Chapter 6

Logic and Action

Overview An action is something that takes place in the world, and that makes a difference to what the world looks like. Thus, actions are maps from states of the world to new states of the world. Actions can be of various kinds. The action of spilling coffee changes the state of your trousers. The action of telling a lie to your friend changes your friend's state of mind (and maybe the state of your soul). The action of multiplying two numbers changes the state of certain registers in your computer. Despite the differences between these various kinds of actions, we will see that they can all be covered under the same logical umbrella.

6.1 Actions in General

> Sitting quietly, doing nothing,
> Spring comes, and the grass grows by itself.
>
> *From: Zenrin kushu, compiled by Eicho (1429-1504)*

Action is change in the world. Change can take place by itself (see the poem above), or it can involve an agent who causes the change. You are an agent. Suppose you have a bad habit and you want to give it up. Then typically, you will go through various stages. At some point there is the *action* stage: you do what you have to do to effect a *change*.

Following instructions for how to combine certain elementary culinary actions (chopping an onion, firing up a stove, stirring the contents of a saucer) may make you a successful cook. Following instructions for how to combine communication steps may make you a successful salesperson, or a successful barrister. Learning to combine elementary computational actions in clever ways may make you a successful computer programmer.

Actions can often be characterized in terms of their results: "stir in heated butter and sauté until soft", "rinse until water is clear". In this chapter you will learn how to use logic for

analyzing the interplay of action and static descriptions of the world before and after the action.

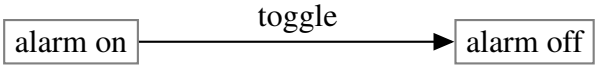
It turns out that *structured actions* can be viewed as compositions of basic actions, with only a few basic composition recipes: *conditional execution*, *choice*, *sequence*, and *repetition*. In some cases it is also possible to *undo* or reverse an action. This gives a further recipe: if you are editing a file, you can undo the last 'delete word' action, but you cannot undo the printing of your file.

Conditional or guarded execution ("remove from fire when cheese starts to melt"), sequence ("pour eggs in and swirl; cook for about three minutes; gently slide out of the pan"), and repetition ("keep stirring until soft") are ways in which a cook combines his basic actions in preparing a meal. But these are also the strategies for a lawyer when planning her defence ("only discuss the character of the defendant if the prosecution forces us", "first convince the jury of the soundness of the alibi, next cast doubt on the reliability of the witness for the prosecution"), or the basic layout strategies for a programmer in designing his code. In this chapter we will look at the logic of these ways of combining actions.

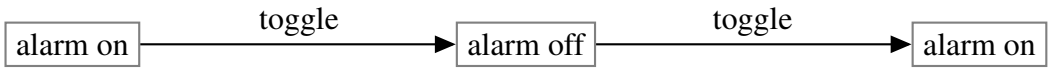
Action structure does not depend on the nature of the basic actions: it applies to *actions in the world*, such as preparing breakfast, cleaning dishes, or spilling coffee over your trousers. It also applies to *communicative actions*, such as reading an English sentence and updating one's state of knowledge accordingly, engaging in a conversation, sending an email with cc's, telling your partner a secret. These actions typically change the cognitive states of the agents involved. Finally, it applies to *computations*, i.e., actions performed by computers. Examples are computing the factorial function, computing square roots, etc. Such actions typically involve changing the memory state of a machine. Of course there are connections between these categories. A communicative action will usually involve some computation involving memory, and the utterance of an imperative ('Shut the door!') is a communicative action that is directed towards action in the world.

There is a very general way to model action and change, a way that we have in fact seen already. The key is to view a changing world as a set of situations linked by labeled arcs. In the context of epistemic logic we have looked at a special case of this, the case where the arcs are epistemic accessibility relations: agent relations that are reflexive, symmetric, and transitive. Here we drop this restriction.

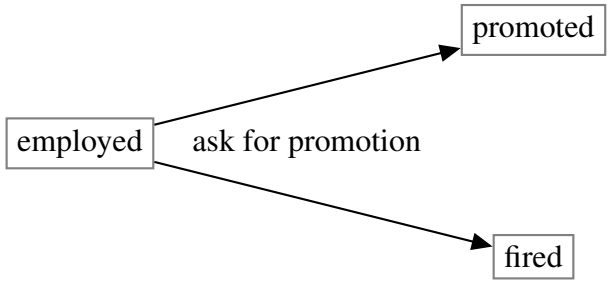
Consider an action that can be performed in only one possible way. Toggling a switch for switching off your alarm clock is an example. This can be pictured as a transition from an initial situation to a new situation:



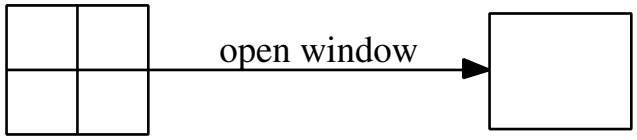
Toggling the switch once more will put the alarm back on:



Some actions do not have determinate effects. Asking your boss for a promotion may get you promoted, but it may also get you fired, so this action can be pictured like this:

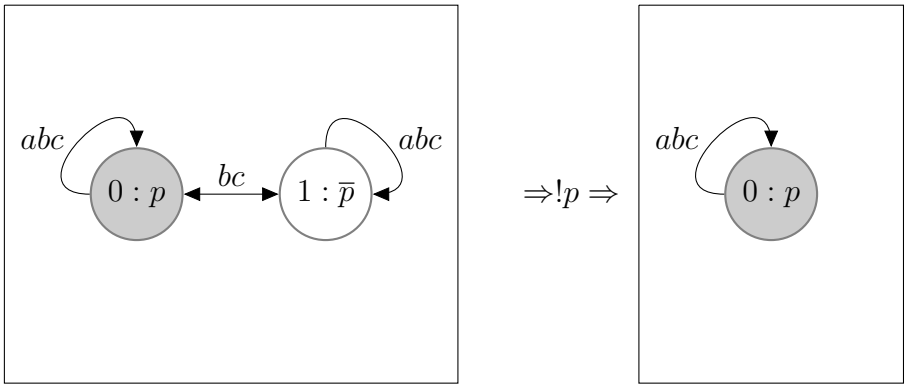


Another example: opening a window. This brings about a change in the world, as follows.



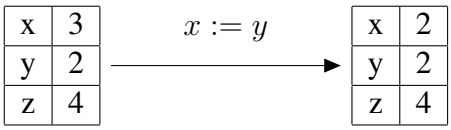
The action of window-opening changes a state in which the window is closed into one in which it is open. This is more subtle than toggling an alarm clock, for once the window is open a *different* action is needed to close it again. Also, the action of opening a window can only be applied to *closed* windows, not to open ones. We say: performing the action has a *precondition* or *presupposition*.

In fact, the public announcements from the previous chapter can also be viewed as (communicative) actions covered by our general framework. A public announcement is an action that effects a change in an information model.



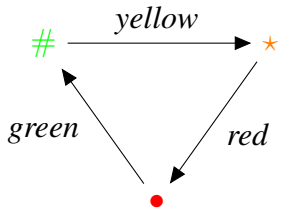
On the left is an epistemic situation where p is in fact the case (indicated by the grey shading), but b and c cannot distinguish between the two states of affairs, for they do not know whether p. If in such a situation there is a public announcement that p is the case, then the epistemic situation changes to what is pictured on the right. In the new situation, everyone knows that p is the case, and everyone knows that everyone knows, and so on. In other words: p has become common knowledge.

Here is computational example. The situation on the left in the picture below gives a highly abstract view of part of the memory of a computer, with the contents of three registers x, y and z. The effect of the assignment action $x := y$ on this situation is that the old contents of register x gets replaced by the contents of register y. The result of the action is the picture on the right.



The command to put the value of register y in register x makes the contents of registers x and y equal.

The next example models a traffic light that can turn from green to yellow to red and again to green. The transitions indicate which light is turned on (the light that is currently on is switched off). The state # is the state with the green light on, the state ⋆ the state with the yellow light on, and the state • the state with the red light on.



These examples illustrate that it is possible to approach a wide variety of kinds of actions from a unified perspective. In this chapter we will show that this is not only possible, but also fruitful.

In fact, much of the reasoning we do in everyday life is reasoning about change. If you reflect on an everyday life problem, one of the things you can do is run through various scenarios in your mind, and see how you would (re)act if things turn out as you imagine. Amusing samples are in the Dutch 'Handboek voor de Moderne Vrouw' (The Modern Woman's Handbook). See `http://www.handboekvoordemodernevrouw.nl/`.

Here is a sample question from 'Handboek voor de Moderne Vrouw': 'I am longing for a cosy Xmas party. What can I do to make our Xmas event happy and joyful?' Here is the recommendation for how to reflect on this:

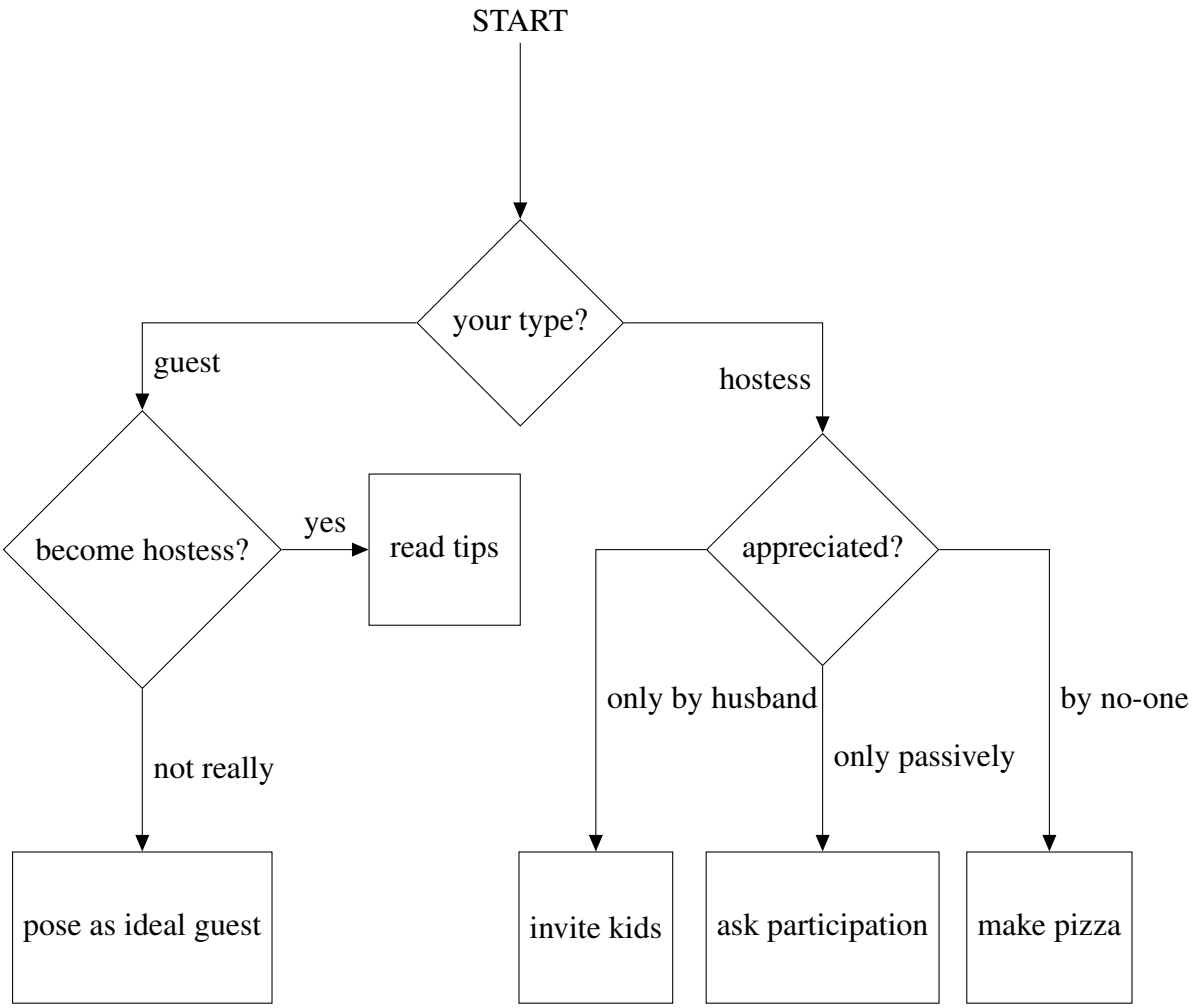


Figure 6.1: Flow Diagram of ‘Happy Xmas Procedure’

Are you the type of a 'guest' or the type of a 'hostess'?
  If the answer is 'guest': Would you like to become a hostess?
    If the answer is 'not really' then
      your best option is to profile as an ideal guest
      and hope for a Xmas party invitation elsewhere.
    If the answer is 'yes' then here are some tips on how to become a great hostess: ...
  If the answer is 'hostess', then ask yourself:
    Are your efforts truly appreciated?
      If the answer is 'Yes, but only by my own husband' then
        probably your kids are bored to death.
        Invite friends with kids of the same age as yours.
      If the answer is 'Yes, but nobody lifts a finger to help out' then
        Ask everyone to prepare one of the courses.
      If the answer is 'No, I only gets moans and sighs' then
        put a pizza in the microwave for your spouse and kids
        and get yourself invited by friends.

