has the value 0 and "y" has the value 9; other variable names are undefined.

We will allow a 'state' to map the same variable to multiple values, e.g. as in $[x \mapsto 0\ ,\ x \mapsto 9]$, to describe a state where we have multiple incarnations of the same variable name. For example, consider the nested block $\{x = 0\ ;\ ...\{x = 9\ ;\ ...\}\}$. The first pair $x \mapsto v$ in the state s is called *the most recent* mapping of x, and v is called the most recent value of x in s.

If x is defined on the state s, we can query its value. We denote this as $s\ x$, which returns x's most recent value in s. For example, $[x \mapsto 0\ ,\ y \mapsto 9]\ x$ and $[x \mapsto 0\ ,\ x \mapsto 9]\ x$ both yields the value 0. I will leave the formal definition of $s\ x$ to you.

1. $\langle c, s\rangle \to c$, where c is a constant like 0,1,2...
2. $\langle x, s\rangle \to s\ x$, where x is a variable name
3. $\dfrac{\langle e_1, s\rangle \to v_1 \quad , \quad \langle e_2, s\rangle \to v_2}{\langle e_1 + e_2, s\rangle \to v_1 + v_2}$

Figure 1.1: *Big steps semantic of* LAsg *expressions.*

We can now express the meaning of the expression $x + y + 1$ on the state $[x \mapsto 0\ ,\ y \mapsto 9]$:

$$\langle x + y + 1\ ,\ [x \mapsto 0\ ,\ y \mapsto 9]\rangle \ \to\ 10$$

More generally, the meaning of LAsg's *expr* can be defined by the rules shown in Figure 1.1.

With those rules, the meaning of for example $x + (y + 1)$ on the state $[x \mapsto 0\ ,\ y \mapsto 9]$ can be derived as shown in the derivation tree below:

$$\dfrac{\langle x\ ,\ [x \mapsto 0\ ,\ y \mapsto 9]\rangle \to 0 \qquad \dfrac{\langle y\ ,\ [x \mapsto 0\ ,\ y \mapsto 9]\rangle \to 9 \quad \langle 1\ ,\ [x \mapsto 0\ ,\ y \mapsto 9]\rangle \to 1}{\langle y + 1\ ,\ [x \mapsto 0\ ,\ y \mapsto 9]\rangle \to 10}}{\langle x + (y + 1)\ ,\ [x \mapsto 0\ ,\ y \mapsto 9]\rangle \ \to\ 10}$$

The meaning of statements is slightly more involved. We need few more support concepts. If a is an item, and s is a list, $a : s$ is a new list that is constructed by appending a in front of (as the head of) s. The update $s\ x\ v$ operation is used to update a state s, so that the most recent value of x in s is changed to v. The formal definition is shown below. This function assumes that x is defined in s.

$$\begin{aligned} \mathsf{update}\ []\ x\ v \quad &= \quad [] \\ \mathsf{update}\ (x' \mapsto w\ :\ s)\ x\ v \quad &= \quad \begin{cases} x \mapsto v\ :\ s & \text{, if } x = x' \\ x' \mapsto w\ :\ \mathsf{update}\ s\ x\ v & \text{, otherwise} \end{cases} \end{aligned} \tag{1.5}$$

For example, to execute the assignment $x := x{+}1$ on the state $s = [x \mapsto 1]$, we first evaluate the meaning of $x{+}1$ on the state. This yields the value 2. Then we update the state s by changing the mapping of x so that it is now mapped to the new value 2. More generally, if $\langle x{+}1, s\rangle \to v$, the assignment $x := x{+}1$ on the state s yields a new state, namely update $s\ x\ v$.

If s is a state, remove $x\ s$ will remove the most recent mapping of x from s, if x is defined on s. Otherwise it simply results in s. Note that if s contains multiple mappings of x, we only remove the most recent one (and not all of them!).

The rules in Figure 1.2 formally define the semantic of assignment, sequential composition of statements (with ";"), and blocks.

The semantic of a block is a bit more complicated. Consider the block $B = \{x = y{+}1\ ;\ S\}$. This block introduces a local variable x whose scope is limited to be within that block. The value of x in initialized by $y{+}1$. Imagine that the block is executed on s as the starting state. We first calculate the value of $y{+}1$ on this state; suppose that this yields the value v. Then, we need to extend the state s with the newly declared variable x, namely:

$$x{\mapsto}v\ :\ s$$

If s already contains a mapping for x, notice that the above extension will make it so that the newly added x will shadow existing ones. Then, S is executed on this new state. Suppose the execution of S yields some new state t. After this, we close the block, and return to the context where B was invoked. This means that the new mapping of x (above) should be removed again from the state. These all are captured by the 3rd rule in Figure 1.2.

In LAsg, a program is a block, so the semantic is similar. However, a program produces a result (and a block does not), which is the final value of the single variable declared by the block. This is captured by the 4th rule in Figure 1.2.

1. $$\frac{\langle e, s\rangle \rightarrow v}{\langle x := e\ ,\ s\rangle\ \rightarrow\ \mathsf{update}\ s\ x\ v}$$

2. $$\frac{\langle S_1, s\rangle \rightarrow s'\ \ ,\ \ \langle S_2, s'\rangle \rightarrow s''}{\langle S_1; S_2\ ,\ s\rangle\ \rightarrow\ s''}$$

3. $$\frac{\langle e, s\rangle \rightarrow v\ \ ,\ \ \langle S\ ,\ x \mapsto v : s\rangle \rightarrow t}{\langle \{x{=}e\ ;\ S\}\ ,\ s\rangle\ \rightarrow\ \mathsf{remove}\ x\ t}$$

4. Let P be a program, and let $P = \{x{=}e\ ;\ S\}$. Its semantic:

$$\frac{\langle e, []\rangle \rightarrow v\ \ ,\ \ \langle S, [x \mapsto v]\rangle \rightarrow t}{\langle P\rangle\ \rightarrow\ t\ x}$$

Figure 1.2: *Big steps semantic of* LAsg *programs and statements.*

1.1.2 Structural Operational Semantics

A structural operational semantic describes the semantic of a program in terms of how each step in its execution proceeds. In the natural operational semantic we used the notation $\langle e, s\rangle \rightarrow v$ to define the value of the expression e when it is evaluated on the state s. In the structural semantic we will use the notation $\langle e, s\rangle \Rightarrow ...$ to describe how e is executed in steps. The structures to the left and right of the $\Rightarrow$ symbol are also called *configurations*. A structural semantic can be seen as a set of rules describing how we go from one configuration to another, until the whole program is processed.

Let's also introduce an additional detail. Suppose we want to use a stack to calculate the value of an LAsg's *expr* (which is by the way also a common way to evaluate expressions in many languages). For example when x (a variable named "x") is the expression to evaluate, we first inspect its value on the current state, and then push this value to the stack. If the expression to evaluate is $e_1 + e_2$, we first evaluate e_1, and then push its resulting value to the stack. We do the same with e_2. Then, we handle the "+" by taking out two top most values from the stack (these should be the result of e_1 respectively e_2), add them, and push the result to the stack. We will now represent configurations so that they include the stack, e.g. $\langle e, \sigma, s\rangle$ where e is the expression to execute, s is the current state of the program, and σ is the current state of the stack. The rules to execute LAsg's *expr* are shown in Figure 1.3.

1. $\langle c, \sigma, s\rangle \Rightarrow \langle c{:}\sigma\ ,\ s\rangle$, where c is a constant like 0,1,2....
2. $\langle x, \sigma, s\rangle \Rightarrow \langle s\ x : \sigma\ ,\ s\rangle$, where x is a variable name.
3. $$\frac{\begin{array}{c}\langle e_1, \sigma, s\rangle \Rightarrow \langle v_1{:}\sigma\ ,\ s\rangle \\ \langle e_2, v_1{:}\sigma, s\rangle \Rightarrow \langle v_2{:}v_1{:}\sigma\ ,\ s\rangle\end{array}}{\langle e_1{+}e_2\ ,\ \sigma\ ,\ s\rangle\ \Rightarrow\ \langle v_1{+}v_2 : \sigma\ ,\ s\rangle}$$

Figure 1.3: *Small steps semantic of* LAsg *expressions.*

Aside from the added detail of using a stack, arguably the above semantic looks pretty similar to the natural operational semantic given in Figure 1.1. So, what is the difference? Admittedly the difference is in this case a bit subtle. We use derivation rules to present both semantics. For example, a derivation rule:

$$\frac{f_1\ ,\ f_2}{f_3}$$

means that statement f_3 can be derived from f_1 and f_2. Unless f_1 and f_2 are explicitly given or postulated, we will in turn have to derive f_1 and f_2 first. The notation does not however specify in which order f_1 and f_2 are to be inferred. Now, the derivation rules for $e_1 + e_2$ of both of

1. $\langle ce, \sigma, s\rangle \Rightarrow \langle e\ ,\ c{:}\sigma\ ,\ s\rangle$, where c is a constant like 0,1,2...
2. $\langle xe, \sigma, s\rangle \Rightarrow \langle e\ ,\ s\ x : \sigma\ ,\ s\rangle$, where x is a variable name
3. $\langle +e\ ,\ v_2{:}v_1{:}\sigma\ ,\ s\rangle \Rightarrow \langle e\ ,\ v_1{+}v_2 : \sigma\ ,\ s\rangle$
4. $\langle \epsilon, \sigma, s\rangle \Rightarrow \langle \sigma, s\rangle$, where ϵ stands for an empty *expr*

Figure 1.4: *Reversed Polish, small steps semantic of* LAsg *expressions.*

the natural semantic version and the structural semantic version say that we first need calculate the semantic of e_1 and that of e_2. As remarked before, the rules does not explicitly specify in what order e_1 and e_2 has to be handled. In fact, for the natural semantic version it actually does not matter. In contrast, the structural semantic version actually imposes an order: e_1 has to be executed first, because the execution of e_2 uses the stack that results from e_1. The order could have been the other way around, but the point is the structural semantic defines the meaning of an *expr* in terms of how it is executed, and the way it defines the execution requires an ordering to be defined (which also implies that the semantic defines a sequential execution, not parallel execution).

The distinction in the two styles is perhaps more visible if you later compare the semantics of statements.

Just for the purpose of demonstrating the difference in the two styles, Figure 1.4 shows an different structural semantic for *expr*. We first need to pre-process the given instance of *expr* so that it is presented in the suffix notation (also known as the reversed Polish notation [4]). For example, $x + (y + 1)$ is represented by $xy1 + +$, and $(x + y) + 1$ is represented by $xy + 1+$. This semantic shows more explicitly how a configuration involving *expr* is being transformed until it becomes empty (and thus we are done with the execution/evaluation of the *expr*).

The structural semantic of statements is shown in Figure 1.5. The semantic of assignment directly specifies the resulting state after the assignment. In contrast, block is handled by reducing it. Consider a block $\{x{=}e\ ;\ S\}$. The first step in executing this block is to evaluate e. After that, the rule for block says that we should proceed with executing S. Applying the set of rules recursively will tell us how to completely execute S, each time in a small step as we did with the block example above. More precisely, we append S to S ; del x. The last is added to explicitly close the block, namely by removing x from the current state. The 5th rule shows how to handle del.

An LAsg program is a block. However, a program is executed on an empty stack and state, and it should return the final value of of the variable declared by the block. This is shown in the last rule in Figure 1.5.

1.1.3 Proving properties of a language

Now that we have seen some examples of formal semantics, you may wonder what they are used for. Semantics are useful for proving properties about the language they describe, or about the semantics themselves. Below we will show two examples.

Example 1. We will show that our structural semantic for *expr* is equivalent with the natural semantic we gave earlier. More precisely:

$$\text{For all } \sigma\text{:}\ \ \langle e, s\rangle \to v \quad\equiv\quad \langle e, \sigma, s\rangle \Rightarrow (v{:}\sigma, s)$$

Properties over semantics are typically proven through an induction over the structure of the language in question. As an example, below is the proof of the above claim. The proof is inductive over the structure of *expr*. There is no need to induct over the structure of statements, since the claim only concerns *expr* and *expr* does not contain any statement.

1. $$\frac{\langle e,\sigma,s\rangle \Rightarrow \langle v{:}\sigma,s\rangle}{\langle x := e\ ,\ \sigma\ ,\ s\rangle\ \Rightarrow\ \langle \sigma\ ,\ \mathsf{update}\ s\ x\ v\rangle}$$
2. $$\frac{\langle S_1,\sigma,s\rangle \Rightarrow \langle \sigma,s'\rangle}{\langle S_1;S_2\ ,\ \sigma\ ,\ s\rangle\ \Rightarrow\ \langle S_2\ ,\ \sigma\ ,\ s'\rangle}$$
3. $$\frac{\langle S_1,\sigma,s\rangle \Rightarrow \langle S',\sigma,s'\rangle}{\langle S_1;S_2\ ,\ \sigma\ ,\ s\rangle\ \Rightarrow\ \langle S';S_2\ ,\ \sigma\ ,\ s'\rangle}$$
4. $$\frac{\langle e,\sigma,s\rangle \Rightarrow \langle v{:}\sigma,s\rangle}{\langle \{x{=}e\ ;\ S\}\ ,\sigma,\ \ s\}\rangle\ \Rightarrow\ \langle S\ ;\ \mathsf{del}\ x\ ,\ \sigma\ ,\ x{\mapsto}v : s\rangle}$$
5. $\langle \mathsf{del}\ x,\sigma,s\rangle \Rightarrow \langle \sigma, \mathsf{remove}\ x\ s\rangle$
6. Let P be a program and let $P = \{x{=}e\ ;\ S\}$. The semantic of P:
$$\frac{\langle e,[],[]\rangle \Rightarrow \langle [v],[]\rangle\ ,\ \langle S,[],[x\mapsto v]\rangle \Rightarrow \langle [],t\rangle}{\langle P\rangle\ \Rightarrow\ \langle t\ x\rangle}$$

Figure 1.5: *Small steps semantic of* LAsg *programs and statements.*

Proof:
We will use the structural semantic defined in Figure 1.3.

1. The base case when e is a constant c. The natural semantic gives $\langle c,s\rangle \rightarrow c$, whereas the structural semantic gives $\langle c,\sigma,s\rangle \Rightarrow \langle c{:}\sigma,s\rangle$. The equivalence obviously holds.
2. The base case when e is a variable x. The natural semantic gives $\langle x,s\rangle \rightarrow s\ x$, whereas the structural semantic gives $\langle x,\sigma,s\rangle \Rightarrow \langle s\ x : \sigma,s\rangle$. Again, the equivalence holds.
3. The induction case when e is e_1+e_2. We will show the equivalence by proving each direction of the implications.

   The natural semantic of + gives $\langle x_1{+}x_2,s\rangle \rightarrow v_1{+}v_2$, where v_1 and v_2 are obtained from $\langle x_1,s\rangle \rightarrow v_1$ and $\langle x_2,s\rangle \rightarrow v_2$. We will show that this implies $\langle x_1{+}x_2,\sigma,s\rangle \Rightarrow \langle v_1{+}v_2 : \sigma,s\rangle$. By the induction hypothesis we get: (a) $\langle x_1,\sigma,s\rangle \Rightarrow \langle v_1{:}\sigma,s\rangle$ and (b) $\langle x_2,v_1{:}\sigma,s\rangle \Rightarrow \langle v_2{:}v_1{:}\sigma,s\rangle$. Then, applying the structural semantic rule for +,see (**??**), on (a) and (b) gives $\langle x_1{+}x_2,\sigma,s\rangle \Rightarrow \langle v_1{+}v_2 : \sigma,s\rangle$. So, the claimed implication is proven.

   Now the other direction. The structural semantic of + gives $\langle x_1{+}x_2,\sigma,s\rangle \Rightarrow \langle v_1{:}v_2{::}\sigma,s\rangle$, where v_2 is obtained from $\langle x_2,v_1{:}\sigma,s\rangle \Rightarrow \langle v_2{:}v_1{:}\sigma,s\rangle$ and v_1 is obtained from $\langle x_1,\sigma,s\rangle \Rightarrow \langle v_1{:}\sigma,s\rangle$. We will show that these imply $\langle x_1{+}x_2,s\rangle \rightarrow v_1{+}v_2$. Applying the induction hypothesis gives: (a) $\langle x_2,s\rangle \rightarrow v_2$ and (b) $\langle x_1,s\rangle \rightarrow v_1$. Applying the natural semantic on + on (a) and (b) gives $\langle x_1{+}x_2,s\rangle \rightarrow v_1{+}v_2$. So, the claimed implication is proven.

   We have now proves both sides of the equivalence, so we are done for this case.

□

Example 2. This example is a bit more involved. Consider the structural semantic with the variant where the semantic of *expr* is defined via its reversed Polish translation as in Figure 1.4. We will show that we the maximum size of the stack in the structural semantic can be determined statically, namely $H{+}1$ where H is the maximum depth of the expressions that occur in the LAsg program to execute. (Actually the stack will never exceed H; but proving this is more involved; we will not do this here.)

The depth of an *expr* is defined as follows:

$$\begin{array}{lcll}
\mathsf{depth}\ c & = & 1 & \text{, where } c \text{ is a constant} \\
\mathsf{depth}\ x & = & 1 & \text{, where } x \text{ is a variable} \\
\mathsf{depth}\ (e_1{+}e_2) & = & 1 + \mathsf{max}\ (\mathsf{depth}\ e_1, \mathsf{depth}\ e_2) &
\end{array}$$

Proof:

We first observe that only the execution of *expr* changes the size of the stack. The execution rules of statements (Figure 1.5) tell us that if we execute a statement on some state s and some stack σ, the execution will end with the same stack. Since the execution of a program starts with an empty stack, every statement will thus start and ends with an empty stack as well.

There are two places where an *expr* is executed: in an assignment, and in the declaration of a local variable in a block. From the above observation it follows that the execution of an *expr* will also start with an empty stack. So, it is sufficient to show that when executing an *expr* the stack will never grow beyond $H{+}1$.

Let us first give a definition of how the reversed Polish version of an expression can be constructed:

$$\begin{array}{lcl} \mathsf{revpol}\ c & = & [c] \\ \mathsf{revpol}\ x & = & [x] \\ \mathsf{revpol}\ (e_1{+}e_2) & = & \mathsf{revpol}\ e_1 \mathbin{+\!\!+} \mathsf{revpol}\ e_2 \mathbin{+\!\!+} [+] \end{array}$$

So for example revpol $(((1 + 2) + 3) + 4)$ is 12+3+4+, whereas $1 + (2 + (3 + 4))$ translates to 1234+++. We can extend the translation so that the 'tokens' in the translation is decorated with the depth of the 'right associative nesting' that leads to it (if you view the expression as a parse tree, this will be the depth/length of the 'right tree spine' that leads to the corresponding token). The two mentioned examples are to be translated as shown below. The numbers in the superscript position are the added depth information.

$((1 + 2) + 3) + 4$	, reversed Polish:	12+3+4+
	, extended reversed Polish:	$1^0\ 2^1\ +^0\ 3^1\ +^0\ 4^1\ +^0$
$1 + (2 + (3 + 4))$	, reversed Polish:	1234+++
	, extended reversed Polish:	$1^0\ 2^1\ 3^2\ 4^3\ +^2\ +^1\ +^0$

Notice that the extended translation still produces the same sequence of tokens as the original reversed Polish; we only extend each token with some information.

To produce the new translation we modify the function revpol as shown below. It uses a worker function rv, that recurses down the structure of the target *expr*, and carries information on the length of the right spine so far:

$$\begin{array}{lcl} \mathsf{revpol}\ e & = & rv\ 0\ e \\ \mathsf{rv}\ k\ c & = & [c^k] \\ \mathsf{re}\ k\ x & = & [x^k] \\ \mathsf{rv}\ k\ (e_1{+}e_2) & = & \mathsf{rv}\ k\ e_1 \mathbin{+\!\!+} \mathsf{rv}\ (k{+}1)\ e_2 \mathbin{+\!\!+} [+^k] \end{array}$$

We will call the result of revpol e (using the extended definition of revpol) an *extended reversed Polish expression.* Below we will give several properties of such extended reversed Polish; we will not prove them. Let α, β denote instances of *expr* which are either a single constant or a single variable.

> **Lemma 1**: an extended reversed Polish expression is a sequence of length at least 1, and it starts with α^0.
>
> **Lemma 2**: if a α occurs in an an extended reversed Polish expression, it will be followed by either (a) β^{k+1} or (b) $+^{k-1}$. Case (a) is the result of a sub-expression of the form $\alpha + \beta$, or $\alpha + (\beta + ...)$. Case (b) comes from a sub-expression of the form $(... + \alpha)$
>
> **Lemma 3**: if $+^k$ occurs in an an extended reversed Polish expression, it will be followed by either (a) nothing, or (b) α^{k+1} or (c) $+^{k-1}$. In the latter two cases, $k > 0$. Case (a) is the result of the $+$ at the top level in the original expression. Case (b) comes

from a sub-expression of the form $(... + ...) + \alpha$, and (c) comes from a sub-expression of the form $... + (... + ...)$.

Lemma 4: an extended reversed Polish expression ends with a $+^k$.

Lemma 5: let $z = \mathsf{revpol}\ e$. For any subscript k that occur in z we have $k < \mathsf{depth}\ e$.

For the purpose of proof, we modify the semantic in Figure **??** so that it now uses the extended reversed Polish notation. Nothing else is changed, so essentially this is still the same semantic, except that it gets additional information.

1. $\langle c^k e, \sigma, s\rangle \Rightarrow \langle e\ ,\ c{:}\sigma\ ,\ s\rangle$
2. $\langle x^k e, \sigma, s\rangle \Rightarrow \langle e\ ,\ s\ x : \sigma\ ,\ s\rangle$
3. $\langle +^k e\ ,\ v_2{:}v_1{:}\sigma\ ,\ s\rangle \Rightarrow \langle e\ ,\ v_1{+}v_2 : \sigma\ ,\ s\rangle$
4. $\langle \epsilon, \sigma, s\rangle \Rightarrow \langle \sigma, s\rangle$

If z is an extended reversed Polish expression, its *execution sequence* is a sequence that shows how the above semantic rules are applied step by step on z, to reduce it to ϵ. For example, suppose $1^0\ 2^1\ +^0\ 3^1\ +^0\ 4^1\ +^0$ is the extended reversed Polish expression to execute. Its execution sequence is shown below:

$$\begin{array}{ll}
 & \langle 1^0\ 2^1\ +^0\ 3^1\ +^0\ 4^1\ +^0\ ,\ [], s\rangle \\
\Rightarrow & \langle 2^1\ +^0\ 3^1\ +^0\ 4^1\ +^0\ ,\ [1], s\rangle \\
\Rightarrow & \langle +^0\ 3^1\ +^0\ 4^1\ +^0\ ,\ [2,1], s\rangle \\
\Rightarrow & \langle 3^1\ +^0\ 4^1\ +^0\ ,\ [3], s\rangle \\
\Rightarrow & \langle +^0\ 4^1\ +^0\ ,\ [3,3], s\rangle \\
\Rightarrow & \langle 4^1\ +^0\ ,\ [6], s\rangle \\
\Rightarrow & \langle +^0\ ,\ [4,6], s\rangle \\
\Rightarrow & \langle \epsilon\ ,\ [10], s\rangle
\end{array}$$

Note that the initial stack in such an execution sequence is always empty (we observed earlier, that when an *expr* is executed, we start the execution with an empty stack).

Let $\langle z, [], s\rangle$ be the starting configuration in an execution sequence of an extended reversed Polish expression z. We will prove the following property:

Lemma 6: consider any configuration $\langle z', \sigma, s\rangle$ in the execution sequence of z, which is not the last configuration. Since this is not the last configuration, then z' is not empty. Its first element is either α^k or $+^k$, for some k. We will prove that in the first case $|\sigma| = k$, whereas for the second case $|\sigma| = k{+}2$.

We will prove this by induction over the execution sequence:

1. By Lemma 1, the property asserted property in Lemma 5 holds on the first configuration in the execution sequence.
2. Suppose $C = \langle \alpha^k : z', \sigma, s\rangle$ is the current configuration in the execution sequence. We will prove that Lemma 6 holds on the next configuration.

   The induction hypothesis says that it holds on the current configuration. Hence: $|\sigma| = k$.

   Applying the semantic rules on the current configuration we obtain the next configuration, namely: $D = \langle z', \alpha{:}\sigma, s\rangle$. By Lemma 2, the first element of z' must be either (a) β^{k+1} or (b) $+^{k-1}$.

   So, for case (a) the configuration D is then $\langle \beta^{k+1}{:}z'', \alpha{:}\sigma, s\rangle$ for some z''. The size of the stack in D is $k{+}1$, as it should, conforms Lemma 6, since the first symbol in the configuration is β^{k+1}. So in this case Lemma 6 indeed holds on the new configuration D.

   For case (b), the configuration D is then $\langle +^{k-1}{:}z'', \alpha{:}\sigma, s\rangle$ for some z''. Again, the size of the stack in D is $k{+}1$. This is as it should, conforms Lemma 6, since the first symbol in the configuration is $+^{k-1}$. So also in this case Lemma 6 holds on the new configuration D.

3. Suppose $C = \langle +^k : z', \sigma, s\rangle$ is the current configuration in the execution sequence. We will prove that Lemma 6 holds on the next configuration.

   The induction hypothesis says that it holds on the current configuration. Hence: $|\sigma| = k{+}2$. So, $\sigma = v_1{:}v_2{:}\sigma'$, for some values v_1 and v_2, and a stack σ'. Note that $|\sigma'| = k$.

   Applying the semantic rules on the current configuration we obtain the next configuration, namely: $D = \langle z', v_1{+}v_2 : \sigma', s\rangle$. By Lemma 3, the first element of z' must be either (a) nothing, or (b) α^{k+1} or (c) $+^{k-1}$.

   If (a) is the case, then D is the last configuration, which is excluded in Lemma 6.

   For case (b), then configuration D is then $D = \langle \alpha^{k+1} : z'' \ , \ v_1{+}v_2 : \sigma' \ , \ s\rangle$ for some z''. The size of the stack in D is $k + 1$, as it should, conforms Lemma 6, since the first symbol in the configuration is α^{k+1}. So in this case Lemma 6 indeed holds on the new configuration D.

   For case (b), then configuration D is then $D = \langle +^{k-1} : z'' \ , \ v_1{+}v_2 : \sigma' \ , \ s\rangle$ for some z''. The size of the stack this D is also $k + 1$. Again, this is as it should, conforms Lemma 6, since the first symbol in the configuration is $+^{k-1}$. So for this case, Lemma 6 also holds on the new configuration D.

The above prove Lemma 6.
Let us prove one more property:

> **Lemma 7**: let $\langle \epsilon, \sigma, s\rangle$ be the last configuration in the execution sequence of z. Let K be the depth of the *expr* from which we obtain z. Then $|\sigma| \leq K$. (Actually $|\sigma| = 1$, but this is more difficult to prove, and we don't need it.)

Lemma 7 can be proven as follows. By Lemma 4, z must ends with $+^k$ for some k. The configuration before the last one must be $\langle +^k, \sigma, s\rangle$ for some σ. By Lemma 6, $|\sigma| = k{+}2$. So, σ can be written as $\sigma = v_1{:}v_2{:} : \sigma'$ for some σ'. Note that $\sigma' = k$. Applying the semantic rule for $+$, the next (and last) configuration is $\langle \epsilon, v_1{+}v_2 : \sigma', s\rangle$. So, the size of the stack in the final configuration is $k{+}1$. By Lemma 5 we know that $k < K$, so the length of the stack in the final configuration is $k{+}1 \leq K$. Done.

Combining Lemma 6 and 7, it follows that the length of the stack at any moment during the execution of an extended reversed Polish expression that corresponds to an *expr* e will never exceed $K{+}1$, if K is the depth of e. So, if H is the depth of the deepest expression in an LAsg program, during its execution the size of the stack will never exceed $H{+}1$ either.
□

1.2 Denotational Semantics

Consider a statement. A structural semantic would describe how the statement is executed. A natural semantic describes the result of such execution. A denotational semantic tries to answer: what is a statement? Or at least it tries to give a model of what a statement is.

We have two main elements in our LAsg language: *expr* and statements. They are quite different: an *expr* does not change the program state, whereas a statement is intended change the state. For each, we will define a 'domain' with which we will express its meaning.

Since the *expr* in LAsg is limited to integer expressions, the meaning of an *expr* is in principle an integer. However, since it may also contain variables, the integer we obtain also depends on the state on which the expression is evaluated.

In Section 1.1.1 when we discussed about natural semantic we already offer an abstract and mathematical representation of states, namely: a state is represented as list of pairs, each map a variable name to a value. Let us just reuse that representation here. Let call the domain of all possible states State.

An expression can thus be seen as a function of type State $\rightarrow$ Int. In other words, we are now proposing to use the latter as the semantical domain of *expr*. Notation like $\mathcal{E}[\![e]\!]$ is commonly used

$\mathcal{E} : expr \to \mathsf{State} \to \mathsf{Int}$

1. $\mathcal{E}[\![c]\!] \;=\; (\lambda s.\ c)$, where c is a constant like $0, 1, 2....$
2. $\mathcal{E}[\![x]\!] \;=\; (\lambda s.\ s\ x)$, where x is a variable name.
3. $\mathcal{E}[\![e_1{+}e_2]\!] \;=\; (\lambda s.\ \mathcal{E}[\![e_1]\!]\ s + \mathcal{E}[\![e_2]\!]\ s)$.

Figure 1.6: *Denotational semantic of* LAsg *expressions.*

to express the denotational semantic of some construct e. The function $\mathcal{E}$ is called a *valuation function* [22], that maps syntactical constructs to some semantical domain. The denotational semantic of *expr* is shown in Figure 1.6.

For example, the semantic of the expression $x + 1$, according to the definitions in Figure 1.6 is:

$$\begin{aligned}
&\mathcal{E}[\![x{+}1]\!] \\
=\;& (\lambda s.\ \mathcal{E}[\![x]\!]\ s + \mathcal{E}[\![x]\!]\ s) \\
=\;& (\lambda s.\ (\lambda s.\ s\ x)\ s + (\lambda s.\ 1)\ s) \\
=\;& (\lambda s.\ s\ x + 1)
\end{aligned}$$

So, it is a function that maps *every* state s to the value $s\ x + 1$. This raises a question, perhaps a somewhat technical one: what if x is not defined in the state s (s does not contain any mapping for x)? We actually also have the same issue when we discussed the operational semantics of LAsg; we just ignored it to focus on other issues. But we actually can motivate why it is safe to ignore the issue. The derivation rules of our operational semantics, both the natural and structural versions, thread the current state as we move from one configuration to another. If we impose that any real LAsg program should be 'well defined' in the sense that every variable that occurs in an expression refers to a variable defined in its own block, or some other block that surrounds it (an ancestor block), it can be proven that the variable is defined in the state that is threaded in through the derivations. A similar argument can be applied here. As we will argue below, a statement can be represented as a function that maps the current state to a new state. It can be proven, and this is of course as you can expect, that such a function will produce essentially the same state as produced by our operational semantic, and therefore the conclusion is the same: in reality the above function, the denotational meaning of $x+1$, will only be applied to a state where x is defined. Therefore, there is no need to define what would happen if this is not the case.

A statement can be modeled by a function of type $\mathsf{State} \to \mathsf{State}$. Note however that such a functional semantic does not work if statements may behave non-deterministically (a function is deterministic), but this is not the case in LAsg. The semantic of statements and programs in shown in Figure 1.7.

$\mathcal{S} : statement \to \mathsf{State} \to \mathsf{State}$
$\mathcal{P} : program \to \mathsf{Int}$

1. $\mathcal{S}[\![x := e]\!] \;=\; (\lambda s.\ \mathsf{let}\ v = \mathcal{E}[\![e]\!]\ s\ \mathsf{in\ update}\ s\ x\ v)$.
2. $\mathcal{S}[\![S_1; S_2]\!] \;=\; (\lambda s.\ S_2\ (S_1\ s))$.
3. $\mathcal{S}[\![\{x{=}e\ ;\ S\}]\!] \;=\; (\lambda s.\ \begin{array}{l}\mathsf{let}\ s' = (x \mapsto \mathcal{E}[\![e]\!]\ s) : s \\ \mathsf{in} \\ \mathsf{remove}\ x\ (\mathcal{S}[\![S]\!]\ s'))\end{array}$
4. Let P be a program and let $P = \{x{=}e\ ;\ S\}$. The semantic of P:

$$\mathcal{P}[\![P]\!] \;=\; \mathsf{let}\ s = [x \mapsto \mathcal{E}[\![e]\!][\,]]\ \mathsf{in}\ \mathcal{S}[\![S]\!]\ s\ x$$

Figure 1.7: *Denotational semantic of* LAsg *statements and programs*

For example, the semantic of the statement $\{x = 1\ ; y := x + 1\}$, according to the definitions in Figure 1.7 is:

$$\begin{array}{l}
\mathcal{S}[\![\{x = 1\,; y := x+1\}]\!] \\
= \ (\lambda s.\ \mathsf{let}\ s' = (x \mapsto \mathcal{E}[\![0]\!]\ s) : s\ \mathsf{in}\ \mathsf{remove}\ x\ (\mathcal{S}[\![y := x+1]\!]\ s')) \\
= \ (\lambda s.\ \mathsf{let}\ s' = (x \mapsto 0) : s\ \mathsf{in}\ \mathsf{remove}\ x\ (\mathcal{S}[\![y := x+1]\!]\ s')) \\
= \ (\lambda s.\ \mathsf{let}\ s' = (x \mapsto 0) : s\ \mathsf{in}\ \mathsf{remove}\ x\ ((\lambda t.\ \mathsf{let}\ v = \mathcal{E}[\![x+1]\!]\ t\ \mathsf{in}\ \mathsf{update}\ t\ y\ v)\ s')) \\
= \ (\lambda s.\ \mathsf{let}\ s' = (x \mapsto 0) : s\ \mathsf{in}\ \mathsf{remove}\ x\ ((\lambda t.\ \mathsf{let}\ v = t\ x+1\ \mathsf{in}\ \mathsf{update}\ t\ y\ v)\ s')) \\
= \ (\lambda s.\ \mathsf{let}\ s' = (x \mapsto 0) : s\ \mathsf{in}\ \mathsf{remove}\ x\ ((\lambda t.\ \mathsf{update}\ t\ y\ (t\ x+1))\ s')) \\
= \ (\lambda s.\ \mathsf{let}\ s' = (x \mapsto 0) : s\ \mathsf{in}\ \mathsf{remove}\ x\ (\mathsf{update}\ s'\ y\ (s'\ x+1)))
\end{array}$$

1.3 Axiomatic Semantic

An axiomatic semantic defines the meaning of a program in terms of certain 'properties'. These properties can be anything depending on the kind of analysis that you want to make. For example it could be the program's performance, or it could be its functional correctness. Furthermore, the semantic is given abstraction in terms of 'axioms' of how a property of interest can be inferred from the program and its fragments. In addition to axioms, people also use 'inference rules'.

As an example, let's take a look at a 'performance' property, defined by the number of hypothetical CPU cycles that a program uses during its execution. We will assume that accessing memory store takes one CPU cycle. Likewise, storing a value to it takes one cycle. Executing + also takes one cycle. The set of 'axioms' below describe how we can calculate how many CPU cycles an LAsg program will use:

1. $cost(c) = 0$, where c is a constant.
2. $cost(x) = 1$, where x is a variable.
3. $cost(e_1 + e_2) = cost(e_1) + cost(e_2)$
4. $cost(S_1; S_2) = cost(S_1) + cost(S_2)$
5. $cost(x := e) = 1 + cost(e)$
6. $cost(\{x = e; S\}) = 1 + cost(e) + cost(S)$
7. Let $P = \{x = e; S\}$ be a program, then: $cost(P) = 2 + cost(e) + cost(S)$. Two cycles are added: one for storing the value of e to x, and one more to fetch the final value of x as the final result of P.

As a second example, which is a bit more involved, let's take a look at functional correctness as properties. A functional specification of a program expresses what the program is expected to achieve. For example, if it is a sorting program, then if we give it an array a to sort, then at the end we expect that a is indeed sorted. Let's say that you want to be able to prove that the program indeeds meets this specification. To be able to do this, first you will need a way to formally express the specification itself. This is not always easy. In fact, in many cases this requires much effort. Try it yourself by trying to express 'sorted' using a mathematical formula (keep in mind that the resulting a should not be just being sorted, but it should be a sorted variant of its original content).

To formally express a specification we will at least need a language to do that. The expression language of LAsg ($expr$) is a bit too poor for this purpose; for example, it does not even have Boolean expressions. Let us now introduce the language Pred for expressing state predicates. This language extends $expr$:

$$\begin{array}{lll}
pred & \to & expr = expr \mid expr > expr \\
 & \mid & \neg pred \mid pred \wedge pred \mid pred \Rightarrow pred \\
 & \mid & (\forall identifier.\ pred)
\end{array} \tag{1.6}$$

The intention is indeed that all instances of *pred* are expressions of type Boolean. An example of a *pred* is: $(\forall k.\ k > 0 \Rightarrow x + k > x)$, stating that for all non-negative k, $x + k > x$. You may notice that the formula is valid for all values of x. Such a formula is called *valid*. Not all instances of *pred* are, nor they need to be, valid.

We will express the specification of a statement in the form of a triple: $\{*\ P\ *\}\ S\ \{*\ Q\ *\}$ where S is a statement, and P, Q are instances of *pred*. Such a triple is also known as Hoare triple [13]. The meaning is as follows: if S is executed on a state that satisfies P (that is, P results in true when evaluated on this state), it will terminate on a new state that satisfies Q. An example of such a specification is shown below:

$$\{*\ x > 1\ *\}\quad x := x{-}1; x := x{-}1\quad \{*\ \neg 0 > x\ *\}$$

stating that the said statement will terminate with a non-negative x, if it is executed on a state where $x > 1$.

Below is a set of inference rules for Hoare triples. They define an axiomatic semantic for LAsg with respect to such specifications. I will now give too much explanation on them now. Later, we will spend an entire chapter, Chapter 3, on this subject.

1. Below, $Q[e/x]$ means the formula that we would get if we replace all *free* occurrences of x in Q with e. For example $x > 0[x{+}1/x]$ yields $x{+}1 > 0$. Only free variables are to be replaced. For example, in $(\forall x.\ x > 0 \Rightarrow y + x > 0)$ the variable y occurs free, but x is *not* free. So, $(\forall x.\ x > 0 \Rightarrow y + x > 0)[x{+}1/x]$ does not yield a new formula, whereas $(\forall x.\ x > 0 \Rightarrow y + x > 0)[y{+}1/y]$ produces $(\forall x.\ x > 0 \Rightarrow y + 1 + x > 0)$.

$$\frac{P \Rightarrow Q[e/x] \text{ is valid}}{\{*\ P\ *\}\ x := e\ \{*\ Q\ *\}}$$

2. $$\frac{\{*\ P\ *\}\ S_1\ \{*\ Q\ *\}\quad ,\quad \{*\ Q\ *\}\ S_2\ \{*\ R\ *\}}{\{*\ P\ *\}\ S_1; S_2\ \{*\ R\ *\}}$$

3. Below, x' and x'' are fresh variables (they do not occur in S, nor in e, and are not the same as x).

$$\frac{\{*\ P\ *\}\quad x' := e\ ;\ x'' := x\ ;\ x := x'\ ;\ S\ ;\ x := x''\quad \{*\ Q\ *\}}{\{*\ P\ *\}\ \{x = e\ ;\ S\}\ \{*\ Q\ *\}}$$

4. Let $Pr = \{x = e\ ;\ S\}$ be an LAsg program. We will introduce a special variable name return to represent the value produced by Pr. This variable name may occur in Q. Furthermore, we assume that P does not contain any free variable (no program variable is allocated when Pr starts), and Q mention no free variable except return (by the time Pr ends, any variable allocated during its execution would have been removed from the state).

$$\frac{\{*\ P\ *\}\quad x := e\ ;\ S\ ;\ \mathsf{return} := x\quad \{*\ Q\ *\}}{\{*\ P\ *\}\ Pr\ \{*\ Q\ *\}}$$

The idea is that a specification that can be inferred through the above rules are a valid specification. Unfortunately, we cannot prove that it is really valid (at least, not without the help of some form of mathematical model which we can use as a reference to define what a 'valid' Hoare triple means), which is why the semantic is called 'axiomatic': we simply postulate that inferable specifications are valid. Despite the mentioned drawback, the good thing about an axiomatic semantic is that it is typically much more abstract. For example, there is no mentioning of states nor stack anywhere in the inference rules, as opposed to the operational semantics from Section 1.1 and the denotational semantic from Section 1.2. A more abstract set of rules are in principle easier to understand and use.

Let's for example use the semantic to prove the specification $\{* \; x = 0 \; *\} \; x := x{+}1 \{* \; x > 0 \; *\}$. According to the rule for assignment, it is sufficient to prove that this formula is valid:

$$x{=}0 \;\Rightarrow\; (x > 0)[x{+}1/x]$$

If we apply the substitution $[x{+}1/x]$ the above formula reduces to $x{=}0 \Rightarrow x{+}1 > 0$. This is indeed a valid formula. Hence, the asked specification is valid.

As a second example, let us try to prove a slightly more complicated case. Prove that the following Hoare triple is valid:

$$\{* \; \mathit{true} \; *\} \; \{x = 2 \; ; \; x := x{-}1 \; ; \; x := x{-}1\} \; \{* \; \neg \; \mathsf{return}{<}0 \; *\}$$

Proof:
by the rule for block, it is sufficient to prove:

$$\{* \; \mathit{true} \; *\} \;\; x := 2 \; ; \; x := x{-}1 \; ; \; x := x{-}1 \; ; \; \mathsf{return} := x \;\; \{* \; \neg \; \mathsf{return}{<}0 \; *\} \tag{1.7}$$

Proving the validity of the above will involve using the rule for ";", which then requires us to prove the validity of each assignment in the above composition. We will work this out in reverse direction (from the last assignment, towards the first):

- By applying the rule for assignment the following can be proven to be valid. It can be proven in the same way as we did in the first example above.

  $$\{* \; \neg \; x{<}0 \; *\} \; \mathsf{return} := x \; \{* \; \neg \; \mathsf{return}{<}0 \; *\}$$

- And this one is also valid:

  $$\{* \; \neg \; x{-}1{<}0 \; *\} \; x := x{-}1 \; \{* \; \neg \; x{<}0 \; *\}$$

- And this one:

  $$\{* \; \neg \; x{-}1{-}1{<}0 \; *\} \; x := x{-}1 \; \{* \; \neg \; x{-}1{<}0 \; *\}$$

- And finally this one:

  $$\{* \; \mathit{true} \; *\} \; x := 2 \; \{* \; \neg \; x{-}1{-}1{<}0 \; *\}$$

By the rule for ";", the above four specifications imply (1.7). So we are done. $\square$

1.4 Exercises

1. Suppose we extend LAsg with a while loop. We also extend *expr* to also include expressions of the form $e_1 = e_2$; this returns true if the evaluating e_1 and e_2 returns the same integer. Else the expression returns false. Such an expression is used to guard a while loop.

   (a) Extend the natural semantic for LAsg to accommodate the new constructs.
   When defining the semantic of a composite statement, e.g. $S_1; S_2$, you will have to deal with the possibility that one of the sub-statement may not terminate. A 'partial' way to deal with this is to simply put such a case outside the scope of your semantic (in other words, you assume that both S_1 and S_2 terminate).
   In this exercise, I want you to provide a more complete semantic, that does not make such an assumption. My proposal is to base this by making the following distinction: a computation that terminates produces a new state, whereas a computation that does not terminate is treated as if it produces a 'special state' denoted by $\perp$[1].

[1] This does assume that we have a 'magical' ability to determine if a statement would terminate. In general, this is undecidable, though in our case, because the limited expressiveness of LAsg *expr*, we can actually decide it. But this is not really our concern right now.

   (b) Do the same with the denotational semantic of LAsg.

2. We extend LAsg with a S_1 handle S_2 construct. It is similar with the try-catch construct in e.g. Java. If S_1 terminates without throwing an exception, then we are done. Otherwise, the execution continues with the handler S_2. We also extend *expr* with two new forms of expressions: divisions e_1/e_2 and function calls $f\ e$ where f is a name that represent the name of an external, uninterpreted, and total function of type Int $\rightarrow$ Int. We assume this function takes exactly one argument.

   'Uninterpreted' here means that we do not care about what it does. We do know its type, and that it is a total function.

   Division by 0 throws an exception.

   (a) Extend the natural semantic for LAsg to accommodate the new constructs.
   (b) Do the same with its structural semantic.
   (c) Do the same with its denotational semantic.

3. Suppose we extend LAsg with program calls. We will alter the syntax of program to:

$$program \ \rightarrow \ \{ \ identifier \ ; \ statements \ \}$$

   So it declares a local variable, whose name stated by the identifier above, but we now do not initialize its value.

   A program call takes the form of $x := P(e)$, where P is a program, e is an *expr*, and x is a variable name. This will assign the value of e to the top most variable declared by P. After executing the body of P, the final value of this variable is then assigned to x.

   Such a call counts as a statement.

   Furthermore, we extend LAsg with if-then-else statements. We also extend *expr* to also include expressions of the form $e_1 = e_2$; this returns true if the evaluating e_1 and e_2 returns the same integer. Else the expression returns false. Such an expression is used to guard an if-then-else statement.

   (a) Extend the natural semantic for LAsg to accommodate the new constructs. Note that a program can be recursive. For simplicity, we will exclude the case of mutual recursion.

       When defining the semantic of a composite statement, e.g. $S_1; S_2$, you will have to deal with the possibility that one of the sub-statement may not terminate, e.g. when it contains a non-terminating recursion. For simplicity, here we will only 'partially' deal with this, by putting such a case outside the scope of your semantic (in other words, you assume that both S_1 and S_2 terminate).
   (b) Do the same with the denotational semantic of LAsg.

Chapter 2

Testing

2.1 Introduction

Let us abstractly view a program P as offering a set of *operations*. In Java you would call them methods. Each operation can take parameters, and when it is called it will trigger some execution of the program, hence moving it from one state to another. While it does so, the program may produce various responses, e.g. in the form of a return value, returned by the operation, or in the form of observable events.

These operations are assumed to be the only interface we have to interact with P. Therefore we can test P by testing the operations. This is done by exposing P to one or more *test suites*. A test suite is just a fancy term people use to mean a set of *test cases*. A test case is a procedure describing:

1. how to drive P through a sequence of calls to P's operations.

2. how to collect P's responses, and query P's states.

3. how to check if those collected responses and states satisfy certain correctness expectations.

Ideally, we want the test-cases to be fully 'automated'[1]. That is, they can be executed by a machine without requiring human interactions. A large test-suite that requires many user interactions is expensive and prone to human mistakes. In some situations, this may unfortunately be the only way to test a system.

In testing jargon, the target program that we test is often called *System Under the Test* or SUT. In reality, there are various kinds of programs, and therefore also various kinds of SUTs. This can greatly influence your testing approach. For example testing an OO program requires a very different approach than testing a functional program. Testing a SUT that only has one operation is different than testing one that has multiple operations. In the latter case, it also matters whether the operations can be called in any order, or whether they require specific orders.

Bug, Error, Fault, Failures

These terms are often used, including by myself, interchangebly. For the sake of completeness, let me give the definition of the terms, so that when distinction is called for, we know what we mean.

A *fault* in software is a mistake you make in the code. This mistake causes the software to move to a wrong state, which is termed *error*. This error may be observed by a user e.g. as the software gives a wrong answer, or by crashing, etc. This observed effect is called software *failure*. 'Bug' a popular term with no precise definition; we can use it if we do not really care for the distinction between the other three terms.

[1]The term automated testing is unfortunately ambigous. It may be used to refer to automation in the test execution, or to refer to automatically generating test-cases.

The important thing with those terms is that testing is aimed to find errors. But to actually fix those errors you still need to locate the faults that cause them. Locating faults is not always easy.

Test Level

People usually distinguish between unit testing, integration testing, and system testing. Generally, you can think a software to consist of components, which in turn may consist of lower level components. So you have components of various levels.

It is a good idea to test your low level components, e.g. your classes, in isolation. This is what people usually mean with *unit testing*. However, to test e.g. a class in isolation means that you would have to make some assumptions on how this class is used. In general it is hard to anticipate all possible ways this can happen, even in the context of your own application. That means, when the classes are composed to form a higher level component, new errors can be made. Therefore it is also useful to test your intermediate level components. This called *integration testing*. Finally, you would want to test the highest integration level, which is the whole software itself. This is called *system testing*.

Errors are more easily found and fixed at the unit level. If they leak out to e.g. the system testing level, figuring out where the originating faults are is usually much harder (thus much more expensive). Therefore, it makes sense to invest in extensive unit testing.

Maintenance

Wen you change your program, you may have to update your test-suites as well. Simply renaming some methods in the program can easily be reflected on its test-suites; this can even be done automatically. The same goes with simple refactoring like 'move method'. Changes on the program's semantic is however hard to reflect automatically on the test-suites, if at all possible. This can potentially trigger a lot of work.

As said, a test case checks if the SUT's states and responses satisfy certain expectations. In testing jargon, these expectations are called *oracles*. Indeed, we can then 'simply' use P's specifications as oracles. However, this requires that the specifications are expressed in a machine readable form. In practice, people often do not have informal specifications, let alone formal and machine readable ones.

Therefore, in practice oracles are often expressed in terms of concrete values, which are compared with SUT's responses or states, e.g. (in Pseudo code):

```
testcase1() {
  List s = [3,2,1] ;
  t = s.sort() ;              -- t is the SUT's reponse
  assert (t == [1,2,3])       -- [1,2,3] is our expected result --> concrete oracle
}
```

But note that concrete oracles have to be manually calculated for every test-case, and they are usually unique for each test-case. This is very labour intensive, and creates obviously a serious maintenance problem: if the SUT's logic changes, we have to re-calculate the corresponding oracles.

Of course, you can also write *partial* specifications to complement traditional test cases with concrete oracles. So that in case we are having problems in calculating the latter, we still have the partial specifications to fall back. For example, the oracle below states that the result of `s.sort()` is a non-empty list; which we can re-use on all test-cases for which `s` is non-empty.

```
nonEmpty(t) { return t<>[] } -- a partial specification stating t is non-empty

testcase2() { List s = [1]     ; assert (nonEmpty(s.sort())) }
testcase3() { List s = [3,2,1] ; assert (nonEmpty(s.sort())) }
```

Testability

Your SUT is highly testable if its operations can be easily and systematically approached from a test case, e.g. if we can simply call them. For example, a pure Java class as the SUT is highly testable. This makes programming test cases much easier.

On the other hand, a GUI application is often much less testable. Graphic-based games (e.g. Mario, CoD, Angrybird) are even worse. Clicking on menus and buttons may require the mouse to be positioned on certain positions, and the SUT's reponses may have to be interpreted from the screen's bitmaps. This makes programming a test case much more difficult. There are so-called capture-and-replay tools that allow a tester to interact directly with the SUT. The interactions are recorded and can be replayed as a test-case. Thus, the tester does not have to calculate e.g. the click-coordinates herself. However, test-cases with concrete click-coordinates are fragile. They won't work on devices with different screen sizes, and they will break when you change the layout of your GUI elements.

SUTs with low testability is painful to test. It can be mitigated by developing an additional layer that can be turned on to programatically expose the operations. But such invesment may not come cheaply.

Non-determinism

When the SUT is deterministic, every test-case will trigger the same execution, and thus the same verdict, each time you execute it. Many programs are however non-deterministic, e.g. due to concurrency or interactions with non-deterministic external components (e.g. external services). When the degree of non-determinism is very high, triggering a bug may become difficult, as it may require a very specific schedule of the operations. For the same reason, *reproducibility* also becomes a serious issue. When your test-case discovers a bug, to debug what causes it you need to be able to reproduce the execution that producuces the bug. With a non-deterministic SUT, this may take repeated runs of the test-case before you 'accidentally' manage to reproduce the execution. With a highly non-deterministic SUT, this can be very difficult.

This can be mitigated by bulding an 'overlay' that you can turn on to disable or at least reduce the amount of non-determism. But again, this entails additional investment.

2.2 Test Adequacy

Testing is inherently incomplete. That is, it does not guarantee that you have found all errors. However, it is often the only feasible option we have to guard the quality and reliability of our software. When testing, you will need some way to asses how much you have tested, and decide if it is reasonable to stop testing. So, we need some concept of test adequacy.

Suppose we define one or more 'test coverage requirements'. An example of such a requirement is that our testing (thus, executing our test suites) have to pass every line of the SUT's code. We want these requirements to be *measurable* and *quantifiable*, so that we can infer how much, e.g. in percentage, of these requirements are fulfilled. So, we could then say that our testing delivers 90% coverage. With respect to the given 'test coverage requirements', testing is adequate if it delivers sufficiently high coverage. Ideally you would want 100% coverage.

There are various coverage criteria, e.g. line coverage, statement coverage, branch coverage, state coverage etc. Importantly, you should realize that no coverage criterion can provide a complete guarantee on the correctness of an SUT. If you use a weak criterion, you can finish testing quicker, but your chance of overlooking errors is also higher. On the other hand, picking a strong criterion means that you have to test more, which also increases your cost. From the pragmatic perspective, you would have to find a combination that balances the quality you aim for, and the resources that you have.

Several commonly used coverage criteria:

- *Function coverage*

Suppose it is possible to view the SUT as a collection of 'functions'. E.g. if the SUT is a Java application, we can use its collection of methods.

The *function coverage* of testing is the percentage of the functions from that collection that got executed by the testing.

This is a very abstract concept of coverage, as it does not imply that every line in every function has been tested.

- *Line coverage*

  The percentage of the lines in the SUT source code which are visited during the testing.

- *Branch coverage*, also known as *descision coverage*.

  A branch statement is a statement like if-then-else and loops; it allows an execution to branch out to follow different paths. If we have branches in our program, ideally we should test all of them.

  The branch coverage of testing is the percentage of the branches in the SUT that are visited during the testing.

There are various tools available for measuring those coverage criteria. E.g. for Java you have Emma, Corbetura, and Clover.

Of the above criteria, branch coverage is the strongest of course. E.g. consider this example:

```
P(x,y) {
  if (even(x) && y>999) return x ;
  else                  return 0 ;
}
```

A single test case is sufficient to give 100% line coverage; but you would only get 50% branch coverage.

Branch coverage can be further strengthened by looking at the individual 'elementary conditions' that make up the guards of our branch statements. E.g. in the above example two test cases are sufficient to give 100% branch coverage, e.g.:

1 : `P(0,1000)`
2 : `P(0,0)`

However, there are actually two ways the if-then-else above is diverted to its else-branch: (1) because `y` ≤ `999`, or (2) because `x` is odd. We have not yet tested the last scenario; from this perspective the above test cases are inadequate.

The condition `even(x)` && `y`>`999` can be seen as consiting of two elementary boolean conditions: `even(x)` and `x` > `999`. Branch coverage does not automatically impose that each value of each elementary condition has been tested, e.g. as shown by the test cases below (which gives 100% branch coverage):

	`even(x)`	`y>999`	branch
1 : `P(0,1000)`	*T*	*T*	*then*
2 : `P(1,1000)`	*F*	*T*	*else*

There is a slightly stronger concept, called *condition/decision coverage* that requires: (1) all branches to be covered, and (2) both values (*T* and *F*) of every elementary condition are also covered. The above test suite does not indeed meet this criteria; but the one blow does:

	`even(x)`	`y>999`	branch
1 : `P(0,1000)`	*T*	*T*	*then*
2 : `P(1,1000)`	*F*	*F*	*else*

An even stronger variant is the *modified condition/decision coverage* (MC/DC). We first need some defintions. Let C be a condition, consisting of elementary conditions $c_1, c_2, ...$; they may be composed in any combination of Boolean operators. An elementary condition c is said to be *activated* when the other elementary conditions are set to certain values, such that changing c will also change the overall value of C. MC/DC requires that every elementary condition is activated, and tested on its both values. There might be a corner case where we have an elementary condition that cannot be activated; this condition is thus logically superflous (it can never influence C), so we do not need to cover it either. For the example program, the above two test suites do not give us full MC/DC covereage. We would need for example the following. The first table are test cases where the elementary condition `even(x)` is activated; and the table below it are test cases where the other elementary condition is activated.

	test case	`even(x)`	`y>999`	the whole "`even(x) && y>999`"	branch
1 :	`P(0,1000)`	*T*	*T*	*T*	*then*
2 :	`P(1,1000)`	*F*	*T*	*F*	*else*

	test case	`even(x)`	`y>999`	the whole "`even(x) && y>999`"	branch
3 :	`P(0,1000)`	*T*	*T*	*T*	*then*
4 :	`P(0,0)`	*T*	*F*	*F*	*else*

However, test case 1 and 3 are identical. So, only three test cases are needed, namely $\{1,3,4\}$. Note that MC/DC implies branch coverage.

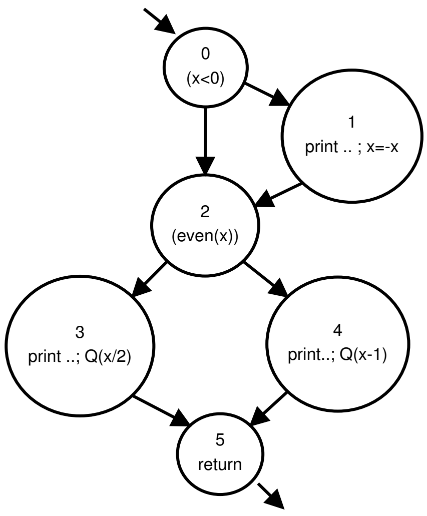
2.2.1 Path-based coverage

Consider now this program:

```
Q(x) {
  /* 0  */
  if (x<0)        { /* 1 */ print("negative!") ; x = -x ; }
  /* 2 */
  if (even(x))    { /* 3 */ print(".") ; Q(x/2) ; }
  else            { /* 4 */ print(".") ; Q(x-1) ; }
 /* 5 */
}
```

Two test cases are sufficient to give full branch coverage: `Q(−1)` and `Q(0)`. But with just these we have not explored all possible control paths through the program, which, depending on your situation, may be considered as insufficient.

Let us abstractly view a program as a so-called *control flow graph* (CFG). The CFG of the above program is shown below. As the name says, it abstractly describes how the control flows within a program.



The nodes in the graph represents a sequential *block* in `Q`. Such a block is a *maximal* segment of elementary statements in `Q` that execute sequentially[2]. Furthermore:

- No statement in the middle of the block should jump/branch out from the block.
- No statement in the middle of the block should be the target of a jump from some block. Only the start of the block can be a jump target.

A block can be quite long, but it can also be very short, consisting of just a single expression or elementary statement. The program `Q` has 6 blocks, I will number with 0..5. I have marked the start of each block in the program `Q` with a comment, as you can see in the code above.

A CFG describes how the control within `Q` flows from one block to another. So if there is an arrow from block b to c, it means that after doing b, the program *may* continue to c. If it is the only outgoing arrow from b, then it *will* proceed to c. If we have e.g. two arrows from b, going to c and d, it means that after doing b the program may branch to either d or e (our CFG's abstraction does not however tell when the program will choose one branch or the other; this is simply because we do not want to bother with it in the following discussion).

Let a CFG G be described by a tuple (N, E, S) where N is its set of nodes, and $E : N \to Pow(N)$ describes the arrows, such that $E(x)$ gives us the set of nodes to which x is connected to in the graph. $S \in N$ is the starting node: the program represented by G always starts at this node. An *exit node* is a node with no outgoing arrow. G is assumed to have at least one exit node, which is reachable from S.

A path in G is just a sequence of nodes such that consecutive nodes are connected by an arrow. A path should contain at least one node. If σ is a path, its length is defined as:

$$length(\sigma) \;=\; \#\sigma - 1 \tag{2.1}$$

So, a path of length 0 consists of one node.

A path is *maximal* if it starts at G's starting node, and ends at an exit node. For example, the program Q shown before has four maximal paths:

$$[0, 1, 2, 3, 5]$$
$$[0, 1, 2, 4, 5]$$
$$[0, 2, 3, 5]$$
$$[0, 2, 4, 5]$$

[2]Note this is how *we* here define CFG. Depending on the purpose, in the literature CFGs may be defined with higher or lower 'granularity'. E.g. for a different purpose you may want a node to represent a bytecode instruction.

From this perspective, at least four test-cases would be needed to cover all those paths, e.g.:

1 : `Q(−2)`
2 : `Q(−1)`
3 : `Q(2)`
4 : `Q(1)`

When a test case is executed, the execution will tarverse some path in the program's CFG. Let's call this *execution path.* If the execution terminates, its path is thus maximal. We define:

Definition 2.2.1 : Tour
A path τ tours another path σ if σ is a subpath of τ. That is, if:

$$\tau \ = \ t +\!\!+ \ \sigma' +\!\!+ \ u$$

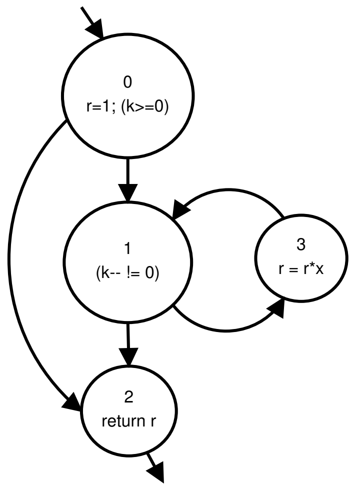
for some paths t and u.
□

Path coverage is the percentage of maximal paths through the CFG which are toured during the testing. Full path coverage obviously implies full branch coverage, but not the other way around, as have been demonstrated before.

As a side note, we can also express statement and branch coverage in terms of CFG. *Node coverage* (or 'block coverage') is the percentage of the nodes in the CFG which are visited during the testing. This corresponds to statement coverage. *Edge coverage* is the percentage of the edges (arrows) in the CFG which are passed during the testing. This corresponds to branch coverage.

Consider now this program with a loop, with its CFG:

```
power(float x, int k) {
  int r = 1 ;
  if (k>=0) {
    while (k-- != 0) r = r*x ;
  }
  return r ;
}
```



Due to the cycle in the CFG, there are now infinite number of maximal paths in the graph. Covering all of them is impossible. A reasonable alternative is to require something along the line of iterating every cycle at least once. But note that a loop may have branches inside, or another loop nested in it. To just requiring it to be iterated at least once sounds too simplistic in this situation (it does not feel adequate).

An *elementary* path in a CFG is a path where no node occurs more than once, except if it appears as the first and last one. In the latter case, the path forms a cycle. But such a cycle is also 'elementary' in the sense that it cannot contain another cycle. An elementary path is *prime* if it is not a subpath of another elementary path. The concept of 'prime path' is originally proposed by Amman and Offutt [1]. *Prime path coverage* is the percentage of the number of prime paths of the program which are toured during the testing.

For example, all paths (4) in the program `Q` are also prime paths. On the other hand, the program `power` has infinite number of paths, but it has only 6 prime paths:

$[0,2]$, $[0,1,2]$, $[0,1,3]$
$[1,3,1]$
$[3,1,3]$, $[3,1,2]$

The path $[0,1,2]$ corresponds to the execution where the loop immediately terminates (it does 0 iteration). The cycles $[1,3,1]$ and $[3,1,3]$ represent a whole class of executions where the loop does at least one iteration.

Note that an execution σ that tours one of the cycles ($[1,3,1]$ or $[3,1,3]$) will not tour $[0,1,2]$ (the 0-iteration case), since the latter cannot be a subpath of such a σ. So, $[0,1,2]$ really represents a different kind of executions.

However, arguably the two cycles $[1,3,1]$, and $[3,1,3]$ actually represent the same cycle, but the definition of prime path coverage treats them as different. Touring $[1,3,1]$ would require an execution that loops at least once, whereas touring $[3,1,3]$ requires at least two iterations. But then we can point out that any execution that tours the latter also tours the first. In fact, it will also tour $[3,1,2]$. Alternatively we can require to only cover a smallest set of prime paths instead:

Definition 2.2.2 : Smallest Set of Prime Paths
Let J be the set of all prime paths in G. A subset $I \subseteq J$ is a smallest set of prime paths if for any path $\tau_1 \in J/I$, there exists $\tau_2 \in I$ such that every execution path that tours τ_2 will also tour τ_1.
□

So, such a set can be ontained by dropping any prime path τ_1 if there is another prime path τ_2 such that any execution path that tours τ_2 will also tours τ_1[3].

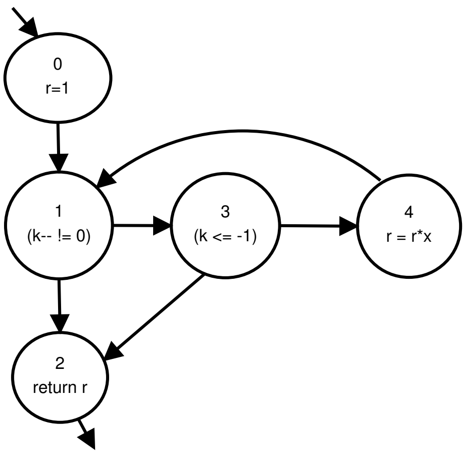
A smallest set of prime paths of `power` is:

$[0,2]$, $[0,1,2]$, $[3,1,3]$

At least three test-cases are needed to give full coverage on this set.

Let us now consider a more complicated loop. Cosider this slight variation of the program `power`:

```
powerz(float x, int k) {
  int r = 1 ;
  while (k-- != 0) {
    if (k<=-1) break ;
    r = r*x ;
  }
  return r ;
}
```



A smallest set of prime paths of `powerz` is:

$[0,1,2]$
$[0,1,3,2]$
$[4,1,3,4]$
$[4,1,3,2]$
$[4,1,2]$

[3]But an objection to this alternative is if τ_2 turns out to be unfeasible. But in practice, it is not so likely that we have a loop that always terminate after just one iteration.

Five test-cases can fully cover them (four can also do the job; less than four is not possible).

The first path, $[0, 1, 2]$, can only be toured by an execution that does not enter the loop. The second, $[0, 1, 3, 2]$ can only be toured by an execution that enters the loop, and then immediately breaks.

The cycle $[4, 1, 3, 4]$ can be toured by any execution that iterates at least twice. The last two require that we should have one execution that iterates then terminates normally, and another that iterates and then terminates through the break. One of these can also cover $[4, 1, 3, 4]$, by iterating twice.

Unfeasible paths and detour

Some paths may actually be impossible to cover by real executions. E.g. if we require a pre-condition `k /=0` on the program `powerz` shown before, then it is impossible to cover the path $[0, 1, 2]$. Such a path is called *unfeasible*.

Let us first consider two weaker variants of 'tour'. Let σ be a path. The induced edges-sequence of σ is:

$$[(\sigma_i, \sigma_{i+1}) \mid 0 \leq i < N]$$

where N is the path's length (which is the length of the list σ minus 1).

Definition 2.2.3 : Detour
An execution path τ is said to *detour* a path σ, if σ is a sub-sequence of τ. That is, σ can be obtained by deleting some elements of τ.
□

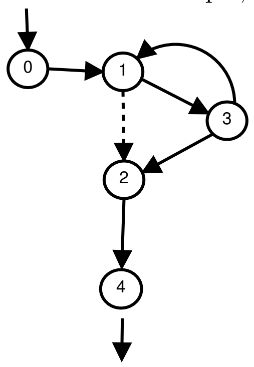
Definition 2.2.4 : Sidewalk
Let σ, τ be two paths, and s, t are their respective edge-sequences. The path τ is said to *sidewalk* σ, if s is a sub-sequence of t.
□

For example, the execution path $[0, 1, 3, 4, 1, 2]$ in `powerz` (it does one iteration, then terminates) does not tour $[0, 1, 2]$, but it still sidewalks the latter.

Of course, if τ tours σ then it also sidewalks and detours σ. Sidewalking is stronger than detouring: if τ sidewalks σ then it also detours σ. The reverse directions are not necessarily true.

If *powerz* turns out to require $k > 0$ as pre-condition, then the path $[0, 1, 3, 2]$ is infeasible. Thus, no actual execution path in its CFG can tour it. Actually, no actual execution path can even sidewalk it. However, the path $[0, 1, 3, 4, 1, 2]$, which is feasible, can still detour it ($[0, 1, 3, 2]$).

As another example, consider the CFG below:



Suppose that the cycle, if entered, it must be iterated at least once. Then the prime path $[0, 1, 3, 2, 4]$ cannot be toured. However, it can be sidewalked by the path $[0, 1, 3, 1, 3, 2, 4]$.

If the dashed edge $(1, 2)$ is unfeasible, then the prime path $[0, 1, 2, 4]$ cannot be toured, neither can it be sidewalked. However, the previous execution $[0, 1, 3, 1, 3, 2, 4]$ detours $[0, 1, 2, 4]$.

In general, when we have difficulty to tour a certain path σ, it could be (though we may not know for sure) that it is because the path is actually unfeasible. We can then at least check if it can be covered if we weaken our definition of touring (so, either sidewalking or detouring). Of course, we are still unable to tour σ, but in my oppinion knowing that it can still be sidewalked or detoured is still useful information.

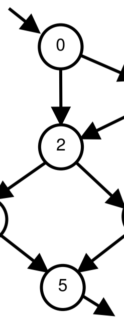
Unrelated with the feasibility issue, prime path coverage by detour is strictly more powerful than node coverage, and prime path coverage by sidewalking is strictly more powerful than edge coverage. Consider again the above example.

- A single execution $[0, 1, 3, 2, 4]$ is enough to give a full node coverage, but it will not detour the prime path $[3, 1, 3]$.
- Two executions paths $[0, 1, 2, 4]$ (no iteration) and $[0, 1, 3, 1, 3, 2, 4]$ (one iteration) gives a full branch coverage. However, the prime-path $[3, 1, 2, 4]$ are not side-walked by either of those execution paths.

2.2.2 Linearly Independent Paths

The number of prime paths is exponential. So, if you have a program with e.g. 10 if-then-else in a series, you'll have a lot of work to do. In practice such a program does not occur very often; in which case prime path coverage is not that bad.

An often suggested alternative is to use linearly independent paths. A set Σ of paths are linearly independent to each other if every path $\sigma \in \Sigma$ has at least one unique edge that is not present in all the other paths in Σ. For example, consider this CFG:



The paths $[0, 2, 3, 5]$ and $[0, 1, 2, 4, 5]$ are linearly independent to each other. Two test cases are sufficient to cover them. But it is not the only such set of paths, e.g. these paths:

$$[0, 2, 3, 5],\ [0, 1, 2, 3, 5],\ [0, 2, 4, 5]$$

also from a set of linearly independent paths. For this set, three test cases would be needed. You already know the popular McCabe complexity number. This number gives an upper bound to the size of the set of linearly independent paths in an CFG.

McCabe complexity number is $e - n + 2$, where e is the number of edges in the CFG, and n is the number of its nodes. It is much less than the number of prime paths.

2.3 White box and black box testing

There is nothing special about writing test cases. They are just programs. E.g. a test suite for the program `power` could look like this:

```
power_test() {
   assert power(2,0)==1 ;
   assert power(2,1)==2 ;
```

```
        assert power(2,3)==8 ;
        assert power(2,-1)==1 ;
    }
```

But how do we come up with these tests in the first place? Remember that you would want your tests to deliver full coverage. The obvious source of information is the source code of your SUT. If that is how you infer your test cases then we call your testing *white box testing.*

However, source code is not always available. And even if it is, you may deliberately decide not to look at it, e.g. because it would flood you with too many details. Your testing is then called *black box testing.*

2.3.1 Classification tree for black box testing

A practical approach you can use for black box testing is by using classification trees [14]. Consider a hypothetical program to register grades for students, with the following header:

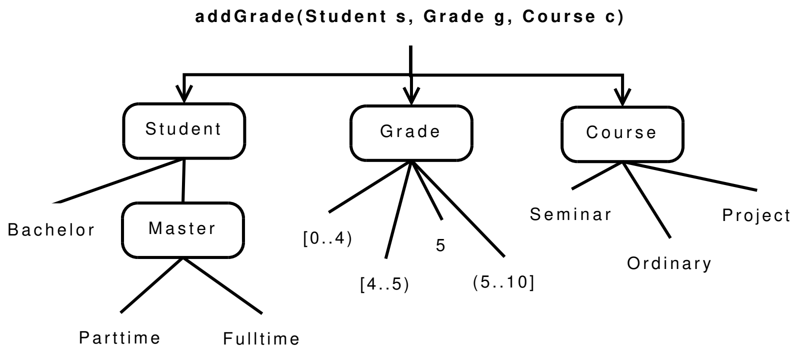
```
    addGrade(Student s, Grade g, Course c)
```

We say that this program has three input domains: that of students, grades, and courses. By e.g. the 'domain' of students we simply mean the set of all possible students which are valid as inputs for this program.

Rather than testing this program with arbitrarty combinations of students, grades, etc, we try to identify logical partitions of each domain into subdomains. For example, it may make sense to distinguish between bacelor and master students.

Remember that you cannot, or do not want to, look into the source code to come up with a sensical partitioning. So, you have to rely on other sources, e.g. specifications or software documentation.

The partitioning can be carried out recursively as needed. For example, we could partition the domains of `addGrade` as shown in the tree below:

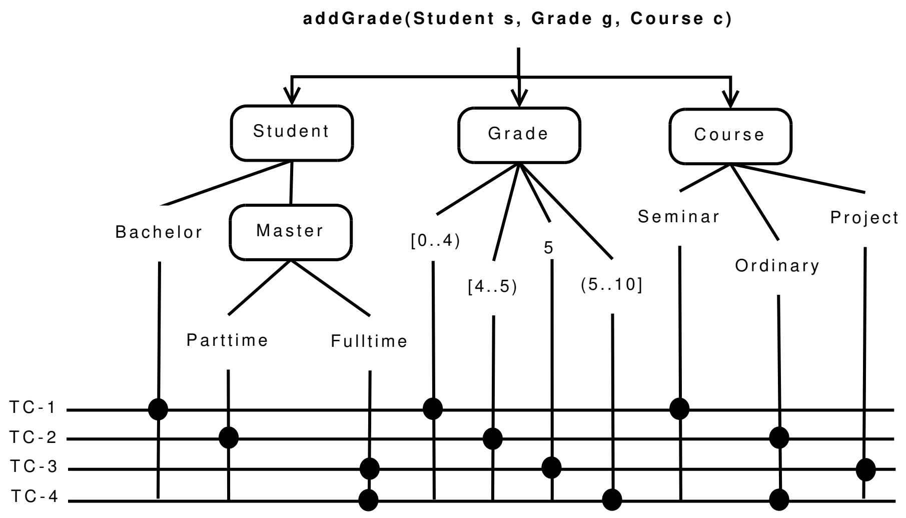


The tree above is called *classification tree* (CT). The boxes and the leaves represent domains or subdomains. In CT jargon they are called 'classes'. The top level classes are called *categories* (above: *Student*, *Grade*, and *Course*), each representing a parameter of `addGrade`.

For each leaf-class you would need to supply at least one value to be used as an input for `addGrade`. So, you would need to supply at least one bachelor student, one parttime master student, one fulltime master student, etc.

Obviously, by combining supplied inputs from different leaf-classes you can produce various test cases. You can generate all combinations to give full combinatorial CT coverage (`addGrade` has in total 36 combinations). But since the total number of combinations explodes fast if you have a large classification tree, you may opt to just generate combinations e.g. such that every class would be tested at least once to give full CT class coverage, or such that every pair of leaf-classes are tested at least once.

There is also a tool called Classification Tree Editor (CTE) from `systematic-testing.com` that allows you to generate such sets of combinations of leaf-classes, or to specify a more advanced set by imposing constraints on the combinations you want, or to even edit them manually if need to. Desinging test cases using CTE looks more or less like the picture below:



Each horizontal like describe a single test case. The black circles on the line specify the input combinations the test case take. E.g. the first test case take a bachelor student, a grade in the range [0..4), and a seminar as inputs for `addGrade`.

The picture above would then describe a set of four test cases. By the way, this suite would give you full TC class coverage (but not full combinatorial coverage, of course).

2.4 Regression

A software's life does not ends at its delivery. Most software is so complicated that it is not feasible to deliver it bug free right on. So, after delivery maintenance continues to fix bugs found afterwards. This means modifications are introduced. Furthermore, users may request changes in features or functionalities, or new ones to be added. This again triggers modifications in the software.

Supposed you have introduced a set of modifications. We would want to test that at least they do not break the functionalities that are supposed to be preserved. A pragmatic way of doing this is by re-running the test suite of the previous version. This is called *regression testing.* If an error is found, this does not necessarily mean that it is a bug in the new version. It could also be that it is part of the intended behavior of the new version which was not accepted or even present in the old version. But at least, it warns us that something needs to be investigated. Also, when no error is found in regression testing, we should keep in mind that any new feature would not be covered in such a test. So, additional test-cases may be needed.

When the SUT is large, its test-suite T is typically very large. Re-running the whole can take hours, or even days. Since modifications are typically limitted to just some places in the SUT, it is likely that most test-cases in T are actually irrelevant for testing the modifications. But how can we figure out which of them are should be included (without having to first execute the whole T to find that out)? Unfortunately, in general this is an undecidable problem due to the halting problem it implies. So, we will consider a slightly different problem instead.

2.4.1 Test cases selection

Let's first define some concepts. For simplicity, let's assume the SUT to be deterministic. A test-case t is asbtractly described by a pair (i, o) where i is the input (or inputs) for the SUT, and o is the resulting result of the test-case, which is either success or fail. For simplicity, we assume that crash and non-termination to be detectable, and result in fail.