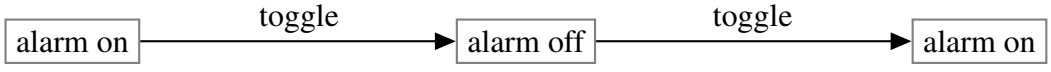
Figure 6.1 gives a so-called flow diagram for the recommendations from this example. Note that the questions are put in $\diamond$ boxes, that the answers are labels of outgoing arrows of the $\diamond$ boxes, and that the actions are put in $\square$ boxes.

6.2 Sequence, Choice, Repetition, Test

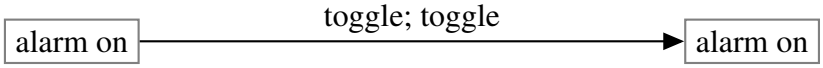
In the logic of propositions, the natural operations are *not*, *and* and *or*. These operations are used to map truth values into other truth values. When we want to talk about action, the repertoire of operations gets extended. What are natural things to do with actions?

When we want to talk about action at a very general level, then we first have to look at how actions can be structured. Let's assume that we have a set of basic actions. Call these basic actions a, b, c, and so on. Right now we are not interested in the internal structure of basic actions. The actions a, b, c could be anything: actions in the world, basic acts of communication, or basic changes in the memory state of a computer. Given such a set of basic actions, we can look at natural ways to combine them.

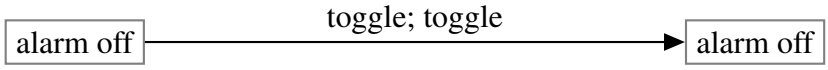
Sequence In the first place we can *perform one action after another*: first eat breakfast, then do the dishes. First execute action a, next execute action b. First toggle a switch. Then toggle it again. Consider again the alarm clock toggle action.



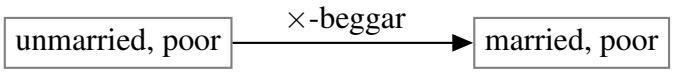
Writing the sequence of two actions a and b as $a; b$, we get:

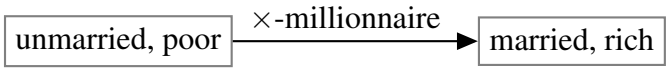


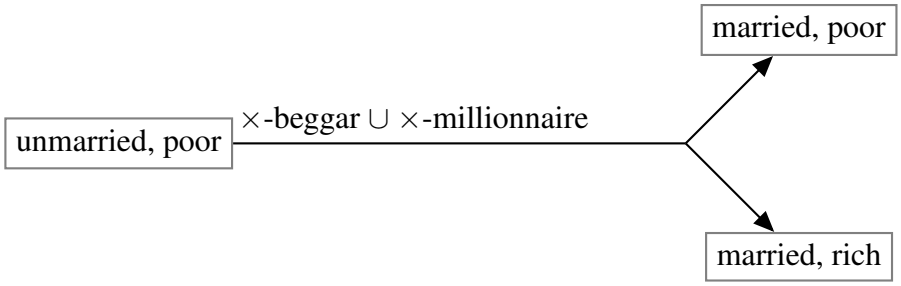
Starting out from the situation where the alarm is off, we would get:



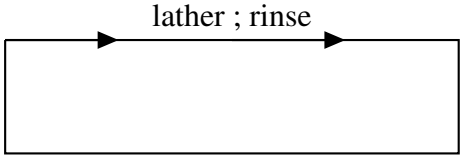
Choice A complex action can also consist of a *choice* between simpler actions: either drink tea or drink coffee. Either marry a beggar or marry a millionnaire.



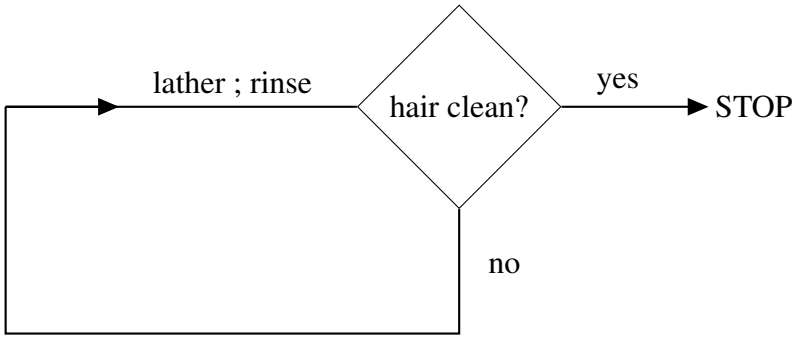




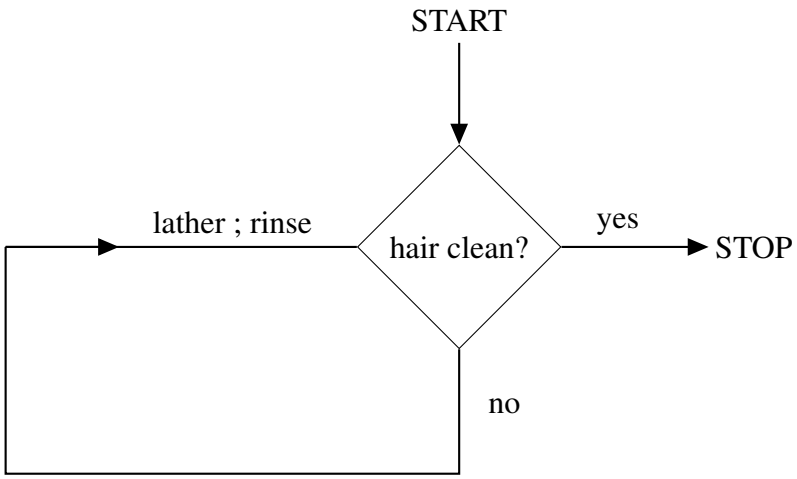
Repetition Actions can be repeated. The phrase 'lather, rinse, repeat' is used as a joke at people who take instructions too literally: the stop condition 'until hair is clean' is omitted. There is also a joke about an advertising executive who increases the sales of his client's shampoo by adding the word 'repeat' to its instructions. If taken literally, the compound action 'lather, rinse, repeat' would look like this:



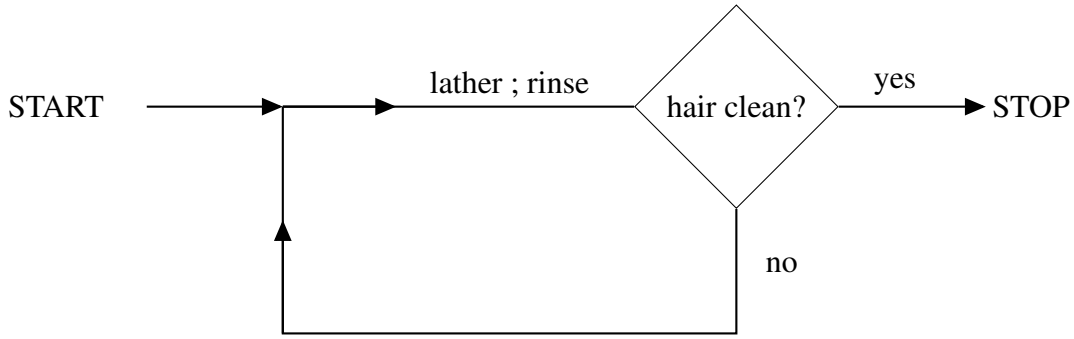
Repeated actions usually have a *stop condition*: repeat the lather rinse sequence until your hair is clean. This gives a more sensible interpretation of the repetition instruction:



Looking at the picture, we see that this procedure is ambiguous, for where do we start? Here is one possibility:



And here is another:



The difference between these two procedures is that the first one starts with a ‘hair clean?’ check: if the answer is ‘yes’, nothing happens. The second procedure starts with a ‘lather; rinse’ sequence, no matter the initial state of your hair.

In many programming languages, this same distinction is made by means of a choice between two different constructs for expressing 'condition controlled loops':

while `not hair_clean` **do** { `lather; rinse` }

repeat { `lather ; rinse` } **until** `hair_clean`

The first loop does not guarantee that the 'lather ; rinse' sequence gets performed at least once; the second loop does.

Test The 'condition' in a condition-controlled loop (the condition 'hair_clean', for example) can itself be viewed as an action: a test whether a certain fact holds. A test to see whether some condition holds can also be viewed as a basic action. Notation for the action that tests condition φ is $?\varphi$. The question mark turns a formula (something that can be true or false) into an action (something that can succeed or fail).

If we express tests as $?\varphi$, then we should specify the language from which φ is taken. Depending on the context, this could be the language of propositional logic, the language of predicate logic, the language of epistemic logic, and so on.

Since we are taking an abstract view, the basic actions can be anything. Still, there are a few cases of basic action that are special. The action that always succeeds is called *SKIP*. The action that always fails is called *ABORT*. If we have tests, then clearly *SKIP* can be expressed as $?\top$ (the test that always succeeds) and *ABORT* as $?\bot$ (the test that always fails).

Using test, sequence and choice we can express the familiar 'if then else' from many programming languages.

if `hair_clean` **then** `skip` **else** { `lather ; rinse` }

This becomes a choice between a test for clean hair (if this test succeeds then nothing happens) and a sequence consisting of a test for not-clean-hair followed by a lather and a rinse (if the hair is not clean then it is first lathered and then rinsed).

`?hair_clean` $\cup$ { `?¬hair_clean ; lather ; rinse` }

The general recipe for expressing **if** φ **then** α_1 **else** α_2 is given by:

$$?\varphi; \alpha_1 \ \cup \ ?\neg\varphi; \alpha_2.$$

Since exactly one of the two tests $?\varphi$ and $?\neg\varphi$ will succeed, exactly one of α_1 or α_2 will get executed.