Let P be the SUT, and T be test suite we used to test it. We assume that P has passed T. That is, none of the test-case in T failed. Let P' be the new version of P after some modifications. Some of the test-cases in T may become *obsolete* for testing P', e.g. they become broken, or are inconsistent with the intended behavior of P'. In practice, identifying the latter case is hard, so it is not done. Let us here just assume that it is possible to identify obsolete test cases, and that they have been removed from T.

What we want is to select a subset $S \subseteq T$ (without re-executing T to do so) to do regression testing on P'. With a smaller S we hope to reduce the test execution time[4]. But we should also include the cost of the selection itself. The total time we spend on selecting S and executing it should not exceed the time needed to simply re-execute the whole T.

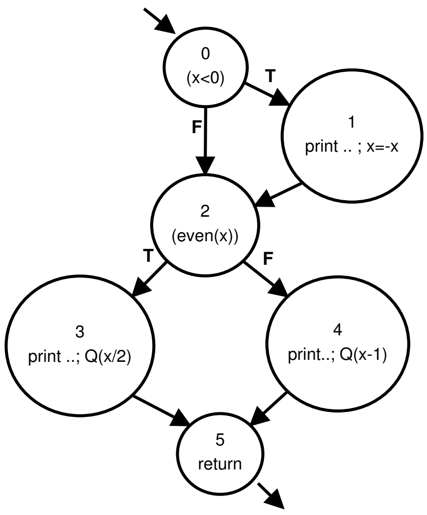
A test-case $t \in T$ is *fault revealing* (with respect to P') if it fails on P'. A selection algorithm is *safe* if the resulting S is guaranteed to include all fault revealing test-cases from T. It is *minimal* if it only contains fault revealing test-cases. Finding the minimal S is in general, again, an undecidable problem.

Suppose we can instrument the SUT, so that when we execute a test-case we knows which 'locations' in the SUT were visited by the execution. So, for each test-case in T, we have this data when we used it to test P. For a test-case t to be fault revealing when used to to test P', it must pass a *modified part.* This is a part which either new in P', or it was in P but deleted in P', or it is present in both but is at least textually different in both version (it has been modified in P'). A test-case that does not pass any modified part will have exactly the same result in P', and thus cannot be fault revealing.

A test-case that passes a modified part, is called *modification revealing.* But note that when the code is textually modified, the semantic can still be the same. So, a modification revealing test-case is not necessarily fault revealing. In other words, the first concept is weaker than the latter.

Selecting all modification revealing test-cases is safe, but this is less trivial than it may seem due to deletion and insertion of new statements. Let's first introduce some concepts.

We will first tweak our representation of CFG a bit. A node that has multiple outgoing arrows represents a descision node. If we first desugar conditional statements with multiple alternatives like `case` to cascades of `if-then-else`, all decision nodes will then only have two outgoing arrows. We will label them with `T` and `F`, to mark the 'then' branch and the 'else' branch. Arrows from non-decision nodes have empty label ϵ. So, for example:



[4]For simplicity, here we will assume that all test-cases take about the same amount of time to execute. Therefore, a smaller subset of T would execute faster. In practice, the assumption may not be true. Suppose we have two candidate-subsets of T, namely S_1 and S_2, such that S_1/S_2 and S_2/S_1 are both non-empty. Even if $|S_1| < |S_2|$, it does not imply that S_1 would therefore execute faster than S_2. Though, both would execute faster than T.

```
select($G_1, G_2, T$) {
   modified := $\emptyset$
   visited  := $\emptyset$
   compare($\mathsf{root}(G_1), \mathsf{root}(G_2)$)
   **return** $\{t \mid e{\in}$modified$,\ \ t{\in}T,\ \ t$ passes $e\}$

   **where**
   compare($n_1, n_2$) {
      visited := visited $\cup$ $\{(n_1, n_2)\}$
      **for**($o_1{\in}\mathsf{next}(G_1, n_1),\quad o_2{\in}\mathsf{next}(G_2, n_2)$) {
         **if** ($label(n_1{\to}o_1){=}label(n_2{\to}o_2)$)
            **if** ($(o_1, o_2){\notin}$visited)
               **if** ($n_1{\not\equiv}n_2$)
                  modified := modified $\cup$ $\{(n_1, n_2)\}$
               **else**
                  compare($o_1, o_2$)
      }
   }
}
```

Figure 2.1: *Rothermel and Harrold selection algorithm.*

Two nodes u, v are syntactically equivalent, denoted by $u \equiv v$, if the statements they represent are textually the same.

Let G_1 and G_2 be the CFGs of P and P'. Consider a test-case t. Suppose when executed on P and P' it traverses the path τ_1 respectively τ_2. If t is modification revealing, these paths must pass some nodes that are syntactically different. In particular, there must a pair of nodes $v_1 \in G_1$ and $v_2 \in G_2$ which the traversed paths differ for the first time from each other. This implies that τ_1 and τ_2 must be of this form:

$$\tau_1 \;=\; \sigma \mathbin{+\!\!+} [v_1] \mathbin{+\!\!+} v_1$$

$$\tau_2 \;=\; \sigma \mathbin{+\!\!+} [v_2] \mathbin{+\!\!+} v_2$$

where $v_1 \not\equiv v_2$, and σ is a common prefix of τ_1 and τ_2. Furthermore, branch-decissions made along σ and its next node must have been identical in both executions. That is, the list of labels over the edges in $\sigma \mathbin{+\!\!+} [v_1]$ (in G_1) and the list of labels over the edges in $\sigma \mathbin{+\!\!+} [v_2]$ (in G_2) must be the same.

So, what we can do is to quantify over all paths $\tau_1 \in G_1$, and to find the pair (v_1, v_2) as above, which is the first pair of nodes where it would differ than its counterpart $\tau_2 \in G_2$. Although there may be infinitely many paths in G_1 and G_2, there will only be finite number of such (v_1, v_2).

Figure 2.1 shows an algorithm by Rothermel and Harrold [21][5] It takes as inputs the CFG G_1 of the program P, the CFG G_2 of the modified program P', and the test suite T of P. The roots are assumed to be identical (else add fake roots which are not allowed to be modified). The algorithm returns a subset of T consisting of all modification revealing test-cases.

[5]It is not literally the same algorithm. I change it a but to improve the presentation; but it should achieve the same.

Chapter 3

Hoare Logic and Predicate Transformer

Hoare logic is a simple logic to prove the correctness of sequential imperative programs. It often forms the basis for other logics for imperative programs. In this logic, programs are specified by so-called *Hoare triples* , e.g.:

$$\{*\ \#\mathtt{S} > \mathtt{0}\ *\}\quad \mathtt{getMax(S)}\quad \{*\ \mathtt{return}{\in}\mathtt{S} \wedge (\forall \mathtt{x} : \mathtt{x}{\in}\mathtt{S} : \mathtt{return}{\geq}\mathtt{x})\ *\}$$

Generally a specification takes the form $\{*\ P\ *\}\ S\ \{*\ Q\ *\}$ where S is the program being specified, and P, Q are predicates over the states of S. P is also called pre-condition, and Q post-condition. The specification means that if S is executed in a state satisfying P, on termination its state will satisfy Q. So the above example means that if `S` is non-empty, then the program will return an element of `S`, which furthermore is less than or equal to other elements of `S`. In other words, the program returns the minimum element of `S`.

Note that in the above definition termination is assumed. You can also require that such a specification should also imply that S terminates, when it is executed from P. Requiring termination is obviously stronger. Hoare triple interpretation where termination is assumed is also called *partial correctness* interpretation, and otherwise it is called *total correctness* interpretation.

3.1 Some Inference Rules

To prove the validity of specifications, Hoare logic provides a set of inference rules. An example of such a rule is shown below, which allows us to infer a new specification with a weaker post-condition. Such a rule is read as follows: if the formulas above the line (so-called 'premises') can be inferred (can be proven to be valid), then the formula under the line can be concluded to be valid as well.

Rule 3.1.1 : Post-condition Weakening

$$\frac{\begin{array}{c}\vdash\ Q \Rightarrow Q' \\ \{*\ P\ *\}\ S\ \{*\ Q\ *\}\end{array}}{\{*\ P\ *\}\ S\ \{*\ Q'\ *\}}$$

Here, I will only discuss several inference rules. A more comprehensive list of rules can be found in Section 3.6. For example, analogous to the above rule, you can also strengthen a pre-condition. Hoare triples are also conjunctive and disjunctive. Rules expressing these properties can be found in Section 3.6.

Hoare logic only allows us to infer Hoare triples. It has by itself no rules to infer predicates over states. For example, one of the premises above is $\vdash Q \Rightarrow Q'$ (the predicate Q implies Q').

Hoare logic does not provide any rule to prove such a claim. We have to fall back to for example Predicate Logic to prove it[1] I will assume that you are familiar with Predicate Logic.

The rule below shows how we can prove the correstness of a sequential composition of two statements:

Rule 3.1.2 : SEQ

$$\frac{\{*\ P\ *\}\ S_1\ \{*\ P'\ *\} \qquad \{*\ P'\ *\}\ S_2\ \{*\ Q\ *\}}{\{*\ P\ *\}\quad (S_1;\ S_2)\quad \{*\ Q\ *\}}$$

Dealing with sequential composition is very important, because that is probably the most important way to compose your programs from elementary statements. Notice that to prove $\{*\ P\ *\}\ (S_1;\ S_2)\ \{*\ Q\ *\}$ the rule requires that you come up with an intermediate predicate $\mathtt{P}'$, and proves the correctness of the components S_1 and S_2 with respect to this intermediate predicate. The rule itself does not give you that P'. We cannot choose P' arbitrarily either, since that may render the first or the second premise invalid. Unfortunately, there is no general algorithm to calculate P' (though in some situations we can do it).

The rule below shows how to deal with conditional statements:

Rule 3.1.3 : ITE

$$\frac{\{*\ P \land g\ *\}\ S_1\ \{*\ Q\ *\} \qquad \{*\ P \land \neg g\ *\}\ S_2\ \{*\ Q\ *\}}{\{*\ P\ *\}\quad \texttt{if}\ g\ \texttt{then}\ S_1\ \texttt{else}\ S_2\quad \{*\ Q\ *\}}$$

The actual work in a program is done by assignments; these are the only units that change the state of your program. The rule below deals with assignments. The notation $Q[e/x]$ means the expression that we would obtain by replacing all *free* occurrences of the variable x in the expression Q by the expression e. For example, "$(\forall x : x{\in}S : x{>}y)[y{+}1/y]$" results in the expression $(\forall x : x{\in}S : x{>}y{+}1)$. [2]

Rule 3.1.4 : ASG

$$\frac{P \Rightarrow Q[e/x]}{\{*\ P\ *\}\quad x := e\quad \{*\ Q\ *\}}$$

Furthermore it is assumed here that the assignment $x := e$ would only affect the meaning of x (it does not have side effect on the variable y, for example).

3.2 Inference rule for loop

Here is the inference rule to handle loop. It assumes the partial correctness interpretation; so with it you still have not proven that the loop terminates. To prove termination you would need to prove some additional conditions.

[1] I explicitly write $\vdash P$ to mean that the predicate P is valid. If P is a state predicate, this means it is supposed to be true when evaluated on every possible state. On the other hand, when we simply write P, it is not necessarily a valid predicate. In the rules above we should perhaps also write $\vdash\ \{*\ P\ *\}\ S\ \{*\ Q\ *\}$ etc. I do not do this since in this case the notation seems to be too verbose.

Technically, in logic $\vdash P$ means that P is inferrable (provable) using the rules of whatever the logic we use. It does not immediately imply that P is valid when interpreted over whatever its intended semantic. The notation $\models P$ is often used to mean that P is valid. Assuming that you use a 'sound' and 'complete' Predicate Logic, then any P provable in the logic is also valid (and any valid P is provable).

[2] Furthermore, we will also insist that this substitution operator can only replace *free* variables. So, $(\forall x : x{\in}S : x{>}y)[0/x]$ will result in the same expression: $(\forall x : x{\in}S : x{>}y)$. Similarly, free variables in e in a substitution $[e/y]$ are meant to be free. So, they should not become bound (captured) after the substitution. For example $(\forall x : x{\in}S : x{>}y)[x{-}1/y]$ is *not* an allowed substitution, So, you should *not* conclude that it results in: $(\forall x : x{\in}S : x{>}x{-}1)$. We can however first rename the bound variable x, for example to $(\forall x' : x'{\in}S : x'{>}y)$. Now we can apply the subtitution: $(\forall x' : x'{\in}S : x'{>}y)[x{-}1/y]$, which results in $(\forall x' : x'{\in}S : x'{>}x{-}1)$.

Rule 3.2.1 : Loop

$$\frac{\vdash\ P \Rightarrow I \qquad \{*\ I \wedge g\ *\}\ S\ \{*\ I\ *\} \qquad \vdash I \wedge \neg g \Rightarrow Q}{\{*\ P\ *\}\ \texttt{while}\ g\ \texttt{do}\ S\ \ \{*\ Q\ *\}}$$

Notice that the rule requires you to come up with an I called *loop invariant* that satisfies all those three premises above. Unfortunately, there is no general algorithm to infer I.

Example

Prove the validity of this specification:

$\{*\ \texttt{i}{\geq}\texttt{0}\ *\}\ \ \texttt{while i>0 do i} := \texttt{i}-\texttt{1}\ \ \{*\ \texttt{i} = \texttt{0}\ *\}$

Answer: using the pre-condition $\texttt{i}{\geq}\texttt{0}$ itself as the invariant I will do.

Example

The following program sums the elements of the array `a`; prove its correctness:

$\{*\ \texttt{s} = \texttt{0}\ *\}\ \ \texttt{i} = \#\texttt{a};\ \ \texttt{while i>0 do}\ \{\texttt{i} := \texttt{i}-\texttt{1};\ \texttt{s} := \texttt{s}+\texttt{a[i]}\}\ \ \{*\ \texttt{s} = (\Sigma\texttt{j} : \texttt{0}{\leq}\texttt{j}{<}\#\texttt{a} : \texttt{a[j]})\ *\}$

Answer: using this as invariant:

$\texttt{0}{\leq}\texttt{i}{\leq}\#\texttt{a}\ \ \wedge\ \ \texttt{s} = (\Sigma\texttt{j} : \texttt{0}{\leq}\texttt{j}{<}\texttt{i} : \texttt{a[j]})$

(prove that it satisfies all the conditions in the Loop Rule above).

Example

The following program checks if a boolean array `b` contains a `true`. Prove that it is correct.

$\{*\ \texttt{true}\ *\}$

$\texttt{i} = \texttt{0};\ \texttt{found} := \texttt{false}$

$\texttt{while i<}\#\texttt{b} \wedge \neg\texttt{found do}\ \{\texttt{found} := \texttt{b[i]};\ \texttt{i} := \texttt{i}+\texttt{1}\}$

$\{*\ \texttt{found} = (\exists\texttt{j} : \texttt{0}{\leq}\texttt{j}{<}\#\texttt{b} : \texttt{b[j]})\ *\}$

Answer: using this as invariant:

$\texttt{0}{\leq}\texttt{i}{\leq}\#\texttt{b}\ \ \wedge\ \ \texttt{found} = (\exists\texttt{j} : \texttt{0}{\leq}\texttt{j}{<}\texttt{i} : \texttt{b[j]})$

(prove that it satisfies all the conditions in the Loop Rule above).

3.3 Invariant as abstraction

Loop invariant can be seen as an abstraction of a loop. Knowing the invariant gives you sufficient information to know the goal of the loop, which you can infer from $I \wedge \neg g$, without having to look at the loop itself, including its various implementation details.

E.g. in the third example above, the loop is optimized to break as soon as the right element has been found. If you just want to know what the loop mainly does, such a detail is less important. The invariant gives you in this case a more abstract view.

3.4 Rule for proving loop termination

To prove that a loop terminates the idea is to come up with an integer expression m (interpreted over your program states) called *termination metric.* This m should be such that its value decreases at each iteration. However, your invariant implies that m has a lower bound, e.g. 0. These imply that you cannot iterate forever, as m would then breach its lower bound, which is a contradiction.

This is formally captured by this strengthened version of the Loop Rule:

Rule 3.4.1 : Loop with Total Correctness

$$\frac{\begin{array}{c} \vdash P \Rightarrow I \\ \{* \; I \wedge g \; *\} \;\; (C := m; S) \;\; \{* \; m < C \; *\} \\ \vdash I \wedge g \Rightarrow m > 0 \\ \vdash I \wedge \neg g \Rightarrow Q \\ \{* \; I \wedge g \; *\} \;\; S \;\; \{* \; I \; *\} \end{array}}{\{* \; P \; *\} \;\; \texttt{while} \;\; g \; \texttt{do} \; S \;\; \{* \; Q \; *\}}$$

In general m can also be an expression that returns a value from some domain D, equiped with a well-founded relation $\prec$ over D, which means that every non-empty subset $X \subseteq D$ must have a minimum member (in terms of the relation $\prec$). The second premise above becomes a requirement that S has to decrease the value of m, over $\prec$. The well-foundedness then implies that the decreasing chain of the corresponding values of m, as we iterate the loop, must have a minimum value. However, since the chain can still be infinitely long, we still need to require that every decreasing chain in D should be finite. And the third premise, saying that m has a lower bound is no longer needed. It can be replaced with a requirement that I implies that m is closed in D.

There is unfortunately no general algorithm to infer m (due to the Halting Problem).

Example

Prove that this loop terminates:

$\{* \; \texttt{i} \geq 0 \; *\}$ `while i>0 do i := i−1` $\{* \; \texttt{i} = 0 \; *\}$

Answer: we can `i≥0` as the invariant I. As the termination metric, the expression `i` will do.

Example

Prove that the program below terminates.

```
{* r≥0 ∧ b≥0 *}

 while r+b > 0 do {
   if r>0 → r := r − 1
   [] b>0 → {b := b − 1; r := r + 1}
   fi
  }

{* r=0 ∧ b=0 *}
```

The `if` above is non-deterministic: if both guards are true, which is possible, then one of the alternatives will be non-deterministically selected. If neither is true, the statement does nothing (skip).

To prove the validity of the above specification, these invariant and termination metric will do:

I : `r≥0 ∧ b≥0`

m : `r + 2b`

To handle the non-deterministic `if`, we can use a generalized version of the ITE Rule (3.1.3, as shown below.

Rule 3.4.2 : NDIF

$$\frac{\begin{array}{c}\{*\ P \wedge g_1 \ *\}\ S_1\ \{*\ Q\ *\} \\ \{*\ P \wedge g_2 \ *\}\ S_2\ \{*\ Q\ *\} \\ P \wedge \neg g_1 \wedge \neg g_2 \ \Rightarrow\ Q\end{array}}{\{*\ P\ *\} \quad \texttt{if}\ g_1 \to S_1\ []\ g_2 \to S_2\ \texttt{fi} \quad \{*\ Q\ *\}}$$

3.5 Model of programs

A *model* of an artifact abstractly represents the artifact. Being 'abstract' implies that there are some properties of the artifact that are not captured by the model. A model is thus simpler (that is why it is just a 'model'); but still, one is often useful to study the real artifact. Below we will discuss a simple way to model programs, which we can use as a reference to study logics about programs, e.g. the Hoare logic. It can be used for example to prove the soundness of the inference rules we have. The purpose of this section is simply to show you a starting point. We will not discuss the soundness of Hoare logic itself in details. You should also keep in mind that the model is simplistic; for modeling complex programming concepts or constructs, you will need to extend it.

Let us abstractly model the *state* of program by a record that maps the program's variables to their values in that state. E.g. $s = \{x{=}0, y{=}0\}$ represents a state of some program with two variables x, y, and in that state the value of x is 0, and y is 9. We will use the notation $s.x$ and $s.y$ to refer to the fields of this record.

Let us for simplicity assume that we have a fixed set of program variables; we name this set Var. All statements we will discuss are assumed to operate on states over Var. Let Σ be the set of all possible states over this Var.

An expression can be abstractly modelled by a function e of type $\Sigma \to val$, where val is the type of the value returned by the expression. A *predicate* is just an expression that returns a boolean value. For example the predicate $x{>}y$ is modelled by the function $(\lambda s.\ s.x > s.y)$.

To keep the notation light, I will usually use the term 'predicate' and its model interchangebly. When I name a predicate P, I often use the same symbol to denote its model. You'll have read it from the context which one is meant. The same goes with other program elements, e.g. state or expression.

A predicate P induces a set, namely the set of all states that satisfies it: $S = \{s \mid P\ s = \texttt{true}\}$. Conversely, if we know the set, it defines the corresponding predicate, namely: $P = (\lambda s.\ s \in S)$. In other words, a predicate and the set it induces are isomorphic. This allows us to treat predicates as if they are sets; e.g. we can pretend applying set operators on them, which is sometimes convenient.

In terms of sets, conjunction and disjunction of two predicates correspond to set intersection and union. So, e.g. $P \wedge Q = P \cap Q$. For implication:

$$P \Rightarrow Q \quad = \quad (\texttt{true}/P) \cup Q$$

The validity of an implication corresponds to the subset relation. Let us write $\models P$ to mean that P is a valid predicate. That is, it is true on all states in Σ. So, $\models P = (\forall s :: P\ s = \texttt{true})$. Now, the validity of an implication means:

$$\models (P \Rightarrow Q) \quad = \quad P \subseteq Q$$

A program can be abstractly modelled by a function of type $\Sigma \to \Sigma$. However, with this we cannot model a non-deterministic program. To allow the latter, we take this model; if S is a statement or a program:

$$S : \Sigma \to Pow(\Sigma)$$

where $Pow(\Sigma)$ is the power set of Σ. $S\ s$ describes the set of all possible end-states when S is executed on s. In this discussion we will just assume that our programs always terminate; so $S\ s$ is never empty.

For example the simple statement `x++` is modelled by:

$$(\lambda s.\ \{\{x{=}s.x{+}1,\ y{=}s.y\}\})$$

Whereas the following is the model of a statement that non-deterministically chooses between doing a skip or `x++`:

$$(\lambda s.\ \{s,\ \{x{=}s.x{+}1,\ y{=}s.y\}\})$$

Models of 'if' and sequential compositions:

$$\texttt{if}\ g\ \texttt{then}\ S\ \texttt{else}\ T \quad = \quad (\lambda s.\ \texttt{if}\ g\ s\ \texttt{then}\ S\ s\ \texttt{else}\ T\ s) \tag{3.1}$$

$$S_1\ ;\ S_2 \quad = \quad (\lambda s. \bigcup\{S_2\ t \mid t \in S_1\ s\}) \tag{3.2}$$

Now we have enough to give an abstract model of Hoare triple:

$$\{*\ P\ *\}\ S\ \{*\ Q\ *\} \quad = \quad (\forall s :: s{\in}P \Rightarrow S\ s \subseteq Q) \tag{3.3}$$

Now that we have these models, we can justify the programming rules that we had. For example, let us see how we can prove the soundness of the SEQ Rule 3.1.2. For your convenience, here is the rule again:

$$\frac{\begin{array}{c}\{*\ P\ *\}\ S_1\ \{*\ P'\ *\}\\ \{*\ P'\ *\}\ S_2\ \{*\ Q\ *\}\end{array}}{\{*\ P\ *\}\quad (S_1;\ S_2)\quad \{*\ Q\ *\}}$$

By the definition in (3.3), it is sufficient to prove that $((S_1;\ S_2)\ s) \subseteq Q$, for any $s \in P$. Now, the first premise implies that $S_1\ s \subseteq P'$. The second premise implies that for any $t \in S_1\ s$, $S_2\ t \subseteq Q$. Consequently, $R = \bigcup\{S_2\ t \mid t \in S_1\ s\}$ is also a subset of Q. But by (3.2), R is just the same as $(S_1;\ S_2)\ s$. So, we are done.

As another example, consider the Post-condition Weakening Rule 3.1.1. Here it is once more:

$$\frac{\begin{array}{c}\vdash\ Q \Rightarrow Q'\\ \{*\ P\ *\}\ S\ \{*\ Q\ *\}\end{array}}{\{*\ P\ *\}\ S\ \{*\ Q'\ *\}}$$

Here is how we can prove the rule. Firstly, assuming the used Predicate Logic is sound and complete, $\vdash Q \Rightarrow Q'$ is equivalent to $\models Q \Rightarrow Q'$. So it means that Q is a subset of Q'. Next, we have to prove that $S\ s \subseteq Q'$, for any $s \in P$. The second premise implies that $S\ s \subseteq Q$. But as remarked, $Q \subseteq Q'$. So, we are done.

3.6 Inference Rules of Hoare Logic

Rule 3.6.1 : Post-condition Weakening

$$\frac{\begin{array}{c}\vdash\ Q \Rightarrow Q'\\ \{*\ P\ *\}\ S\ \{*\ Q\ *\}\end{array}}{\{*\ P\ *\}\ S\ \{*\ Q'\ *\}}$$

Rule 3.6.2 : Pre-condition Strengthening

$$\frac{\begin{array}{l}\vdash\ P \Rightarrow P'\\ \{*\ P'\ *\}\ S\ \{*\ Q\ *\}\end{array}}{\{*\ P\ *\}\ S\ \{*\ Q\ *\}}$$

Rule 3.6.3 : Hoare Triple Disjunction

$$\frac{\begin{array}{l}\{* \; P_1 \; *\} \; S \; \{* \; Q_1 \; *\} \\ \{* \; P_2 \; *\} \; S \; \{* \; Q_2 \; *\}\end{array}}{\{* \; P_1 \vee P_2 \; *\} \; S \; \{* \; Q_1 \vee Q_2 \; *\}}$$

Rule 3.6.4 : Hoare Triple Conjunction

$$\frac{\begin{array}{l}\{* \; P_1 \; *\} \; S \; \{* \; Q_1 \; *\} \\ \{* \; P_2 \; *\} \; S \; \{* \; Q_2 \; *\}\end{array}}{\{* \; P_1 \wedge P_2 \; *\} \; S \; \{* \; Q_1 \wedge Q_2 \; *\}}$$

Rule 3.6.5 : SEQ

$$\frac{\begin{array}{c}\{* \; P \; *\} \; S_1 \; \{* \; P' \; *\} \\ \{* \; P' \; *\} \; S_2 \; \{* \; Q \; *\}\end{array}}{\{* \; P \; *\} \quad (S_1; \; S_2) \quad \{* \; Q \; *\}}$$

Rule 3.6.6 : If-Then-Else 1

$$\frac{\begin{array}{c}\{* \; P \wedge g \; *\} \; S_1 \; \{* \; Q \; *\} \\ \{* \; P \wedge \neg g \; *\} \; S_2 \; \{* \; Q \; *\}\end{array}}{\{* \; P \; *\} \quad \texttt{if } g \texttt{ then } S_1 \texttt{ else } S_2 \quad \{* \; Q \; *\}}$$

Rule 3.6.7 : If-Then-Else 2

$$\frac{\begin{array}{c}\{* \; P_1 \; *\} \; S_1 \; \{* \; Q \; *\} \\ \{* \; P_2 \; *\} \; S_2 \; \{* \; Q \; *\}\end{array}}{\{* \; (g \Rightarrow P_1) \wedge (\neg g \Rightarrow P_2) \; *\} \quad \texttt{if } g \texttt{ then } S_1 \texttt{ else } S_2 \quad \{* \; Q \; *\}}$$

Rule 3.6.8 : Assignment

$$\frac{-}{\{* \; Q[e/v] \; *\} \quad v := e \quad \{* \; Q \; *\}}$$

Rule 3.6.9 : Loop 1 (Partial Correctness)

$$\frac{\begin{array}{c}\vdash \; P \Rightarrow I \\ \{* \; I \wedge g \; *\} \; S \; \{* \; I \; *\} \\ \vdash I \wedge \neg g \Rightarrow Q\end{array}}{\{* \; P \; *\} \; \texttt{while } g \texttt{ do } S \; \{* \; Q \; *\}}$$

Rule 3.6.10 : Loop 2 (Partial Correctness)

$$\frac{\begin{array}{c}\{* \; I \wedge g \; *\} \; S \; \{* \; I \; *\} \\ \vdash I \wedge \neg g \Rightarrow Q\end{array}}{\{* \; I \; *\} \; \texttt{while } g \texttt{ do } S \; \{* \; Q \; *\}}$$

Rule 3.6.11 : Loop 3 (Total Correctness)

$$\frac{\begin{array}{c}\vdash P \Rightarrow I \\ \{* \; I \wedge g \; *\} \; (C := m; S) \; \{* \; m < C \; *\} \\ \vdash I \wedge g \Rightarrow m > 0 \\ \vdash I \wedge \neg g \Rightarrow Q \\ \{* \; I \wedge g \; *\} \; S \; \{* \; I \; *\}\end{array}}{\{* \; P \; *\} \; \texttt{while } \; g \texttt{ do } S \; \{* \; Q \; *\}}$$

Rule 3.6.12 : Assignment Targeting an Array's Element
Treat an assignment $a[e_1] := e_2$ as an assignment:

$$a := a(e_1 \texttt{ repby } e_2)$$

where $a(e_1 \texttt{ repby } e_2)$ is defined as follows:

$$a(e_1 \texttt{ repby } e_2)[j] \; = \; (j = e_1) \rightarrow e_2 \mid a[j]$$

Rule 3.6.13 : Assignment Targeting an Record's Element
Treat an assignment $r.fname := e$ as an assignment:

$$r := r(fname\ \texttt{repby}\ e)$$

where `repby` is defined as follows:

$$\begin{array}{lcll} r(fname\ \texttt{repby}\ e).fname & = & e & \\ r(fname\ \texttt{repby}\ e).gname & = & r.gname & \text{, if } fname \neq gname \end{array}$$

3.7 Some Commonly Used Inference Rules and Theorems of Predicate Logic

Rule 3.7.1 : Excluded Middle

$$\frac{-}{P \vee \neg P}$$

Rule 3.7.2 : Modus Ponens ($\Rightarrow$ Elimination)

$$\frac{\begin{array}{l} P \\ P \Rightarrow Q \end{array}}{Q}$$

Rule 3.7.3 : Contradiction

$$\frac{\begin{array}{l} P \\ \neg P \end{array}}{\texttt{false}} \qquad \frac{P \Rightarrow \texttt{false}}{\neg P}$$

Rule 3.7.4 : True Consequence

$$\frac{P = \texttt{true}}{P} \qquad \frac{\texttt{true} \Rightarrow P}{P}$$

Rule 3.7.5 : $\wedge$ Elimination

$$\frac{P \wedge Q}{Q} \qquad \frac{P \wedge Q}{P}$$

Rule 3.7.6 : Conjunction ($\wedge$ Introduction)

$$\frac{\begin{array}{l} P \\ Q \end{array}}{P \wedge Q}$$

Rule 3.7.7 : $\vee$ Elimination

$$\frac{\begin{array}{l} \neg P \\ P \vee Q \end{array}}{Q}$$

Rule 3.7.8 : $\vee$ Introduction

1. $\dfrac{\neg P \Rightarrow Q}{P \vee Q}$
2. An easier version. But it is weaker, since you may fail to prove Q without the help of $\neg P$:

$$\frac{Q}{P \vee Q}$$

Rule 3.7.9 : Case Split

$$\frac{\begin{array}{l} P \Rightarrow Q \\ \neg P \Rightarrow Q \end{array}}{Q} \qquad \frac{\begin{array}{l} P_1 \vee P_2 \\ P_1 \Rightarrow Q \\ P_2 \Rightarrow Q \end{array}}{Q}$$

Rule 3.7.10 : Specialization ($\forall$-Elimination)

$$\frac{\begin{array}{l} P\ e \\ (\forall i : P\ i : Q\ i) \end{array}}{Q\ e} \qquad \frac{(\forall i : P\ i : Q\ i)}{P\ e\ \Rightarrow\ Q\ e}$$

Rule 3.7.11 : $\exists$ Introduction

$$\frac{P\ e}{(\exists i :: P\ i)} \qquad \frac{\begin{array}{l} P\ e \\ Q\ e \end{array}}{(\exists i : P\ i : Q\ i)}$$

Theorem 3.7.12 : Basic equalities of Boolean connectors

1. $\vdash \neg\neg P = P$
2. $\vdash P \vee Q = Q \vee P$
3. $\vdash \texttt{true} \vee Q = \texttt{true}$
4. $\vdash \texttt{false} \vee Q = Q$
5. $\vdash (P \vee Q) \vee R = P \vee (Q \vee R) = P \vee Q \vee R$

6. $\vdash P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$
7. $\vdash P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
8. $\vdash P \wedge Q = Q \wedge P$
9. $\vdash \texttt{true} \wedge Q = Q$
10. $\vdash \texttt{false} \wedge Q = F$
11. $\vdash (P \wedge Q) \wedge R = P \wedge (Q \wedge R) = P \wedge Q \wedge R$
12. $\vdash P \Rightarrow Q = \neg P \vee Q$
13. $\vdash \neg(P \Rightarrow Q) = P \wedge \neg Q$
14. $\vdash P \Rightarrow Q \Rightarrow R = P \Rightarrow (Q \Rightarrow R) = P \wedge Q \Rightarrow R$
15. $\vdash (P = Q) = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$

Theorem 3.7.13 : de Morgan

1. $\vdash \neg(P \vee Q) = \neg P \wedge \neg Q$
2. $\vdash \neg(P \wedge Q) = \neg P \vee \neg Q$

Theorem 3.7.14 : Contra Position

$\vdash P \Rightarrow Q = \neg Q \Rightarrow \neg P$

Theorem 3.7.15 : COND conversion

1. $\vdash \quad P \quad \Rightarrow \quad (P \rightarrow e_1 \mid e_2 \quad = \quad e_1)$
2. $\vdash \quad \neg P \quad \Rightarrow \quad (P \rightarrow e_1 \mid e_2 \quad = \quad e_2)$
3. $\vdash P \rightarrow e \mid e \;=\; e$
4. $\vdash f\ (P \rightarrow e_1 \mid e_2) \;=\; P \rightarrow f\ e_1 \mid f\ e_2$
5. $\vdash P \rightarrow e_1 \mid e_2 \;=\; \neg P \rightarrow e_2 \mid e_1$

Theorem 3.7.16 : COND Split

$\vdash \quad P \rightarrow Q \mid R \quad = \quad (P \Rightarrow Q) \wedge (\neg P \Rightarrow R)$

P, Q, and R have to be predicates.

Theorem 3.7.17 : Renaming bound variables

1. $\vdash (\forall i : P : Q) = (\forall i' : P[i'/i] : Q[i'/i])$
2. $\vdash (\exists i : P : Q) = (\exists i' : P[i'/i] : Q[i'/i])$

i' should not occur free in P and Q.

Theorem 3.7.18 : Negate $\forall$

$\vdash \neg(\forall i : P\ i : Q\ i) = (\exists i : P\ i : \neg(Q\ i))$

Theorem 3.7.19 : Negate $\exists$

$\vdash \neg(\exists i : P\ i : Q\ i) = (\forall i : P\ i : \neg(Q\ i))$

Theorem 3.7.20 : Nested quantifications

1. $\vdash (\forall i,j :: P\ i\ j) = (\forall i :: (\forall j :: P\ i\ j))$
2. $\vdash (\exists i,j :: P\ i\ j) = (\exists i :: (\exists j :: P\ i\ j))$

Theorem 3.7.21 : Quantification over singleton domain

1. $\vdash (\forall i : i = e : P\ i) = P\ e$
2. $\vdash (\exists i : i = e : P\ i) = P\ e$

Theorem 3.7.22 : Range Split

1. $\vdash (\forall i : P\ i : Q_1\ i \wedge Q_2\ i) = (\forall i : P\ i : Q_1\ i) \wedge (\forall i : P\ i : Q_2\ i)$
2. $\vdash (\exists i : P\ i : Q_1\ i \vee Q_2\ i) = (\exists i : P\ i : Q_1\ i) \vee (\exists i : P\ i : Q_2\ i)$

Theorem 3.7.23 : Domain split

1. $\vdash (\forall i : P\ i \vee Q\ i : R\ i) = (\forall i : P\ i : R\ i) \wedge (\forall i : Q\ i : R\ i)$
2. $\vdash (\exists i : P\ i \vee Q\ i : R\ i) = (\exists i : P\ i : R\ i) \vee (\exists i : Q\ i : R\ i)$

Theorem 3.7.24 : Quantification over empty domain

1. $\vdash (\forall i : \texttt{false} : P\ i) = \texttt{true}$
2. $\vdash (\exists i : \texttt{false} : P\ i) = \texttt{false}$

Theorem 3.7.25 : Domain Shift

1. $\vdash (\forall i : P\ i : Q\ i) = (\forall i :: P\ i \Rightarrow Q\ i)$
2. $\vdash (\forall i : P_1\ i \wedge P_2\ i : Q\ i) = (\forall i : P_1\ i : P_2\ i \Rightarrow Q\ i)$
3. $\vdash (\exists i : P\ i : Q\ i) = (\exists i :: P\ i \wedge Q\ i)$
4. $\vdash (\exists i : P_1\ i \wedge P_2\ i : Q\ i) = (\exists i : P_1\ i : P_2\ i \wedge Q\ i)$

3.8 Predicate Transformer

Imagine a function **p** that maps programs to functions of type $Pred \rightarrow Pred$. So, if S is a statement or a program, **p** S is a function that takes a state predicate, and returns a state predicate. The latter function is called a *predicate transformer*. The function **p** itself can be thought as defining predicate transformer semantics to programs. However, people often simply use the term *predicate transformer* to also refer to this **p**.

The idea was first proposed by Dijkstra [6]; he uses such a transformer to calculate a pre-condition of S, given some post-condition. Conversely, we can also have a transformer that calculate post-condtions. Let's start with the latter, because it is more intuitive to explain. Let $\mathbf{cp}\rhd\ S$ be a transformer, such that when given a pre-condition P, $\mathbf{p}\rhd\ S$ it would calculate a consistent post-condition. We also say that the transformer is *sound*; it is as such that the following hold:

$$\{*\ P\ *\}\ S\ \{*\ \mathbf{cp}\rhd\ S\ P\ *\} \tag{3.4}$$

Whereas a program takes a concrete input and results in a conrete output, a predicate transformer can be thought as a symbolic counterpart of the program it represents. For example, the transformer $\mathbf{cp}\rhd\ S$ takes a formula P that symbolically represents a class of inputs, and it calculates the corresponding symbolic characterization of the outputs. In contrast to a concrete execution of S, note that P can easily characterize an infinitely large set of states. A computation that operates on predicates, rather than on concrete values, is also called *symbolic computation.*

Of particular interest is the so-called *strongest post-condition transformer*, often denoted by **sp**, which produces a post-condition that is consistent and strongest. It follows, that a verification problem can be equivalently expressed as the problem of proving the validity of the implication below:

$$\{*\, P \,*\}\ S\ \{*\, Q\,*\} \quad \equiv \quad \mathbf{sp}\, S\ P\ \Rightarrow\ Q \tag{3.5}$$

A forward transformer satisying the above equivalence is said to be *sound* and *complete.* It is complete if it satisfies the $\Rightarrow$ direction, and sound if it satisfies the $\Leftarrow$ direction; note that the latter is equivalent to the fomulation in (3.4).

We can also have transformers that work in the opposite direction, backward rather than forward as $\mathbf{cp}\triangleright$. Imagine a transformer $\triangleleft\mathbf{cp}\ S$ that takes a post-condition and calculates a consistent pre-condition. We also say that the transformer is *sound*; it is as such that the following hold:

$$\{*\ \triangleleft\mathbf{cp}\ S\ Q\,*\}\ S\ \{*\, Q\,*\} \tag{3.6}$$

Analogous to the concept of the strongest post-condition, we can also define the *weakest post-condition transformer*, often denoted by **wp**, which produces the weakest pre-condition that is consistent with the given post-condition. It follows, that a verification problem can be equivalently expressed as the problem of proving the validity of the implication below:

$$\{*\, P \,*\}\ S\ \{*\, Q\,*\} \quad \equiv \quad P \Rightarrow \mathbf{wp}\ S\ Q \tag{3.7}$$

A backward transformer satisying the above equivalence is said to be *sound* and *complete.* It is complete if it satisfies the $\Rightarrow$ direction, and sound if it satisfies the $\Leftarrow$ direction; note that the latter is equivalent to the fomulation in (3.6).

We have not said whether in all the definitions given above, the Hoare triples should be interpreted partially (termination assumed) or totally (termination required). The good thing is that the above concepts are in themselves independent of this choice. However, if the partial interpretation is chosen, the corresponding transformers are often called the liberal variant. For example, the 'weakest liberal pre-condition transformer', often denoted by **wlp**, yields the weakest pre-condition with respect to the given post-condition, but termination is assumed.

Dijkstra mainly focused on the **wp** and **wlp** transformers, and use them purely for solving verification problems. Notice that the nice thing about these transformers is that they basically convert the problem of program verification to a problem in plain Predicaate Logic. We will need an implementation these transformers, for which we would first need to have some idea how they can be formulated. This will be discussed in the subsection below. However, let me first point out that it is unfortunately not possible to have a truely weakest pre-condition transformer that always terminates. A terminating weakest pre-condition transformer would otherwise solve the Halting problem. So, pragmatically some trade off has to be made. What is also important, predicate transformers have other applications beyond the traditional verification. This will be discussed later.

3.8.1 wp-transformer for GCL

Let us consider a very simple imperative language, similar to Dijkstra's Guarded Command Language (GCL) [6]. We will simply call our variant GCL as well. Below we will show how a **wp**-transformer can be formulated for this language; you should then get an idea how this transformer can be turned into an actual tool.

Having a small langauge definitely helps to focus our discussion. However, the language is actually expressive enough to let you handle real life programming languages, by first translating them to GCL. This has actually been done. The tool ESC/Java (Extended Static Checker for

Java) [15] uses predicate transformers to verify absence of runtime exceptions (e.g. caused by null-dereference, or by access to an array beyond its valid indices). A predicate transformer can of course be defined directly at the Java level. A major drawback of this approach is that the resulting implementation would be large and complex. Hence it is more difficult to experiment with, harder to maintain, and it is also more difficult to decouple its parts to be reused for other purposes (e.g. if you want to reuse parts to build your own variant of Java logic). So, in ESC/Java, the transformer is defined over some GCL-like language. Java source code is first translated to GCL, and the transformer takes over from that point on.

Spec# follows the same approach, for C#. Their GCL-layer is called Boogie, and is also released as a separete component, allowing you to reuse the same verification engine for other languages.

For now, our GCL has the following statements:

1. `skip`.
2. Assignment: $var := expr$.
3. $stmt_1$; $stmt_2$.
4. Throwing an exception: `raise`.
5. Try $stmt_1$, if it throws an exception, handles it with $stmt_2$: $stmt_1$! $stmt_2$.
6. Requiring that a predicate holds: `assert` *expr*. When the expression holds when this statement is executed, it does nothing (so we can proceed to the next statement). If the expression does not hold, the statement 'crashes'[3].
7. Assuming that a predicate holds: `assume` *expr*. When the expression holds when the statement is executed, nothing happens; we can proceed with whatever the next statement is. If the expression does not hold, the situation simply does not fit into the assumption we posed; it is then treated as a magic statement that can realize any post-condition[4].
8. Non-deterministically choose between $stmt_1$ and $stmt_2$: $stmt_1$ [] $stmt_2$.
9. Introducing a block with local variables: `var` $variable^+$ `in` *stmt* `end`. The variables are uninitialized. If they must be initialized, this is done explicitly in *stmt*.
10. A loop with an optional invariant $expr_1$: [`inv` $expr_1$] `while` $expr_2$ `do` *stmt*

Note that an if-then-else structure can be encoded as follows:

$$\texttt{if}\ g\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \quad = \quad \{\texttt{assume}\ g\ ;\ S_1\}\ []\ \{\texttt{assume}\ \neg g\ ;\ S_2\}$$

We can also easily express non-deterministic 'if', e.g.:

$$\{\texttt{assume}\ g\ ;\ S_1\}\ []\ \{\texttt{assume}\ g\ ;\ S_2\}\ []\ \{\texttt{assume}\ \neg g\ ;\ \texttt{skip}\}$$

Let us first exclude exceptions and loops. The weakest pre-condition transformer for GCL can be defined recursively as shown below. For simplicity, we will take the partial correctness interpretation (termination is assumed).

[3]In principle, this implies that we also need a way to talk about 'crashes' in our logic. How to do this is a topic of its own. For the sake of simplicity, we will choose to avoid this issue; we could say that we want to focus on reasoning about the program's states when it does not crash. Keep in mind that for a different purpose, for example if you need to investigate what you find at a crash, and you need a logic to reason about such things, then obviously the aspect needs to be included in the logic.

[4]This statement is introduced only as a theoretical concept, used later for expressing other things. No such magical statement can in reality exist. But in terms of the simple model from Section 3.5, there is one statement that can do such magic, namely `magic` = $(\lambda s.\ \emptyset)$, so `assume` e can be modelled by `if` e `then skip else magic`.

1. $\mathsf{wlp}\ \mathtt{skip}\ Q \quad = \quad Q$
2. $\mathsf{wlp}\ (x := e)\ Q \quad = \quad Q[e/x]$, assuming, as before, the assignment would only affect the meaning of x.
3. $\mathsf{wlp}\ (\mathtt{assert}\ P)\ Q \quad = \quad P \wedge Q$
4. $\mathsf{wlp}\ (\mathtt{assume}\ P)\ Q \quad = \quad P \Rightarrow Q$
5. $\mathsf{wlp}\ (S_1; S_2)\ Q \quad = \quad \mathsf{wlp}\ S_1\ (\mathsf{wlp}\ S_2\ Q)$
6. $\mathsf{wlp}\ (S_1[]S_2)\ Q \quad = \quad \mathsf{wlp}\ S_1\ Q\ \wedge\ \mathsf{wlp}\ S_2\ Q$

   It follows that the **wlp** of if-then-else is:

   $$\mathbf{wlp}\ (\mathtt{if}\ g\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2)\ Q \quad = \quad (g \Rightarrow \mathbf{wlp}\ S_1\ Q)\ \wedge\ (\neg g \Rightarrow \mathbf{wlp}\ S_2\ Q)$$

   You can prove that it is equivalent to the following (due to g and $\neg g$ lhs of the implications):

   $$\mathbf{wlp}\ (\mathtt{if}\ g\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2)\ Q \quad = \quad (g \wedge \mathbf{wlp}\ S_1\ Q)\ \vee\ (\neg g \wedge \mathbf{wlp}\ S_2\ Q)$$

7. For simplicity we assume that x is a fresh variable; else rename it to make it fresh[5].

   $\mathsf{wlp}\ (\mathtt{var}\ x\ \mathtt{in}\ S\ \mathtt{end})\ Q \quad = \quad (\forall x :: \mathsf{wlp}\ S\ Q)$

Notice that above definition of **wlp** is 'syntax driven'. That is, **wlp** is defined per syntax construct of statements. This makes implementing this **wlp** quite straight forward.

3.8.2 Liberal Pre-conditon of Loops

Consider first a loop $\mathtt{inv}\ I\ \mathtt{while}\ g\ \mathtt{do}\ S$, where I is a candidate invariant that the programmer has annotated; so we 'expect' it to satisfy $\{*\ I \wedge g\ *\}\ S\ \{*\ I\ *\}$; see Section 3.2. Well, just because the programmer annotes it, does not mean that I is indeed an invariant of the loop; so this needs to be verified. If it is, we can *choose* to define the **wlp** of the loop to be equal to I. However, we should then also require that when the loop terminates, and that the invariant implies the given post-condition Q. For the latter, it means that we additionally need to verify that $I \wedge \neg g \Rightarrow Q$ is valid. If this turns out to be invalid, or if I is not an invariant, then it can still work out if the loop immediately exit when it starst in Q. We can capture these as the following choice of $'\mathbf{wlp}'$:

$$\mathbf{wlp}\ (\mathtt{inv}\ I\ \mathtt{while}\ g\ \mathbf{do}\ S)\ Q \quad = \quad \begin{cases} I & \text{, provided } \vdash I \wedge \neg g \Rightarrow Q \\ & \quad \text{and } \vdash I \wedge g \Rightarrow \mathbf{wlp}\ S\ I \\ & \\ \neg g \wedge Q & \text{, otherwise} \end{cases} \tag{3.8}$$

The two conditions may be non-trivial to prove. We can also take a more pragmatic position, by trusting the programmer's annotation, and thus dropping the second condition.

The function above does not actually always return the weakest pre-condition, which would happen if the annotated I turns out to be not an invariant, or if it is, but is simply not the weakest one. Requiring the programmer to accurately annotate the weakest invariant seems unpractical. So, naming the function **wlp** is not really appropriate. Nevertheless, we will keep the name, to avoid the verbosity we would get if we introduce a new name. This does mean that we lose the equivalence in (3.7). We still have soundness (3.6), which implies the following property:

$$\{*\ P\ *\}\ T\ \{*\ Q\ *\} \quad \Leftarrow \quad P \Rightarrow \mathbf{wlp}\ T\ Q \tag{3.9}$$

So, the validity of the implication $P \Rightarrow \mathbf{wlp}\ T\ Q$ implies the validity of the Hoare triple specification. However, if the implication turns out to be invalid, it does not necessarily imply that the specification is invalid. It could be because some invariants we gave are actually too strong.

[5] Imposing that you rename the local variable to make it fresh makes the **wlp** formula much simpler. We can drop the requirement though; but then you need to do some substitutions in the **wlp** te prevent accidental interpretation of upper level variables as local variables.

3.8.3 Reducing Loops

What if the programmer does not annotate the loops he wrote? In practice, this is typically the case. There are some heuristics to derive invariants based on some analysis on the shape of the post-condition, the shape of the loop, and variables that are involved in them [9, 19, 18]. These are beyond the scope of this book. I will instead show few non-heuristic approaches. They are all pretty basic, so if you build your own verification tool, these can be easily added as part of your initial infrastructure to deal with loops.

Calculating I with fixpoint iterations

A loop `while` g `do` S behaves the same as:

```
if g then {S ; while g do S}
     else skip
```

So, if W is the **wlp** of the loop, it would also be equal to:

$$\begin{aligned} W &= (g \wedge \textbf{wlp}\ S\ (\textbf{wlp}\ (\texttt{while}\ g\ \texttt{do}\ S)\ Q)) \ \vee\ (\neg g \wedge Q) \\ &= (g \wedge \textbf{wlp}\ S\ W) \ \vee\ (\neg g \wedge Q) \end{aligned}$$

So, W is a solution of the equation below. Actually, it is the greatest solution of it, with respect to the $\subseteq$ relation (interpreting predicates as sets):

$$W = (g \wedge \textbf{wlp}\ S\ W) \ \vee\ (\neg g \wedge Q) \tag{3.10}$$

Let me first detour to some bits of fix point theory. Let $(A, \leq)$ be a set with a partial order over it. This pair is called a *complete lattice* if every subset of A has a supremum (least upper bound) and infimum (greatest lower bound) with respect to $\leq$. Note that it follows that a complete lattice cannot be empty. Let $f : A{\rightarrow}A$ be a function. An x satisfying $x = f(x)$ is called a *fix point* of f. The function is *monotonic* with respect to $\leq$ if $x \leq y$ implies $f(x) \leq f(y)$. Such a function has a fix point [23]:

Theorem 3.8.1 : KNASTERTARSKI
If $(A, \leq)$ is a complete lattice, and $f : A{\rightarrow}A$ is monotonic with respect to $\leq$, then f has a fix point. Additionally, let P be the set of all fix points of f. These hold:

1. $(P, \leq)$ forms a complete lattice (thus, is not empty).
2. The supremum of P is also a member of P; this is thus the greatest fix point of f.
3. The infimum of P is also a member of P; this is the least fix point of f.

2 and 3 are actually corollaries of 1. □

If A is a set, let $\mathcal{P}(A)$ denote the power-set over A. Notice that $(\mathcal{P}(A), \subseteq)$ forms a complete lattice. Also recall that a (state) predicate can be seen as a set of states. So, **wlp** is a function of type $\mathit{Stmt}{\rightarrow}\mathcal{P}(\mathit{state}){\rightarrow}\mathcal{P}(\mathit{state})$. So, by the theorem above, if it is monotonic over its second argument, the equation (3.10) is guaranteed to have a solution.

Consider now the lattice $(\mathcal{P}(A), \subseteq)$. A function $f : \mathcal{P}(A){\rightarrow}\mathcal{P}(A)$ is $\cup$-continous if for every increasing (with respect to $\subseteq$) chain $X_0, X_1, ...$:

$$f(\bigcup_i X_i) = \bigcup_i f(X_i)$$

Similarly, f is $\cap$-continous if for every decreasing chain $X_0, X_1, ...$:

$$f(\bigcap_i X_i) = \bigcap_i f(X_i)$$

You can prove (easily) that if f is $\cup$-continous or $\cap$-continous then it is also monotonic. For such a function we have a stronger result [2], see below. The notation f^i means as follows: for any X, $f^0(X)$ is defined as X; and $f^{i+1} = f(f^i(X))$.

Theorem 3.8.2 :
If $f : \mathcal{P}(A) \rightarrow \mathcal{P}(A)$ is $\cup$-continous, then the least fix-point of f is $\bigcup_{i \leq 0} f^i(\emptyset)$. $\Box$

Theorem 3.8.3 :
If $f : \mathcal{P}(A) \rightarrow \mathcal{P}(A)$ is $\cap$-continous, then the greatest fix-point of f is $\bigcap_{i \leq 0} f^i(A)$. $\Box$

Since $f^0(\emptyset) = \emptyset$, then obviously $f^0(\emptyset) \subseteq f^1(\emptyset)$. You can then inductively prove that $f^i(\emptyset) \subseteq f^{i+1}(\emptyset)$. So, $\bigcup_{i \leq 0 \leq k} f^i(\emptyset)$ is just equal to $f^k(\emptyset)$. So, the following algorithm can be used to calculate the least fix point:

```
Result: The least fix point of $f$
$X := \emptyset$ ;
while $f(X) \neq X$ do
  | $X := f(X)$
end
return $X$
```

Algorithm 1: Least fix point iteration.

Analogously, to calculate the greatest fix point:

```
Result: The greatest fix point of $f$
$W := A$ ;
while $f(W) \neq W$ do
  | $W := f(W)$
end
return $W$
```

Algorithm 2: Greatest fix point iteration.

This gives us a way to calculate the (greatest) solution of (3.10). Notice that A (the initial value of W) is in this case equal to the entire *state*, which corresponds to the predicate `true`. Neither algorithm is guaranteed to terminate though, unless, for example, A is finite.

Using Algorithm 2 actually requires that **wlp** is $\cap$-continous over its post-condition parameter. In a simplified form, the property requires that **wlp** S $(Q_1 \wedge Q_2)$, where $Q_1 \Leftarrow Q_2$, is equal to (**wlp** S Q_1) $\wedge$ (**wlp** S Q_2). We do not prove this, but in any case this is an intuitive property that you would expect from any $\triangleleft$**cp**-type transformer. The **wlp** does have this property [7, 10]; actually it is even universally conjunctive, at least for the basic statement constructs defined so far. Extending with new statement constructs does bring in the requirement to prove that the new constructs maintain the conjunctivity of **wlp**.

Eliminating Loops with Finite Unrolling

Let's first define $[\texttt{while}]^k$ as follows:

1. $[\texttt{while}]^0(g, S) \quad = \quad \texttt{assert}\ \neg g$
2. $[\texttt{while}]^{k+1}(g, S) \quad = \quad \texttt{if}\ g\ \texttt{then}\ \{S\ ;\ [\texttt{while}]^k(g, S)\}\ \texttt{else skip}$

For example, $[\texttt{while}]^1(g, S)$ would expand to `if` g `then` $\{S\ ;\ \texttt{assert}\neg g\}$ `else skip`. The statement $[\texttt{while}]^k$ is the k-unrolling of the loop `while` g `do` S; it is the version of the latter where we require that the loop should terminate in at most k steps. It follows that any Hoare triple specification satisfied by $\texttt{while}^k$ will also be satisfied by the original `while` (but not the other way around). We can choose to define the **wlp** of `while` to be the **wlp** of $[\texttt{while}]^k$ instead, for some chosen k:

1. $\textbf{wlp}\ ([\texttt{while}]^0(g,S))\ Q \quad = \quad \neg g \wedge Q$
2. $\textbf{wlp}\ ([\texttt{while}]^{k+1}(g,S))\ Q \quad = \quad (g \wedge \textbf{wlp}\ S\ (\textbf{wlp}\ ([\texttt{while}]^k(g,S))\ Q))\ \vee\ (\neg g \wedge Q)$

The resulting predicate calculated by the above **wlp** is sound (we will still have property (3.6)). So, if the right hand side in (3.7) can be verified to be valid, then so is the original specification (the left hand side in (3.7)). However, if the right hand side in (3.7) is invalid, this does not necessarily mean that the original specification is also invalid.

Alternatively, we can *assume* (rather than *requiring*) that the loop terminates in at most k iterations. Consider this definition, which only differs with the previous one at the base case, for which we use `assume` instead of `assert`:

1. $\langle\texttt{while}\rangle^0(g,S) \quad = \quad \texttt{assume}\ \neg g$
2. $\langle\texttt{while}\rangle^{k+1}(g,S) \quad = \quad \texttt{if}\ g\ \texttt{then}\ \{S\ ;\ \langle\texttt{while}\rangle^k(g,S)\}\ \texttt{else skip}$

The corresponding **wlp** is:

1. $\textbf{wlp}\ (\langle\texttt{while}\rangle^0(g,S))\ Q \quad = \quad \neg g \Rightarrow Q$
2. $\textbf{wlp}\ (\langle\texttt{while}\rangle^{k+1}(g,S))\ Q \quad = \quad (g \wedge \textbf{wlp}\ S\ (\textbf{wlp}\ (\langle\texttt{while}\rangle^k(g,S))\ Q))\ \vee\ (\neg g \wedge Q)$

Using **wlp** $\langle\texttt{while}\rangle^k$ as the **wlp** of `while` is complete but not sound. That is, we still have the $\Rightarrow$ relation in (3.7), but we lose (3.6). This means, proving that the right hand side of (3.7) is valid does not give us any information on whether the original specification (the left hand side of (3.7)) is valid. However, if the right hand side is invalid, then so is the original specification of the loop.

Dynamic Inference

Another possibility is to 'learn' the invariant from execution logs. To do this we first 'instrument' the program (which means we inject additional code into it), such that the values of the variables in the loop in question are logged, everytime the loop starts an iteration. We then execute the program some number of times, with different inputs, and thus generating logs about the target loop. Suppose you can now construct a whole set of formulas which are candidates invariants (this can potentially a very large set!). We can check which of them are not violated by the logs. These are formulas that can potentially be valid invariants for the loop. We subsequently verify each of them until we find one that satisfies the constraints in (3.8); it will then be the chosen **wlp** of the loop. Daikon [8] is a popular tool people use to learn invariants (but also pre- and post-conditions) from logs.

3.8.4 Program call

So far we have been a bit vague about what a 'program' is. We have been mainly discussing about statements. If we define a 'program' to be a set of computer instructions to do something, then statements are also programs. Let us now use the term *program* to specifically mean parameterized statements. We will describe it with this structure: $Pr(x_1, x_2, ...\mid y_1, y_2, ...)\{\ S\ \}$. Pr is the name of the program, and S is a statement which is the body of this program. It has two sets of parameters: $x_1, x_2, ...$ are input parameters, and $y_1, y_2, ...$ are output/return parameters. When the program finishes, a vector containing return parameters is available so that their values can be inspected. A program that simply returns a single value can be represented by a structure $Pr(x_1, ...|\textbf{return})$.

A program can be executed, by passing concrete values for the parameters it expects. We will also allow programs to be called from other programs. If this is possible, what is then the **wlp** of such a call?

Let us first introduce simultant assignment. For example $x, y := y, x{+}y$ is an assignment that assigns the value of y and $x{+}y$ before the assignment, simultaneously to x and y respectively. Note that the effect is not the same as doing the assignments sequentially, in one order or the other.

$$\textbf{wlp}\ (x, y := e_1, e_2)\ Q \ = \ Q[e_1, e_2/x, y] \tag{3.11}$$

where $Q[e_1, e_2/x, y]$ denotes the expression we would get by simultaneously replacing all free occurrences of x and y with respectively e_1 and e_2. For example, $(x = y)[y, x{+}y/x, y]$ would yield $y = x{+}y$. In comparison, $(x = y)[y/x][x{+}y/y]$ applies the substitution sequentially: first $[y/x]$, then $[x{+}y/y]$; this results in $x{+}y = x{+}y$, or simply `true` after simplification.

Let $Pr(x \mid r_1, r_2)\{\ S\ \}$ be a program with body S. It that takes one parameter, namely x, and returns two values, namely r_1 and r_2. Parameters are always read by value, and the program has no side effect. A call to such a program has this form:

$$y, z := Pr(e)$$

which would simultaneously assign the return parameters r_1 and r_2 to x and y respectively.

Notice that the semantic of this call can be equivalently modelled by the unfolding below. The variables x, r_1, r_2 are assumed to be <u>fresh</u>, else rename them in S to make them fresh.

$$y, z := Pr(e) \quad \equiv \quad \{\texttt{var}\ x, r_1, r_2\ ;\ x := e\ ;\ S\ ;\ y, z := r_1, r_2\} \tag{3.12}$$

This means that we do not need a special formula to express the **wlp** of program calls, since they can just be expressed in terms of the constructs that we already have. Note also that a program with side effect can be expressed with the above kind of programs. For example a program $inc(x)$ that increases the value of x can be represented by the program $inc'(x|r)\{r := x{+}1\}$, and then calling it as $x := inc'(x)$.

The above way of calculating the **wlp** of a program call assumes that we know the program's body. But what if the body S is unknown (or, if we prefer to abstract away from it)? Assume that we at least know that $Pr(x \mid r_1, r_2)$ satisfies the specification below:

$$\{*\ P\ *\}\ Pr(x \mid r_1, r_2)\ \{*\ Q\ *\}$$

We furthermore require that P and Q only contain x, r_1, r_2 as free variables. This can be encoded as the following body:

$$Pr(x \mid r_1, r_2)\{\texttt{assert}\ P\ ;\ \texttt{assume}\ Q\}$$

Since we now have a body, then the above construction (3.12) can again be used to reason about calls to Pr.

3.8.5 Arrays and objects

Let a be an array. For simplicity, let us first assume that it is infinitely large, so $a[i]$ is always defined, for any integer i. Consider the assignment $a[i] := 0$. This assignment is problematical for both the Rule 3.1.4 and the **wlp** defined in Subsection 3.8.1, because it may affect the meaning of e.g. $a[j]$ if i and j evaluate to the same value. But note that you may not know this during the calculation of the **wlp**. 'Evaluating' requires us to be able to look into the actual program state; but in **wlp** we are statically analyzing a program, without executing it.

To deal with this, we treat such an assignment as an assignment that targets the whole array instead:

$$a[i] := e \quad \equiv \quad a := a(i\ \texttt{repby}\ e) \tag{3.13}$$

where $a(i\ \texttt{repby}\ e)$ denotes an array which is identical with a, except at position i, its value is e. The corresponding **wlp** is thus just:

$$\textbf{wlp}\ (a[i] := e)\ Q \quad = \quad Q[a(i\ \texttt{repby}\ e)/a] \tag{3.14}$$

For example **wlp** $(a[i] := 0)$ $(a[i] = a[3])$ is $(a[i] = a[3])[a(i\ \texttt{repby}\ 0)/a]$. Applying the substitution we obtain:

$$\begin{array}{l} a(i\ \texttt{repby}\ 0)[i] \;=\; a(i\ \texttt{repby}\ 0)[3] \\ = \\ 0 \;=\; a(i\ \texttt{repby}\ 0)[3] \\ = \\ 0 \;=\; (3{=}i \rightarrow 0 \mid a[3]) \end{array}$$

Many programming languages are Object Oriented (OO); so you would ask: how can you deal with OO programs? We can extend our Hoare logic and **wlp** calculation natively with OO constructs, e.g. as in the work by Pierik-Boer [17]. This has the advantage that we can more abstractly reason about objects. However, the logic also becomes complicated. Alternatively, we can also try to express OO constructs in terms of the constructs that we already have. Objects will be encoded as e.g. members of arrays; so, this is less abstract. On the other hand, the same Hoare logic and predicate transformers can be used.

Let us introduce a global (and infinite) array H representing the objects heap[6]. Objects are kept and manipulated in this array. We additionally introduce another global variable N, representing the number of objects currently in H. H is a two dimensional array of type:

$$[int][fieldname] \rightarrow value$$

A member of *value* can be either a primitive value such as an integer or a boolean value, or a 'reference' to another object. A reference is represented by an integer; more precisely an index to the first-dimension of the array H. This idea was proposed by Leino et al [15], which later becomes the base of the famous verification tool ESC/Java.

Suppose we have an object o with two fields $f1$ and $f2$. This object will be stored in $H[o]$ such that $o.f1$ denotes $H[i][f1]$, and $o.f2$ denotes $H[i][f2]$.

Below I give the encoding of some basic OO primitives into the constructs that we already have.

1. Assigning to an object variable:

   $$o := e$$

   does not require any special translation; o is just an index to H, and e should thus be an expression that returns an integer.

2. Assigning to an object's field:

   $$o.f := e \quad \equiv \quad H[o][f] := e \tag{3.15}$$

3. Object creation. Suppose the class C has two fields, f_1 and f_2. For simplicity, suppose they are both of type *int*. When a new instance of C is created, these fields will be automatically intialized to 0.

   $$o = \texttt{new}\ C \quad \equiv \quad \{\ H[N][f_1] := 0\ ;\ H[N][f_2] := 0\ ;\ o := N\ ;\ N := N{+}1\ \} \tag{3.16}$$

4. Let $Pr(x \mid r_1, r_2)$ be a method of the class C. This program implicitly has the signature $Pr(this, H, x \mid H', r_1, r_2)$, where *this* represents the target object (an instance of C) on which the method is called, H is the heap before the call, and H' represents the new heap after the call. The call $y, z := o.Pr(x)$ is represented by:

   $$y, z := o.Pr(x) \quad \equiv \quad H, y, z := Pr(o, H, x) \tag{3.17}$$

[6] OO languages like Java store and manage objects in a global structure called 'heap'. Abstractly, for us it is just a set of existing objects. We will also ignore garbage collection in our discussion. So for simplicity, let's assume that objects are never garbage collected.

3.8.6 Extended Static Checking

Many modern programming languages come with type checkers, that check the consistency of the types of the expressions used in programs. Without them, many programs would be much more buggy than they are now. However, not all errors can be found by a type checker. An Extended Static Checker (ESC) is used to statically check a set of generic correctness properties, which would normally be beyond the range of a type checker. An example of such a property is the absence of crash due to division by zero, or due to an access to an array outside its valid indices. An ESC is usually not intended to verify the functionality of a program (e.g. that it should sort an array). In principle, verifying absence of division by zero can be just as hard as verifying a functionality. However, we can expect that in practice the resulting proof obligations would often be simple enough to be proven automatically by a back-end theorem prover.

There may be many possible causes for a program to crash; some maybe impractical to exclude with static checking. For example a program may crash because it calls some OS function, and the function crashes. If no formal specification is known for the function, then the best an ESC can say is that any program that calls the function is unsafe, which of course exagerates. So, to be pragmatic we typically want to limit an ESC to check only for certain types of chrashes (you decide which ones). This does mean that it would then be blind to other types of crashes.

The tool ESC/Java uses a **wlp** transformer to do such static checking. It works essentially as follows. Given a program Pr to check, we will first annotate its body S with assertions expressing conditions to guarantee non-crashing executions of various expressions and substatements in S.

For any sub-statement T in S, we transform it to `assert` α ; T, where α is a sufficient condition for any execution of T to proceed without crashing. For example, an assignment $x := e$ where e contains a sub-expression e_1/e_2, should be transformed to `assert` $e_2 \neq 0$; $x := e$. Simlarly, $a[i] := 0$ should be transformed to `assert` $0 \leq i < \#a$; $a[i] := 0$. Note that such assertions can be automatically inserted.

If S' is the transformed S, then **wlp** S' `true` gives a pre-condition that would guanrantee non-crasching execution of S. If a pre-condition P for Pr is given, absence of crash can be verified by proving $P \Rightarrow$ **wlp** S' `true`.

Notice that the calculation of **wlp** can proceed as usual.

Loops do pose a problem. We cannot expect programmers to annotate their loops with invariants. The fix point algorithm does not require such annotation, but it may not terminate. Loop unfolding is an alternative, in particular the $\langle\texttt{while}\rangle^k$ unfolding, though as remarked earlier in Subsection 3.8.3 it is unsound. In this case, crashes that happen in iterations above k will not be detected.

3.8.7 Security Checking

Imagine a security critical program. In such a program, accessing some variables, or calling some sub-programs may require certain credentials, and some objects can only be accessed by a certain group of users, and similarly for methods. These are security constraints. A violation on such a security constraint will be called a *breach*. Can we use something similar to the ESC approach described above to check absence of breaches?

Let $Users$ be the set of all users of a program. Notice that such a constraint basically just specifies a set of users that is allowed to do a certain thing. We will therefore represent a *credential* as a subset of $Users$. Credentials are ordered by $\subseteq$: if c and d are crendentials, and $c \subseteq d$ then d has the same or 'higher' credential than c. That is, any operation that requires c can be done by a user with credential d. For example, the *root* user would have the entire $Users$ as his credential. For an operation, requiring the credential $\emptyset$ is a non-constraint; this is the same as requiring no credential at all. Notice that $\subseteq$ is only a partial ordering. For example the credential $\{u\}$ and $\{t\}$ are incomparable. An operation that requires the credential $\{u\}$ can be done by a user with credential $\{u, t\}$, but not by a user with just credential $\{t\}$.

More generally, we can talk about credentials in terms of a domain $(\mathcal{C}, \preceq, \bot)$ where $\mathcal{C}$ represents the set of all possible credentials. The relation $\preceq$ is a partial order; $c \preceq d$ expresses that d has at least the same level of credential as c. The $\bot$ is the least element of $\mathcal{C}$ (representing absence of any credential). For example, $(\mathcal{P}(Users), \subseteq, \emptyset)$ is such a domain.

Let us consider a simple setup where only variables can have credential requirements —we can think that some variables represent persistent resources, such as a file or a database, whose access typically needs to be secured.

Every GCL program is now assumed to implicitly have a parameter called *cred* representing the credential of the user that invokes it. We will not concern ourselves with how this *cred* is determined in the first place; it suffices here to assume that it is correctly determined. 'Implicit' here means that the parameter is automatically inserted by our 'hypothetical GCL compiler'. Despite this, in our presentation we will make *cred* explicit nonetheless, so that we can more explicitly talk about it.

In GCL, varaiables can be declared in two places: in a `var`-construct, and as formal parameters of a program. When they are declared, variables can now be annotated with their required credential level, e.g.:

$$\texttt{var}\ x\ c{\preceq}y\ d{\preceq}z\ \texttt{in}\ S\ \texttt{end}$$

The above declares three variables, x, y and z, where and x requires no credential, y requires credential of at least c, and z requires credential of at least d. We will treat such declaration as a syntactic sugar (a short hand) for:

$$\texttt{var}\ x\ c_x\ y\ c_y\ z\ c_z\ \texttt{in}\ c_x := \bot\ ;\ c_y := c\ ;\ c_z = d\ ;\ S\ \texttt{end}$$

New variables C_x, C_y, C_z are introduced to explicitly represent the credential requirements of x, y, and z.

The example below shows the definition and some usage of a GCL program to reset the value of an integer variable if it becomes too large:

$$reset(cred, b \mid d{\preceq}b')\{\ \texttt{if}\ b \geq 10^6\ \texttt{then}\ b' := 0\ \texttt{else}\ b' = b\ \}$$

$$\texttt{if}\ x{>}0\ \texttt{then}\ y := reset(cred, y)\ \texttt{else}\ z := reset(cred, z)$$

Quite obviously, using *reset* would require a credential of at least d[7]. But note that a call $z := reset(cred, z)$, for example, also puts a constraint on C_z.

Credential declarations as in the example above induces constraints on when it is valid to read or write to certain variables. These are as follows:

1. Any access read or write to a variable y now requires that $C_x \preceq cred$.
2. We will allow information to flow between variables with the same credential. So, if y and y' have the same credential, then e.g. an assignment $y := y'{+}1$ is allowed.

   How about information flow between variables of different credentials? This can be risky. Allowing information to flow from variables of a ower credential to those with a higher credential may raise the concern of contamination. This may allow a user with a low credential to force a program to do something that is supposed to require a higher credential.

   Allowing the opposite flow may raise the concern of information leak, which may allow a user with a low credential to get information that is supposedly only available with a higher credential.

   On the other hand, forbidding one flow or the other may in some situations be too restrictive. For example, suppose z is a high credential variable and k is a low credential variable. An assignment e.g. $z := z{+}k$ is arguably safe if at that moment k holds no 'dangerous' value.

[7]The program header gives a hint for this, but actually this depends on the program's actually execution. In this example, any execution of *reset* will access b'. If hypothetically there is an execution that does not do that, then this execution would not need any credential at all.

We can allow the declaration of a variable to be extended to also specify the kind of information flows allowed for that variable. This would allow different flow policies to be expressed for different variables. However, my purpose here is just to demonstrate that a predicate transformer-based approach can be used for verifying security-related constraints. So, for simplicity in this book will assume that for all variables, contamination is not allowed. Leakage is tolerated. An example of where such a setup is still useful is the control system of my heater. I don't mind if you peek into its state, but I absolutely do not want an unauthorized user to be able to override the control.

We consider a program to be safe if all its security-related constraints, as defined above, are not broken. To verify this, every read and write to a variable is first extended with the corresponding credential checks. For example, the assignment:

$x := y - x$

is transformed to:

```
assert $C_x \preceq cred \wedge C_y \preceq cred$ ;
assert $C_x \preceq C_y$ ;
$x := y - x$
```

As another example, consider again the previous example, with the first call to *reset* highlighted:

```
if $x>0$ then $\boxed{y := reset(cred, y)}$ else $z := reset(cred, z)$
```

Expanding the first call gives the code below[8]; we again highlight some fragments:

```
if $\boxed{x>0}$
then var $b\ C_b\ b'\ C_{b'}$ in
        $C_b := \bot$ ; $C_{b'} := d$ ;
        $b := y$ ;
        if $b \geq 10^6$ then $b' := 0$ else $b' = b$ ;
        $y := b'$
        end
else $z := reset(cred, z)$
```

Inserting the security constrains generated by the boxed parts results in the code below. Since each variable access generates at least one constraint, the code will indeed become quite verbose; but the growth is still linear.

```
assert $C_x \preceq cred$ ;
if $\boxed{x>0}$
then var $b\ C_b\ b'\ C_{b'}$ in
        $C_b := \bot$ ; $C_{b'} := d$ ;
          assert $C_b \preceq cred \wedge C_y \preceq cred$ ;
          assert $C_b \preceq C_y$ ;
          $b := y$ ;
          assert $C_b \preceq cred$ ;
          if $b \geq 10^6$
            then {assert $C_{b'} \preceq cred$ ; $b' := 0$}
            else {  assert $C_{b'} \preceq cred \wedge C_b \preceq cred$ ;
                    assert $C_{b'} \preceq C_b$ ;
                    $b' = b$ }
          assert $C_y \preceq cred \wedge C_{b'} \preceq cred$ ;
          assert $C_y \preceq C_{b'}$ ;
          $y := b'$
        end
else $z := reset(cred, z)$
```

[8]To reduce verbosity, we do not expand the handling of the parameter *cred*, treating it instead as a global variable.

To verify the safety of the program, we can now calculate the **wlp**. If we are only interested in verifying the security constraints (and not so much in whatever the actual post-condition of the above program), we can just use **true** as the post-condition. Let's call the above statement T; so we are interested in calculating **wlp** T **true**. If T has some given pre-condition P, then the final formula to verify is:

$$P \Rightarrow \mathbf{wlp}\ T\ \mathbf{true} \tag{3.18}$$

Despite the verbosity above, the resulting **wlp** can be simplified quite a lot, and you will end up with this:

$$\begin{array}{ll} & C_x \preceq cred \\ \wedge & (x{>}0 \Rightarrow (C_y \preceq cred \wedge d \preceq cred \wedge (y{<}10^6 \Rightarrow d \preceq \bot) \wedge C_y \preceq d)) \\ \wedge & (x{\leq}0 \Rightarrow ...) \end{array}$$

The first conjunct says that the user's credential should be enough to be able to access the variable x. This comes originally from the guard of the top-level `if` that indeed mentioned x.

The third conjunct comes from the second call of *reset*; we will skip its dicussion since it is just analogous to that of the first call. The second conjunct comes from this call `if` $x{>}0$ `then` $y := reset(cred, y)$. The conjunct takes the form of an implication $x{>}0 \Rightarrow ...$, which means that the constraints listed after the implication only need to be valid when $x{>}0$. One of the constraints requires that the user's credential should be enough to access y, and another requires that the user's credential has to be at least d. These are as expected, since the call accesses y, and the formal parameter b' of *reset* requires the credential d.

However, if $d \neq \bot$ we may actually run into a problem, namely if $reset(cred, y)$ is called with $y{<}10^6$. The corresponding condition $d \preceq \bot$ would not even be satisfiable. So basically, the example code is problematic. The following change in the definition of *reset* can solve that:

$$reset(cred, \boxed{d \preceq b} \mid d \preceq b')\{\ \texttt{if}\ b \geq 10^6\ \texttt{then}\ b' := 0\ \texttt{else}\ b' = b\ \}$$

With the above change, the resulting **wlp** woud be:

$$\begin{array}{ll} & C_x \preceq cred \\ \wedge & (x{>}0 \Rightarrow (C_y \preceq cred \wedge d \preceq cred \wedge d \preceq C_y \wedge C_y \preceq d)) \\ \wedge & (x{\leq}0 \Rightarrow ...) \end{array}$$

Whether this **wlp** is now 'ok' still depends on the given pre-condition P in (3.18); but at least the above **wlp** is satisfiable[9].