Using the operation for turning a formula into a test, we can first test for p and next test for q by means of $?p; ?q$. Clearly, the order of testing does not matter, so this is equivalent to $?q; ?p$. And since the tests do not change the current state, this can also be expressed as a single test $?(p \wedge q)$.

Similarly, the choice between two tests $?p$ and $?q$ can be written as $?p \cup ?q$. Again, this is equivalent to $?q \cup ?p$, and it can be turned into a single test $?(p \vee q)$.

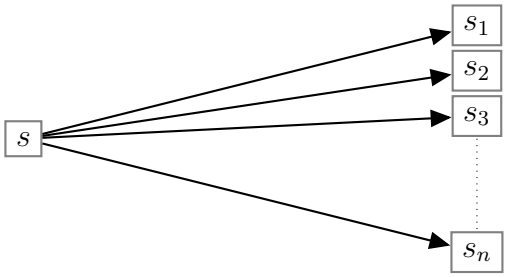
Converse Some actions can be undone by reversing them: the reverse of opening a window is closing it. Other actions are much harder to undo: if you smash a piece of china then it is sometimes hard to mend it again. So here we have a choice: do we assume that basic actions can be undone? If we do, we need an operation for this, for taking the *converse* of an action. If, in some context, we assume that undoing an action is generally impossible we should omit the *converse* operation in that context.

Exercise 6.1 Suppose $\breve{}$ is used for reversing basic actions. So $a\breve{}$ is the converse of action a, and $b\breve{}$ is the converse of action b. Let $a; b$ be the sequential composition of a and b, i.e., the action that consists of first doing a and then doing b. What is the converse of $a; b$?

6.3 Viewing Actions as Relations

As an exercise in abstraction, we will now view actions as binary relations on a set S of states. The intuition behind this is as follows. Suppose we are in some state s in S. Then performing some action a will result in a new state that is a member of some set of new states $\{s_1, \ldots, s_n\}$.

If this set is empty, this means that the action a has aborted in state s. If the set has a single element s', this means that the action a is deterministic on state s, and if the set has two or more elements, this means that action a is non-deterministic on state s. The general picture is:



Clearly, when we extend this picture to the whole set S, what emerges is a binary relation on S, with an arrow from s to s' (or equivalently, a pair (s, s') in the relation) just in case performing action a in state s may have s' as result. Thus, we can view binary relations on S as the interpretations of basic action symbols a.

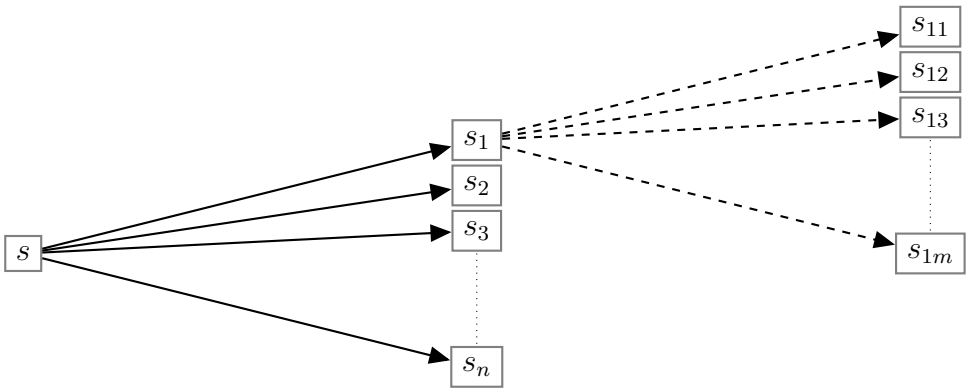
The set of all pairs taken from S is called $S \times S$, or S^2. A binary relation on S is simply a set of pairs taken from S, i.e., a subset of S^2.

Given this abstract interpretation of basic relations, it makes sense to ask what corresponds to the operations on actions that we encountered in Section 6.2. Let's consider them in turn.

Sequence Given that action symbol a is interpreted as binary relation R_a on S, and that action symbol b is interpreted as binary relation R_b on S, what should be the interpretation of the action sequence $a; b$? Intuitively, one can move from state s to state s' just in case there is some intermediate state s_0 with the property that a gets you from s to s_0 and b gets you from s_0 to s'. This is a well-known operation on binary relations, called *relational composition*. If R_a and R_b are binary relations on the same set S, then $R_a \circ R_b$ is the binary relation on S given by:

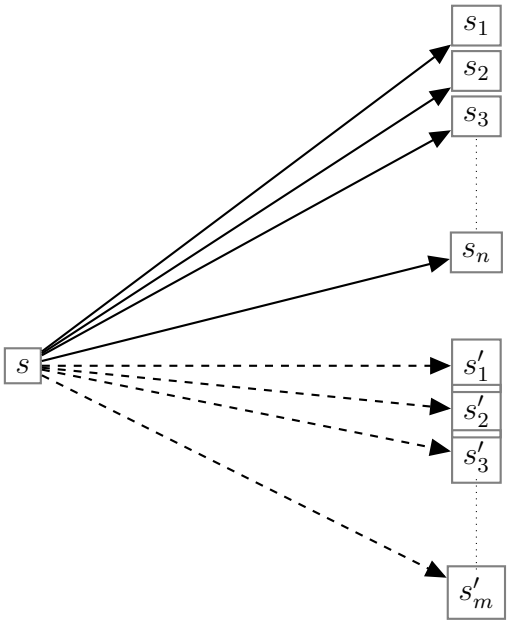
$$R_a \circ R_b = \{(s, s') \mid \text{ there is some } s_0 \in S : (s, s_0) \in R_a \text{and } (s_0, s') \in R_b\}.$$

If basic action symbol a is interpreted as relation R_a, and basic action symbol b is interpreted as relation R_b, then the sequence action $a; b$ is interpreted as $R_a \circ R_b$. Here is a picture:



If the solid arrows interpret action symbol a and the dashed arrows interpret action symbol b, then the arrows consisting of a solid part followed by a dashed part interpret the sequence $a; b$.

Choice Now suppose again that we are in state s, and that performing action a will get us in one of the states in $\{s_1, \ldots, s_n\}$. And supposse that in that same state s, performing action b will get us in one of the states in $\{s'_1, \ldots, s'_m\}$.



Then performing action $a \cup b$ (the choice between a and b) in s will get you in one of the states in $\{s_1, \ldots, s_n\} \cup \{s'_1, \ldots, s'_m\}$. More generally, if action symbol a is interpreted as the relation R_a, and action symbol b is interpreted as the relation R_b, then $a \cup b$ will be interpreted as the relation $R_a \cup R_b$ (the union of the two relations).

Test A notation that is often used for the equality relation (or: identity relation is I. The binary relation I on S is by definition the set of pairs given by:

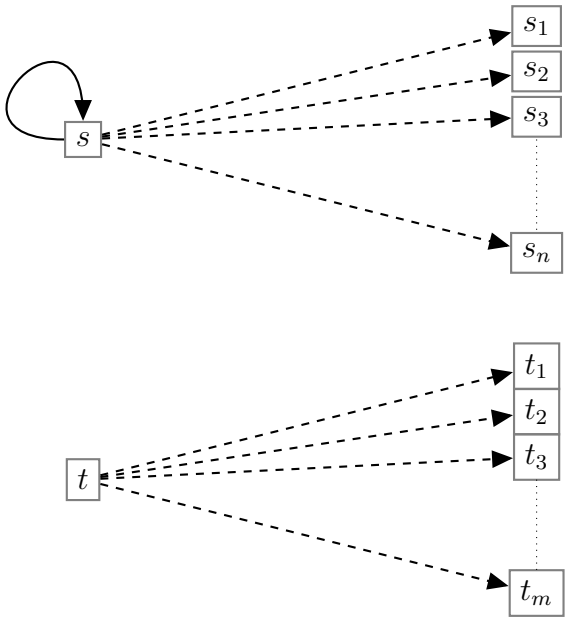
$$I = \{(s, s) \mid s \in S\}.$$

A test $?\varphi$ is interpreted as a subset of the identity relation, namely as the following set of pairs:

$$R_{?\varphi} = \{(s, s) \mid s \in S, s \models \varphi\}$$

From this we can see that a test does not change the state, but checks whether the state satisfies a condition.

To see the result of combining a test with another action:



The solid arrow interprets a test $?\varphi$ that succeeds in state s but fails in state t. If the dashed arrows interpret a basic action symbol a, then, for instance, (s, s_1) will be in the interpretation of $?\varphi; a$, but (t, t_1) will not.

Since $\top$ is true in any situation, we have that $?\top$ will get interpreted as I (the identity relation on S). Therefore, $?\top; a$ will always receive the same interpretation as a.

Since $\bot$ is false in any situation, we have that $?\bot$ will get interpreted as $\emptyset$ (the empty relation on S). Therefore, $?\bot; a$ will always receive the same interpretation as $?\bot$.

Before we handle repetition, it is useful to switch to a more gereral perspective.

6.4 Operations on Relations

Relations were introduced in Chapter 4 on predicate logic. In this chapter we view actions as binary relations on a set S of situations. Such a binary relation is a subset of $S \times S$, the set of all pairs (s, t) with s and t taken from S. It makes sense to develop the general topic of operations on binary relations. Which operations suggest themselves, and what are the corresponding operations on actions?

In the first place, there are the usual set-theoretic operations. Binary relations are sets of pairs, so taking unions, intersections and complements makes sense (also see Appendix A). We have already seen that taking unions corresponds to choice between actions.

Example 6.2 The union of the relations 'mother' and 'father' is the relation 'parent'.

Example 6.3 The intersection of the relations $\subseteq$ and $\supseteq$ is the equality relation $=$.

In Section 6.3 we encountered the notation I for the equality (or: identity) relation on a set S. We have seen that tests get interpreted as subsets of I.

We also looked at composition of relations. $R_1 \circ R_2$ is the relation that performing an R_1 step followed by an R_2 step. To see that order of composition matters, consider the following example.

Example 6.4 The relational composition of the relations 'mother' and 'parent' is the relation 'grandmother', for 'x is grandmother of y' means that there is a z such that x is mother of z, and z is parent of y.

The relational composition of the relations 'parent' and 'mother' is the relation 'maternal grandparent', for 'x is maternal grandparent of y' means that there is a z such that x is parent of z and z is mother of y.

Exercise 6.5 What is the relational composition of the relations 'father' and 'mother'?

Another important operation is *relational converse*. The relational converse of a binary relation R, notation $R^\smile$, is the relation given by:

$$R^\smile = \{(y, x) \in S^2 \mid (x, y) \in R\}.$$

Example 6.6 The relational converse of the 'parent' relation is the 'child' relation.

Exercise 6.7 What is the relational converse of the $\subseteq$ relation?

The following law describes the interplay between composition and converse:

Converse of composition $(R_1 \circ R_2)^\smile = {R_2}^\smile \circ {R_1}^\smile$.

Exercise 6.8 Check from the definitions that $(R_1 \circ R_2)^\smile = {R_2}^\smile \circ {R_1}^\smile$ is valid.

There exists a long list of logical principles that hold for binary relations. To start with, there are the usual Boolean principles that hold for all sets:

Commutativity $R_1 \cup R_2 = R_2 \cup R_1$, $R_1 \cap R_2 = R_2 \cap R_1$,

Idempotence $R \cup R = R$, $R \cap R = R$.

Laws of De Morgan $\overline{R_1 \cup R_2} = \overline{R_1} \cap \overline{R_2}$, $\overline{R_1 \cap R_2} = \overline{R_1} \cup \overline{R_2}$.

Specifically for relational composition we have:

Associativity $R_1 \circ (R_2 \circ R_3) = (R_1 \circ R_2) \circ R_3$.

Distributivity

$$\begin{aligned} R_1 \circ (R_2 \cup R3) &= (R_1 \circ R_2) \cup (R_1 \circ R_3) \\ (R_1 \cup R_2) \circ R3) &= (R_1 \circ R_3) \cup (R_2 \circ R_3). \end{aligned}$$

There are also many principles that seem plausible but that are invalid. To see that a putative principle is invalid one should look for a counterexample.

Example 6.9 $R \circ R = R$ is invalid, for if R is the 'parent' relation, then the principle would state that 'grandparent' equals 'parent', which is false.

Exercise 6.10 Show by means of a counterexample that $R_1 \cup (R_2 \circ R_3) = (R_1 \cup R_2) \circ (R_1 \cup R_3)$ is invalid.

Exercise 6.11 Check from the definitions that $R_1 \circ (R_2 \cup R_3) = (R_1 \circ R_2) \cup (R_1 \circ R_3)$ is valid.

Exercise 6.12 Check from the definition that $R^{\smile} = R$ is valid.

Exercise 6.13 Check from the definitions that $(R_1 \cup R_2)^{\smile} = {R_1}^{\smile} \cup {R_2}^{\smile}$ is valid.

Transitive Closure A relation R is transitive if it holds that if you can get from x to y in two R-steps, then it is also possible to get from x to y in a single R-step (see page 4-20 above). This can be readily expressed in terms of relational composition.

R is transitive iff $R \circ R \subseteq R$.

The *transitive closure* of a relation R is defined as the smallest transitive relation S that contains R. This means: S is the transitive closure of R if

(1) $R \subseteq S$,

(2) $S \circ S \subseteq S$,

(3) if $R \subseteq T$ and $T \circ T \subseteq T$ then $S \subseteq T$.

Requirement (1) expresses that R is contained in S, requirement (2) expresses that S is transitive, and requirement (3) expresses that S is the *smallest* transitive relation that contains R: any T that satisfies the same requirements must be at least as large as S.

The customary notation for the transitive closure of R is R^+. Here is an example.

Example 6.14 The transitive closure of the 'parent' relation is the 'ancestor' relation. If x is parent of y then x is ancestor of y, so the parent relation is contained in the ancestor relation. If x is an ancestor of y and y is an ancestor of z then surely x is an ancestor of z, so the ancestor relation is transitive. Finally, the ancestor relation is the smallest transitive relation that contains the parent relation.

You can think of a binary relation R as a recipe for taking R-steps. The recipe for taking double R-steps is now given by $R \circ R$. The recipe for taking triple R-steps is given by $R \circ R \circ R$, and so on.

There is a formal reason why the order of composition does not matter: $R_1 \circ (R_2 \circ R_3)$ denotes the same relation as $(R_1 \circ R_2) \circ R_3$. because of the above-mentioned principle of associativity.

The n-fold composition of a binary relation R on S with itself can be defined from R and I (the identity relation on S), by recursion (see Appendix, Section A.6), as follows:

$$\begin{aligned} R^0 &= I \\ R^n &= R \circ R^{n-1} \text{ for } n > 0. \end{aligned}$$

Abbreviation for the n-fold composition of R is R^n. This allows us to talk about taking a specific number of R-steps.

Notice that $R \circ I = R$. Thus, we get that $R^1 = R \circ R^0 = R \circ I = R$.

The transitive closure of a relation R can be computed by means of:

$$R^+ = R \cup R^2 \cup R^3 \cup \cdots$$

This can be expressed without the $\cdots$, as follows:

$$R^+ = \bigcup_{n \in \mathbb{N}, n>0} R^n.$$

Thus, R^+ denotes the relation of doing an arbitrary finite number of R-steps (at least one).

Closely related to the transitive closure of R is the reflexive transitive closure of R. This is, by definition, the smallest relation that contains R and that is both reflexive and transitive. The reflexive transitive closure of R can be computed by:

$$R^* = I \cup R \cup R^2 \cup R^3 \cup \cdots$$

This can be expressed without the $\cdots$, as follows:

$$R^* = \bigcup_{n \in \mathbb{N}} R^n.$$

Thus, R^* denotes the relation of doing an arbitrary finite number of R-steps, including zero steps.

Notice that the following holds:

$$R^+ = R \circ R^*.$$

Exercise 6.15 The following identity between relations is not valid:

$$(R \cup S)^* = R^* \circ S^*.$$

Explain why not by giving a counter-example.

Exercise 6.16 The following identity between relations is not valid:

$$(R \circ S)^* = R^* \circ S^*.$$

Explain why not by giving a counter-example.

For Loops In programming, repetition consisting of a specified number of steps is called a *for loop*. Here is an example of a loop for printing ten lines, in the programming language *Ruby*:

```
#!/usr/bin/ruby

for i in 0..10
   puts "Value of local variable is #{i}"
end
```

If you have a system with *Ruby* installed, you can save this as a file and execute it.

While Loops, Repeat Loops If R is the interpretation of a ('doing a once'), then R^* is the interpretation of 'doing a an arbitrary finite number of times', and R^+ is the interpretation of 'doing a an arbitrary finite number of times but at least once'. These relations can be used to define the interpretation of *while loops* and *repeat loops* (the so-called *condition controlled loops*), as follows.

If a is interpreted as R_a, then the condition-controlled loop 'while φ do a' is interpreted as:

$$(R_{?\varphi} \circ R_a)^* \circ R_{?\neg\varphi}.$$

First do a number of steps consisting of a $?\varphi$ test followed by an a action, next check that $\neg\varphi$ holds.

Exercise 6.17 Supposing that a gets interpreted as the relation R_a, $?\varphi$ as $R_{?\varphi}$ and $?\neg\varphi$ as $R_{?\neg\varphi}$, give a relational interpretation for the condition controlled loop 'repeat a until φ'.

6.5 Combining Propositional Logic and Actions: PDL

The language of propositional logic over some set of basic propositions P is given by:

$$\varphi \ ::= \ \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \text{ where } p \text{ ranges over } P.$$

If we assume that a set of basic action symbols A is given, then the language of actions that we discussed in Sections 6.2 and 6.3 above can be formally defined as:

$$\alpha \ ::= \ a \mid ?\varphi \mid \alpha; \alpha \mid \alpha \cup \alpha \mid \alpha^* \text{ where } a \text{ ranges over } A.$$

Note that the test $?\varphi$ in this definition refers to the definition of φ in the language of propositional logic. Thus, the language of propositional logic is embedded in the language of actions.

Now here is a new idea, for also doing the converse: extend the language of propositional logic with a construction that describes the results of executing an action α.

If α is interpreted as a binary relation then in a given state s there may be several states s' for which (s, s') is in the interpretation of α.

Interpret $\langle\alpha\rangle\varphi$ as follows:

> $\langle\alpha\rangle\varphi$ is true in a state s if for *some* s' with (s, s') in the interpretation of α it holds that φ is true in s'.

For instance, if a is the action of asking for promotion, and p is the proposition expressing that one is promoted, then $\langle a\rangle p$ expresses that asking for promotion may result in actually getting promoted.

Another useful expression is $[\alpha]\varphi$, with the following interpretation:

> $[\alpha]\varphi$ is true in a state s if for *every* s' with (s, s') in the interpretation of α it holds that φ is true in s'.

For instance, if a again expresses asking for promotion, and p expresses that one is promoted, then $[a]p$ expresses that, in the current state, the action of asking for a promotion *always* results in getting promoted.

Note that $\langle a\rangle p$ and $[a]p$ are not equivalent: think of a situation where asking for a promotion may also result in getting fired. In that case $\langle a\rangle p$ may still hold, but $[a]p$ does not hold.

If one combines propositional logic with actions in this way one gets a basic logic of change called Propositional Dynamic Logic or PDL. Here is the formal definition of the language of PDL:

Definition 6.18 (Language of PDL — propositional dynamic logic) Let p range over the set of basic propositions P, and let a range over a set of basic actions A. Then the formulas φ and action statements α of propositional dynamic logic are given by:

$$\begin{array}{lcl} \varphi & ::= & \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \langle\alpha\rangle\varphi \mid [\alpha]\varphi \\ \alpha & ::= & a \mid ?\varphi \mid \alpha_1; \alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha^* \end{array}$$

The definition does not have $\rightarrow$ or $\leftrightarrow$. But this does not matter, for we can introduce these operators by means of abbreviations or shorthands.

$\top$ is the formula that is always true. From this, we can define $\bot$, as shorthand for $\neg\top$.

Similarly, $\varphi_1 \rightarrow \varphi_2$ is shorthand for $\neg\varphi_1 \vee \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$ is shorthand for $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$.

Propositional dynamic logic abstracts over the set of basic actions, in the sense that basic actions can be anything. In the language of PDL they are atoms. This means that the range of applicability of PDL is vast. The only thing that matters about a basic action a is that it is interpreted by some binary relation on a state set.

Propositional dynamic logic has two basic syntactic categories: *formulas* and *action statements*. Formulas are used for talking about states, action statements are used for classifying transitions between states. The same distinction between formulas and action statements can be found in all imperative programming languages. The statements of C or Java or Ruby are the action statements. Basic actions in C are assigning a value to a variable. These are instructions to change the memory state of the machine. The so-called Boolean expressions in C behave like formulas of propositional logic. They appear as conditions or tests in conditional expressions. Consider the following C statement:

```
if (y < z)
  x = y;
else
  x = z;
```

This is a description of an action. But the ingredient `(y<z)` is not a statement (description of an action) but a Boolean expression (description of a state) that expresses a test.

Propositional dynamic logic is an extension of propositional logic with action statements, just like epistemic logic is an extension of propositional logic with epistemic modalities. Let a set of basic propositions P be given. Then appropriate states will contain valuations for these propositions. Let a set of basic actions A be given. Then every basic action corresponds to a binary relation on the state set. Together this gives a labeled transition system with valuations on states as subsets from P and labels on arcs between states taken from A.

Exercise 6.19 Suppose we also want to introduce a shorthand α^n, for a sequence of n copies of action statement α. Show how this can be defined by induction. (Hint: use $\alpha^0 := ?\top$ as the base case.)

Let's get a feel for the kind of things we can express with PDL. For any action statement α,

$$\langle\alpha\rangle\top$$

expresses that the action α has at least one successful execution. Similarly,

$$[\alpha]\bot$$

expresses that the action fails (cannot be executed in the current state).

The basic actions can be anything, so let us focus on a basic action a that is interpreted as the relation R_a. Suppose we want to say that some execution of a leads to a p state and another execution of a leads to a non-p state. Then here is a PDL formula for that:

$$\langle a \rangle p \wedge \langle a \rangle \neg p.$$

If this formula is true in a state s, then this means that R_a forks in that state: there are at least two R_a arrows starting from s, one of them to a state s_1 satisfying p and one of them to a state s_1 that does not satisfy p. For the interpretation of P we need properties of states, for p is like a one-place predicate in predicate logic.

If the basic actions are changes in the world, such as spilling milk S or cleaning C, then $[C;S]d$ expresses that cleaning up followed by spilling milk always results in a dirty state, while $[S;C]\neg d$ expresses that the occurrence of these events in the reverse order always results in a clean state.

6.6 Transition Systems

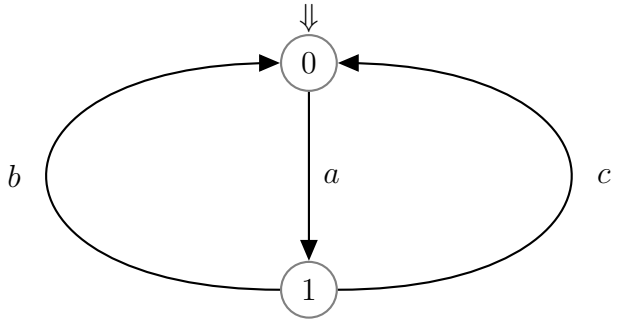
In Section 6.7 we will define the semantics of PDL relative to labelled transition systems, or process graphs.