Unauthorized use of resource

Imagine that you bought a mobile app in an app store. The app said that it will not use the Internet, so you agreed to install it into your smartphone. A rogue app may however still try to use the Internet behind your back, and thus cost you your Internet's quota.

We can extend the above approach to also check for unauthorized use of a resource. We can model resources with objects. Operations on them are can be modelled as methods. Furthermore, we want to be able to express the credential requirement for invoking each operations. For example, suppose f represents a file as a resource. Possible operations on a file are read and write, modelled by the methods $f.read(-|x')$ and $f.write(x|-)$ (I only show their headers; so, *read* takes no input parameter, and has one output parameter; *write* is the other way around). To express the

[9] You may notice that when $x{>}0$, the above predicate requires that C_y should be exactly d. In other words, the patched *reset* can only be used to reset the value of a variable whose credential requirement is d (it cannot be used to reset a variable y' whose credential requirement is higher, or even lower, than d). This may seem restrictive, as you may expect that calling it to reset y' with a lower credential should be ok. This is because our GCL does not have a concept of reference (pointer) passing. Reseting y is modelled by copying it to a local variable b, then we reset b, then we copy b back to y. So there is information flow from y to b, as well as from b back to y. Because we have adopted the policy that forbids information to flow from a lower credential variable to that of a higher credential, the net effect is that this creates a constraint that C_y has to be exactly equal to C_b.

credential requirements for these methods, we can add fake parameters to the methods (they are not actually used for anything by the methods) and specify the required credentials, e.g.:

$$\begin{array}{ll} f.read & (c{\preceq}fake,\ -\mid x') \\ f.write & (d{\preceq}fake,\ x\mid -) \end{array}$$

When calling the methods, the value passed to $fake$ is irrelevant (so you can pass any value). However, its usage will induce the corresponding security constraint. So, calling *read* now requires a credential of at least c, whereas calling *write* requires d. Now we can proceed with the same verification procedure as before.

Input injection attack

In real programs, many tasks are actually carried out by external components, e.g. a method from an external library, an OS function, a query to a back-end DB engine, a service, etc. Some of these tasks may be security critical, e.g. reading or writing to a file, or to a database. A well known form of attack is input injection. Consider a program $Pr(x)$ that takes input x from the user (which could be an input that user can submit through Internet). Imagine that Pr internally interacts with some back-end database. As common now, a database has its own database engine. Pr does not directly work on the data in the database. Instead, each time it wants to do something with the database, it sends some (possibly complex) expression to it, which the database engine would interpret. Ultimately, this expression depends on the value of x that the user gave to Pr. It may be possible to construct a certain value of x, such that the resulting expression sent to the database engine would do something totally unintended, such as retrieving the whole content of the database, rather that just some intended small fraction of it.

An example of such attack is the notorious SQL injection attack. Imagine that Pr above construct the following string to be sent and executed by a back-end SQL database engine:

```
query := "SELECT * FROM users WHERE name ='" + userName + "';"
return query
```

which is supposed to return the user's own personal data, given his username. Suppose that this username is the input x of Pr, a malicious user can send the following string as username: `"' or '1'='1"`, causing Pr to construct this query string:

```
SELECT * FROM users WHERE name ='' or '1'='1''
```

with the effect that it will fetch the data of *all* users, and Pr then happily returns then all to its caller.

To defend against input injection, Pr should not blindly trust user inputs. It should first inspect and clean these inputs from any pattern that looks dangerous. This process is called *sanitizing* inputs.

Proving the soundness of such process is very hard and will require the formalization of what 'dangerous' means. We will not go into that. Here I will only discuss a pragmatic approach where we *assume* that the functions used to sanitize are correct. So the only thing left to verify is that we do not forget to do that, when it matters.

To model sanitizing in our formalism, we proceed as follows. Firstly, parameters representing user inputs are assumed to have $\perp$ credential. So, their values cannot be used to update variables with non-$\perp$ credentials. We introduce a function $sanitize(u, c)$ which will evaluate the expression u, then return the sanitized version of the result. Sanitizing can be as simple as removing all quote-like characters in a string, or it can be quite complicated. We will abstract away from such details by modelling *sanitize* as an *uninterpreted* function (it is a function, but we don't know its definition). Whatever the result of $sanitize(u, c)$, it is considered to safe enough to flow into variables with credential requirements of *at most* c. Other than this, we now know nothing about the result of *sanitize*.

For example, consider the assignment $y := x{+}1$, where x is user input. As before, this would be transformed to (keeping in mind that $C_x = \bot$):

```
assert $C_y \preceq cred \ \wedge \ C_y \preceq \bot$ ;
$y := x{+}1$
```

Assuming C_y is non-$\bot$, the second conjunct in `assert` would be invalid (it is even unsatisfiable). In terms of verification, this means that our logic will discover that we are trying to pass information from an unsecure to a secure veriable.

It is a good thing that the logic can detect the vulnerability! Now, let us now proceed to patching the statement so as to remove the vulnerability. Let us change the above assignment to:

$$y := sanitize(x, c){+}1$$

This would be transformed to:

```
assert $C_y \preceq cred \ \wedge \ C_y \preceq c$ ;
$y := sanitize(x, c){+}1$
```

If this is all the program whose security we have to verify, then the **wlp** (with respect to **true** as the post-condition) is just the asserted constraints above. We do not in this case suffer from the fact the abstraction that *sanitize* introduces.

But suppose this is the program to verify:

```
assert $C_y \preceq cred \ \wedge \ C_y \preceq c$ ;
$y := sanitize(x, c){+}1$ ;
if $y{=}0$ then assert $C_x \preceq cred$ ; ...
else ...
```

The resulting **wlp** will contain $sanitize(s, c){+}1 = 0 \ \Rightarrow \ C_x \preceq cred$. The validity of this fragment may depend on the result of *sanitize*, of which we have no information. Whether this is actually problematic still depends on the actual pre-condition of the program. If this pre-condition implies $C_x \preceq cred$, then we are good. If this pre-condition says something like:

$$y = 0 \Rightarrow \ C_x \preceq cred$$

There there is no way for us to prove:

$$sanitize(s, c){+}1 = 0 \ \Rightarrow \ C_x \preceq cred$$

without more information on what *sanitize* does.

Dealing with Composite Data

So far, our credential system can be used to guard access to program variables. Suppose we have a variable x that contains an array. Many languages allow references (pointers) to a composite structure such as an array or an object to be passed around. In such a setup x does not literally contain the array, but only a reference to the array. But then, theye may be other variables, so-called aliases, that holds a reference pointing to the same array. Variables of primitive type, e.g. *int* or *String*, are not typically allowed to have aliases.

It a setup that allows aliases, it makes sense that objects (and arrays) should be allowed to state their own credential requirement, to make sure that multiple variables pointing to them respect this requirement. Here, let us just limit ourselves to objects. We extend objects, such that every object will have a field called *cred*, expressing the minimum credential needed to access the object.

For example, suppose o is a variable pointing to an object, and $o.f$ is a primitive-typed field. The assignment $o.f := 0$ requires access to the object that o points to, and is then be transformed to:

```
assert $C_o \preceq cred \wedge o.cred \preceq cred$ ; $o.f := 0$
```

As another example, consider $o.f := n.f{+}1$, where $n.f$ is a primitive-types field. The assignment requires access to the objects o and n. So the assignment is then transformed to:

> `assert` $C_o \preceq cred \;\wedge\; C_n \preceq cred \;\wedge\; C_o \preceq C_n$;
> `assert` $o.cred \preceq cred \;\wedge\; n.cred \preceq cred \;\wedge\; o.cred \preceq n.cred$;
> $o.f := n.f{+}1$

Accessing an object through its method should also be guarded to have a proper credential. For example, a call $o.m()$ should then be transformed to:

> `assert` $o.cred \preceq cred$; $o.m()$

This however raises another issue. In an OO language, it is quite common to build an object from smaller ones. The bigger object is often called a *container*, and its subobjects are called *components*. It seems reasonable to require that a component can only be accessed with a credential that is at least equal to its own, and furthermore at least equal to the credential requirements of its container. Additionally, in our particular setup we have earlier decided to adopt the policy that requires information to only flow from high credential 'items' to low credential ones (and forbid the opposite flow). Effectively, this means that in our setup a container and all objects it contains should have the same credential requirement. One way to verify this is to keep track the components of each container, and the containers of each component. But this is rather complicated.

An alternative to that is to require that we can only add an object o as a component of n if $n.cred = o.cred$. This makes sure that every access to o will also respect the credential requirement of the container n[10].

For example, suppose $n.next$ is a reference-typed field. Consider this assignment $n.next := o$. First, this requires access to the the object n, so this should be properly guarded. Second, the assignment links o to become a 'child' of n. In other words, o becomes a component of the container n; this should also be guarded. The assignment should be transformed to:

> `assert` $C_n \preceq cred \;\wedge\; C_o \preceq cred \;\wedge\; C_n \preceq C_o$;
> `assert` $n.cred \preceq cred \;\wedge\; n.cred = o.cred$;
> $n.next := o$

3.8.8 Dealing with Exception

We have postponed the discussion about exception. From the verification perspective, it is probably the most annoying construct. In e.g. Leino etal [15], they extend the concept of post-condition. It is now a pair: $\{*\, P \,*\}\ S\ \{*\, N, E \,*\}$, to mean that upon normal termination, P should realize the post-condition N. However, if it terminates due to an exception, then it should realize the post-condition E. In favor of simplicity, we will not do so here. We will just use the traditional single post-condition.

Exception is thrown by the statement `raise`. The intended semantic is as usual: the execution would jump to the closest enclosing handler, if there is one. Else the program terminates. This however complicates the formulation of the **wlp**. We prefer to define **wlp** in a syntax driven way, because such a definition is nice, and can be directly mapped to an implementation. However, having jumps screws this up. Therefore, we will take a rather indirect approach. Let us call the version of GCL without exception GCLnormal. We will try to define GCL in terms of GCLnormal. If we can do this, then we can simply use the **wlp** as we already have; no need to change anything. This however means that `raise` is not going to be interpreted as a direct jump, since there is no such jump in GCLnormal.

We will introduce a global boolean variable *exc*. A state where *exc* is true is called an *exceptional* state, else it is called a normal state. The statement `raise` sets the variable to true.

[10]However, this approach assumes that we can find all such additions. For example, if an external program, which is beyond the reach of our logic can put o inside n, the potential vulnerability of this is not visible to our logic.

Sequential composition is now changed, such that at each ';' it checks if the state is normal. If so it executes the next statement, as usual. But if the state is exceptional, it skips the next statement. The next ';' does the same. The statement $S!H$ simply executes S sequentially, with the above modified meaning of ';'. When it reaches the !, the program again checks the state. If the state is exceptional, it resets *exc* to false, and continues with H. Else it skips over H. The resulting effect of the execution is the same with that of the jump semantic.

The full translation from GCL to GCLnormal is given below.

1. We assume that when a program is called, it always starts in a normal state. The declaration of a GCL program $Pr(x \mid r)\ S$ is thus treated as the declaration $Pr(x \mid r)\{\texttt{assume}\ \neg exc\ ;\ S\}$ in GCLnormal.

2. The statement *raise* is translated to $exc := \texttt{true}$ in GCLnormal.

3. The construct $S!H$ (do S, with H as the handler if S throws an exception) is treated as this in GCLnormal:

   $S; \texttt{if}\ exc\ \texttt{then}\ \{exc := \texttt{false}\ ;\ H\}\ \texttt{else skip}$

4. The composition $S_1; S_2$ is treated as this in GCLnormal:

   $S_1; \texttt{if}\ exc\ \texttt{then skip else}\ S_2$

   Effectively this means that if $S_1; S_2; ...$ is a sequence of statements, and S_k is the first that throws an exception, the execution will skip over $S_{k+1}; S_{k+2}; ....$

   The annoying thing with this is that every ';' now doubles the size of the formula. Since a program usually contains a lot of uses of the ';' operator, this blows up the resulting pre-condition. What may still be exploited is the fact that in GCL, only `raise` can throw an exception. So, we can statically inspect whether S_1 above contains a `raise`, or call to a program that may transitively trigger the execution of `raise`. If this is not the case, we do not need to expand $S_1; S_2$ as above. It can remain as it is.

5. The loop `while` g `do` S is treated as this in GCLnormal:

   $\texttt{while}\ g \wedge \neg exc\ \texttt{do}\ S$

   So, if S throws an exception somewhere in its middle, firstly, the rest of S will not be executed. Then, as the control returns to the guard g, due to the extension of the guard to $g \wedge \neg exc$ the loop will then stops, as it should.

6. Other constructs are not changed.

Recall that in GCL, `raise` is the only construct that can throw an exception. Expressions that in a real programming language may throw an exception should therefore be encoded explicitly. For example, the assignment $x := x/y$ should be encoded as:

$\texttt{if}\ y = 0\ \texttt{then raise else skip}\ ;\ x := x/y$

Similarly, $a[i] = 0$ should be encoded as:

$\texttt{if}\ i < 0 \vee i{\geq}\#a\ \texttt{then raise else skip}\ ;\ a[i] = 0$

These encoding should be done before the transformation to GCLnormal. Conditionals and loops may require more elaborate encoding. For example:

$\texttt{while}\ 1/y > 0\ \texttt{do}\ S$

should be encoded as:

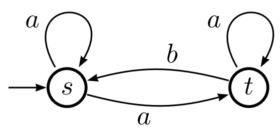
$\texttt{if}\ y = 0\ \texttt{then raise else skip}\ ;$
$\texttt{while}\ 1/y > 0\ \texttt{do}\ \{\ S\ ;\ \texttt{if}\ y = 0\ \texttt{then raise else skip}\ \}$

Chapter 4

Model Checking

4.1 Automata

Figure 4.1 shows some an example of a *finite state automaton* (FSA), also known as finite state machine (FSM). Such an automaton is often used to (abstractly) model a program. The nodes in the automaton are the automaton's states, and the arrows depict how the automaton can go from one state to another. They are also called *transitions* or *actions.* The short arrow pointing to s (on the left) is used to denote that s is the automaton starting/initial state. As the name suggests, these automata have finite number of states.



$$\begin{tikzcd} \arrow[r] & s \arrow[loop above, "a"] \arrow[r, bend right, "a"'] & t \arrow[loop above, "a"] \arrow[l, bend right, "b"'] \end{tikzcd}$$

Figure 4.1: *A finite state automaton*

Definition 4.1.1 : Automaton
Let us define an *automaton* M as a tuple (S, I, Σ, R) where:

- S is the set of states. I is a non-empty subset of S, specifying M's possible initial states.
- Σ are the set of actions of M. Every arrow in M is labelled with one action.
- $R : S \to \Sigma \to Pow(S)$ describes the arrows. If s is a state, and a is an action, $R(s, a)$ is the set of possible next-states if we execution the action a on the state s.

M is an FSA, if the set S is finite. $\Box$

An *execution* of M is a path through M that starts in an initial state. A path is a sequence of connected arrows in the automaton. It can be finite or infinite. The i-th action of the execution is the action that label its i-th arrow. The i-th state in the execution is the starting state of the i-th arrow in the sequence. If the execution is finite, its last state is the end-state of its last arrow.

If τ is an execution, we will use the notation τ_i or $state(\tau, i)$ to denote it's i-th state, and τ^i or $act(\tau, i)$ to denote it's i-th action.

Sometimes we are more interested in the sequence of actions taken along an execution. We will call this sequence *sentence*; sometimes it is also called *trace*. So, if τ is an execution, its induced sentence is the sequence $\tau^0, \tau^1, ...$ We say that s is a sentence of M if there is an execution of M that induces it.

Sometimes we are more interested in the sequence of states along the execution. Rather than inventing yet another name, let us also just call this sequence of states '*execution*'. It should be clear from the context which one is meant.

In the literature you will find many variations of the above definition; figure 4.2 shows some such variations.

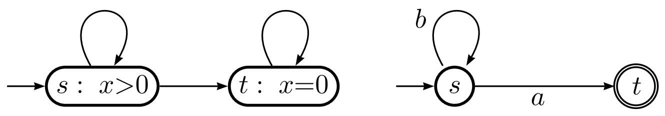


Figure 4.2: *Finite state automata*

The one on the left has no label/decoration on the arrows, but has labels on the states instead. The one on the right has a so-called 'accepting state', marked with the double-lined border.

Let's call a state with no successor a *terminal state*: no further execution is possible from such a state. Accepting and terminal states are not the same concept. Accepting states are often used to express certain goals; they may coincide with terminal states, but not necessarily so.

Depending on what aspects you need to capture in your FSAs you may or may not need such extensions. Sometimes we only want to consider finite executions, sometimes only infinite executions, depending on your modeling purpose. If the automaton has a concept of accepting states (as the right automaton in Figure 4.2), we may only want to consider executions that end in an accepting states.

The right automaton in Figure 4.2 is *deterministic*, because from any state u, the action α uniquely determines what the next state is (in other words, $R(u, \alpha)$ has at most one state). Otherwise, the automaton is *non-deterministic*. The left automaton in Figure 4.1 can be called non-deterministic.

4.2 Some Basic Operations on Automata

4.2.1 Intersection of automata

We can define the intersection of two automata M and N simply by taking the intersection of the set of arrows of both automata. Whether this definition is good depends of course on our purpose: what do we want to do with it?

For our purpose later, we will take a bit more complicated concept of intersection. The idea is that $M \cap N$ should be an automaton whose sentences are exactly those sentences allowed by both M and N.

The idea is to construct $M \cap N$ by executing both M and N. We will keep track where we are (in which states we are) in M and in N; and we will do a transition labelled with an action a, only if this action is possible in both M and N.

Definition 4.2.1 : INTERSECTION
Let $M = (S_1, I_1, \Sigma_1, R_1)$ and $N = (S_2, I_2, \Sigma_2, R_2)$. We define $M \cap N = (S, I, \Sigma, R)$ where:

- $S = S_1 \times S_2$. So, the states of M are pairs (s, t). This is how we keep track of the states of M and N; s is where we are in M; and t is where we are in N.
- $I = \{(s_0, t_0) \mid s_0 \in I_1 \wedge t_0 \in I_2\}$
- $\Sigma = \Sigma_1 \cap \Sigma_2$
- R is such that $(s', t') \in R((s, t), a)$ if and only if $s' \in R_1(s, a)$ and $t' \in R_2(t, a)$.

□

$M \cap N$ is also called the automaton obtained by 'lock-step execution' of M and N.

4.2.2 Interleaving of automata

Let M and N be two automata with no common action. The interleaving of them, denoted by $M||N$ is the automaton whose execution is obtained by interleaving the arrows of M and N. $M||N$ is also called the product automaton of M and N.

$M||N$ represents a composite system, obtained by executing M and N in parallel. We do assume here that they operate on their own private states, and that they can do their actions independently (no need to synchronize). The states of $M||N$ are formed by tupling the states of M and N.

Definition 4.2.2 : INTERLEAVING
Let $M = (S_1, I_1, \Sigma_1, R_1)$ and $N = (S_2, I_2, \Sigma_2, R_2)$, such that they have no common action. We define $M||N = (S, I, \Sigma, R)$ where:

- $S = S_1 \times S_2$. So, each state of $M||N$ is a pair (s, t). Think this as a joint state, where the s component tells us what the state of M is within the composite $M||N$, and similarly the t component tells us the state of N.
- $I \;=\; \{(s,t) \mid (s,t) \in I_1 \times I_2\}$
- $\Sigma \;=\; \Sigma_1 \cup \Sigma_2$
- R is such that $R((s,t),a) \;=\; \{(s',t) \mid s' \in R_1(s,a)\} \;\cup\; \{(s,t') \mid t' \in R_2(t,a)\}$

□

Projecting the sentences of $M||N$ to Σ_1 will give us all the sentences of M; projecting them to Σ_2 gives us the sentences of N.

Synchronized actions

A variations of the above modeling of parallel composition is when M and N share some actions. We first need to decide how such a common action is executed in a parallel execution of M and N. One possibility, which is quite common in various modelling approaches, is to consider a common action as an action that must be executed *together*. So if a is a common action, the composite $M||N$ can make a transition:

$$(s,t) \quad \xrightarrow{a} \quad (s',t')$$

if and only if M can go from s to s', with the action a, *and* if N too can go from t to t' with the action a. This is also called *synchronous* execution of a.

You need to adapt the definition of interleaving $M||N$ accordingly.

When the automata operate on a common state space

Another variation is if M and N actually operate on the same set of states, rather than on their respective private states. So, $S_1 = S_2$. In this case we should take $S = S_1$ as $M||N$'s set of states; thus not $S_1 \times S_2$. The arrows of $M||N$ are then either the arrows of M or the arrows of N, labelled by non-common actions, or synchronous transitions by both M and N over their common actions.

4.3 Temporal Properties

We will introduce a certain type of properties over executions, called 'temporal' properties. To discuss about this, it is convenient to decorate the states of our FSAs with more information. We won't need labels on the arrows. So let us first introduce the type of FSAs called Kripke structures.

A real program P operates on variables. The program's *concrete state* is determined by the values of its variables. When we model the program with an automaton M, we basically map P's concrete states to M's states. This mapping can be one-to-one. That is, each P's concrete state is mapped to a unique state in M. Such an M will have a large number of states (though this number may still be finite).

We can also choose to model more abstractly, e.g. by only specifying a fixed set of properties over P's concrete states. We can query which properties hold or do not hold on a given state in the model M, but we will have no information on properties which were not included in the model. For describing this kind of models we use a variant of automaton called *Kripke structure.*

Definition 4.3.1 : KRIPKE STRUCTURE
A Kripke structure K is a finite state automaton, described by a tuple $(S, I, R, Prop, V)$ where:

- S is a finite set of states, with I as the initial states.
- $R : S \to Pow(S)$ describes the arrows. If s is a state in S, the $R(s)$ gives the set of all possible next-states.

  The notation $Pow(S)$ denotes the set of all subsets of S. It is sometimes also denoted as 2^S.
- $Prop$ is a set of 'atomic' propositions. Each describes a property of K's states.
- $V : S \to Pow(Prop)$ is called a *labelling function.* If s is a state in K, $V(s)$ is the set of all propositions that hold in s.

  Furthermore, we take the convention that if $p \notin V(s)$ then p does *not* hold in s.

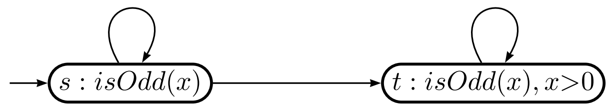
  The labelling is assumed to be consistent. That is, for every state s, the conjunction:

$$\bigwedge_{p \in V(s)} p \quad \wedge \quad \bigwedge_{q \notin V(s)} \neg q$$

  should be satisfiable (in other words, the conjuction should not be empty/contradictive).

□

Example:



We have above two states, named s and t. We have two propositions, $isOdd(x)$ and $x{>}0$. We can see above that $isOdd(x)$ holds in s, but $x{>}0$ does not, whereas in state t both propositions hold. Note that these propositions are 'independent' according to the definition above.

Formally, the R of the above Kripke structure is:

$$\begin{array}{lll} s & \mapsto & \{s, t\} \\ t & \mapsto & \{t\} \end{array}$$

And the V:

$$\begin{array}{lll} s & \mapsto & \{isOdd(x)\} \\ t & \mapsto & \{isOdd(x), x{>}0\} \end{array}$$

A Kripke structure is *non-deterministic*, if there is state that has multiple next-states. The above example is non-deterministic.

The arrows in a Kripke structure are not labelled, and the structure has no concept of acceptance state either. States with no successor implicitly model terminal states.

Checking whether a proposition p holds in some or all states of a give Krike structure is straight forward: we can just iterate over all states and check if p decorates them. We have afterall only finite number of states.

Checking that a certain property (e.g. that p will occur infinitely many often) hold on all executions is different. Although we have finite number of states, the number of executions the automaton generates may be infinite (as the automa Figure 4.1). So, simply enumerating the executions will not work. Furthermore, some executions may be infinite. We will return to this issue later.

A *temporal property* is a 'timing' dependent property. Usually, relative timing is meant, as opposed to real time properties. Abstractly it can be characterized as a property over the executions of an automaton. Examples of such a property are:

- Whenever R receives a value through a channel c, the value it receives is never 0.
- $S||R$ will not deadlock.

In contrast, Hoare triple only specifies a relation between the initial and end-state of an execution. In that respect, Hoare triple is just a special case of temporal properties. However, most temporal properties cannot be expressed with Hoare triples.

One way to specify temporal properties is by using *Linear Temporal Logic* (LTL). It provides a number of operators to express temporal properties, e.g. $p \rightarrow \Diamond q$ means that if p holds initially, then eventually q will hold.

An LTL *formula* is an expression with the syntax below; ϕ, ψ range over LTL formulas, and p ranges over atomic propositions.

$$\begin{array}{lll} \phi & ::= & p, \text{ where } p \text{ is an atomic proposition (state predicate)} \\ & ::= & \neg\phi \;\mid\; \phi \wedge \psi \;\mid\; \mathbf{X}\,\phi \;\mid\; \phi\,\mathbf{U}\,\psi \end{array} \tag{4.1}$$

Intuitively, $\mathbf{X}\ \phi$ means that ϕ holds on the next state. Whereas $\phi\ \mathbf{U}\ \psi$ means that either ψ holds now, or in the future. If it holds in the future, then from now until the point just before it holds, ϕ holds.

Notice that the syntax allows you to nest temporal operators; e.g. you can write: $p\ \mathbf{U}\ (q\ \mathbf{U}\ r)$.

Additionally, we will also introduce the following derived operators:

- Disjunction and implication:

$$\begin{array}{lll} \phi \vee \psi & = & \neg(\neg\phi \wedge \neg\psi) \\ \phi \rightarrow \psi & = & \neg\phi \vee \psi \end{array}$$

- Eventually ϕ:

$$\Diamond\phi \;=\; \texttt{true}\ \mathbf{U}\ \phi$$

- Always ϕ:

$$\Box\phi \;=\; \neg\Diamond\neg\phi$$

- ϕ weak-until ψ; which means either ϕ holds forever, or we switch over to ψ (but we don't have to switch over):

$$\phi\ \mathbf{W}\ \psi \;=\; (\Box\phi) \vee (\phi\ \mathbf{U}\ \psi)$$

The operator is also called 'unless'.

- ϕ releases ψ; which means either ψ holds forever, or at some point it is released by $\phi \wedge \psi$:

  $$\phi \mathbf{R} \psi \quad = \quad \psi \mathbf{W} (\phi \wedge \psi)$$

Some examples of properties you can express with LTL:

- $\Box(p \rightarrow \Diamond q)$: whenever p holds, then eventually q will hold.
- $p \mathbf{U} (q \mathbf{U} r)$: we start with a possibly empty duration where p holds constantly, followed immediately by a (also possibly empty) duration where q holds constantly, which will be ended by a point where r holds.
- $\Diamond\Box p$: eventually we will come to p, afterwhich we will remain in p (eventually, we stabilize into p).

To simplify our discussion, we will only define the semantics of LTL operators with respect to infinite executions. If the given automaton contains a terminal state, and we can add a self-looping arrow on that state, and thus making all of its executions infinite[1]. We can then add a label that only holds on the terminal state, and thus can recover the ability to specify properties on the terminal state by expressing it through this unique label.

We will model the target system with a Kripke structure $M = (S, s_0, R, \mathit{Prop}, V)$, with no terminal state. If the system has concurrent components, and you have modelled each component with its own automaton, we assume that M is the combined automaton of all those components, e.g. using the interleaving operation from Section 4.2.

An *execution* τ of a Kripke structure is as defined generically in Section 4.2, except that the arrows of a Kripke structure has no label. This τ induces a sequence of propositions that hold along the execution, which is the sequence:

$$V(\tau_0), V(\tau_1), ...$$

We will call the latter *abstract execution*, or just *execution* if it is obvious from the context which one is meant.

4.3.1 Valid Property

We will write $M \models \phi$ to denote that the property ϕ is a valid property of the Kripke structure M. This means that it is valid on all abstract executions of M.

Semantically, an LTL property is a predicate over (abstract) executions. If Π is an abstract execution of M, we write $\Pi \models \phi$ to mean that ϕ is a valid property on Π.

Note that since an execution is infinite, its sufix is also infinite. We write $\Pi, i \models \phi$ to mean that ϕ is a valid property of the sufix of Π, starting from its i-th element. So:

$$\Pi \models \phi \quad = \quad \Pi, 0 \models \phi$$

Let $\Pi(i)$ denotes the i-th element of the sequence Π. Note that this is a set of proposition (from M's *Prop*). Now we can define the meaning of our LTL formulas:

1. If p is an atomic proposition (from *Prop*), $\Pi, i \models p$ means that p holds on Π's i-th state. So:

   $$\Pi, i \models p \quad = \quad p \in V(\Pi(i))$$

2. $\neg\phi$ is valid on Π, i, if its negation is not valid. So:

   $$\Pi, i \models \neg\phi \quad = \quad \text{not } (\Pi, i \models \phi)$$

   E.g. in the case of atomic proposition p, then:

   $$\Pi, i \models \neg p \quad = \quad p \notin V(\Pi(i))$$

[1] In literature this is also called: extending executions with 'stuttering'.

3. $\phi \wedge \psi$ is valid on Π, i if each is individually valid. So:

$$\Pi, i \models \phi \wedge \psi \quad = \quad (\Pi, i \models \phi) \text{ and } (\Pi, i \models \psi)$$

4. $\mathbf{X}\ \phi$ is valid on Π, i if ϕ holds from the next state:

$$\Pi, i \models \mathbf{X}\ \phi \quad = \quad \Pi, i{+}1 \models \phi$$

5. $\phi\ \mathbf{U}\ \psi$ is valid on Π, i if at some time in the future of i, ψ holds, and in the mean time ϕ holds:

$$\begin{array}{lcl} \Pi, i \models \phi\ \mathbf{U}\ \psi & = & \text{there is a } j \geq i, \text{ such that } \Pi, j \models \psi \\ & & \textbf{and } (\forall h : i{\leq}h{<}j : \Pi, h \models \phi) \end{array}$$

Below are some well known equivalence between formulas [3]. The equivalence $\phi = \psi$ below means that for any Π, $\Pi, i \models \phi$ if and only if Π, $\Pi, i \models \psi$. Consequently, any occurence of ϕ is a formula can be equivalently replaced by ψ, vice-versa.

Duality:	$\neg\mathbf{X}\ \phi$	$=$	$\mathbf{X}\ \neg\phi$
	$\neg\Diamond\phi$	$=$	$\Box\neg\phi$
	$\neg\Box\phi$	$=$	$\Diamond\neg\phi$
Idempotency:	$\Diamond\Diamond\phi$	$=$	$\Diamond\phi$
	$\Box\Box\phi$	$=$	$\Box\phi$
	$\phi\ \mathbf{U}\ (\phi\ \mathbf{U}\ \psi)$	$=$	$\phi\ \mathbf{U}\ \psi$
	$(\phi\ \mathbf{U}\ \psi)\ \mathbf{U}\ \psi$	$=$	$\phi\ \mathbf{U}\ \psi$
Absorbtion:	$\Diamond\Box\Diamond\phi$	$=$	$\Box\Diamond\phi$
	$\Box\Diamond\Box\phi$	$=$	$\Diamond\Box\phi$
Expansion:	$\phi\ \mathbf{U}\ \psi$	$=$	$\psi\ \vee\ (\phi\ \wedge\ \mathbf{X}\ (\phi\ \mathbf{U}\ \psi))$
	$\phi\ \mathbf{W}\ \psi$	$=$	$\psi\ \vee\ (\phi\ \wedge\ \mathbf{X}\ (\phi\ \mathbf{W}\ \psi))$
	$\Diamond\psi$	$=$	$\psi\ \vee\ \mathbf{X}\ (\Diamond\psi)$
	$\Box\phi$	$=$	$\phi\ \wedge\ \mathbf{X}\ (\Box\phi)$
Distributivity:	$\mathbf{X}\ (\phi\ \mathbf{U}\ \psi)$	$=$	$(\mathbf{X}\ \phi)\ \mathbf{U}\ (\mathbf{X}\ \psi)$
	$\Diamond(\phi \vee \psi)$	$=$	$\Diamond\phi\ \vee\ \Diamond\psi$
	$\Box(\phi \wedge \psi)$	$=$	$\Box\phi\ \wedge\ \Box\psi$
Others:	$\Box\phi$	$=$	$\phi\ \mathbf{W}\ \mathit{false}$
	$\phi\ \mathbf{U}\ \psi$	$=$	$(\phi\ \mathbf{W}\ \psi)\ \wedge\ \Diamond\psi$

We additionally have the following dualities for $\mathbf{U}$ and $\mathbf{W}$ [3]:

$$\neg(\phi\ \mathbf{U}\ \psi) \ = \ (\phi \wedge \neg\psi)\ \mathbf{W}\ (\neg\phi \wedge \neg\psi)$$

The latter should also be equivalent with the simpler formula $\neg\psi\ \mathbf{W}\ (\neg\phi \wedge \neg\psi)$.

$$\neg(\phi\ \mathbf{W}\ \psi) \ = \ (\phi \wedge \neg\psi)\ \mathbf{U}\ (\neg\phi \wedge \neg\psi)$$

The latter should also be equivalent with the simpler formula $\neg\psi\ \mathbf{U}\ (\neg\phi \wedge \neg\psi)$.

4.4 Buchi Automaton

Some automata have *acceptance states.* We can define an execution of M as *accepting* if it ends in an acceptance state of M. A sentence of M is said to be *accepted* by M if it is induced by an accepting execution. So, while M can produce lots of sentences, not all are considered as 'accepted sentences'. Let $Lang(M)$, 'the *language* of M', denote the set of all accepted sentences of M.

The above is the standard definition of acceptance. Note that only finite sentences can thus be accepted (because it must *ends* in some state). Because in LTL we are dealing with inifinite sentences we will tweak the concept of 'acceptance'. An automaton with this tweaked acceptance is called *Buchi Automaton.*

Definition 4.4.1 : BUCHI AUTOMATON
A Buchi automaton is a finite state automaton, described by a tuple $M = (S, I, F, R, \Sigma)$. Most of the components are the same as before, except that there is no labelling function, and that now we have F.

- S is the set of states. I is a non-empty subset of S, specifying M's possible initial states.
- F is a subset of S, this is the set of M's acceptance states.
- Σ is the 'alphabet' of M. It is the set of labels of M's arrows. Every arrow in M is labelled with one symbol from Σ.
- $R : S \to \Sigma \to Pow(S)$ describes the arrows. If s is a state, and a is a symbol, $R(s, a)$ is the set of possible next-states we arrive at by 'cosuming' the symbol a on the state s. We can only move to a next state by consuming a symbol.

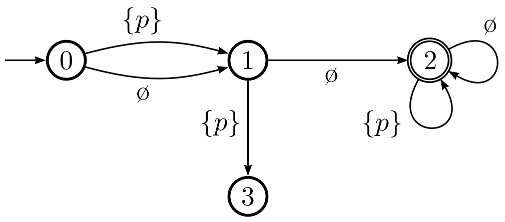
  Note that such an R allows M to be non-deterministic.

□

A Buchi automaton M only accepts infinite sentences. A infinite execution τ of M is accepting if it passes through F infinitely many times[2]. An infinite sentence is accepted if it is induced by an accepting execution. $Lang(M)$ is defined as before.

We can use a Buchi automaton to represent an LTL formula. Let ϕ be an LTL formula over *Prop* as its set of atomic propositions. Now, you have seen that semantically an LTL formula is defined as a predicate over abstract executions. An abstract execution is an infinite sequence over $pow(Prop)$. The idea is to construct a Buchi automaton M_ϕ that accepts all possible infinite sentences over $pow(Prop)$ on which ϕ holds. This automaton would then fully describe ϕ. This automaton is indeed a bit wierd because its alphabet Σ must be $pow(Prop)$ (and *not Prop*).

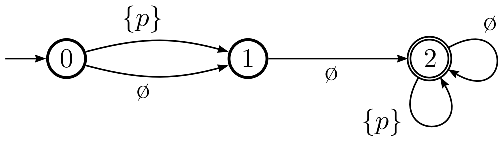
For example, the Buchi automaton below fully describes the abstract executions of the formula $\mathbf{X}\ \neg p$, over $Prop = \{p\}$:



Note that the alphabet Σ of the automaton is $pow(Prop)$. So, in this case $\Sigma = \{\emptyset, \{p\}\}$, which means that we have *two* possible symbols to label each arrow: "∅" or "{p}". The second one represents the case when p holds, whereas the first one represents the case when p does not hold.

The automaton can be a bit simpler though. In state-2 we describe where the automaton would go if it consumes each of possible symbol. But because the transition to state-3 will never lead to an accepting execution, we can just as well remove it. So, we obtain this:

[2]It does not have to pass *every* state in F inifinitely many times. Since F is finite, τ just needs to pass at least one state in F inifinitely many times.



If we now take a larger $Prop = \{p, q\}$, the automaton will look quite verbose. E.g. we would now have four arrows from state-0 to state-2, labelled by each of the subsets of $\{p, q\}$. The same with the loop from state-2 to state-2. To simplify the drawing, we will use the following abbreviations:

- $s \xrightarrow{*} t$

  Intuitively, this means that we can go from s to t by consuming any symbol.

  Technically, it represents a bunch of arrows from state s to t: for each subset A of $Prop$, we implicitly have the arrow $s \xrightarrow{A} t$.

- $s \xrightarrow{p,q\in} t$

  We can go from s to t if the propositions p and q hold.

  It represents a bunch of arrows from state s to t: for each subset A of $Prop$ such that $p, q \in A$, we implicitly have the arrow $s \xrightarrow{A} t$.

- $s \xrightarrow{p,q\notin} t$

  We can go from s to t if the proposition p and q both do not hold.

  It represents a bunch of arrows from state s to t: for each subset A of $Prop$ such that $p \notin A$ and $q \notin A$, we implicitly have the arrow $s \xrightarrow{A} t$.

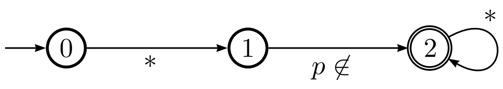
- $s \xrightarrow{C_1,C_2,...} t$

  We can go from s to t if all the conditions hold.

  It represents a bunch of arrows from state s to t: for each subset A of $Prop$ such that A satisfies all the conditions $C_1, C_2, ...$, we implicitly have the arrow $s \xrightarrow{A} t$.