Definition 6.20 (Labelled transition system) Let P be a set of basic propositions and A a set of labels for basic actions. Then a labelled transition system (or LTS) over atoms P and agents A is a triple $M = \langle S, R, V \rangle$ where S is a set of states, $V : S \to \mathcal{P}(P)$ is a valuation function, and $R = \{\stackrel{a}{\to} \subseteq S \times S \mid a \in A\}$ is a set of labelled transitions, i.e., a set of binary relations on S, one for each label a.

Another way to look at a labelled transition system is as a first order model predicate for a language with unary and binary predicates.

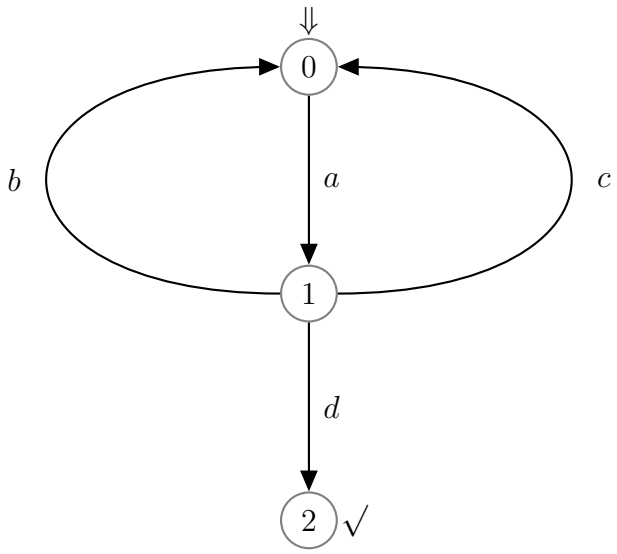
LTSs with a designated node (called the *root node*) are called *pointed LTSs* or *process graphs*.

The process of repeatedly doing a, followed by a choice between b and c can be viewed as a process graph, as follows:



The root note 0 is indicated by $\Downarrow$. There are two states 0 and 1. The process start in state 0 with the execution of action a. This gets us to state 1, where there are two possible actions b and c, both of which get us back to state 0, and there the process repeats. This is an infinite process, just like an operating system of a computer. Unless there is a system crash, the process goes on forever.

Jumping out of a process can be done by creating an action that moves to an end state.



We can think about $\surd$ as a proposition letter, and then use PDL to talk about these process graphs. In state 1 of the first model $\langle d\rangle\surd$ is false, in state 1 of the second model $\langle d\rangle\surd$ is true. This formula expresses that a d transition to a $\surd$ state is possible.

In both models it is the case in state 0 that after *any number* of sequences consisting of an a step followed by either a b or a c step, a further a step is possible. This is expressed by the following PDL formula: $[(a;(b\cup c))^*]\langle a\rangle\top$.

Exercise 6.21 Which of the following formulas are true in state 0 of the two models given above:

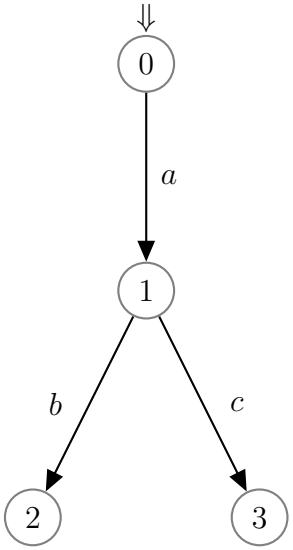
(1) $\langle a;d\rangle\surd$.

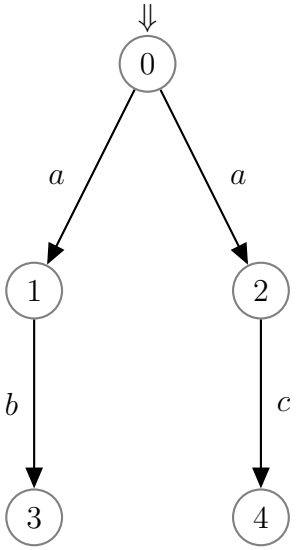
(2) $[a;d]\surd$.

(3) $[a](\langle b\rangle\top\wedge\langle c\rangle\top)$.

(4) $[a]\langle d\rangle\surd$.

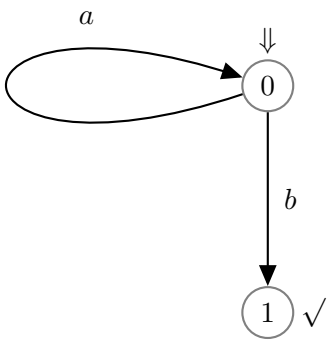
The following two pictures illustrate an important distinction:

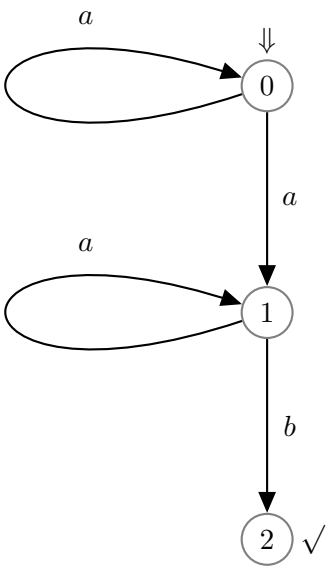




In the picture on the left, it is possible to take an a action from the root, and next to make a choice between doing b or doing c. In the picture on the right, there are two 'ways' of doing a, one of them ends in a state where b is the only possible move and the other one ending in a state where c is the only possible move. This difference can be expressed in a PDL formula, as follows. In the root state of the picture on the left, $[a](\langle b\rangle\top \wedge \langle c\rangle\top)$ is true, in the root state of the picture on the right this formula is false.

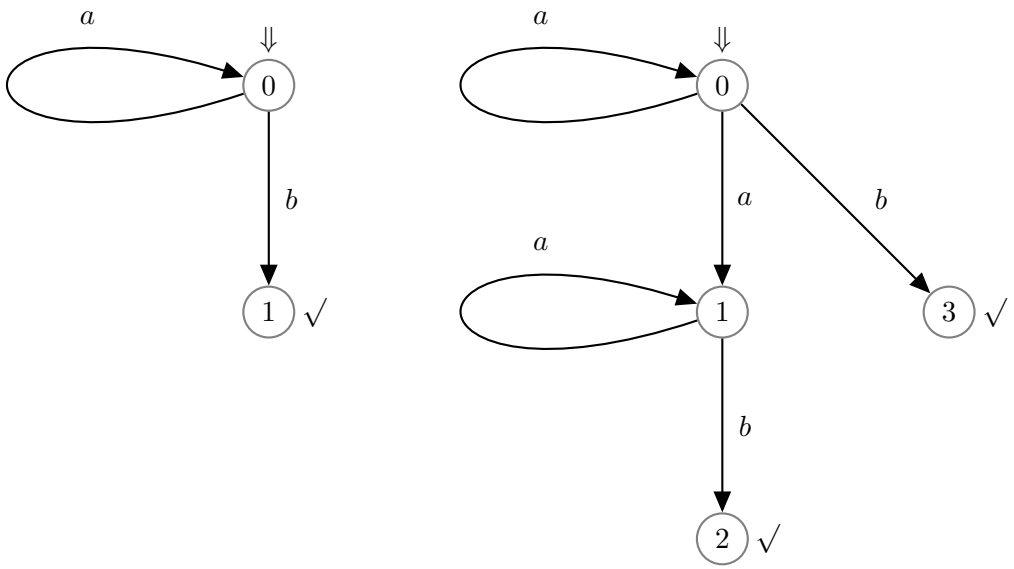
Exercise 6.22 Find a PDL formula that can distinguish between the root states of the following two process graphs:





The formula should be true in one graph, false in the other.

Exercise 6.23 Now consider the following two pictures of process graphs:



Is it still possible to find a PDL formula that is true in the root of one of the graphs and false in the root of the other? If your answer is 'yes', then give such a formula. If your answer is 'no', then try to explain as clearly as you can why you think this is impossible.

6.7 Semantics of PDL

The formulas of PDL are interpreted in states of a labeled transition system (or: LTS, or: process graph), and the actions a of PDL as binary relations on the domain S of the LTS. We can think of an LTS as given by its set of states S, its valuation V, and its set of labelled transitions R. We will give the interpretation of basic actions a as $\xrightarrow{a}$.

If an LTS M is given, we use S_M to refer to its set of states, we use R_M to indicate its set of labelled transitions, and we use V_M for its valuation.

Definition 6.24 (Semantics of PDL) Given is a labelled transition system $M = \langle S, V, R\rangle$ for P and A.

$$
\begin{array}{lcl}
M, s \models \top & & \text{always} \\
M, s \models p & \iff & p \in V(s) \\
M, s \models \neg\varphi & \iff & M, s \not\models \varphi \\
M, s \models \varphi \vee \psi & \iff & M, s \models \varphi \text{ or } M, s \models \psi \\
M, s \models \varphi \wedge \psi & \iff & M, s \models \varphi \text{ and } M, s \models \psi \\
M, s \models \langle\alpha\rangle\varphi & \iff & \text{for some } t, (s,t) \in [\![\alpha]\!]^M \text{ and } M, t \models \varphi \\
M, s \models [\alpha]\varphi & \iff & \text{for all } t \text{ with } (s,t) \in [\![\alpha]\!]^M \text{ it holds that } M, t \models \varphi.
\end{array}
$$

where the binary relation $[\![\alpha]\!]^M$ interpreting the action α in the model M is defined as

$$\begin{aligned}
[\![a]\!]^M &= \xrightarrow{a}_M \\
[\![?\varphi]\!]^M &= \{(s,s) \in S_M \times S_M \mid M, s \models \varphi\} \\
[\![\alpha_1;\alpha_2]\!]^M &= [\![\alpha_1]\!]^M \circ [\![\alpha_2]\!]^M \\
[\![\alpha_1 \cup \alpha_2]\!]^M &= [\![\alpha_1]\!]^M \cup [\![\alpha_2]\!]^M \\
[\![\alpha^*]\!]^M &= ([\![\alpha]\!]^M)^*
\end{aligned}$$

Note that the clause for $[\![\alpha^*]\!]^M$ uses the definition of reflexive transitive closure that was given on page 6-16.

These clauses specify how formulas of PDL can be used to make assertions about PDL models.

Example 6.25 The formula $\langle a \rangle \top$, when interpreted at some state in a PDL model, expresses that that state has a successor in the $\xrightarrow{a}$ relation in that model.

A PDL formula φ is *true* in a model if it holds at every state in that model, i.e., if $[\![\varphi]\!]^M = S_M$.

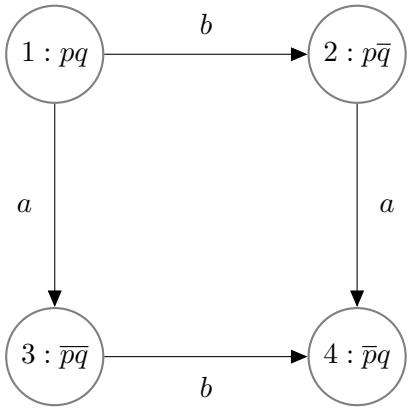
Example 6.26 Truth of the formula $\langle a \rangle \top$ in a model expresses that $\xrightarrow{a}$ is serial in that model. (A binary relation R is serial on a domain S if it holds for all $s \in S$ that there is some $t \in S$ with sRt.)