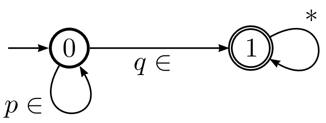
  E.g. we can thus write $s \xrightarrow{p\in,q\notin} t$

So now we can draw the previos automaton less verbosely, and moreover the drawing is now independent of the choice of $Prop$:

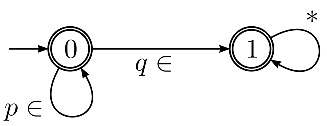


Here are few more examples:

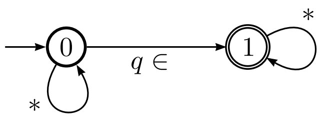
- A Buchi automaton that describes the formula p **U** q:



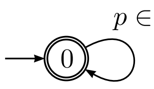
- A Buchi automaton that describes the formula p **W** q; notice that we have two accepting states:



- A Buchi automaton that describes the formula $\Diamond q$:



- A Buchi automaton that describes the formula $\Box p$:



4.4.1 Generalized Buchi Automata

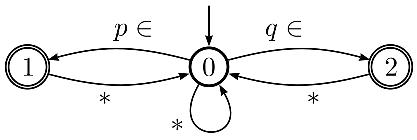
To represents some formulas, in particular conjunctions, it is convenient to use a *generalized Buchi automaton.* It is a Buchi automaton with multiple sets of accepting states. Note that we are talking about set of sets here. An ordinary Buchi automaton may have one, or multiple, accepting states. They are grouped in a single *set* of accepting states. A generalized Buchi has multiple of such sets.

Definition 4.4.2 : Generalized Buchi Automaton (GBA)
Is a finite state automaton, described by a tuple $M = (S, I, \mathcal{F}, R, \Sigma)$. All components except $\mathcal{F}$ have the same meaning as in the ordinary Buchi automaton.

$\mathcal{F}$ is a set of sets. Each member F of $\mathcal{F}$ is a set of acceptance states.
$\Box$

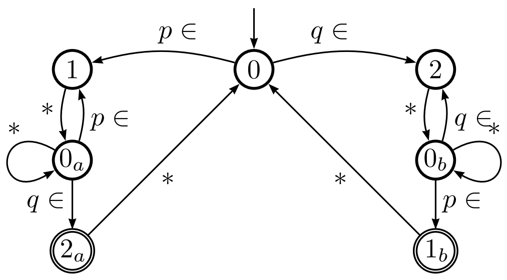
An execution is accepting by the GBA M if it passes through *each* member of $\mathcal{F}$ infinitely many times.

For example a GBA describing the conjunction $(\Box\Diamond p) \wedge (\Box\Diamond q)$ is:



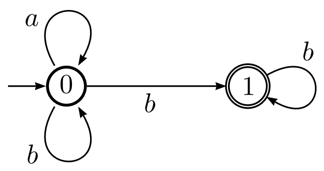
The $\mathcal{F}$ is $\{\{1\},\{2\}\}$. So, we must visit state-1 infinitely many times, *and* also visit state-2 infinitely many times. Note that this is different that saying, as in the ordinary Buchi automaton, that $\{1, 2\}$ is our set of accepting states, for which passing state-1 infinitely many times, but ignoring state-2, will do.

Every generalized Buchi automaton can be converted into an equivalent ordinary Buchi automaton (that is, they accept the same language). Below is an ordinary Buchi automaton that accepts the same language as the one above. It has a single set of accepting states, which is $F = \{2_a, 1_b\}$.



4.4.2 Non-deterministic vs Deterministic Buchi Automata

Whereas a standard non-deterministic FSA can be converted into an equivalent deterministic FSA, this does not in general apply to Buchi automata. For example, consider the Buchi automaton below:



Sentences accepted by this automaton has the form $s{+\!\!+}[b, b, b, ...]$, where s is any finite prefix over a and b. No deterministic Buchi automaton can generate these sentences.

4.4.3 Constructing Buchi Automaton

Every LTL formula can be represented by a (generalized) Buchi automaton. There are algorithms to systematically construct such an automaton. This will be discussed in the lecture.

4.5 LTL Model Checking

Let M be a Kripke structure. We want to verify whether $M \models \phi$. We can do this by first constructing the (generalized) Buchi automaton that represents the negation of ϕ; let's call this automaton $B_{\neg\phi}$. The intersection of these two automata produces sentences which can be produced by both M and $B_{\neg\phi}$. If one of these sentences, say σ, is also an *accepted* sentence for $B_{\neg\phi}$, then it follows that this σ represents an execution of M that satisfies $\neg\phi$, and thus violates ϕ. Therefore if such a σ can be found, then ϕ is not valid. Formally, we have this relation:

Theorem 4.5.1 :

$$M \models \phi \text{ if and only if } Lang(M \cap B_{\neg\phi}) = \emptyset$$

□

The good news is, intersection can be calculated —see Definition 4.2.1. Checking if the language of a Buchi automaton is empty can also be done in finite time. So, we can do verification, despite the infinite number of executions that M may generate!

Before we go into the algorithm for the latter, there is one technical detail we have to deal with. We cannot directly do the intersection $M \cap B_{\neg\phi}$ because they are two different kinds of automata. There is a slight mismatch in their concepts. In a Kripke structure, propositions label the states. Whereas in a Buchi automaton they label the arrows. To make them match, we will transform the Kripke structure M to a Buchi automaton, without changing its meaning (they will generate the same sentences).

Let $M = (S, s_0, R, Prop, V)$ be a Kripke structure; we assume that it has no terminal state (we want to restrict to only infinite executions). Its corresponding Buchi automaton is $Buchi(M) = (S', I, F, R', \Sigma)$ where:

- $S' = S \cup \{z_0\}$, where z_0 is a new dummy innitial state.
- $I = \{z_0\}$.
- We want to consider all possible executions of M. So, it does not have any constraint on which sentences it 'accepts'. Therefore we set F to be the entire S'. In other words, every state is an acceptance state.
- $\Sigma = pow(Prop)$.

- We add a new arrow from z_0 to each actual innitial state s_0, labelled by $V(s_0)$. So:

  $$R'(z_0, A) \;=\; \begin{cases} \{s_0\} & \text{, if } A = V(s_0) \\ \emptyset & \text{otherwise} \end{cases}$$

  We keep the arrows in R, each will be labelled with the label of its destination. So;

  $$R'(s, A) \;=\; \{t \mid t \in R(s) \text{ and } A{=}V(t)\}$$

Note that M and $Buchi(M)$ generate the same set of sentences.

Now we can construct $C = Buchi(M) \cap B_{\neg\phi}$, as described in Definition 4.2.1. This results in a new Buchi automaton. Because $Buchi(M)$ puts no constraint on its acceptance states, the acceptance states or sets of the combined automaton C is just the same as those of $B_{\neg\phi}$.

Now the only thing left to do is to check whether $Lang(C)$ is empty; the is explained in the next section.

4.5.1 Emptiness of Buchi Automaton

Consider your Buchi automaton C. It may generate infinitely many sentences. So we cannot enumerate them to check if there is one that would be accepted by C. However, the acceptance criterion of a Buchi automaton is rather unique. Combined with the fact that, like all the other automata we discuss here, C has a finite number of states, we can conclude the following:

Theorem 4.5.2 :
The language of a Buchi automaton is *not* empty if and only if it has *one* reachable accepting state, that cycles to itself.
□

Good! This comes thus down in finding cycles in your automaton, which is a solvable problem. Notice that an automaton is also a finite and directed *graph*.

A directed graph G has nodes, connected by arrows. We can describe it with a tuple $(S, I, next)$ where S is its set of nodes, I is its set of roots (initial nodes), and $next : S \to pow(S)$ describes the arrows. For every node u, $v \in next(u)$ means that there is an arrow from u to v. So, $next(u)$ is just the set of all 'next'-nodes of u. A Buchi automaton $C = (S, I, F, R, \Sigma)$ induces a graph $G = (S, I, next)$ where $next(u) = (\cup a :: R(u, a))$.

There are algorithms to calculate the set $\mathcal{C}$ of strongly connected components (SCC) of a graph. An SCC is a subgraph such that any two nodes u, v in the SCC are reachable from each other. It follows that if u is in an SCC, then there is a cycle from u to itself. So, all we need to do then is to go through this set $\mathcal{C}$ to find one SCC that contains an accepting state, which is furthermore reachable from an innitial state.

That was one way to do it. The model checker SPIN uses a different algorithm, namely depth first search (DFS). DFS is a generic algorithm on a graph. It explores a given graph starting from a given node r. It follows the arrows as deep as possible, and then backtracks. When it terminates, it would have visited all nodes reachable from r. The algorithm is shown below.

```
explore(G) {
  visited := $\emptyset$ ;
  for($r \in$root($G$)) DFS($r$)
    where
    DFS($u$) {
      if ($u \in$visited) return ;   -- (*)
      visited := visited $\cup$ $\{u\}$   ;
      for ($s$  :  $G.next(u)$) DFS($s$)  ;
    }
}
```

Importantly, we maintain the set of the nodes we have seen (the variable `visited`). In this way the algorithm avoids visiting the same node twice, and thus avoiding to go round and round in a cycle. At the end, `visited` will contain all nodes reachable from the root. Note that every node and edge in the graph are processed at most once. Consequently, its (worst) time complexity is $O(|S| + E)$, where E is the number of edges in G.

Notice that when the algorithm encounters a node u which is already in `visited` (in the step marked with (*) above), it implies that we have found a cycle, of which u is an element of. If you recall Theorem 4.5.2 we are indeed interested in finding cycles, but not just any cycle. We want to find cycles through an accepting state, which are also reachable from the root. For this purpose we modify the DFS algorithm to the double DFS algorithm shown below. The set of nodes in G that correspond to C's accepting states is denoted by acceptStates(G).

```
checkEmpty(G) {
  visited_1 := ∅ ;
  for(r∈root(G)) DFS1(r)

    where
    DFS1(u) {
      if (u∈visited_1) return ;
      visited_1 := visited_1 ∪ {u} ;
      for (s : G.next(u)) {
         if (u∈acceptStates(G)) {
             astate := u ;
             visited_2 := ∅ ; DFS2(s)
             where
             DFS2(v) {
                if (v = astate) throw CycleFound ;  -- (*) cycle!
                if (v∈visited_2) return ;
                visited_2 := visited_2 ∪ {v}
                for (s : G.next(v)) DFS2(s) ;
             }
         }
         DFS1(s) ;
      }
    }
}
```

When it throws `CycleFound` in the step marked by (*), we can conclude that:

1. there is a cycle in G that contains an accepting state v.
2. v is reachable from the root.

So, by Theorem 4.5.2 this proves that the Buchi automaton C is not empty. What is even better, we can easily extend DFS so that it also gives us the path that leads to the cycle we found above. This allows us to show a concrete 'witness' of proving the non-emptiness of C. In terms of the orginal verification problem posed in Theorem 4.5.1, where we ask the question whether the LTL formula ϕ is valid on M, this path/witness corresponds to an execution in M that demonstrates a violation of ϕ. In other words, it concretely shows you an execution producing the error, which is of course a very useful information for debugging.

We can also note that invoking the cycle detection (`DSF2`) can be somewhat wasteful when there are two successors of u in the algorithm above, e.g. s_1 and s_2 such that s_2 can also be reached from s_1. In this scenario, invoking `DFS2`(s_1) will eventually also inspect s_2. So, when `DSF2`(s_1) returns, invoking it again on s_2 would be wasteful. This can be avoided by doing the cycle detection before or after the entire loop **for** ($s : G.next(u)$). We check the cycle by invoking `DFS2`(u), but `DFS2` must then be modified not to count that immediately as a cycle.

```
checkEmpty(G) {
  visited$_1$ := $\emptyset$ ;
  for($r \in$root($G$)) DFS1($r$)

    where
    DFS1($u, \sigma$) {
      if ($u \in$visited$_1$) return ;
      visited$_1$ := visited$_1$ $\cup$ $\{u\}$ ;
      for ($s$ : $G.next(u)$) {

         if ($u \in$acceptStates($G$)) {
            astate := $u$ ;
            visited$_2$ := $\emptyset$ ; DFS2($s, [u]$)
            where
            DFS2($v, \tau$) {
               if ($v =$ astate) throw CycleFound($\sigma$++$\tau$++$[v]$) ;   -- (*) cycle!
               if ($v \in$visited$_2$) return ;
               visited$_2$ := visited$_2$ $\cup$ $\{v\}$
               for ($s$ : $G.next(v)$) DFS2($s$, $\tau$++$[v]$) ;
            }
         }

         DFS1($s$, $\sigma$++$[u]$) ;
      }
    }
}
```

Figure 4.3: *Double DFS model-checking algorithm.*

Figure 4.3 shows a version of the above double DFS algorithm, this time extended so that we also keep track of the path that leads to the node we are inspecting. More precisely, we maintain these invariant:

In DFS1: σ is a path from the root up to (but not including) u.

In DSF2: σ++τ is a path from the root up to (but not including) v.

When an accepting cycle is found (the step marked with (*) in Figure 4.3), we also report the full path to the cycle, and the cycle itself, which is:

σ ++ τ ++ $[v]$

Let the 'size' of an automaton K be denoted by $|K|$, and is defined as the sum of the number of its states and the number of its edges.

Consider the verification of an LTL property ϕ on an automaton M. When converting $\neg\phi$ to a Buchi automaton we may end up with an automaton B whose size is exponential with respect to the size of ϕ (denoted by $|\phi|$), which is defined as the number basic terms it contains. Still, the size of M (the automaton whose correctness we want to verify) is usually still larger than this B The size of the intersection $M \cap B$ is then at most $|M|$. It follows that our DFS algorithm is also $O(|M|)$. Do note that M can be very large, in particular if it is the result of a parallel composition of several other automata.

4.5.2 By-passing the Intersection

The algorithm in Figure 4.3 assumes that G is the graph representing the intersection automaton $Buchi(M) \cap B_{\neg\phi}$. However, we do not actually need to construct this intersection first. This may save us memory and CPU time, if a counter example is found. Effectively, the algorithm only needs the following information from G: $root(G)$, $G.next(u)$, and $\mathsf{acceptState}(G)$. These can be directly expressed in terms of M and $B_{\neg\phi}$. We will represents the nodes in G by pairs (s_1, s_2) where s_1 is a state in M and s_2 is a state in $B_{\neg\phi}$.

1. $(r_1, r_2) \in root(G)$ can be equivalently checked by $r_1 \in init(Buchi(M)) \ \wedge \ r_2 \in init(B_{\neg\phi})$
2. $(s_1, s_2) \in next((u_1, u_2))$ can be equivalently checked by checking the existence a transition $u_1 \xrightarrow{A} s_1$ in $Buchi(M)$ and a transition $u_2 \xrightarrow{A} s_2$. Note that labels of these transition must be the same.
3. $(u_1, u_2) \in \mathsf{acceptState}(G)$ can be equivalently checked by checking that u_2 is an accepting state in $B_{\neg\phi}$.

So, we can replace the above checks in the algorithm in Figure 4.3 with their equivalent counterparts.

4.5.3 Model Checking on Concrete States

Even with the improvement proposed above our model checking algorithm still assumes that M is a Kripke structure. If we have a real program P, usually it is not expressed as a Kripke structure. Can we then systematically construct a Kripke structure out of it?

First, note that we have defined Kripke structures to have finite number of states. Actually, this was because the model checking algorithm really requires you to have finite number of states. If P has infinite number of states, in theory the corresponding Kripke structure M can still be finite, depending on the chosen abstraction $Prop$. But to automatically construct M will still be problematic as we will need to quantify over P's concrete states, whose number is infinite.

If P has finite number of states (or, you only consider a subset of the original program so that you only have to deal with finite number of states), we can construct M e.g. as shown below:

1. Let's introduce two sets S, R, intialized to empty. S will be a set of state, and R a set of arrows/transitions between the states.
2. For every initial state $s_0 \in init(P)$, we explore s_0. To explore a state s we do:
   (a) if we have seen s before, we won't explore it again. Else we add it to S.
   (b) We check the code of P to see what's the next 'atomic' statement[3] to execute. If there is none, we are done exploring s.
   If there is only one atomic statment a possible, we execute it. Suppose t is the resulting state. We add t to S, and the arrow (s, a, t) to R. Then we explore t.
   If there are multiple atomic statements possible on s, we do as above for every choice b that we have.

This procedure terminates, and at the end $(init(P), S, R)$ defines an FSA K that is equivalent to P. If $Prop$ is given, it is straight forward to reduce K to a Kripke structure M. What is not so nice here is that intermediate FSA K is typically very big. Now, if we look again at the improvement discussed in Subsection 4.5.2, the only part of model checking that actually requires information from M is this part:

> Check the existence a pair of transitions, $u_1 \xrightarrow{A} s_1$ in $Buchi(M)$, and $u_2 \xrightarrow{A} s_2$ in $B_{\neg\phi}$, labelled by the same A.

[3]An atomic statement is a statement whose execution cannot be interrupted.

Recall that the orginal place in the algorithm in Figure 4.3 where the above is used is in this quantification:

```
for $((s_1, s_2) \in next(u_1, u_2))$ body
```

Using the above improvement in Subsection 4.5.2 it is then replaced by:

```
forall ($a_1$ is an outgoing arrow in $Buchi(M)$ from $u_1$)  -- *1
   forall ($a_2$ is an outgoing arrow in $B_{\neg\phi}$ from $u_2$) {
      if ($label(a_1) \neq label(a_2)$) continue ; -- *2
      $s_1$ := $destination(a_1)$ ;  -- *3
      $s_2$ := $destination(a_2)$ ;
      body
   }
}
```

Notice that only the parts marked by * actually requires information from $Buchi(M)$. The first one can be equivalently done by quantifying over all atomic statements α_1 of P that can be executed on the the current state u_1. However now, we no longer keep track of the states of M, but rather the concrete states of P.

In *2 we have to check whether the labels of a_1 and a_2 are the same. However, now we have α_1 instead of a_1, and being a statement of a real program, it does not have a label as a Buchi automaton would have. But actually the purpose of this checking is to see if all propositions p in $label(a_2)$ hold in the concrete state that we get if we execute α_1. Checking this can be done simply by evaluating p on that state.

Finally, in *3 we need to know the next state of α_1, which we can do simply by executing it on u_1.

To summarize, the above code fragment can thus be replaced with the following:

```
forall ($\alpha_1$ is an atomic statement in $P$ that can be executed on $u_1$)
   forall ($a_2$ is an outgoing arrow in $B_{\neg\phi}$ from $u_2$) {
      $s_1$ := execute $a_1$ on $u_1$ ;
      if (there is a $p \in label(a_2)$ that does not hold on $s_1$) continue ;
      $s_2$ := $destination(a_2)$ ;
      body
   }
}
```

With this modification, the model checking algorithm is run directly on P. It does not construct any intermediate FSA. This approach is also called *on-the-fly* model checking. Notice that the 'explore' algorithm presented before (for constructing P's FSA) is actually also an instance of DFS. We can thus see the inner DFS in the model checking algorithm as being superimposed on the exploration DFS. From this perspective, it is as if we do model checking on-the-fly as we explore P.

Keep in mind that the above procedure of model checking directly on P presumes P to have finite number of states. For example, P can be a model expressed in Promela, which is the modelling language used by the model checker SPIN. Promela has the familiar style of a structured programming language, though it provide only a small set of programming constructs. In any case, models written in Promela always have finite number of states. On the other hand, if P is e.g. a Java program, then you will need to somehow constrain it to a finite subset of its full behavior. The result of your verification will indeed be incomplete, but with respect to the space you are checking you will still be complete.

Chapter 5

CTL Model Checking

5.1 CTL

Computation Tree Logic(CTL) is another popular class of temporal logic. It shares some similarity with LTL, but is not entirely the same. For example, navigations properties on a GUI or a web site can be better expressed with CTL than LTL.

We will again interpret CTL on Kripke structures. Remember that a Kripe structure K is always defined with respect to a set *Prop* of atomic propositions.

A CTL formula is not interpreted on execution sequences, but rather on an execution tree. An execution 'sequence' represents 'the' current execution, whereas an execution tree rooted in a certain state s represents all possible executions that can branch out from s.

In CTL it is called 'computation tree' rather than 'execution tree'.

Let K be a Krike structure. A computation tree is a tree rooted in an initial state of K, and such that every branch in the tree is an arrow in K, and it is maximal (it keep expanding until it hits terminal states). Such a tree can be infinitely deep. As with LTL, to simplify the discussion we will assume that K has no terminal state; so we will only have infinite computation trees.

The computation trees that start in K's initial states are of course a bit special, since this represents K's all possible full executions.

The syntax of CTL:

$$\begin{array}{lll} \phi & ::= & p, \quad \text{where } p \text{ is an atomic proposition} \\ & & \mid \neg\phi \mid \phi \wedge \psi \\ & & \mid \mathbf{EX}\ \phi \mid \mathbf{E}(\phi\ \mathbf{U}\ \psi) \mid \mathbf{A}(\phi\ \mathbf{U}\ \psi) \end{array} \tag{5.1}$$

Intuitively, $\mathbf{EX}\ \phi$ means that ϕ holds on one of K's next states. $\mathbf{E}(\phi\ \mathbf{U}\ \psi)$ means that there is one execution path in K's computation tree (rooted at one of K's initial states), such that ψ eventually holds, and until then ϕ holds (along the path).

$\mathbf{A}(\phi\ \mathbf{U}\ \psi)$ means as the same with **EU**, but the until-property has to hold on all possible paths that branch out from K's initial states.

To define these formally, we will first introduce some auxiliary notations. We write $K \models \phi$ to mean that ϕ is a valid CTL property of the Kripke structure K. We write $K, t \models \phi$ to mean that ϕ is a valid CTL property on the computation tree t, where t is an computation tree of K.

$paths(t)$ is the set of all paths through the tree t, which starts at its t's root.

If s is a state in K, $tree(s)$ is the q computation tree (of K), rooted in s (there is only one such tree).

We define:

$$K \models \phi \quad = \quad (\forall s_0 : s_0 \in init(K) :\ K, tree(s_0) \models \phi) \tag{5.2}$$

where $init(K)$ denotes the set of K's initial states.

The meaning of various CTL formulas are formally defined as follows:

1. If p is an atomic proposition (from K's *Prop*), it holds on a computation tree t if it holds on t's root:

$$K,t \models p \quad = \quad p \in V(root(t))$$

2. $\neg\phi$ holds on a computation tree t, if ϕ does not hold:

$$K,t \models \neg\phi \quad = \quad \text{not } (K,t \models \phi)$$

3. $\phi \wedge \psi$ holds on t if each holds individually:

$$K,t \models (\phi \wedge \psi) \quad = \quad (K,t \models \phi) \text{ and } (K,t \models \psi)$$

4. **EX** ϕ holds on t if ϕ holds on the tree rooted at one of t's root next-state:

$$K,t \models \mathbf{EX}\ \phi \quad = \quad \text{there is a state } v \in R(root(t)) \text{ such that } K, tree(v) \models \phi$$

   This is called the 'exists-next' operator.

5. $\mathbf{E}(\phi\ \mathbf{U}\ \psi)$ holds on t if there is one path π, on which the until-property holds:

$$\begin{aligned} K,t \models \mathbf{E}(\phi\ \mathbf{U}\ \psi) \quad = \quad & \text{there is a } p \in paths(t) \text{ and a } j \geq 0 \text{ such that:} \\ & K, tree(p_j) \models \psi \\ & \text{and, for all } i,\ 0 \leq i < j\text{:} \quad K, tree(p_i) \models \phi \end{aligned}$$

   This is called the 'exists-until' operator.

6. $\mathbf{A}(\phi\ \mathbf{U}\ \psi)$ holds on t if the until-property holds on all paths starting at t's root:

$$\begin{aligned} K,t \models \mathbf{A}(\phi\ \mathbf{U}\ \psi) \quad = \quad & \text{for all } p \in paths(t) \text{, there is a } j \geq 0 \text{ such that:} \\ & K, tree(p_j) \models \psi \\ & \text{and, for all } i,\ 0 \leq i < j\text{:} \quad K, tree(p_i) \models \phi \end{aligned}$$

   This is called the 'always-until' operator.

Notice that the 'E' and 'A' can be seen as a separate operator that quantifies over $paths(t)$. The first existentially quantifies over it, and the second universally.
The 'X' and 'U' have the same intuition as in LTL. However CTL's syntax forbids you to write nested 'until' within the same 'E' or 'A' quantifier. E.g. this has no meaning in CTL:

$$\mathbf{E}(p\ \mathbf{U}\ (q\ \mathbf{U}\ r))$$

Furthermore note that if a computation tree t has v as its root, then it is the *only* computation tree rooted at v. In otherwords, the root and the tree fully determines each other. So, let us add this notation:

$$K,v \models \phi \quad = \quad K, tree(v) \models \phi$$

In literature, CTL is often defined in terms of the root of the corresponding computation tree; so now you know what is meant.
We can furthermore define a number of handy derived operators:

- Disjunction and implication:

$$\begin{aligned}\phi \vee \psi &= \neg(\neg\phi \wedge \neg\psi)\\ \phi \rightarrow \psi &= \neg\phi \vee \psi\end{aligned}$$

- Always-next ϕ:

$$\mathbf{AX}\ \phi = \neg(\mathbf{EX}\ \neg\phi)$$

- Exists-eventually and always-eventually:

$$\mathbf{EF}\ \phi = \mathbf{E}(\mathtt{true}\ \mathbf{U}\ \phi)$$

$$\mathbf{AF}\ \phi = \mathbf{A}(\mathtt{true}\ \mathbf{U}\ \phi)$$

- Exists-globally and always-globally:

$$\mathbf{EG}\ \phi = \neg(\mathbf{AF}\ \neg\phi)$$

$$\mathbf{AG}\ \phi = \neg(\mathbf{EF}\ \neg\phi)$$

Actually we only need either the exist-until or the always-until (rather than both) as a primitive operator, since one can be expressed in terms of the other. We have this equality:

$$\mathbf{A}(\phi\ \mathbf{U}\ \psi) = \neg\mathbf{E}(\neg\psi\ \mathbf{U}\ (\neg\phi \wedge \neg\psi)) \ \wedge\ \neg\mathbf{EG}\ \neg\psi \tag{5.3}$$

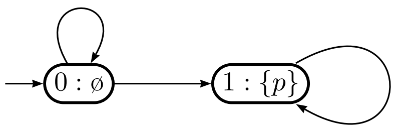
5.1.1 CTL vs LTL

Some properties can be expressed in both CTL and LTL, e.g:

CTL	LTL
$\mathbf{AG}\ p$	$\Box p$
$\mathbf{AF}\ p$	$\Diamond p$
$\mathbf{AX}\ p$	$\mathbf{X}\ p$
$\mathbf{A}(p\ \mathbf{U}\ q)$	$p\ \mathbf{U}\ q$

where p and q are atomic propositions.

Some CTL properties cannot be expressed in LTL. For example, consider the automaton below, with $Prop = \{p\}$.

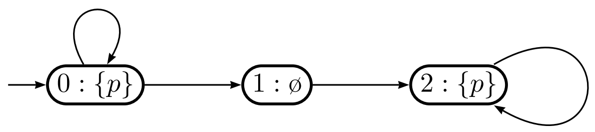


The CTL property $\mathbf{EF}\ p$ holds on this automaton, which means that there is one execution on which the proposition p will eventually hold.

We cannot express this in LTL. E.g. The closest we have in LTL is $\Diamond p$, but it would require that p holds on *all* executions, which is too strong, and is not valid for the above automaton. Another option is try to express it with negation: $\neg\Box\neg p$; but this just the equivalent of $\Diamond p$. So that won't work either.

Essentially LTL cannot express the property because it cannot existentially quantify over the set of all executions. An LTL property is always defined with respect to *all* executions.

Some LTL properties cannot be expressed in CTL. Consider this automaton:



This LTL property is valid on this automaton: $\Diamond\Box p$. It says, over any execution of this automaton, along the execution eventually p will continue to hold. There is a nested quantinfications over each execution here.

However CTL has no mechanism to express such a nested quantifications over a single path. The closest formula in CTL to express $\Diamond\Box p$ is $\mathbf{AF}\ (\mathbf{AG}\ p)$. But this requires that all execution paths from state-0 satisfies the $\mathbf{F}(\mathbf{AG}\ p)$ part. In particular, consider the infinite path where we just keep cyling in state-0. Let's call this path π. By the definition of $\mathbf{AF}$, there should be some $i \geq 0$, such that $tree(\pi_i) \models \mathbf{AG}\ p$. But $\pi = 0, 0, 0, ...$ So, its i-th state is just state-0, and $\mathbf{AG}\ p$ does *not* hold on $tree(0)$!

5.1.2 CTL*

CTL* is a superset of both CTL and LTL. The syntax is below. A CTL* formula is a state formula, which in turns is built from path fomulas.

The syntax of state formulas:

$$\begin{array}{lll} \phi & ::= & p, \quad \text{where } p \text{ is an atomic proposition} \\ & & \mid \neg\phi \mid \phi \wedge \psi \\ & & \mid \mathbf{E}f \mid \mathbf{A}f \end{array} \tag{5.4}$$

where f is a *path* formula, with the following syntax:

$$\begin{array}{lll} f & ::= & \phi, \quad \text{where } \phi \text{ is a state formula} \\ & & \mid \neg f \mid f \wedge g \\ & & \mid \mathbf{X}\ f \mid f\ \mathbf{U}\ g \end{array} \tag{5.5}$$

In CTL* we can express e.g. $\mathbf{E}(p\ \mathbf{U}\ (p\ \mathbf{U}\ q))$. Furthermore, CTL* is decidable.

5.1.3 Expansion Laws

Let's first consider LTL's 'until' operator, because it is a bit easier to explain. It satisfies this 'expansion' property:

$$\phi\ \mathbf{U}\ \psi \quad = \quad \psi \vee (\phi \wedge \mathbf{X}\ (\phi\ \mathbf{U}\ \psi)) \tag{5.6}$$

It says, $\phi\ \mathbf{U}\ \psi$ holds if ψ holds immediately, or if ϕ holds now the $\phi\ \mathbf{U}\ \psi$ holds with respect to all executions from one-step ahead. Notice the recursive relation here.

CTL's 'until', and all its derived operators, also has a similar property, as shown below. We will see an application of the property in the next subsection.

Theorem 5.1.1 : CTL's Expansion Property

- $\mathbf{E}(\phi\ \mathbf{U}\ \psi) \quad = \quad \psi \vee (\phi \wedge \mathbf{EX}\ (\mathbf{E}(\phi\ \mathbf{U}\ \psi)))$
- $\mathbf{A}(\phi\ \mathbf{U}\ \psi) \quad = \quad \psi \vee (\phi \wedge \mathbf{AX}\ (\mathbf{A}(\phi\ \mathbf{U}\ \psi)))$
- $\mathbf{EF}\ \phi \quad = \quad \phi \vee \mathbf{EX}\ (\mathbf{EF}\ \phi)$
- $\mathbf{AF}\ \phi \quad = \quad \phi \vee \mathbf{AX}\ (\mathbf{AF}\ \phi)$
- $\mathbf{EG}\ \phi \quad = \quad \phi \vee \mathbf{EX}\ (\mathbf{EG}\ \phi)$

□

5.1.4 CTL Model Checking

Consider a Krikpe structure $K = (S, I, R, Prop, V)$. Let's treat a CTL formula as a predicate on K's states. Let's write W_ϕ to mean the set of K's states such that ϕ holds. That is:

$$W_\phi \;=\; \{s \mid s \in S \text{ and } K, s \models \phi\}$$

Notice that we then have this relation:

$$K \models \phi \;=\; s_0 \in W_\phi, \text{for all initial state } s_0 \tag{5.7}$$

Well, this gives an idea as to how to check whether ϕ is a valid property of K. The above says that you can thus proceed by first calculating the set W_ϕ, and then we simply check whether $s_0 \in W_\phi$.

We can calculate W_ϕ recursively, over the structure of ϕ. We start by calculating W_p for each atomic proposition in ϕ, then works our way up to the W of the next level of the subformulas of ϕ, and proceed until we come to the 'root' level ϕ and calculate W_ϕ. See the algorithm below:

1. For an atomic proposition p, calculating W_p is straight forwards:
$$W_p \;=\; \{s \mid s \in S \text{ and } p \in V(s)\}$$
2. $W_{\mathbf{EX}\ \phi}$ is calculated as below. Note that by this time, W_ϕ must have already been calculated.
$$W_{\mathbf{EX}\ \phi} \;=\; \{s \mid s \in S \text{ and } R(s) \cap W_\phi \neq \emptyset\}$$
which simply says that we take all states s (of K) that has at least one successor in W_ϕ.
3. Calculating $W_{\mathbf{AX}\ \phi}$ is also straight forward:
$$W_{\mathbf{AX}\ \phi} \;=\; \{s \mid s \in S \text{ and } R(s) \subseteq W_\phi\}$$
which says that we take all states s (of K) that has *all* its successors in W_ϕ.
4. Calculating $W_{\mathbf{E}(\phi\ \mathbf{U}\ \psi)}$ and $W_{\mathbf{A}(\phi\ \mathbf{U}\ \psi)}$ is more complicated, and is separately explained below.
5. $W_{\phi\wedge\psi} \;=\; W_\phi \cap W_\psi$
6. $W_{\neg\phi} \;=\; S/W_\phi$

Calculating W of always-until

To calculate the W of 'until', we are helped by its expansion property —see Theorem 5.1.1. It implies that the same relation holds for W of 'until'. So (let's take the 'always-until' variant):

$$W_{\mathbf{A}(\phi\ \mathbf{U}\ \psi)} \;=\; W_\psi \cup (W_\phi \cap \{s \mid s \in S \text{ and } R(s) \subseteq W_{\mathbf{A}(p\ \mathbf{U}\ q)}\})$$

This is a bit long formula. Let's abbreviate: $Z = W_{\mathbf{A}(\phi\ \mathbf{U}\ \psi)}$. Then the equation becomes:

$$Z \;=\; W_\psi \;\cup\; (W_\phi \;\cap\; \underbrace{\{s \mid s \in S \text{ and } R(s) \subseteq Z\}}_{f(Z)}\;)$$

If we assume that we have already calculated W_ϕ and W_ψ, then all elements at the right hand side of the equation above, except for Z, are known. So, more abstractly the above equation has this form:

$$Z \;=\; W_\psi \;\cup\; (W_\phi \;\cap\; f(Z)) \tag{5.8}$$

where f is a function abbreviates the part of the original right hand side as indicated above. We are looking for the *smallest solution* of the above equation.

To calculate the smallest solution for Z we will proceed iteratively as follows. We will approximate Z with a series $Z_0, Z_1, Z_2, \ldots$. They are calculated as follows:

1. Start with an empty approximation:

$$Z_0 \;=\; \emptyset$$

2. The next approximation is calculated from the previous one, as follows:

$$Z_{i+1} \;=\; W_\psi \;\cup\; (W_\phi \;\cap\; f(Z_i))$$

3. Keep approximating, until we find a $Z_{k+1} = Z_k$. Then Z_k is equal to the Z we are looking for.

Obviously, if the above loop terminates, it will terminate with a solution of (5.8). Because of the way f is defined, you can actually prove that

$$Z_i \;\subseteq\; Z_{i+1}$$

Because K is a finite state automaton (S is finite), we cannot indefinitely grow Z_i. So, the loop must terminate. It can also be proven that it terminates with the smallest solution.

Calculating W of exists-until

Analogously with 'always-until' above, based on the expansion property of 'exists-until', we have:

$$W_{\mathbf{E}(\phi\ \mathbf{U}\ \psi)} \;=\; W_\psi \cup (W_\phi \cap \{s \mid s \in S \text{ and } R(s) \cap W_{\mathbf{A}(p\ \mathbf{U}\ q)} \neq \emptyset\})$$

Let's abbreviate: $Z = W_{\mathbf{E}(\phi\ \mathbf{U}\ \psi)}$. Then the equation becomes:

$$Z \;=\; W_\psi \;\cup\; (W_\phi \;\cap\; \underbrace{\{s \mid s \in S \text{ and } R(s) \cap Z \neq \emptyset\}}_{g(Z)}\;)$$

Or:

$$Z \;=\; W_\psi \;\cup\; (W_\phi \;\cap\; g(Z))$$

We are looking for the smallest solution for this Z. We will calculate it iteratively as before. We will approximate Z with a series $Z_0, Z_1, Z_2, \ldots$. They are calculated as below. It is the same algorithm as in always-until. The only difference is that we need to use g instead of f:

1. Start with an empty approximation:

$$Z_0 \;=\; \emptyset$$

2. The next approximation is calculated from the previous one, as follows:

$$Z_{i+1} \;=\; W_\psi \;\cup\; (W_\phi \;\cap\; g(Z_1))$$

3. Keep approximating, until we find a $Z_{k+1} = Z_k$. Then Z_k is equal to the Z we are looking for.

Or using this iterations

An alternative way to calculate $W_{\mathbf{A}(\phi\ \mathbf{U}\ \psi)}$, which is perhaps more intuitive, is shown below.

1. Start with:

$$Z_1 \;=\; W_\psi$$

2. The next approximation is calculated from the previous one, as follows:

$$Z_{i+2} \;=\; Z_{i+1} \;\cup\; (W_\phi \;\cap\; f(Z_{i+1}))$$

   where $f(Z)$ is defined as before, namely $f(Z) = \{s \mid s \in S \text{ and } R(s) \subseteq Z\}$.

3. Keep approximating, until we find a $Z_{k+1} = Z_k$. Then Z_k is our solution.

$W_{\mathbf{E}(\phi\ \mathbf{U}\ \psi)}$ can be calculated analogously, except we use $g(Z)$ instead of $f(Z)$, where as before $g(Z) = \{s \mid s \in S \text{ and } R(s) \cap Z \neq \emptyset\}$

We can prove by induction, that for every $i > 1$, Z_i obtained both the old and the new procedures are the same. For $i = 1$ this is obvious. Let's denote Z_i obtained by the original fix point iterations Z^o_i, and the one obtained by the alternate iterations above Z_i. We have to prove that $Z^o_{i+2} = Z_{i+2}$:

$$\begin{array}{ll}
& Z^o_{i+2} \\
= & \text{— def. of } Z^o \\
& W_\psi \;\cup\; (W_\phi \;\cap\; f(Z^o_{i+1})) \\
= & \text{— applying the induction hypothesis} \\
& W_\psi \;\cup\; (W_\phi \;\cap\; f(Z_{i+1})) \\
= & \text{— def. of } Z \\
& W_\psi \;\cup\; (W_\phi \;\cap\; f(Z_i \cup (W_\phi \cap Z_i))) \\
= & \text{— we can show that } f(A \cup B) = f(A) \cup f(A \cup B) \\
& W_\psi \;\cup\; (W_\phi \;\cap\; (f(Z_i) \cup f(Z_i \cup (W_\phi \cap Z_i)))) \\
= & \text{— def. of } Z \\
& W_\psi \;\cup\; (W_\phi \;\cap\; (f(Z_i) \cup f(Z_{i+1}))) \\
= & \\
& W_\psi \;\cup\; (W_\phi \;\cap\; f(Z_i)) \;\cup\; (W_\phi \cap f(Z_{i+1})) \\
= & \text{— applying the induction hypothesis} \\
& W_\psi \;\cup\; (W_\phi \;\cap\; f(Z^o_i) \;\cup\; (W_\phi \cap f(Z_{i+1})) \\
= & \text{— def. of } Z^o \\
& Z^o_{i+1} \;\cup\; (W_\phi \;\cap\; f(Z_{i+1})) \\
= & \text{— applying the induction hypothesis} \\
& Z_{i+1} \;\cup\; (W_\phi \;\cap\; f(Z_{i+1})) \\
= & \text{— def. of } Z \\
& Z_{i+2}
\end{array}$$

The above described CTL model checking algorithm has the time complexity of $O((N+E)*|\phi|)$ where N is the number of states in the target Kripke structure, E is its number of arrows, and $|\phi|$ is the size of the CTL formula we are verifying.

Also known as labelling

In the literature, the above approach of model checking by calculating W_ϕ is also explained in terms of a *labelling* process. The model checking proceeds by each state s by ϕ if ϕ holds on this state. Note that this is just equivalent to determining the set W_ϕ itself.

Chapter 6

Symbolic Model Checking and BDD

6.1 Symbolic Representation of State Space

As you can see in the previous model checking algorithms, both for LTL and CTL, we need to have the 'state space' of our target program (or model) in order to apply the algorithm. LTL model checking allows you to lazily construct this state space, but ultimately you need to construct it completely.

Now, the symbolic model checking approach proposes to represent the state space symbolically with a formula. It would be a big formula, but nevertheless you can store in less space than the state space itself. Essentially, the idea is to represent multiple states and multiple arrows in the state space with just a single small formula; this saves space.

However, this does mean that our model checking algorithms have to be adapted so that they work on formulas, rather than an automaton. We will later see how this works, but first let us see how we can symbolically represent an automaton (which is what a state space is) with formulas.

To be more precise, we will represent automatons with *Boolean functions*. Such a function $f(x, y)$ takes parameters of Boolean type, and returns a Boolean value. The body of this function is described by a Boolean formula, with the following syntax:

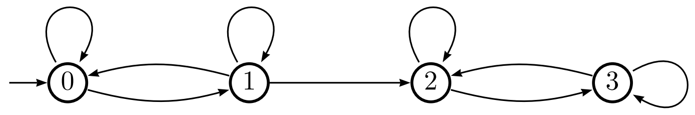
$$p \ ::= \ 0 \ | \ 1 \ | \ \textit{variable} \ | \ \neg p \ | \ p \wedge q \ | \ p \vee q$$

For brevity we will use 1/0 instead of `true` and `false`. We also just write $x.y$ or even xy (if it is clear that x and y are two variables) to mean $x \wedge y$. We write $\bar{e}$ to mean the negation of the expression e. So, $\bar{x}\bar{y}$ means $\neg x \wedge \neg y$, whereas $\overline{xy}$ means $\neg(x \wedge y)$.

Derived operators $\Rightarrow$ and $=$ (equality on Boolean values) are defined as usual; furthermore quantifications over Boolean values are defined as follows:

$$\begin{aligned} (\exists x :: p) &= p[1/x] \vee p[0/x] \\ (\forall x :: p) &= \neg(\exists x :: \neg p) \end{aligned}$$

Please note that x above is of type Boolean, so we only quantify over 0 and 1.
Consider now this automaton K with just four states:



We can use Boolean functions (formulas) over two variables x and y to encode its four states; e.g. with the encoding below:

encoding	state
$\bar{x}\bar{y}$	0
$\bar{x}y$	1
$x\bar{y}$	2
xy	3

The nice thing about this is that we can also use formulas to represent *sets* of states. E.g. the formula x represents the set $\{2,3\}$; the formula $\bar{x}$ represents $\{0,1\}$; the formula $x \lor y$ represents $\{1,2,3\}$.

An arrow relates a state to a possible next-state. If we use a primed names of x and y to represent next-state, we can also encode arrows. E.g:

encoding	arrow
$\bar{x}\bar{y}\bar{x}'\bar{y}'$	the loop from 0 to 0
$\bar{x}\bar{y}\bar{x}'y'$	the arrow from 0 to 1

We can also encode multiple arrows with a single Boolean function, e.g. the two arrows going into state 0 can be encoded by:

$$\bar{x}\bar{x}'\bar{y}'$$

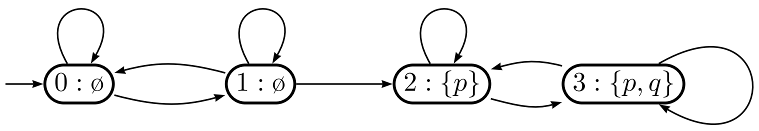
It can be quite short, thus saving space. E.g. all the four arrows between state 2 and 3 can be represented by:

$$xx'$$

The entire automaton can be described by the following Boolean function:

$$R(x,y,x',y') \quad = \quad \bar{x}\overline{x'} \;\lor\; \bar{x}y\overline{y'} \;\lor\; xx'$$

We can easily extend this to encode the propositions that hold in the states. For example, suppose label the states of the automaton that we had before as follows; *Prop* $= \{p, q\}$ is assumed:


$$\begin{tikzcd}
{} \arrow[r] & 0 : \emptyset \arrow[loop above] \arrow[r, bend right] & 1 : \emptyset \arrow[loop above] \arrow[l, bend right] \arrow[r] & 2 : \{p\} \arrow[loop above] \arrow[r, bend right] & 3 : \{p,q\} \arrow[loop right] \arrow[l, bend right]
\end{tikzcd}$$

We see that p holds on states 2,3, whereas q only holds on 3. We can capture this with a following Boolean function $V(x,y,p,q)$:

$$V(x,y,p,q) \quad = \quad (p = x) \land (q = xy)$$

where $f = g$ above is just boolean equality; so it holds if either fg or $\bar{f}\bar{g}$. The above formula says that p holds only on the states s whose encoding xy has its first component (x) true; whereas q holds on the state whose encoding xy has both its components (x and y) true.

Question: how do we handle automaton with concrete states? So, we don't have labeling with propositions. Instead, we have variables that take values. E.g. variables a, b which in every state may take different values[1]

[1]Answer: when you verify a given formula ϕ, you would know what your *Prop* is. Given this *Prop*, the given concrete state FSA can be reduced to a Kripke structure with labelling L.

6.2 CTL Symbolic Model Checking

Let $K = (S, I, R, Prop, V)$ be a Krikpe structure. Let ϕ be a CTL formula we want to verify on K. Recall our model checking approach proceeds by first calculating W_ϕ (the set of all states of K that would satisfy ϕ), and then verify $s_0 \in W_\phi$ for each initial state $s_0 \in I$.

We need to turn this to work on Boolean functions. You have seen that a set of states can be expressed by a Boolean function. Suppose now, as in the above example, that we can use two Boolean variables x and y to encode the states of K. We will first construct a symbolic representation of W_ϕ. This is a Boolean function over x and y, which we will just denote by:

$$W_\phi(x, y)$$

Intuitively this function is as such that $s \in W_\phi$ if and only if $W_\phi(enc(s)) = 1$, where $enc(s)$ is the vector x, y that represents the state s.

Checking $s_0 \in W_\phi$ is now equivalent to checking if $W_\phi(enc(s_0)) = 1$. If we only have one initial state s_0 then we are done. More generally, if we have multiple initial states, represented by a Boolean formula $init(x, y)$, we check if this formula is *valid*:

$$init(x, y) \;\Rightarrow\; W_\phi(x, y)$$

Importantly, observe that this suggests that we can convert the problem of CTL model checking to the problem of evaluating a Boolean function, or the problem of checking validity in the proposition logic. The latter two problems are solvable.

This also suggests that if you can give a symbolical description of your target program, then we can model check it without having to explicitly construct its state space.

For the above to work, we still need to figure out a way to construct the symbolical representation of W_ϕ. We will show the construction below, through examples. Consider again the automaton K from the previous section, with $Prop = \{p, q\}$. In the section we have already given the symbolic description of this K, in the form of the functions:

$$\begin{array}{lcl} R(x, y, x', y') & : & \text{describes } K\text{'s arrows} \\[1em] V(x, y, p, q) & : & \text{describes the labelling with propositions} \end{array}$$

Atomic Proposition

Consider the atomic proposition p. W_p is then just the set of all K's states where p holds:

$$W_p \;=\; \{s \mid p \in V(s)\}$$

But this is a set. We need to re-express this set as a Boolean function, which we can do with this one:

$$W_p(x, y) \;=\; (\exists q :: V(x, y, 1, q))$$

Whereas the first version of W_p above is a set of states, the second one is indeed a function. It is a Boolean function to be precise; but note that this function equivalently encodes the same set of states.

Suppose we only have one initial state namely s_0 enoced by 00, and suppose we want to verify if $K \models p$. This means checking whether $s_0 \in W_p$, which we can symbolically check by checking if this:

$$W_p(0, 0)$$

would evaluate to 1.

If we have multiple initial states, whose set is symbolically represented by the Boolean function $init(x, y)$, then $K \models p$ can be checked by checking if this:

$$init(x, y) \;\Rightarrow\; W_p(x, y)$$

is *valid*.

Conjunction and Negation

Well, assuming you have constructed the Boolean functions $W_\phi(x,y)$ and $W_\psi(x,y)$ then:

$$\begin{aligned} W_{\neg\phi}(x,y) &= \neg W_\phi(x,y) \\ W_{\phi\wedge\psi}(x,y) &= W_\phi(x,y) \ \wedge \ W_\psi(x,y) \end{aligned}$$

Next Property

Assume that we have already calculated $W_\phi(x,y)$. Now, $W_{\mathbf{EX}\ \phi}$ is:

$$W_{\mathbf{EX}\ \phi} \quad = \quad \{s \mid (R(s) \ \cap \ W_\phi) \neq \emptyset\}$$

Let us firs re-express this a bit differently (but equivalent):

$$W_{\mathbf{EX}\ \phi} \quad = \quad \{s \mid (\exists s' :: s' \in R(s) \ \wedge \ s' \in W_\phi)\}$$

We can express this with a Boolean function, namely:

$$W_{\mathbf{EX}\ \phi}(x,y) \quad = \quad (\exists x', y' :: R(x,y,x',y') \ \wedge \ W_\phi(x',y'))$$

The W of **AX** can be calculated indirectly through this equality: $\mathbf{AX}\ \phi = \neg\mathbf{EX}\ \neg\phi$. Or, if we want to express this directly:

$$W_{\mathbf{AX}\ p}(x,y) \quad = \quad (\forall x', y' :: R(x,y,x',y') \ \Rightarrow \ W_\phi(x',y'))$$

To verify e.g. $K \models \mathbf{EX}\ \phi$ we check the validity of this:

$$\mathit{init}(x,y) \ \Rightarrow \ W_{\mathbf{EX}\ \phi}(x,y)$$

Until

Well, assuming you have constructed the Boolean functions $W_\phi(x,y)$ and $W_\psi(x,y)$. Recall that calculate $W_{\mathbf{E}(p\ \mathbf{U}\ q)}$ by iteratively approximating it with Z_i. We will use the second way of iterating these Z_i's, but without the substraction $f(Z_{i+1}/Z_i)$. So:

$$\begin{aligned} Z_0 &= W_\psi \\ Z_{i+1} &= Z_i \ \cup \ (W_\phi \cap \{s \mid R(s) \ \cap \ Z_i \neq \emptyset\}) \end{aligned}$$

The latter can also be written as:

$$Z_{i+1} \quad = \quad Z_i \ \cup \ (W_\phi \cap \{s \mid (\exists s' :: s' \in R(s) \ \wedge \ s' \in Z_i)\})$$

But we can also represents Z_i with Boolean functions:

$$\begin{aligned} Z_0(x,y) &= W_\psi(x,y) \\ Z_{i+1}(x,y) &= Z_i(x,y) \ \vee \ (W_\phi(x,y) \wedge (\exists x', y' :: R(x,y,x',y') \wedge Z_i(x',y'))) \end{aligned}$$

However, we have to stop the iteration if we have reach a fix point. So, a point where $Z_{i+1} = Z_i$. If we have set of states then we know how to do it. However, now we have Boolean functions. To check if you have reached the fix point you would then need to check whether:

$Z_i(x,y)$ and $Z_{i+1}(x,y)$ are equivalent

And you need to be able to do this efficiently. Additionally, note that the formula Z_i grows as we iterate. This formula the be pretty big, which in a way is not surprising as it encodes a part of the total state space of K. This can be a large space to encode. So, additionally we also want to have a space efficient way to represent our boolean formulas.

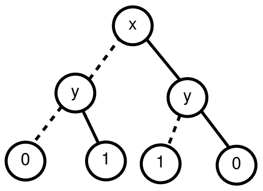
The all-until operator can be handled analogously.

6.3 Representing Boolean Function

One way to check if two Boolean formulas p and q are equivalent is to turn this to a satisfiability problem. So, we check whether $p \neq q$ is satisfiable. If so, then they are not equivalent; else they are. You can use a SAT solver to do this. Note that this problem is in general NP-complete.

Here I will explain a complementary approach (to SAT solving). We will construct some form of 'standard' representations of proposition formulas, in such a way that p and q are equivalent if and only if their standard representations are structurally the same. So, we can instead compare their representations. Such a representation is also called *canonical representation* or *normal form.* Truth table is such a canonical representation, but it won't do for us, due to the exponential size of the table.

A *binary decision tree* is a binary tree whose nodes are labelled with a variable name, and leaves labelled with either 0 or 1. Each node has two child-nodes: its *low* and respectively *high* child. Here the low child is indicated by the dashed line connecting to it. See the example below:



For later, it is easier if we just treat a leaf as a special kind of node. So, a tree has nodes, but some of these nodes are leaves.

Let $Node$ be the set of nodes in the tree. We'll use functions $low, high : Node \to Node$. If v is not a leaf, $low(v)$ and $high(v)$ return v's low and respectively high child. The function $ind(v)$ returns the variable (to be precise: 'variable name'q) that decorates the node v, if it is not a leaf. If v is a leaf, $val(v)$ returns its Boolean value, which is either 0 or 1.

To be a binary decision tree, we also need to require that along every path in the tree, each variable name occurs at most once.

Such a tree can be used to represent a Boolean function. Given a tree, the function represented by it can be recursively reconstructed as follows:

1. If v is a leaf, $func(v) = val(v)$.
2. If v is a non-leaf node, decorated with x (so, $ind(v) = x$) then:

$$func(v) \quad = \quad \bar{x}.func(low(u)) \ \vee \ x.func(high(u))$$

3. If t is a tree with root v_0, the the formula t represents is $func(v_0)$.

For example, the Boolean function represented by the tree above is:

$$f(x,y) \ = \ func(root) \ = \ \bar{x}\bar{y}0 \ \vee \ \bar{x}y1 \ \vee \ x\bar{y}1 \ \vee \ xy0$$

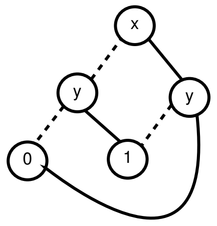
which can be simplified to:

$$f(x,y) \ = \ \bar{x}y \ \vee \ x\bar{y}$$

Every boolean function can be represented with a binary decision tree. Such a tree uses however a lot of space, since it requires exponential number of nodes, with respect to the number of parameters of our Boolean function, but we will improve this below.

A *binary decision graph* (BDD) is just like a *binary decision tree*, except that it is a *directed graph.* So, multiple nodes can share the same child or leaf. There are some additional constraints: the graph must have a single *root*, and all paths in the graph must end in the leaves. The benefit of using BDDs is that they can (but not always) compactly represent boolean functions, and furthermore checking if two BBDs represent equivalent formulas can be done very efficiently.

Below is an example. To make the drawing less verbose, I keep the direction of the edges in my drawings implicit (it's a *directed* graph): it is always from top to bottom. The root is always the topmost node.



A BDD can be used to represent a Boolean function as well. We use the same algorithm as for the tree to reconstruct the function that the graph represents. If G is a BDD with root v_0, then $f(G)$ is just $f(v_0)$.

Because in a graph you can share nodes, it is more space-efficient than a tree. The above graph represents the same Boolean function as the tree you earlier saw:

$$f(x,y) \;=\; func(root) \;=\; \bar{x}y \;\vee\; x\bar{y}$$

but it uses two less nodes.

A BDD is called *reduced* if it cannot be made smaller (without changing its meaning as a Boolean function); see below for the formal definition. For example, the above BDD is reduced. In terms of space usage, it would be nice to have a reduced BDD.

Definition 6.3.1 : Reduced BDD
A BDD G is reduced if:

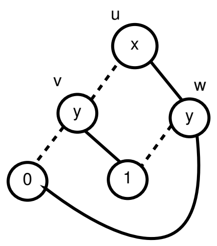
- for any non-leaf node v, $low(v) \neq high(v)$. Otherwise G can be simplified.
- for any distinct nodes v and v', the subgraphs rooted at them are *not* isomorphic. Otherwise G can be simplified. The meaning of 'isomorphic' is explained below.

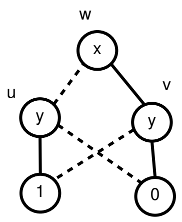
□

Two graphs may look different, but yet structurally the same. They are then called isomorphic. More precisely:

Definition 6.3.2 : Isomorphic BDD
Two BDDs G and H are isomorphic if we can obtain H from G by renaming G's nodes, and vice versa. You are not allowed to rename $var(v)$ nor $val(v)$.
□

For example, the two BDDs below are isomorphic:





Note that since a BDD may only have one root, and the children of any node are distinguished (low and high), the choice for the renaming function σ, that map nodes of G to those of H, is actually quite constrained. It must map G's root to H's root, and the root's low (high) child in G to the corresponding root's low (high) child in H, and so on all the way down to the terminal nodes. Hence, testing isomorphism on BDDs is quite simple; it can be done in $O(N)$, where N is the number of nodes in G or H, whichever the smallest one is.

Suppose we introduce a certain linear ordering among the 'variables' (the decorations of the nodes in a BDD), e.g. x, y, z. You could also choose y, x, z; what matters now is that you fix an ordering. If x and y are two variables, we will write $x \prec y$ to mean that in this ordering x preceeds y.

Definition 6.3.3 : Ordered BDD
An *ordered* BDD (OBDD) is a BDD G such that the variables along any path in the BDD respect the ordering. More precisely, for any non-leaf node v:

$$ind(v) \prec ind(low(v)) \quad \text{and} \quad ind(v) \prec ind(high(v))$$

□

For example, if x, y is our ordering, then all the above BDDs are ordered, and else not.

An important propertry of OBDD is the following:

Theorem 6.3.4 :
If you fix the ordering of the variables, then for every Boolean function f there is a unique (modulo isomorphism) reduced OBDD (with respect to the given ordering) that represents this f.
□

The 'modulo isomorphism' here means that if there are multiple reduced OBDDS representing f, they will all be isomorphic to each other.

To put it differently, OBDD can be used as the cannonical representation for Boolean formulas. Note however, the representation is bound to the variable ordering you chosed. If you change the ordering, you may get a different OBDD representation. Some orderings yield the optimal (smallest) OBDD, some may do the opposite.

The above theorem implies that to check whether two formulas f and g are equivalent, we can do it by comparing if their OBDDs are isomorphic

6.3.1 Checking Satisfaction

A Boolean function f is *satisfiable* if it is possible to find values for its parameters that would make the function to return 1. The function is valid if for all possible values of its parameters, it always returns 1. If we can check satisfiability, we can also check validity: f is valid if and only if $\neg f$ is not satisfiable.

To check if f is satisfiable, we can first construct its OBDD F, and then checks whether the leaf 1 is reachable from F's root. Actually, just checking whether F has 1 as a leaf will do. But usually you also want to know the values of the variables that would statisfy f. This corresponds to finding a path in F that leads to the leaf 1. This can be checked with e.g. a DFS in $O(N + E)$ time.

6.3.2 Constructing OBDD

Given a formula p, we still need an algorithm that would construct its OBDD. One way to do it is to first construct a binary decision tree for it, and then reduce it. Well, the tree will take exponential size, although after the reduction can get back much of the used space. However, it may happen that for example we already have OBDDs for p and q, but now we want to construct the OBDDs of $p \wedge q$; this can be done more efficiently.

Reduce

This algorithm is used to reduce an OBDD. The input of the algorithm is an OBDD G. It returns a OBDD H, such that it represents the same Boolean function as G, and furthermore H is a reduced OBDD.

The idea is to traverse G from bottom to top. At each node v, we check if we have seen another node v' that represent the same formula as v. Those nodes are thus equivalent. Hence, v is redundant, and won't be copied to H. Else it will.

To keep track of which nodes are equivalent, we will maintain a mapping ID. It maps the nodes of G that we have visited, to the nodes of H. If two nodes u and v map to the same w, then they are all equivalent, but we have chosen to pick w to be put in the reduced graph H[2].

We will use the notation $ID(v) = w$ to mean that ID maps v to w. We write $v \in ID$ to mean that ID contains a mapping for v (it maps v to something).

For explaining the algorithm, it is easier if we capture assumptions on ID a bit more formally with invariants, which the algorithm will maintain later.

The fact that ID maps to H is expressed by this invariant:

$$I_0 : \quad v \in ID \ \text{ implies } \ ID(v) \in H \tag{6.1}$$

If $ID(u) = v$ then these nodes are equivalent; thus they represent the same Boolean functions. This is expressed by this invariant:

$$I_1 : \quad u \in ID \text{ implies } func(u) = func(ID(u)) \tag{6.2}$$

For sanity, we add these invariants too:

$$I_2 : \quad u \in H \ \text{ implies } \ ID(u) = u \tag{6.3}$$

$$I_3 : \quad v \in H \text{ and } v \text{ is non-leaf implies } low(v), high(v) \in H \tag{6.4}$$

The algorithm proceeds as follows:

1. We innitialize the mapping ID to empty.
2. For every leaf v in G we do:
   (a) If H contains no leaf of the same value as v, we add v to H. We add an identity-entry $v \mapsto v$ to the mapping ID.
   (b) If H already contains a leaf v' of the same value ($val(v) = val(v')$), then v is redundant. We do *not* add v to H. We do add the entry $v \mapsto v'$ to the mapping ID.
3. We proceed recursively, along the index-numbers of the variables decorating G's nodes. We process the high index first, then move to the lower ones. Visually, this is as if we start from the bottom of the graph and proceed towards the top.

   Now, assume all nodes $u \in G$ such that $ind(u) > i$ have been processed.

   We proceed processing the nodes v with $ind(v) = i$ (there may be multiple such v). Note first that the Boolean formula represented by v is:

   $$func(v) \ = \ ind(v).func(high(v)) \ \vee \ \overline{ind(v)}.func(low(v))$$

   For every such v, note that its children $low(v)$ and $high(v)$ would have higher indices than v. Therefore they are in ID. Next, we do:

[2]To explain the name "ID", it is because that is how it is called in the original algorithm [5]. However, the original ID maps a node to a unique number, which can be thought as a new Identity for the node. We use ID differently, because this is easier. However, abstractly we are still doing the same thing. We still keep the name ID to make it easier for you to relate this account of the algorithm to the original one.

(a) If $ID(high(v)) = ID(low(v))$, it follows from I_1 that:

$$func(high(v)) \;=\; func(low(v)) \;=\; func(ID(low(v)))$$

And therefore:

$$\begin{aligned} func(v) &= ind(v).func(high(v)) \;\vee\; \overline{ind(v)}.func(low(v)) \\ &= ind(v).func(low(v)) \;\vee\; \overline{ind(v)}.func(low(v)) \\ &= func(low(v)) \\ &= func(ID(low(v))) \end{aligned}$$

H must have already contained $ID(low(v))$, due to I_0. But the Boolean formula represented by v is just the same as that represented by $ID(low(v))$. So, v is redundant! Therefore we do *not* add it to H.

We do add the entry $v \mapsto ID(low(v))$ to ID.

(b) If H contains v' decorated with the same variable-name as that of v (so, $ind(v') = ind(v)$), and:

$$ID(low(v)) = low(v') \quad \text{and} \quad ID(high(v)) = high(v')$$

Then it follows, by I_2, that:

$$f(low(v)) = f(low(v')) \quad \text{and} \quad f(high(v)) = f(high(v'))$$

Therefore:

$$\begin{aligned} f(v) &= ind(v).f(high(v)) \;\vee\; \overline{ind(v)}.f(low(v)) \\ &= ind(v').f(high(v)) \;\vee\; \overline{ind(v')}.f(low(v)) \\ &= ind(v').f(high(v')) \;\vee\; \overline{ind(v')}.f(low(v')) \\ &= f(v') \end{aligned}$$

So, the Boolean formula represented by v is just the same as that represented by v'. Then v is redundant. We do not add it to H.

We do add the entry $v \mapsto ID(v')$ to ID.

(c) If neither (a) nor (b) holds, then v is not redundant. It is added to H, and the entry $v \mapsto v$ is added to H.

Furthermore, we change the children of v:

- set $v.low$ to $ID(v.low)$, and
- set $v.high$ to $ID(v.high)$, and

For a more efficient work out of this algorithm, see [5], which is $O(|G| * log|G|)$.

Apply

If we already have the OBDDs for f and g, we can combine them to construct an OBDD for $f \wedge g$. We will give an general algorithm to do this for all Boolean binary operators. It even works for $\neg$ because $\neg p = p$ **xor** 1. Let $\otimes$ below be any Boolean binary operator.

Let f and g be two Boolean functions over the same set of parameters, represented by OBDD F and respectively G. Let x be one of these parameters. First, notice that we have this equality:

$$f \otimes g \;=\; x.(f[1/x] \otimes g[1/x]) \;\vee\; \bar{x}.(f[0/x] \otimes g[0/x]) \tag{6.5}$$

where $f[1/x]$ means the function we would obtain if we replace x with 1.

The above suggests that we can perhaps try to construct the OBDD of $f \otimes g$ recursively.

The algorithm will be called $apply(F, G, \otimes)$. It takes the OBDDs F and G as inputs, and the operation to apply. It will return a new OBDD, such that (specification of 'apply'):

$$func(F) \otimes func(G) \;=\; func(apply(F, G, \otimes))$$

Suppose r_F and r_G are the roots of F and respectively G. We will write $low(F)$ to denote the subgraph of F, rooted at $low(r_F)$. Similarly, we define $high(F)$.

We have a number of cases:

1. If F and G both turn out to only contain one leaf. So, they represent the constant functions, which are either 0 or 1.

   In this case, we directly calculate $val(r_F) \otimes val(r_G)$, and construct the BDD representing the result.

2. Else, at least one of the two graphs are non-trivial. We have these cases:

   (a) The roots r_F and r_G are decorated with the same variable (so, $ind(r_F) = ind(r_G)$); suppose that this variable is x. Now, according to the equality above we have:

   $$f \otimes g \quad = \quad x.(f[1/x] \otimes g[1/x]) \ \vee \ \bar{x}.(f[0/x] \otimes g[0/x])$$

   where $f = func(F)$ and $g = func(G)$.
   However, because x is the variable at the root, notice that:

   $$f[1/x] \ = \ func(high(F)) \quad \text{and} \quad f[0/x] \ = \ func(low(F))$$

   And similarly for g. Therefore:

   $$\begin{aligned} func(F) \otimes func(G) \quad &= \quad x.(f[1/x] \otimes g[1/x]) \ \vee \ \bar{x}.(f[0/x] \otimes g[0/x]) \\ &= \quad \begin{array}{l} x.(func(high(F)) \otimes func(high(G))) \\ \vee \\ \bar{x}.(func(low(F)) \otimes func(low(G))) \end{array} \\ &= \quad x.apply(high(F), high(G), \otimes) \ \vee \ \bar{x}.apply(low(F), low(G), \otimes) \end{aligned}$$

   which shows how to recursively calculate the result of *apply*.

   (b) Both r_F and r_G are decorated with a different variable, say x and y respectively, and suppose $x \prec y$ (the case when $y \prec x$ is symmetrical). Because G is an OBDD (*ordered* BDD), this implies that it has no node labelled with x. So, its function does not depend on x. So:

   $$g[0/x] \ = \ g[1/x] \ = \ g$$

   Therefore:

   $$\begin{aligned} func(F) \otimes func(G) \quad &= \quad x.(f[1/x] \otimes g[1/x]) \ \vee \ \bar{x}.(f[0/x] \otimes g[0/x]) \\ &= \quad x.(func(high(F)) \otimes g) \ \vee \ \bar{x}.(func(low(F)) \otimes g) \\ &= \quad x.apply(high(F), G, \otimes) \ \vee \ \bar{x}.apply(low(F), G, \otimes) \end{aligned}$$

   which shows how to recursively calculate the result of *apply*.

   (c) One of r_F and r_G is a leaf, and the other is not. Suppose r_G is the leaf. This implies that $func(G)$ is a constant function, thus does not depend on x. So:

   $$g[0/x] \ = \ g[1/x] \ = \ g$$

   This is the same situation as we had in (b) above, and can be handled in the same way.

Now this algorithm, if implemented as is, it is inefficient. E.g. consider subgraphs F' and G' of F respectively G. Eventually, we must recurse on them, thus calculating $apply(F', G', \otimes)$. However because we are dealing with graphs, F' and G' may have multiple parents, which will cause $apply(F', G', \otimes)$ to be called multiple times. Because this may happen again and again, as we recurse down the graphs, the overall complexity is exponential.

A simple trick to get around this to store and keep track that we have calculated $apply(F', G', \otimes)$. So the next time we need to do it again, we don't have to calculate, but can just retrieve the result from memory. This reduces the complexity to $O(|F| * |G|)$. For further tweaks, see [5].

Substitution

Let f be a Boolean function over a set of parameters. Let x be one of the parameters. Suppose we have already constructed an OBDD F that represents f. We now want to construct an OBDD representing $f[1/x]$, and analogously $f[0/x]$. Note that here x is not necessarily the root variable of F.

I'll leave this to you :)

Overview of complexity

Here is an overview of the complexity of various operations on OBDD. This is based on the original algorithms in [5]. Below, F and G are OBDDs; and let f and g be the Boolean functions represented them. $|F|$ is the number of nodes in F.

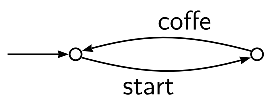
$satisfy(F)$	$O(\|F\|)$	// check if f is satisfiable
$reduce(F)$	$O(\|F\| * log\|F\|)$	
$apply(F, G, \otimes)$	$O(\|F\| * \|G\|)$	
$subst(F, x, constant)$	$O(\|F\| * log\|F\|)$	// construct and reduce the OBDD of $f[0/x]$ or $f[1/x]$

Chapter 7

CSP

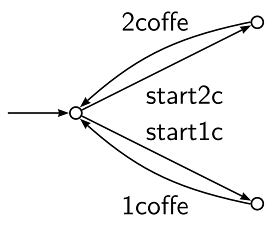
Communicating Sequential Processes (CSP) is a nice formalism you can use to abstractly describe a concurrent system that relies on event synchronization to synchronize its concurrent components. It is introduced by Hoare back in 1978. The original book describing it [11] has been updated (2004) [12] and is available for free at `www.usingcsp.com`.

In CSP a system is described by a 'process'. A CSP process is an *abstraction* of a real program. In this abstraction, the units of execution are *events*. An event has a name, and its execution is atomic (indibisible, cannot be interrupted). For example, a coffee machine can be modelled as a CSP process with two events, start and coffee, which we can describe as the following FSA:



where the event start represents the user action of pushing on the machine's start button, and coffee represents the machine's response of producing coffee.

In CSP there is no concept of program variables as we would have in e.g. C. Events also do not have parameters. You will have to abstract these into undecorated states and unparameterized events such as in the example above. For example, if it is possible for the user of our coffee machine to choose between normal and double quantity coffee, we could model it e.g. like this:



7.1 CSP Syntax and Primitive Operators

However, the above way of describing a process is actually not how CSP does it. It uses a textual expression. The syntax is given below —we will only consider a subset of the full CSP though. A

Process is expressed by one of the following constructs:

$$
\begin{array}{lll}
 & \mathsf{STOP}_{Alphabet} & \\
| & Event \rightarrow Process & \text{– sequential composition, } a \text{ should be in } \alpha P \\
| & Process \,\Box\, Process & \text{– choice, } \alpha P \text{ and } \alpha Q \text{ should be the same} \\
| & Process \sqcap Process & \text{– choice, } \alpha P \text{ and } \alpha Q \text{ should be the same} \\
| & Process \,||\, Process & \text{– parallel composition} \\
| & Process/Alphabet & \text{– hiding} \\
| & Name & \text{– a name of a process}
\end{array}
$$

αP denotes the alphabet of the process P, which is the set of events P can potentially produce; this will be explained more later.

If P and Q are processes, the composition $a{\rightarrow}P$ is a process that does the event a and then it behaves as P. The composition $P\Box Q$ is a process that behaves either as P or as Q. CSP has another choice operator, namely $\sqcap$; we will discuss it later.

$\mathsf{STOP_A}$ is a process that does not do anything; the meaning of A will be explained later. If the choice of A is clear from the context, or if it is not important for the discussion at hand, we will write just STOP instead of STOP_A.

With an equation like:

$$F \;=\; a \rightarrow \mathsf{STOP}_{\{a,b\}}$$

we can give a name to a process and define what it means. So, above we define a process named F, which does the event a then it stops. The process G belows does b then a then it stops:

$$G \;=\; b \rightarrow P$$

We can also define a process recursively, e.g.:

$$Clock \;=\; tick \rightarrow (Clock \,\Box\, \mathsf{STOP}_{\{tick\}})$$

defines a process called *Clock* that can do any number of *tick*s. In the above example it is at least intuitively clear what process is meant. The syntax itself does not prevent someone from writing an equation like this:

$$O \;=\; O \,\Box\, (a \rightarrow \mathsf{STOP}_{\{a,b\}})$$

We will avoid this kind of left recursions, as it is not intuitively clear which process is meant. Theoretically, we can still define the meaning of such an equation as the 'least' process satisfying the equation, if some natural concept of ordering can be defined over processes; but we will not go into this.

7.1.1 Alphabet, External and Internal Events

An *alphabet* is just a set of events. Some events in a process are *external.* The others are *internal.* External events can be observed and interacted to by the 'environment' of the process, which can be a user other processes. In principle, a process described in CSP only specifies its behavior with respect to its external events (so, its behavior as it would appear to its environment). The *alphabet of a process P*, denoted by αP, is the set of P's external events. It is defined as below:

$$
\begin{array}{lcll}
\alpha\mathsf{STOP}_A & = & A & \\
\alpha(a \rightarrow P) & = & \alpha P & , a \in \alpha P \text{ is assumed} \\
\alpha(P \,\Box\, Q) & = & \alpha P & , \alpha P = \alpha Q \text{ is assumed} \\
\alpha(P \,\sqcap\, Q) & = & \alpha P & , \alpha P = \alpha Q \text{ is assumed} \\
\alpha(P \,||\, Q) & = & \alpha P \,\cup\, \alpha Q & \\
\alpha(P/A) & = & \alpha P/A &
\end{array}
$$

If A is an alphabet, the construct P/A coverts the events in A to internal events. The operator '/' is also called *hiding* or *internalization* operator. For example, if we hide *tick* in the *Clock* process defined above, from the environment's perspective the resulting process would be equivalent to STOP. As another example, consider again F and G defined before:

$G/\{b\}$ is equivalent to F

The alphabet of a process defined by an equation $P = rhs$ can be obtained by calculating the alphabet of the right hand side rhs, where occurrences of P are replaced by a STOP with the smallest compatible alphabet. 'Compatible' means that the alphabet of the replacement should respect the composition operators used in the rhs so that the expression is well defined —recall that some composition operators like $\rightarrow$ have constraints on the alphabet of the involved sub-expressions.

For example, consider again the *Clock* example. It alphabet can de determined as follows:

$$\alpha Clock \;=\; \alpha(tick \rightarrow (\mathsf{STOP}_{\{tick\}} \,\Box\, \mathsf{STOP}_{\{tick\}})) \;=\; \{tick\}$$

7.1.2 Parallel Composition

$P||Q$ denotes the *parallel composition* of P and Q. This amounts to interleaved execution of P and Q. However, events that are common in both (so, events in $\alpha P \cap \alpha Q$) must be executed synchronously (together). For example:

$$(a \rightarrow x \rightarrow \mathsf{STOP}_{\{a,x\}}) \mid\mid (b \rightarrow x \rightarrow \mathsf{STOP}_{\{b,x\}})$$

is equivalent to:

$$(a \rightarrow b \rightarrow x \rightarrow \mathsf{STOP}_{\{a,b,x\}}) \,\Box\, (b \rightarrow a \rightarrow x \rightarrow \mathsf{STOP}_{\{a,b,x\}})$$

which is a process that can do either ab or ba (interleaving of respective 'private events') first, then followed by the synchronized execution of the common event x.

In CSP, events can be synchronized by more than just two processes. In the example below, we have three sub-processes that synchronize on the event x:

$$(a \rightarrow x \rightarrow \mathsf{STOP}_{\{a,x\}}) \mid\mid (b \rightarrow x \rightarrow \mathsf{STOP}_{\{b,x\}}) \mid\mid (c \rightarrow x \rightarrow \mathsf{STOP}_{\{c,x\}})$$

On the other hand we may actually intend x to be only synchronized by the first two sub-processes, and the x in the third process is meant to be a different event which happen to have the same name. We can model it as follows, by using the hiding operator:

$$((a \rightarrow x \rightarrow \mathsf{STOP}_{\{a,x\}}) \mid\mid (b \rightarrow x \rightarrow \mathsf{STOP}_{\{b,x\}}))/\{x\} \mid\mid (c \rightarrow x \rightarrow \mathsf{STOP}_{\{c,x\}})$$

7.2 Refinement and Specification

Let us write $P \sqsubseteq Q$, pronounced Q *refines* P, to mean that the process Q behaves in such a way that it can be used to replace P. CSP supports such a concept. With it, we can simply use a process to specify how another process should behave. In $P \sqsubseteq Q$, P would take the role of specification and Q the implementation. For example, to a coffee machine can internally be quite complicated. But we can specify it with respect to its external behavior e.g. as follows:

$$\begin{array}{lcl} CoffeeSpec & \sqsubseteq & CoffeeMachine/\{start, coffee\} \\ CoffeeSpec & = & start \rightarrow coffee \rightarrow CoffeeSpec \end{array}$$

But how do we verify it? We will return to this latter. First, let us first try to give a semantical definition to CSP processes so that we can at least have a formal definition of $\sqsubseteq$. I will first show you a so-called *trace semantic*. This semantic is incomplete, but at least intuitive to explain.

7.2.1 Trace Semantic

A *trace* is just a sequence of events. Let us just stick to the notation used in [12] to denote traces:

$\langle\rangle$	is the empty trace
$\langle a\rangle$	is a singleton trace containing a
$\langle a,b,c\rangle$	is a trace with three elements
$s\,\hat{}\,t$	concatenation of two traces
$s{\restriction}A$	projection or restriction: the trace obtained by throwing away events *not* in A

Furthermore, if A is an alphabet, then A^* denotes all finite traces over the events in A.