A PDL formula φ is *valid* if it holds for all PDL models M that φ is true in that model, i.e., that $[\![\varphi]\!]^M = S_M$.

Exercise 6.27 Show that $\langle a;b \rangle \top \leftrightarrow \langle a \rangle \langle b \rangle \top$ is an example of a valid formula.

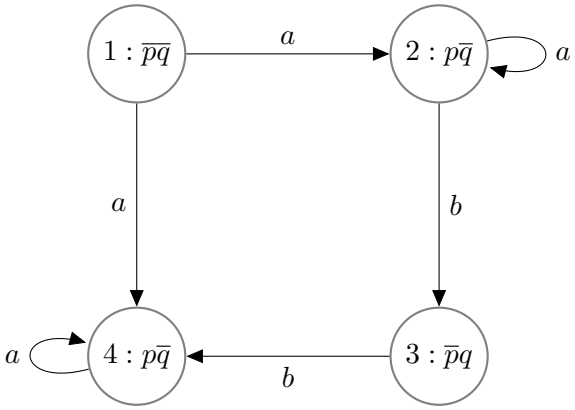
As was note before, ? is an operation for mapping formulas to action statements. Action statements of the form $?\varphi$ are called *tests*; they are interpreted as the identity relation, restricted to the states satisfying the formula.

Exercise 6.28 Let the following PDL model be given:



Give the interpretations of $?p$, of $?(p \vee q)$, of $a; b$ and of $b; a$.

Exercise 6.29 Let the following PDL model be given:



(1) List the states where the following formulas are true:

a. $\neg p$

b. $\langle b \rangle q$

c. $[a](p \rightarrow \langle b \rangle q)$

(2) Give a formula that is true only at state 4.

(3) Give all the elements of the relations defined by the following action expressions:

a. $b; b$

b. $a \cup b$

c. a^*

(4) Give a PDL action expression that defines the relation $\{(1, 3)\}$ in the graph. (Hint: use one or more test actions.)

Converse Let $\breve{}$ (converse) be an operator on PDL programs with the following interpretation:

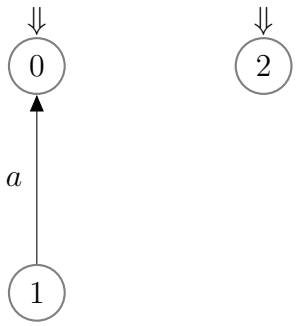
$$[\![\alpha\breve{}]\!]^M \quad = \quad \{(s, t) \mid (t, s) \in [\![\alpha]\!]^M\}.$$

Exercise 6.30 Show that the following equalities hold:

$$\begin{aligned} (\alpha; \beta)\breve{} &= \beta\breve{}; \alpha\breve{} \\ (\alpha \cup \beta)\breve{} &= \alpha\breve{} \cup \beta\breve{} \\ (\alpha^*)\breve{} &= (\alpha\breve{})^* \end{aligned}$$

Exercise 6.31 Show how the equalities from the previous exercise, plus atomic converse $a\breve{}$, can be used to define $\alpha\breve{}$, for arbitrary α, by way of abbreviation.

It follows from Exercises 6.30 and 6.31 that it is enough to add converse to the PDL language for atomic actions only. To see that adding converse in this way increases expressive power, observe that in root state 0 in the following picture $\langle a^{\smile}\rangle\top$ is true, while in root state 2 in the picture $\langle a^{\smile}\rangle\top$ is false. On the assumption that 0 and 2 have the same valuation, no PDL formula without converse can distinguish the two states.



6.8 Axiomatisation

The logic of PDL is axiomatised as follows. Axioms are all propositional tautologies, plus an axiom stating that α behaves as a standard modal operator, plus axioms describing the effects of the program operators (we give box ($[\alpha]$)versions here, but every axiom has an equivalent diamond ($\langle\alpha\rangle$) version), plus a propositional inference rule and a modal inference rule.

The propositional inference rule is the familiar rule of Modus Ponens.

(modus ponens) From $\vdash \varphi_1$ and $\vdash \varphi_1 \to \varphi_2$, infer $\vdash \varphi_2$.

The modal inference rule is the rule of modal generalization (or: necessitation):

(modal generalisation) From $\vdash \varphi$, infer $\vdash [\alpha]\varphi$.

Modal generalization expresses that theorems of the system have to hold in every state.

Example 6.32 Take the formula $(\varphi \wedge \psi) \to \varphi$. Because this is a propositional tautology, it is a theorem of the system, so we have $\vdash (\varphi \wedge \psi) \to \varphi$. And because it is a theorem, it has to hold everywhere, so we have, for any α:

$$\vdash [\alpha]((\varphi \wedge \psi) \to \varphi).$$

Now let us turn to the axioms. The first axiom is the K axiom (familiar from Chapter 5) that expresses that program modalities distribute over implications:

$$\text{(K)} \ \vdash \quad [\alpha](\varphi \to \psi) \to ([\alpha]\varphi \to [\alpha]\psi)$$

Example 6.33 As an example of how to play with this, we derive the equivalent $\langle\alpha\rangle$ version. By the K axiom, the following is a theorem (just replace ψ by $\neg\psi$ everywhere in the axiom):

$$\vdash [\alpha](\varphi \to \neg\psi) \to ([\alpha]\varphi \to [\alpha]\neg\psi).$$

From this, by the propositional reasoning principle of contraposition:

$$\vdash \neg([\alpha]\varphi \to [\alpha]\neg\psi) \to \neg[\alpha](\varphi \to \neg\psi).$$

From this, by propositional reasoning:

$$\vdash [\alpha]\varphi \wedge \neg[\alpha]\neg\psi) \to \neg[\alpha](\varphi \to \neg\psi).$$

Now replace all boxes by diamonds, using the abbreviation $\neg\langle\alpha\rangle\neg\varphi$ for $[\alpha]\varphi$:

$$\vdash \neg\langle\alpha\rangle\neg\varphi \wedge \neg\neg\langle\alpha\rangle\neg\neg\psi) \to \neg\neg\langle\alpha\rangle\neg(\varphi \to \neg\psi).$$

This can be simplified by propositional logic, and we get:

$$\vdash (\neg\langle\alpha\rangle\neg\varphi \wedge \langle\alpha\rangle\psi) \to \langle\alpha\rangle(\varphi \wedge \psi).$$

Example 6.34 This example is similar to Example 5.45 from Chapter 5.

Above, we have seen that $[\alpha]((\varphi \wedge \psi) \to \varphi)$ is a theorem. With the K axiom, we can derive from this:

$$\vdash [\alpha](\varphi \wedge \psi) \to [\alpha]\varphi.$$

In a similar way, we can derive:

$$\vdash [\alpha](\varphi \wedge \psi) \to [\alpha]\psi.$$

From these by propositional reasoning:

$$\vdash [\alpha](\varphi \wedge \psi) \to ([\alpha]\varphi \wedge [\alpha]\psi). \tag{*}$$

The implication in the other direction is also derivable, as follows:

$$\vdash \varphi \to (\psi \to (\varphi \wedge \psi)),$$

because $\varphi \to (\psi \to (\varphi \wedge \psi))$ is a propositional tautology. By modal generalization (necessitation) from this:

$$\vdash [\alpha](\varphi \to (\psi \to (\varphi \wedge \psi))).$$

By two applications of the K axiom and propositional reasoning from this:

$$\vdash [\alpha]\varphi \to ([\alpha]\psi \to [\alpha](\varphi \wedge \psi)).$$

Since $\varphi \to (\psi \to \chi)$ is propositionally equivalent to $(\varphi \wedge \psi) \to \chi$, we get from this by propositional reasoning:

$$\vdash ([\alpha]\varphi \wedge [\alpha]\psi) \to [\alpha](\varphi \wedge \psi). \tag{**}$$

Putting the two principles $(*)$ and $(**)$ together we get:

$$\vdash [\alpha](\varphi \wedge \psi) \leftrightarrow ([\alpha]\varphi \wedge [\alpha]\psi). \tag{***}$$

Let us turn to the next axiom, the axiom for test. This axiom says that $[?\varphi_1]\varphi_2$ expresses an implication:

$$\text{(test)} \ \vdash \quad [?\varphi_1]\varphi_2 \leftrightarrow (\varphi_1 \rightarrow \varphi_2)$$

The axioms for sequence and for choice:

$$\begin{aligned} \text{(sequence)} \ &\vdash \quad [\alpha_1;\alpha_2]\varphi \leftrightarrow [\alpha_1][\alpha_2]\varphi \\ \text{(choice)} \ &\vdash \quad [\alpha_1 \cup \alpha_2]\varphi \leftrightarrow [\alpha_1]\varphi \wedge [\alpha_2]\varphi \end{aligned}$$

Example 6.35 As an example application, we derive

$$\vdash [\alpha;(\beta \cup \gamma)]\varphi \leftrightarrow [\alpha][\beta]\varphi \wedge [\alpha][\gamma]\varphi.$$

Here is the derivation:

$$\begin{array}{lll} [\alpha;(\beta \cup \gamma)]\varphi & \leftrightarrow \text{(sequence)} & [\alpha][\beta \cup \gamma]\varphi \\ & \leftrightarrow \text{(choice)} & [\alpha]([\beta]\varphi \wedge [\gamma]\varphi) \\ & \leftrightarrow (***) & [\alpha][\beta]\varphi \wedge [\alpha][\gamma]\varphi. \end{array}$$

These axioms together reduce PDL formulas without $*$ to formulas of multi-modal logic (propositional logic extended with simple modalities $[a]$ and $\langle a\rangle$).

Example 6.36 We show how this reduction works for the formula $[(a;b) \cup (?\varphi;c)]\psi$:

$$\begin{array}{lll} [(a;b) \cup (?\varphi;c)]\psi & \leftrightarrow \text{(choice)} & [a;b]\psi \wedge [?\varphi;c]\psi \\ & \leftrightarrow \text{(sequence)} & [a][b]\psi \wedge [?\varphi][c]\psi \\ & \leftrightarrow \text{(test)} & [a][b]\psi \wedge (\varphi \rightarrow [c]\psi). \end{array}$$

For the * operation there are two axioms:

$$\begin{aligned} \text{(mix)} \ &\vdash \quad [\alpha^*]\varphi \leftrightarrow \varphi \wedge [\alpha][\alpha^*]\varphi \\ \text{(induction)} \ &\vdash \quad (\varphi \wedge [\alpha^*](\varphi \rightarrow [\alpha]\varphi)) \rightarrow [\alpha^*]\varphi \end{aligned}$$

The mix axiom expresses the fact that α^* is a reflexive and transitive relation containing α, and the axiom of induction captures the fact that α^* is the *least* reflexive and transitive relation containing α.

As was mentioned before, all axioms have dual forms in terms of $\langle\alpha\rangle$, derivable by propositional reasoning. For example, the dual form of the test axiom reads

$$\vdash \langle?\varphi_1\rangle\varphi_2 \leftrightarrow (\varphi_1 \wedge \varphi_2).$$

The dual form of the induction axiom reads

$$\vdash \langle\alpha^*\rangle\varphi \rightarrow \varphi \vee \langle\alpha^*\rangle(\neg\varphi \vee \langle\alpha\rangle\varphi).$$

Exercise 6.37 Give the dual form of the mix axiom.

We will now show that in the presence of the other axioms, the induction axiom is equivalent to the so-called loop invariance rule:

$$\frac{\varphi \to [\alpha]\varphi}{\varphi \to [\alpha^*]\varphi}$$

Here is the theorem:

Theorem 6.38 In PDL without the induction axiom, the induction axiom and the loop invariance rule are interderivable.