Let's use traces to model executions of a process. Each is a sequence of events that the process can produce. For example the process:

$$R_1 \;=\; a \to b \to \mathsf{STOP}$$

when executed can exibit three traces: $\langle\rangle$, $\langle a\rangle$, and $\langle a,b\rangle$. The set of traces that a process P can exibit is denoted by $\mathbf{traces}(P)$. Another example:

$$R_2 \;=\; (a \to a \to \mathsf{STOP}) \;\Box\; (b \to a \to \mathsf{STOP})$$

$\mathbf{traces}(R_2) = \{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle a,a\rangle, \langle b,a\rangle, \}$ In general, the traces of a process can be calculated as follows:

$$\begin{array}{lcl}
\mathbf{traces}(\mathsf{STOP}) & = & \emptyset \\
\mathbf{traces}(a \to P) & = & \{\langle\rangle\} \;\cup\; \{\langle a\rangle\,\hat{}\,s \mid s \in \mathbf{traces}(P)\} \\
\mathbf{traces}(P \;\Box\; Q) & = & \mathbf{traces}(P) \;\cup\; \mathbf{traces}(Q) \\
\mathbf{traces}(P \;\sqcap\; Q) & = & \mathbf{traces}(P) \;\cup\; \mathbf{traces}(Q) \\
\mathbf{traces}(P/A) & = & \{s{\restriction}(\alpha P/A) \mid s \in \mathbf{traces}(P)\}
\end{array}$$

Notice in particular how the semantic of $a \to P$ is defined. Its set of traces consists not only of what it can 'maximall' do, but also all the prefixes in between. For example, the traces of $a \to b \to \mathsf{STOP}$ do not only consist of the maximal $\langle a,b\rangle$, but also all its prefixes $\langle\rangle$ and $\langle a\rangle$. In fact, it is a general property of our trace semantic:

Theorem 7.2.1 : Prefix Closed-ness
If s is a prefix of t and $t \in \mathbf{traces}(P)$, then $s \in \mathbf{traces}(P)$.
□

What is the traces of $P||Q$? Note that that s is a trace of this composition if and only if when projected on αP it must be a trace of P, and similarly when projected on αQ it is a trace of Q. This leads to this formula:

$$\mathbf{traces}(P||Q) \;=\; \{s \mid s \in (\alpha(P||Q))^*,\; s{\restriction}\alpha P \in \mathbf{traces}(P),\; s{\restriction}\alpha Q \in \mathbf{traces}(Q)\;\}$$

For a process defined by a non-recursive equation $P = rhs$, its traces is simply the traces of the right hand side formula rhs. If the equation is recursive, we define its set of traces as the smallest solution to:

$$\mathbf{traces}(P) \;=\; \mathbf{traces}(rhs)$$

For example, consider this variation of *Clock*:

$$Clock_2 \;=\; tick \to tock \to (Clock_2 \;\Box\; \mathsf{STOP})$$

Its traces is then the smallest solution of:

$$\mathbf{traces}(Clock_2) \;=\; \mathbf{traces}(tick \to tock \to (Clock_2 \;\Box\; \mathsf{STOP}))$$

Using the other definitions, we can work out the right hand side, and reduce the above equation to the following:

$$\textbf{traces}(Clock_2) \;=\; \{\langle\rangle, \langle a\rangle\} \;\cup\; \{\langle a,b\rangle \,\hat{}\, s \mid s \in \textbf{traces}(Clock_2)\;\}$$

The smallest solution of the above equation is (which you probably already guess):

$$\textbf{traces}(Clock_2) \;=\; \{\langle a,b\rangle^k \mid k \geq 0\}$$

Now we can define what $\sqsubseteq$ means. Let us choose for a rather conservative definition. Q refines P if it *cannot* do something that P won't be able to do. So, if P cannot blow up your computer, and $P \sqsubseteq Q$, it follows that Q won't blow up the computer either. Note that by this definition, a process that does nothing will refine any specification. We will deal with this later. For now, we define:

Definition 7.2.2 : Trace-based Refinement

$$P \sqsubseteq Q \;=\; \alpha P{=}\alpha Q \;\wedge\; \textbf{traces}(Q) \subseteq \textbf{traces}(P)$$

□

Unfortunately, as we have seen in the above example, the set of traces of even a small process can be infinitely large. So, verifying $P \sqsubseteq Q$ by directly using the above definition is not going to work.

7.3 FSA Semantic

An FSA can also be used to describe an event-based system, e.g. as the coffee machine examples at the beginning of this Chapter. You can perhaps even say that it is a more direct way of describing such a process, since the states and how the events move the process from one state to another are described explicitly. In any case, FSAs seem to be expressive enough to be used as the semantic of CSP processes.

Let's take FSAs with structures as described in Definition 4.1.1. In particular, the arrows are labelled. Such an FSA M is described by a tuple (S, s_0, A, R) where S is a set of states, s_0 is an initial state (we will only use one initial state), A is the alphabet of M (its set of events), and R is a function that described the arrows in M. For any state $s \in S$ and any event $a \in A$, $R(s,a)$ gives the set of possible next states of M if it performs a on the state s.

Additionally, we introduce a special event denoted by τ representing internal event (there may be different sorts of internal events, but we will not distinguish between them). An arrow in M can also be labelled with τ. So, if there is an arrow from the state s to t labelled with τ then $t \in R(s,\tau)$.

A trace of M is a sequence of events that can be produced by following a path in M, starting from its initial state. We however remove τ-events from such a trace (in other words, a 'trace' of M consists only of events in A, and does not contain any τ). We write $\textbf{trace}(M)$ to denote the set of all traces that M can produce.

Let $\textbf{fsa}(P)$ denote the FSA that corresponds to P. We want this FSA to equivalently describe P, in the sense that:

$$\textbf{trace}(\textbf{fsa}(P)) \;=\; \textbf{trace}(P) \tag{7.1}$$

This FSA of P can be constructed as described below:

1. $\textbf{fsa}(\mathsf{STOP}_A) \;=\; (\{s_0\}, s_0, A, R)$. R is the empty relation: $R(s_0, a) = \emptyset$, for all $a \in A$.
2. Let $\textbf{fsa}(P) = (S, s_0, A, R)$. Then $\textbf{fsa}(a \to P)$ is constructed as follows:

   $$(\{t_0\} \cup S, t_0, \{a\} \cup A, R')$$

   where R' is R, but to represent the sequencing, it is extended with the mapping $R(t_0, a) = \{s_0\}$. Notice that the new FSA has a new initial state as well.

3. Let **fsa**$(P) = (S_1, s_1, A, R_1)$ and **fsa**$(Q) = (S_2, s_2, A, R_2)$. Let (S_3, s_3, A, R_3) be the FSA of $P \,\Box\, Q$. Let us first construct it as below. The construction is intuitive, but is actually **not entirely correct**. We will return to this later.

   (a) $S_3 = \{s_3\} \cup S_1 \cup S_2$. Notice that s_3 is the new initial state.

   (b) R_3 is the 'union' of R_1 and R_2. Additionally, to represent the choices we additionally we add τ-arrows from the new initial state s_3 to s_1 and s_2. So:

$$\begin{array}{rcl} R_3(t,a) & = & \begin{cases} R_1(t,a) & \text{, if } t \in S_1 \\ \\ R_2(t,a) & \text{, if } t \in S_2 \end{cases} \\ \\ R_3(s_3,\tau) & = & \{s_1, s_2\} \end{array}$$

   otherwise $R_3(t,a) = \emptyset$.

4. The FSA of $P \sqcap Q$ is constructed in the same way as $P \,\Box\, Q$.

5. Let **fsa**$(P) = (S, s_0, A, R)$. Then **fsa**(P/B) is $(S, s_0, A/B, R')$, where R' is obtained from R by replacing arrows labelled with $a \in B$ with arrows labelled with τ:

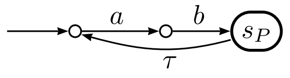
$$\begin{array}{rcll} R'(s,a) & = & R(s,a) & \text{if } a \in A/B \\ R'(s,\tau) & = & R(s,a) & \text{if } a \in A \cap B \end{array}$$

   otherwise $R'(t,a) = \emptyset$.

6. The FSA of $P||Q$ is constructed by taking the interleaving product of **fsa**(P) and **fsa**(Q). This is as discussed in Subsection 4.2.2 (more specifically, the variant with synchronized events).

7. When P is defined by an equation $P = rhs$, its FSA is just the FSA of rhs. However, if the equation is recursive (but not left recursive), in the construction of **fsa**(rhs) we replace every occurrence of P in rhs by a an automaton that only consist of a single state s_P, with no arrow. Then, we add a τ arrow from this s_P to the initial state of **fsa**(rhs).

   For example, the FSA of $P = a \to b \to P$ is (shown graphically):



$$\begin{tikzcd} {} \arrow[r] & \circ \arrow[r, "a"] & \circ \arrow[r, "b"] & \boxed{s_P} \arrow[ll, bend left, "\tau"] \end{tikzcd}$$

As a side note, if we allow quantified equations in CSP, we can write something like:

$$\begin{array}{rcl} Aircraft_0 & = & (flyUp \to Aircraft_1) \,\Box\, (landing \to \mathsf{STOP}) \\ Aircraft_{i+1} & = & (flyUp \to Aircraft_{i+2}) \,\Box\, (flyDown \to Aircraft_i) \end{array}$$

Which gives us additional expressiveness; but, such processes cannot be expressed with a *finite* state automaton.

7.3.1 Refinement Checking with FSA

Since **FSA**(P) has the same traces as P, the problem of checking $P \sqsubseteq Q$ can be reduced to checking traces inclusion in the corresponding FSAs:

Theorem 7.3.1 :

$$P \sqsubseteq Q \quad = \quad \textbf{traces}(\textbf{fsa}(Q)) \subseteq \textbf{traces}(\textbf{fsa}(P))$$

$\Box$

But note this was using $\sqsubseteq$ defined in terms of traces (Definition 7.2.2).

Let us first define some concepts and notations. Let M be an FSA. We write **initials**(M) is the set of events that M can do as its first event. If $M = (S, s_0, A, R)$ is a *deterministic* FSA , its initials can be calculated as follows:

$$\textbf{initials}(M) \;=\; \{a \mid a \in A, R(s_0, a) \neq \emptyset\ \} \tag{7.2}$$

Note by the way that every non-deterministic FSA can be systematically converted into an equivalent deterministic FSA.

Analogously, if s is a state, $\textbf{initials}_M(s)$ is the set of first events that M can do when it is it the state s:

$$\textbf{initials}_M(s) \;=\; \{a \mid a \in A, R(s, a) \neq \emptyset\ \} \tag{7.3}$$

If σ is a trace of M, at the end of σ M must be in a certain state t. Since M was assumed to be deterministic, there is only one possible such t for each σ, denoted by $\textbf{endstate}_M\sigma$. The initials of σ is defined as the initials of t:

$$\textbf{initials}_M(\sigma) \;=\; \textbf{initials}_M(\textbf{endstate}_M(\sigma)) \quad \text{, assuming } \sigma \in \textbf{traces}(M) \tag{7.4}$$

Now, the theorem below can be proven, which shows an alternative way to check traces inclusion (crucial for its proof is the observation that traces of an FSA is prefix-closed):

Theorem 7.3.2 :
Let M and L be two FSAs with the same set of events.

$$\textbf{traces}(M) \subseteq \textbf{traces}(L) \;=\; (\forall\sigma : \sigma \in \textbf{traces}(M \cap L) : \textbf{initials}_M(\sigma) \subseteq \textbf{initials}_L(\sigma))$$

where the intersection $M \cap L$ is as defined in Definition 4.2.1.
□

The above still does not help us, because it requires us to quantify over $\textbf{traces}(M \cap L)$, which can be infinite. However, if we first reduce M and K so that they are now deterministic, then as pointed out before, executing σ would bring M to a unique state s and L to unique state t. The initials of σ of each FSA is just the same as the initials of this s respectively t. Furthermore, note that in the construction of $M \cap L$ as in Definition 4.2.1 is state of $M \cap L$ is actually a pair (v, u) where v is a state of M and u is a state of L. We can prove the following:

Theorem 7.3.3 :
Let M and L be two deterministic FSAs with the same set of events.

$$\textbf{traces}(M) \subseteq \textbf{traces}(L) \;=\; (\forall(v,u) : (v,u) \in \textbf{states}(M \cap L) : \textbf{initials}_M(v) \subseteq \textbf{initials}_L(u))$$

where the intersection $\textbf{states}(M \cap L)$ is the set of all states of $M \cap L$.
□

Importantly, notice that now we quantify over the states of $M \cap L$. There are only finitely many of them. Consequently, the above checking can be done in finite time. Consequently, refinement checking based on the trace semantic can also be done in finite time. Figure 7.1 shows a DFS-based algorithm to do this; Figure 7.1 shows an iterative version of it.

```
--  $M = (S_1, s_1, A, R_1)$ and $L = (S_2, s_2, A, R_2)$ are deterministic FSAs
--  this checks whether $\mathbf{traces}(M) \subseteq \mathbf{traces}(L)$
checkInclusion($M, L$) {
   checked := $\emptyset$ ;
   DFS($s_1, s_2$)
   where
   DFS($t_1, t_2$) {
      if(($t_1, t_2$) $\in$ checked) return ;
      checked := checked $\cup$ $\{(t_1, t_2)\}$ ;
      if($\mathbf{initials}_M(s_1) \not\subseteq \mathbf{initials}_L(s_2)$) throw Violation ;
      forall ($u_1, u_2$) such that $(\exists a :\ a{\in}A :\ u_1{\in}R_1(t_1, a) \ \wedge\ u_2{\in}R_2(t_2, a))$
         DFS($u_1, u_2$)
   }
}
```

Figure 7.1: *A DFS-based recursive algorithm to check traces-inclusion.*

```
--  $M = (S_1, s_1, A, R_1)$ and $L = (S_2, s_2, A, R_2)$ are deterministic FSAs
--  this checks whether $\mathbf{traces}(M) \subseteq \mathbf{traces}(L)$
checkInclusion($M, L$) {
   checked := $\emptyset$ ;
   pending := $\{(s_1, s_2)\}$   ;
   while(pending $\neq \emptyset$) {
      ($t_1, t_2$) := an element retrieved from pending ;
      checked := checked $\cup$ $\{(t_1, t_2)\}$ ;
      if($\mathbf{initials}_M(s_1) \not\subseteq \mathbf{initials}_L(s_2)$) throw Violation ;
      pending := pending
                 $\cup$
                 $\{(u_1, u_2) \mid (\exists a :\ a{\in}A :\ u_1{\in}R_1(t_1, a) \ \wedge\ u_2{\in}R_2(t_2, a) \ \wedge\ (u_1, u_2){\notin}\texttt{checked})\}$
   }
}
```

Figure 7.2: *An iterative algorithm to check traces-inclusion.*

7.4 Enforcing Progress

We have chosen for a conservative refinement relation. It makes sure that e.g. an implementation Q won't blow up anything unless it is allowed by its specification P. Unfortunately, it does not enforce progress. For example a Q that does nothing will by our definition satisfy any specification.

Enforcing progress does bring some new complication. Consider the parallel composition:

$$(a \to b \to \mathsf{STOP}_{\{a,b\}}) \mid\mid (b \to a \to \mathsf{STOP}_{\{a,b\}})$$

Obviously this composition will get stuck. Both a and b are common events of the two subprocesses, so they have to synchronize on them. But this is not possible. In other words, the composition comes to a deadlock after $\langle\rangle$. Consider now this composition:

$$(a \to b \to \mathsf{STOP}_{\{a,b\}}) \mid\mid ((a \to \mathsf{STOP}_{\{a,b\}}) \,\square\, (b \to \mathsf{STOP}_{\{a,b\}}))$$

Obviously, it will deadlock after the first event. But will is also deadlock at $\langle\rangle$? That depends on how the choice in $(a \,\square\, b)$ is made. If the subprocess $(a \,\square\, b)$ itself internally decides which choice it takes, then yes: the above composition *may* deadlock at $\langle\rangle$.

But if it is the 'environment' that dictates the choice, the the above composition won't deadlock at $\langle\rangle$. So, when progress is an issue, then we need to distinguish between 'external' and 'internal' choices. In CSP, the $\square$ operator expresses external choice, whereas $\sqcap$ expresses internal choice. So, the above composition indeed won't deadlock at $\langle\rangle$, whereas its variant below may deadlock at $\langle\rangle$:

$$(a \to b \to \mathsf{STOP}_{\{a,b\}}) \mid\mid ((a \to \mathsf{STOP}_{\{a,b\}}) \;\sqcap\; (b \to \mathsf{STOP}_{\{a,b\}}))$$

When making its choice, CSP assumes the environment to only look at what it can do next (we do not assume the evironment to be smart enough to look beyond that). Consider this example, where $\alpha P = \alpha Q = \{a, b, c\}$:

$$P \mid\mid Q \;\; \text{where: } Q \;=\; ((a \to b \to \mathsf{STOP}_{\{a,b,c\}}) \,\square\, (a \to c \to \mathsf{STOP}_{\{a,b,c\}}))$$

Q offers a choice between doing $\langle a, b\rangle$ or $\langle a, c\rangle$. But P cannot see that far ahead. It only sees that the first possible event to do is a. That means that Q itself will then have to make the choice whether it then go to the first branch or the other. In other words, this choice becomes internal. It could be that Q takes the first branch, and thus its next possible event is b. If $P = a \to c \to \mathsf{STOP}_{\{a,b,c\}}$ it will then deadlock at that point. With respect to P, Q behaves actually as:

$$Q \;=\; a \to ((b \to \mathsf{STOP}_{\{a,b,c\}}) \;\sqcap\; (c \to \mathsf{STOP}_{\{a,b,c\}}))$$

The simplistic trace semantic given in Subsection 7.2.1 cannot distinguish between external and internal choice. In otherwords, the traces of $P \,\square\, Q$ and $P \sqcap Q$ are the same. Below we will show how to extend this semantic so that we can also enforce progress.

7.4.1 Refusals

Consider these processes, with all STOPs are indexed by $\{a, b\}$:

$$\begin{aligned} P_1 &= (a \to \mathsf{STOP}) \,\square\, (b \to \mathsf{STOP}) \\ P_2 &= a \to b \to \mathsf{STOP} \\ P_3 &= (a \to \mathsf{STOP}) \;\sqcap\; (b \to \mathsf{STOP}) \end{aligned}$$

Suppose we compose them in parallel with an environment whose alphabet is also $\{a, b\}$. The process P_1 will not refuse (unable to synchronize) any first event from the environment, whereas P_2 will refuse b. P_3 may refuse a, if it is the only event the environment can do. Similarly, P_3 may refuse b, if b is the only event the environment can do. However, P_3 *cannot* refuse *both* a and

b. That is, if the environment is able to either synchronize with a or b, then despite the internal choice made by P_3, the composition $P||env$ will be able to do its first event.

A *refusal* of P is a subset A of its alphabet (thus, it ia a set of events) such that when the environment of P offers to synchronize over any event in A (P may choose) as the first event to be executed together with P , it is still possible for P, due to some internal choices it makes, to come to some internal state where P cannot do any event in A (it deadlocks on A).

The refusals-set of P, denoted by **refusals**(P) is the set of *all* P's refusals. Consider again the above examples:

$$\begin{array}{lcl} \textbf{refusals}(P_1) & = & \{\emptyset\} \\ \textbf{refusals}(P_2) & = & \{\emptyset, \{b\}\} \\ \textbf{refusals}(P_3) & = & \{\emptyset, \{a\}, \{b\}\} \end{array}$$

Notice that a refusal is a set, and **refusals** is a set of sets.

One more example, with all STOPs are indexed by $\{a, b, c, d\}$:

$$P_4 \;=\; ((a \to \mathsf{STOP}) \,\Box\, (b \to \mathsf{STOP})) \;\sqcap\; ((c \to \mathsf{STOP}) \,\Box\, (d \to \mathsf{STOP}))$$

If the environment offers $\{a, b, c, d\}$, obviously P_4 won't deadlock. But if the environment only offers $\{a, b\}$ then P_4 may deadlock, namely if it chooses to do $c \,\Box\, d$. Similarly, $\{c, d\}$ can also be refused. However, notice that $\{a, c\}$, or $\{a, b, c\}$, or $\{b, c\}$ will not be refused. P_4's full set of refusals consists of:

1. $\{a, b\}$ and all its subsets.
2. $\{c, d\}$ and all its subsets.

We can generalize this concept. A set $A \subseteq \alpha P$ is a refusal of a trace s, if it is possible for P to externally show the behavior s and comes to some internal state where P cannot do any event in A (it deadlocks on A). Note that due to the process' non-determinism, there may be more than one state it may end up with, after doing s. If just one of these states can refuse A, then by the above definition A is a refusal of s.

We define **refusals**(P/s) as the set of *all* refusals of the trace s in P. And of course, **refusals**(P) = **refusals**($P/\langle\rangle$). Refusals have the following properties:

1. Trivially, the empty set is always a refusal. So, for any trace s of P:

$$\emptyset \in \textbf{refusals}(P/s) \tag{7.5}$$

2. If P can refuse A, it can refuse any subset of A:

$$B \in \textbf{refusals}(P/s) \;\wedge\; A \subseteq B \;\Rightarrow\; A \in \textbf{refusals}(P/s) \tag{7.6}$$

3. If after doing s, P cannot refuse $a \in \alpha P$, then sa must be possible:

$$(\forall A :: a \notin \textbf{refusals}(P/s)) \;\Rightarrow\; s\,\hat{}\,\langle a\rangle \in \textbf{traces}(P) \tag{7.7}$$

In particular the last property gives us a hint on how to enforce progress: the left side of $\Rightarrow$ can be seen as a requirement for P to progress from s to sa. Or generally, it appears that specifying refusals can be used as a way to express progress requirement. So, let's do that: we will extend our CSP semantic to also include the refusals. Somewhat misleadingly, the new semantic is called the 'failures' of P.

Definition 7.4.1 : FAILURES

$$\textbf{failures}(P) \;=\; \{(s, X) \mid s \in \textbf{traces}(P),\; X \in \textbf{refusals}(P/s)\;\}$$

□

Notice that since $\emptyset$ is always a refusal, all traces of P are automatically included in its failures.

Like **traces**, **failures** is also prefix-closed. But furthermore, with respect to the refusals it is also subset-closed:

Theorem 7.4.2 : Downward Closed-ness
If (t, B) is a failure of P (it is a member of $\mathbf{failures}(P)$), then for any prefix s of t and any $A \subseteq B$, (s, A) is also a failure of P.
$\Box$

And now we can define a stronger concept of refinement, namely one based on failures:

Definition 7.4.3 : Failures-based Refinement

$$P \sqsubseteq Q \;=\; (\alpha P = \alpha Q) \;\wedge\; \mathbf{failures}(Q) \subseteq \mathbf{failures}(P)$$

$\Box$

Under this concept of refinement, you can no longer trivially implement e.g. $a \to \mathsf{STOP}_{\{a\}}$ with $\mathsf{STOP}_{\{a\}}$. Another example: you can refine e.g.:

$$(a \to \mathsf{STOP}_{\{a,b\}}) \;\sqcap\; (b \to \mathsf{STOP}_{\{a,b\}})$$

by reducing its non-determinism, to:

$$(a \to \mathsf{STOP}_{\{a,b\}}) \;\Box\; (b \to \mathsf{STOP}_{\{a,b\}})$$

This is as expected. This refinement is also possible with the old trace semantic. But with the failure semantic, the other way around is no longer true.

7.5 Refinement Checking Revisited

To check refinement under the new semantic we will have to revisit our refinement checking approach (Section 7.3). To still do the checking through FSAs we have to make sure that our FSA semantic (Section 7.3) translates every CSP process P to an 'equivalent' FSA, which now mean that $\mathbf{fsa}(P)$ should define the same set of failures as P itself. But before we can come to that, a whole bunch of supporting concepts need to be introduced or redefined first.

7.5.1 Fixing the FSA-semantic

The definition of $\mathbf{fsa}(P)$ in Section 7.3 is good, except for the $\Box$ and $\sqcap$ constructs. They are not distinguished. For the new semantic they should be distinguished. We will keep the definition of $\mathbf{fsa}(P \sqcap Q)$ as it was, namely the same as the *old* definition of $\mathbf{fsa}(P \Box Q)$.

We redefine $\mathbf{fsa}(P \Box Q)$ as follows:

Let $\mathbf{fsa}(P) = M_1 = (S_1, s_1, A, R_1)$ and $\mathbf{fsa}(Q) = M_2 = (S_2, s_2, A, R_2)$. Let (S_3, s_3, A, R_3) be the FSA of $P \Box Q$; it is constructed as follows:

1. $S_3 = S_1 \cup S_2 \cup \{s_3\}$
2. R_3 is just the 'union' of R_1 and R_2. However, we make s_3 the new initial state. All outgoing arrows of s_3 are all the initial arrows of M_1 and M_2. So:

$$\begin{array}{lcll} R_3(t,a) & = & R_1(t,a) & \text{, if } t \in S_1 \\ R_3(u,b) & = & R_2(u,b) & \text{, if } u \in S_2 \\ R_3(s_3,a) & = & R_1(s_1,a) \cup R_2(s_2,a) & \end{array}$$

otherwise $R_3(t, a) = \emptyset$.

7.5.2 Fixing the definition of initials

Let $M = (S, s_0, A, R)$ be an FSA. The initials set of s ($\textbf{initials}_M(s)$) is defined as the set of all events that M can do next after doing s. However, recall that the definitions in (7.3) and (7.2) assume M to be deterministic. For refinement checking with the trace semantic, this is good enough. With the failure semantic, non-determinism matters. So we have to redefine it.

We write $s \overset{\sigma}{\to} t$ to mean that from the state s M can reach the state t by doing the trace σ —interleaving the execution with τ-arrows is allowed. We write $s \overset{\tau^+}{\to} t$ to mean that the state t can be reached from the state s by following one or more τ-arrows. We call such a t a τ-closure of s. We define:

$$\textbf{xchoices}_M(s) \;=\; \{a \mid a \in A,\; R(s,a) \neq \emptyset\} \tag{7.8}$$

It is the set of all events (excluding τ-event!) labelling the outgoing arrows from s (this is what **initials** was in the previous definition). The initials of s are the events that M can do either immediately when it is at the state s, or after doing some τ-steps:

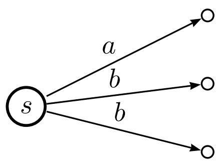
$$\textbf{initials}_M(s) \;=\; \textbf{xchoices}_M(s) \;\cup\; (\cup t : s \overset{\tau^+}{\to} t : \textbf{xchoices}_M(t)) \tag{7.9}$$

Note that this set can be calculated.

7.5.3 Refusals of a state in FSA

A refusal of a state s is a set of events that M can deadlock when it is in that state. The refusals set can be calculated as below. We will distinguish three cases of s. The first two are actually special cases of the third. I show them to help you understanding the third one.

1. Suppose the state s that does not have any outgoing τ-arrow (except to itself), e.g. as the situation below:

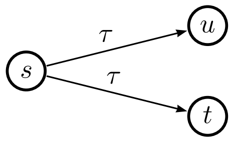


The choice of which arrows to follow will be considered as made by the environment. So, it represents external choices. However, the environment can only choose the event to synchronize. In the above example, the environment can choose between doing a or b; but it cannot choose which of the two b-arrows to follow. The latter is decided internally by M.

In the above example, whan on s, M will thus not refuse any non-empty subset of $\{a, b\}$. Any subset X that does not intersect with $\{a, b\}$ is refused. So, in this case:

$$\textbf{refusals}_M(s) \;=\; \{X \mid X \subseteq A,\; X \cap \textbf{xchoices}_M(s) = \emptyset\;\}$$

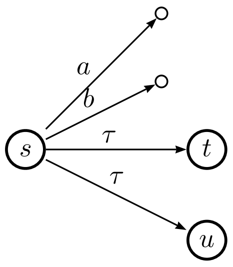
2. Suppose s only have outgoing τ-arrows, e.g. as below:



In this case, when at s, M can refuse any X_1 refused by u and also any X_2 refused by t. But note that $X_1 \cup X_2$ cannot be refused. So:

$$\mathbf{refusals}_M(s) \;=\; \mathbf{refusals}_M(t) \;\cup\; \mathbf{refusals}_M(u)$$

3. Finally, as the third case consider when s has both τ and non-τ arrows. E.g. as below:



The option to do an external event, a or b, is considered to be maintained across τ-transitions. In other words, in the above example the option to do these events is implicitly maintained in the state t and u.

This gives the following definition of refusals of s is:

$$\mathbf{refusals}_M(s) \;=\; (R \cap \mathbf{refusals}_M(t)) \;\cup\; (R \cap \mathbf{refusals}_M(u))$$

where $R = \{X \mid X \subseteq A,\ X \cap \mathbf{xchoices}_M(s) = \emptyset\ \}$.

Finally, we can now define what the failures of an FSA:

Definition 7.5.1 : Failures of FSA
Let $M = (S, s_0, A, R)$ be an FSA.

$$\mathbf{failures}(M) \;=\; \{(\sigma, X) \mid \sigma \in \mathbf{traces}(M), s_0 \overset{\sigma}{\to} t, X \in \mathbf{refusals}_M(t)\ \}$$

We claim that our **fsa** semantic preserves the failures:

$$\mathbf{failures}(P) \;=\; \mathbf{failures}(\mathbf{fsa}(P)) \tag{7.10}$$

Unfortunately, we cannot prove this because we have not given a formal definition of **failures**(P) (the way we did for **traces**(P)). Alternatively, if you consider the way **fsa**(P) and **failures**(M) have been defined as reasonable, we can just as well take the above equation as the definition of **failures**(P).

So now refinement (Definition 7.4.3) can be re-expressed as follows:

$$P \sqsubseteq Q \;=\; \mathbf{failures}(\mathbf{fsa}(Q)) \subseteq \mathbf{failures}(\mathbf{fsa}(P)) \tag{7.11}$$

7.5.4 Fixing DFSA reduction

So, as shown above, refinement checking boils down to checking if the failures of one FSA are included in that of another. To do the latter, we prefer to have deterministic FSAs.

As noted before, any non-deterministic FSA M can be converted to an 'equivalent' deterministic FSA M'. But the standard way of doing such reduction only guarantees that M and M' are traces-equivalent (they produce the same traces). But now we need to make sure that they are also failures equivalent.

Let $M = (S, s_0, A, R)$ be an FSA, let **dt**(M) be a function that construct a deterministic version of M. The construction is shown below; it is the usual way to convert an NDFSA to a DFSA, but it is *extended*. The resulting deterministic FSA is extended with labelling. It has the following structure: $M' = (S', t_0, A, R', lab)$. The elements of M' are as usual, except for the new element $lab : S' \to pow(pow(A))$ that labels every state of M' with a set of refusals. The construction of M' is shown below; each state in S' will be a subset of S.

1. The initial state t_0 is $\{s_0\} \cup \{t \mid s_0 \overset{\tau^+}{\rightarrow} t \text{ in } M\ \}$.
2. We label t_0 with the union of all refusals-set of its members. So:

   $$lab(t_0) \ = \ (\cup t : t \in t_0 : \mathbf{refusals}_M(t))$$

3. We start with $S' = \{t_0\}$ and R' such that we have no arrows. So:

   $$R'(t_0, a) = \emptyset \ \ \text{, for all } a$$

4. For each state $T \in S'$, and for each $a \in A$ such that there is at least one arrow in M from some state in T labelled with a, we construct a new state, namely:

   $$U \ = \ U_{base} \ \cup \ \{u' \mid (\exists u : u \in U_{base} : u \overset{\tau^+}{\rightarrow} u')\}$$

   where

   $$U_{base} \ = \ \{u \mid (\exists t : t \in T : u \in R(t, a))\}$$

   U is then added to S', if it is not already there.

   We add an arrow from T to U, labelled with a. So, we add U to $R(T, a)$.

   The label of U is:

   $$lab(U) \ = \ (\cup u : u \in U : \mathbf{refusals}_M(u))$$

   We repeat this step until no new states can be added to S'; then we are done.

For a deterministic M', recall that a trace σ will lead M' to a unique state u (σ fully determines the state of M' at the end of σ). This u is denoted by $\mathbf{endstate}_{M'}(\sigma)$. Recall that for such M', and a trace σ of M', $\mathbf{initials}(\sigma)$ is defined to be the initials of its end-state. Analogously, we define:

$$\mathbf{refusals}_{M'}(\sigma) \ = \ lab(\mathbf{endstate}_{M'}(\sigma)) \tag{7.12}$$

We claim that the failures of M can be re-expressed in terms of its deterministic version M':

$$\mathbf{failures}(M) \ = \ \{(\sigma, X) \mid \sigma \in \mathbf{traces}(\mathbf{dt}(M)),\ X \in \mathbf{refusals}_{\mathbf{dt}(M)}(\sigma)\ \} \tag{7.13}$$

For checking failures inclusion we can prove the following theorem, which is the analogous of Theorem 7.3.2 —the proof relies on the prefix-closed-ness of traces and downward-closed-ness of refusals.

Theorem 7.5.2 :
Let M and L be two FSAs with the same set of events, $M' = \mathbf{dt}(M)$, and $L' = \mathbf{dt}(L)$.

$$\begin{array}{l} \mathbf{failures}(M) \subseteq \mathbf{failures}(L) \\ = \\ (\forall \sigma : \sigma \in \mathbf{traces}(M' \cap L') : \mathbf{initials}_{M'}(\sigma) \subseteq \mathbf{initials}_{L'}(\sigma) \ \wedge \ \mathbf{refusals}_{M'}(\sigma) \subseteq \mathbf{refusals}_{L'}(\sigma)) \end{array}$$

□

Again, such a theorem cannot directly help us: to check the inclusion we have to quantify over the traces of $M' \cap L'$, whose number can be infinite.

And finally, the analogous of Theorem 7.3.3:

Theorem 7.5.3 :
Let M and L be two FSAs with the same set of events, $M' = \mathbf{dt}(M)$, and $L' = \mathbf{dt}(L)$. Let $lab_{M'}$ and $lab_{L'}$ be the labelling functions of M' respectively L'.

$$\begin{array}{ll}
\mathbf{failures}(M) \subseteq \mathbf{failures}(L) & \\
= & \\
(\forall (V,U) : (V,U) \in \mathbf{states}(M' \cap L') : & \mathbf{initials}_{M'}(V) \subseteq \mathbf{initials}_{L'}(U) \\
& \wedge \\
& lab_{M'}(V) \subseteq lab_{L'}(U)\ )
\end{array}$$

□

The above can be checked systematically, namely by the same algorithm as shown in Figures 7.1 and 7.2, except that we need to extend the check for violation. By the Theorem above, we now need to check:

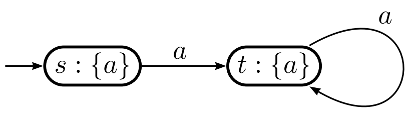
$$\mathbf{initials}_{M'}(V) \subseteq \mathbf{initials}_{L'}(U) \quad \wedge \quad lab_{M'}(V) \subseteq lab_{L'}(U)$$

which indeed can be computed straightforwardly. Thus, we have effectively shown how refinement under the failures semantic can be done.

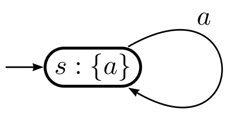
7.5.5 Some notes

The refinement checking algorithms presented here are simpler variations of the one given by Roscoe [20]. In particular, we do not consider the issue of divergence.

The coversion to deterministic FSA as explained in Subsection 7.5.4 corresponds to what in [20] called pre-normal-form transformation. Consider this FSA, which is deterministic:



The alphabet is $\{a\}$. It has two states, s and t, decorated (to save space) only their respective maximal refusals. So, at each at this state, the FSA can either continue doing an a, or to simply stop (because it can refuse $\{a\}$). Our **dt** reduction will not reduce such an FSA further, yet Roscoe pointed out that the two states are in a way equivalent: all their future traces and refusals are indistinguishable. So, he defines the normal form to be the minimum DFSA such that such equivalent states are merged into one. For the above example, the normal form is:



Notice that indeed both FSAs have the same failures sets.

Roscoe orginal algorithm requires that when checking $P \sqsubseteq Q$ that the FSA of P (the specification-side of the refinement) has been normalized as shown in the example above. However, for failures-based checking I think that pre-normal forms will do. Note that using our checking algorithm we can show that each of the above two automata above is refined by the other.

Another difference is that Roscoe does not require the FSA of Q (the implementation-side of the refinement) to be reduced to a DFSA. In contrats, we require both P and Q to be reduced to DFSAs. We did this in favor of simplicity (so it is easier to explain the algorithm). It is possible to drop this requirement for Q, but you also need to tweak the algorithms. You are advised to look at [20].

Bibliography

[1] P. Ammann and J. Offutt. *Introduction to Software Testing.* Cambridge University Press, 2008.

[2] A. Arnold. *Finite Transition Systems, Semantics of Communicating Systems.* Prentice Hall, 1994.

[3] C. Baier and J. P. Katoen. *Principles of Model Checking.* MIT Press, New York, May 2008.

[4] John A Ball et al. *Algorithms for RPN calculators.* Wiley, 1978.

[5] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[6] E. W. Dijkstra. Guarded commands, non-determinacy, and a calculus for the derivation of programs. *Comm. ACM*, 18(8):453–457, August 1975.

[7] E. W. Dijkstra. Semantics of straight-line programs, 1985. EWD 910, `www.cs.utexas.edu/~EWD`.

[8] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[9] Carlo Alberto Furia and Bertrand Meyer. Fields of logic and computation. chapter Inferring Loop Invariants Using Postconditions, pages 277–300. Springer-Verlag, 2010.

[10] W. Hesselink. Predicate-transformer semantics of general recursion. *Acta Informatica*, 26(4):309–332, 1989.

[11] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[12] C.A.R. Hoare. *Communicating Sequential Processes.* 2004. Online at `www.usingcsp.com`.

[13] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[14] E. Lehmann and J. Wegener. Test case design by means of the CTE XL. In *Proceedings of the 8th European Int. Conference on Software Testing, Analysis & Review (EuroSTAR)*, 2000.

[15] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking java programs via guarded commands. Technical Report SRC Technical Note 1999-002, Compaq, 1999. available online at `http://research.microsoft.com/en-us/um/people/leino/papers/SRC-1999-002.pdf`.

[16] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications.* Springer, 2007.

[17] Pierik and de Boer. A proof outline logic for object-oriented programming. *TCS: Theoretical Computer Science*, 343, 2005.

[18] I. S. W. B. Prasetya. Introduction to programming logic, 2014. Lecture Notes for the Course Software Testing and Verification, Utrecht University.

[19] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443–476, 2007.

[20] A. W. Roscoe. Model-checking CSP. In *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378. Prentice Hall, 1994. available online at `www.cs.ox.ac.uk/oucl/work/bill.roscoe/publications/50.ps`.

[21] G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[22] David A. Schmidt. Programming language semantics. In *Encyclopedia of Computer Science*, pages 1463–1466. John Wiley and Sons Ltd., Chichester, UK.

[23] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2), 1955.

Index