Proof. For deriving the loop invariance rule from the induction axiom, assume the induction axiom. Suppose

$$\vdash \varphi \to [\alpha]\varphi.$$

Then by modal generalisation:

$$\vdash [\alpha^*](\varphi \to [\alpha]\varphi).$$

By propositional reasoning we get from this:

$$\vdash \varphi \to (\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)).$$

From this by the induction axiom and propositional reasoning:

$$\vdash \varphi \to [\alpha^*]\varphi.$$

Now assume the loop invariance rule. We have to establish the induction axiom. By the mix axiom and propositional reasoning:

$$\vdash (\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)) \to [\alpha]\varphi.$$

Again from the mix axiom and propositional reasoning:

$$\vdash (\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)) \to [\alpha][\alpha^*](\varphi \to [\alpha]\varphi).$$

From the two above, with propositional reasoning using (***):

$$\vdash (\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)) \to [\alpha](\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)).$$

Applying the loop invariance rule to this yields:

$$\vdash (\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)) \to [\alpha^*](\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)).$$

From this we get the induction axiom by propositional reasoning:

$$\vdash (\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)) \to [\alpha^*]\varphi.$$

This ends the proof. □

Axioms for Converse Suitable axioms to enforce that $a^{\smile}$ behaves as the converse of a are the following:

$$\begin{aligned} \vdash \varphi &\rightarrow [a]\langle a^{\smile}\rangle\varphi \\ \vdash \varphi &\rightarrow [a^{\smile}]\langle a\rangle\varphi \end{aligned}$$

Exercise 6.39 Show that the axioms for converse are sound, by showing that they hold in any state in any LTS.

6.9 Expressive power: defining programming constructs

The language of PDL is powerful enough to express conditional statements, fixed loop statements, and condition-controlled loop statements as PDL programs. More precisely, the conditional statement

$$\text{if } \varphi \text{ then } \alpha_1 \text{ else } \alpha_2$$

can be viewed as an abbreviation of the following PDL program:

$$(?\varphi;\alpha_1) \cup (?\neg\varphi;\alpha_2).$$

The fixed loop statement

$$\text{do } n \text{ times } \alpha$$

can be viewed as an abbreviation of

$$\underbrace{\alpha;\cdots;\alpha}_{n \text{ times}}$$

The condition-controlled loop statement

$$\text{while } \varphi \text{ do } \alpha$$

can be viewed as an abbreviation of

$$(?\varphi;\alpha)^*;?\neg\varphi.$$

This loop construction expressed in terms of reflexive transitive closure works for *finite* repetitions only, for note that the interpretation of " while $\top$ do α" in any model is the empty relation. Successful execution of every program we are considering here involves *termination* of the program.

The condition controlled loop statement

$$\text{repeat } \alpha \text{ until } \varphi$$

can be viewed as an abbreviation of

$$\alpha;(?\neg\varphi;\alpha)^*;?\varphi.$$

Note how these definitions make the difference clear between the *while* and *repeat* statements. A *repeat* statement always executes an action at least once, and next keeps on performing the action until the stop condition holds. A *while* statement checks a continue condition and keeps on performing an action until that condition does not hold anymore. If *while* φ *do* α gets executed, it may be that the α action does not even get executed once. This will happen if φ is false in the start state.

In imperative programming, we also have the *skip* program (the program that does nothing) and the *abort* program (the program that always fails): *skip* can be defined as $?\top$ (this is a test that always succeeds) and *abort* as $\bot$ (this is a test that always fails).

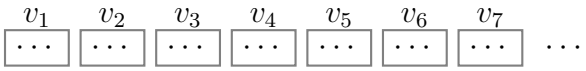
Taking stock, we see that with the PDL action operations we can define the whole repertoire of imperative programming constructs: inside of PDL there is a full fledged imperative programming language.

Moreover, given a PDL program α, the program modalities $\langle\alpha\rangle\varphi$ and $[\alpha]\varphi$ can be used to describe so-called *postconditions of execution* for program α. The first of these expresses that α has a successful exection that ends in an φ state; the second one expresses that every successful execution of α ends in a φ state. We will say more about the use of this in Section 6.10 below.

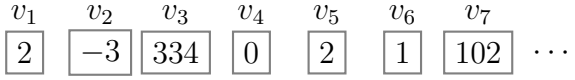
6.10 Outlook — Programs and Computation

If one wishes to interpret PDL as a logic of computation, then a natural choice for interpreting the basic actions statements is as *register assignment statements*. If we do this, then we effectively turn the action statement part of PDL into a very expressive programming language.

Let v range over a set of registers or memory locations V. A V-memory is a set of storage locations for integer numbers, each labelled by a member of V. Let $V = \{v_1, \ldots, v_n\}$. Then a V-memory can be pictured like this:



A V-state s is a function $V \to \mathbb{Z}$. We can think of a V-state as a V-memory together with its contents. In a picture:



If s is a V-state, $s(v)$ gives the contents of register v in that state. So if s is the state above, then $s(v_2) = -3$.

Let i range over integer names, such as 0, -234 or 53635 and let v range over V. Then the following defines arithmetical expressions:

$$a \quad ::= \quad i \mid v \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2.$$

It is clear that we can find out the value $[\![a]\!]_s$ of each arithmetical expression in a given V-state s.

Exercise 6.40 Provide the formal details, by giving a recursive definition of $[\![a]\!]_s$.

Next, assume that basic propositions have the form $a_1 \leq a_2$, and that basic action statements have the form $v := a$. This gives us a programming language for computing with integers as action statement language and a formula language that allows us to express properties of programs.

Determinism To say that program α is deterministic is to say that if α executes successfully, then the end state is uniquely determined by the initial state. In terms of PDL formulas, the following has to hold for every φ:

$$\langle\alpha\rangle\varphi \rightarrow [\alpha]\varphi.$$

Clearly, the basic programming actions $v := a$ are deterministic.

Termination To say that program α terminates (or: halts) in a given initial state is to say that there is a successful execution of α from the current state. To say that α always terminates is to say that α has a successful execution from *any* initial state. Here is a PDL version:

$$\langle\alpha\rangle\top.$$

Clearly, the basic programming actions $v := a$ always terminate.

Non-termination of programs comes in with loop constructs. Here is an example of a program that never terminates:

$$\textit{while } \top \textit{ do } v := v + 1.$$

One step through the loop increments the value of register v by 1. Since the loop condition will remain true, this will go on forever.

In fact, many more properties beside determinism and termination can be expressed, and in a very systematic way. We will give some examples of the style of reasoning involved.

Hoare Correctness Reasoning Consider the following problem concerning the outcome of a pebble drawing action.

> A vase contains 35 white pebbles and 35 black pebbles. Proceed as follows to draw pebbles from the vase, as long as this is possible. Every round, draw two pebbles from the vase. If they have the same colour, then put a black pebble into the vase (you may assume that there are enough additional black pebbles outside of the vase). If they have different colours, then put the white pebble back. In every round one pebble is removed from the vase, so after 69 rounds there is a single pebble left. What is the colour of this pebble?

It may seem that the problem does not provide enough information for a definite answer, but in fact it does. The key to the solution is to discover an appropriate *loop invariant*: a property that is initially true, and that does not change during the procedure.

Exercise 6.41 Consider the property: 'the number of white pebbles is odd'. Obviously, this is initially true. Show that the property is a loop invariant of the pebble drawing procedure. What follows about the colour of the last pebble?

It is possible to formalize this kind of reasoning about programs. This formalization is called Hoare logic. One of the seminal papers in computer science is Hoare's [Hoa69]. where the following notation is introduced for specifying what a computer program written in an imperative language (like C or Java) does:

$$\{P\} \quad C \quad \{Q\}.$$

Here C is a program from a formally defined programming language for imperative programming, and P and Q are conditions on the programming variables used in C.

Statement $\{P\}\ C\ \{Q\}$ is true if whenever C is executed in a state satisfying P and if the execution of C terminates, then the state in which execution of C terminates satisfies Q. The 'Hoare-triple' $\{P\}\ C\ \{Q\}$ is called a *partial correctness specification*; P is called its *precondition* and Q its *postcondition*. Hoare logic, as the logic of reasoning with such correctness specifications is called, is the precursor of all the dynamic logics known today.

Hoare correctness assertions are expressible in PDL, as follows. If φ, ψ are PDL formulas and α is a PDL program, then

$$\{\varphi\}\ \alpha\ \{\psi\}$$

translates into

$$\varphi \to [\alpha]\psi.$$

Clearly, $\{\varphi\}\ \alpha\ \{\psi\}$ holds in a state in a model iff $\varphi \to [\alpha]\psi$ is true in that state in that model.

The Hoare inference rules can now be derived in PDL. As an example we derive the rule for guarded iteration:

$$\frac{\{\varphi \wedge \psi\}\ \alpha\ \{\psi\}}{\{\psi\}\ \textit{while}\ \varphi\ \textit{do}\ \alpha\ \{\neg\varphi \wedge \psi\}}$$

First an explanation of the rule. The correctness of *while* statements is established by finding a *loop invariant*. Consider the following C function:

```
int square (int n)
{
  int x = 0;
  int k = 0;
  while (k < n) {
    x = x + 2*k + 1;
    k = k + 1;
  }
  return x;
}
```

How can we see that this program correctly computes squares? By establishing a loop invariant:

$$\{x = k^2\}\ \texttt{x = x + 2*k + 1; k = k + 1;}\ \{x = k^2\}.$$

What this says is: if the state before execution of the program is such that $x = k^2$ holds, then in the new state, after execution of the program, with the new values of the registers x and k, the relation $x = k^2$ still holds. From this we get, with the Hoare rule for *while*:

$$\{x = k^2\}$$
$$\texttt{while (k < n) \{ x = x + 2*k + 1; k = k + 1; \}}$$
$$\{x = k^2 \wedge k = n\}$$

Combining this with the initialisation:

$$\{\top\}$$
$$\texttt{int x = 0 ; int k = 0;}$$
$$\{x = k^2\}$$
$$\texttt{while (k < n) \{ x = x + 2*k + 1; k = k + 1; \}}$$
$$\{x = k^2 \wedge k = n\}$$

This establishes that the *while* loop correctly computes the square of n in x.

So how do we derive the Hoare rule for *while* in PDL? Let the premise $\{\varphi \wedge \psi\}\ \alpha\ \{\psi\}$ be given, i.e., assume (6.1).

$$\vdash (\varphi \wedge \psi) \rightarrow [\alpha]\psi. \tag{6.1}$$

We wish to derive the conclusion

$$\vdash \{\psi\}\ \textit{while}\ \varphi\ \textit{do}\ \alpha\ \{\neg\varphi \wedge \psi\},$$

i.e., we wish to derive (6.2).

$$\vdash \psi \rightarrow [(?\varphi; \alpha)^*; ?\neg\varphi](\neg\varphi \wedge \psi). \tag{6.2}$$

From (6.1) by means of propositional reasoning:

$$\vdash \psi \to (\varphi \to [\alpha]\psi).$$

From this, by means of the test and sequence axioms:

$$\vdash \psi \to [?\varphi;\alpha]\psi.$$

Applying the loop invariance rule gives:

$$\vdash \psi \to [(?\varphi;\alpha)^*]\psi.$$

Since ψ is propositionally equivalent with $\neg\varphi \to (\neg\varphi \wedge \psi)$, we get from this by propositional reasoning:

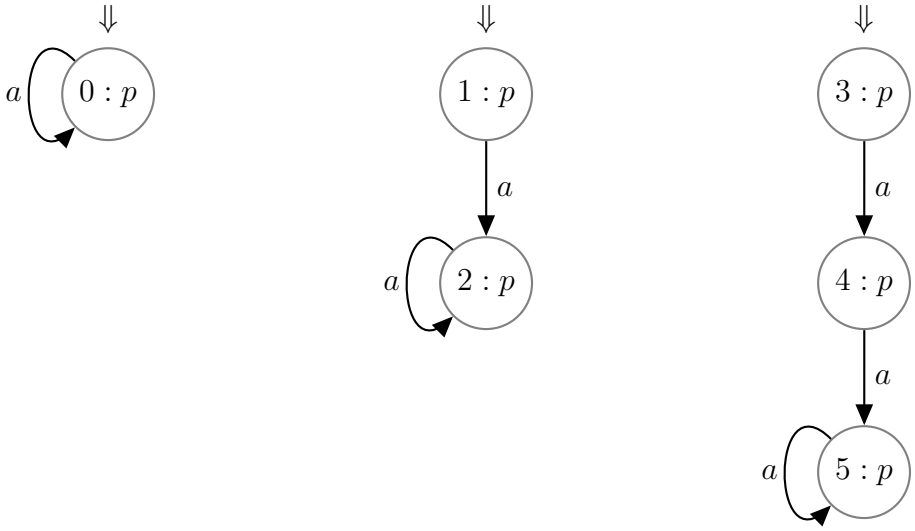
$$\vdash \psi \to [(?\varphi;\alpha)^*](\neg\varphi \to (\neg\varphi \wedge \psi)).$$

The test axiom and the sequencing axiom yield the desired result (6.2).

6.11 Outlook — Equivalence of Programs and Bisimulation

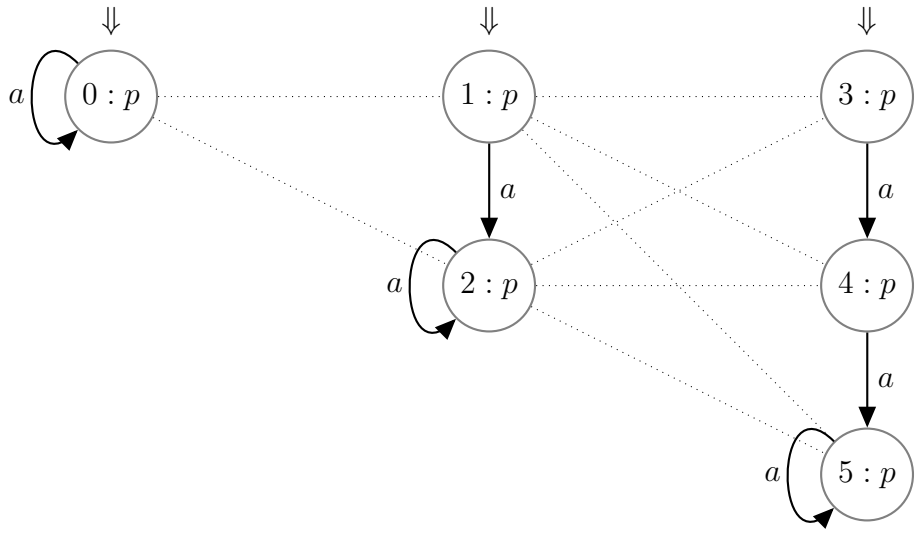
PDL is interpreted in labelled transition systems, and labelled transition systems represent processes. But the correspondence between labelled transition systems and processes is not one-to-one.

Example 6.42 The process that produces an infinite number of a transitions and nothing else can be represented as a labelled transition system in lots of different ways. The following representations are all equivalent, and all represent that process. We further assume that some atomic proposition p is true in all states in all structures.



Each of these three process graphs pictures what is intuitively the following process: that of repeatedly doing a steps, while remaining in a state satisfying p, with no possibility of escape. Think of the actions as ticks of clock, and the state as the state of being imprisoned. The clock ticks on, and you remain in jail forever.

It does not make a difference *for what we can observe directly* (in the present case: that we are in a p state) and *for what we can do* (in the present case: an a action, and nothing else) whether we are in state 0, 1, 2, 3, 4 or 5. From a local observation and action perspective, all of these states are *equivalent*. Below we will make this notion of equivalence precise. For now, we indicate it with connecting lines, as follows:



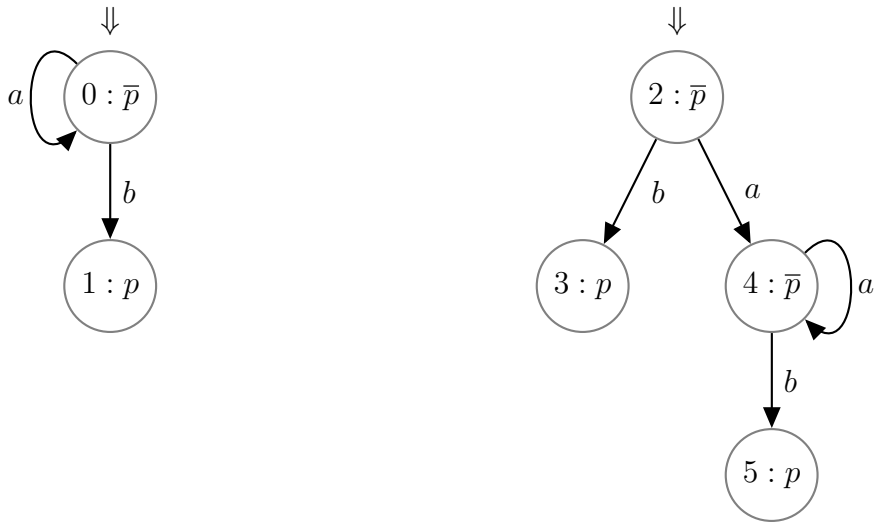
To connect the example to PDL: in all states in each process graph the formulas $\langle a^* \rangle p$, $\langle a; a^* \rangle p$, $\langle a; a; a^* \rangle p$, and so on, are all true. Moreover, it will not be possible to find a PDL formula that sees a difference between the root states of the three process graphs.

We will give a formal definition of this important relation of 'being equivalent from a local action perspective'. We call this relation *bisimulation*, and we say that states that are in the relation are *bisimilar*. Common notation for this is the symbol $\underline{\leftrightarrow}$. Thus, $s \underline{\leftrightarrow} t$ expresses that there is some relation C which is a bisimulation, such that sCt.

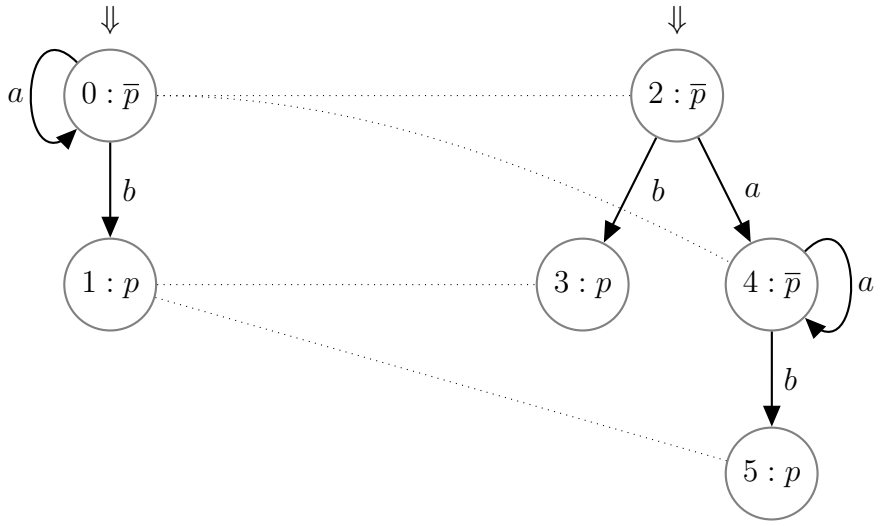
For the picture above we have: $0 \underline{\leftrightarrow} 1$, $0 \underline{\leftrightarrow} 2$, and also, between the middle and the right graph: $1 \underline{\leftrightarrow} 3$, $1 \underline{\leftrightarrow} 4$, $1 \underline{\leftrightarrow} 5$, $2 \underline{\leftrightarrow} 3$, $2 \underline{\leftrightarrow} 4$, $2 \underline{\leftrightarrow} 5$. The composition of two bisimulations is again a bisimulation, and we get from the above that we also have: $0 \underline{\leftrightarrow} 3$, $0 \underline{\leftrightarrow} 4$ and $0 \underline{\leftrightarrow} 5$.

We can also have bisimilarity within a single graph: $1 \underline{\leftrightarrow} 2$, and $3 \underline{\leftrightarrow} 4$, $3 \underline{\leftrightarrow} 5$, $4 \underline{\leftrightarrow} 5$. Note that every node is bisimilar with itself.

Example 6.43 For another example, consider the following picture. Atom p is false in states 0, 2, and 4, and true in states 1, 3 and 5.



In the labelled transition structures of the picture, we have that $0 \underline{\leftrightarrow} 2$, and that $0 \underline{\leftrightarrow} 4$; and $1 \underline{\leftrightarrow} 3$ and $1 \underline{\leftrightarrow} 5$. In a picture:



The notion of *bisimulation* is intended to capture such process equivalences.

Definition 6.44 (Bisimulation) A bisimulation C between LTSs M and N is a relation on $S_M \times S_N$ such that if sCt then the following hold:

Invariance $V_M(s) = V_N(t)$ (the two states have the same valuation),

Zig if for some $s' \in S_M$ $s \xrightarrow{a} s' \in R_M$ then there is a $t' \in S_N$ with $t \xrightarrow{a} t' \in R_N$ and $s'Ct'$.

Zag same requirement in the other direction: if for some $t' \in S_N$ $t \xrightarrow{a} t' \in R_N$ then there is an $s' \in S_M$ with $s \xrightarrow{a} s' \in R_M$ and $s'Ct'$.

The notation $M, s \leftrightarrows N, t$ indicates that there is a bisimulation C that connects s and t. In such a case one says that s and t are bisimilar.

Let M, N be a pair of models and let $C \subseteq S_M \times S_N$. Here is an easy check to see whether C is a bisimulation. For convenience we assume that each model has just a single binary relation (indicated as R_M and R_N). Checking the *invariance* condition is obvious. To check the *zig* condition, check whether

$$C^\smile \circ R_M \subseteq R_N \circ C^\smile.$$

To check the *zag* condition, check whether

$$C \circ R_N \subseteq R_M \circ C.$$

Example 6.45 (Continued from Example 6.43) To see how this works, consider the two models of Example 6.43. Let C be given by

$$\{(0,2),(0,4),(1,3),(1,5)\}.$$

Then the invariance condition holds, for any two states that are C-connected agree in the valuation for p.

Furthermore, $C^\smile \circ R_{M,a} = \{(0,0)\}$ and $R_{N,a} \circ C^\smile = \{(0,0)\}$, so the zig condition holds for the a labels. $C^\smile \circ R_{M,b} = \{(0,1)\}$, and $R_{N,b} \circ C^\smile = \{(0,1)\}$, so the zig condition also holds for the b labels.

Finally, $C \circ R_{N,a} = \{(2,4),(4,4)\}$ and $R_{M,a} \circ C = \{(2,4),(4,4)\}$, so the zag condition holds for the a labels. $C \circ R_{N,b} = \{(0,3),(0,5)\}$, and $R_{M,b} \circ C = \{(0,3),(0,5)\}$, so the zag condition also holds for the b labels.

This shows that C is a bisimulation.

Exercise 6.46 Have another look at Exercise 6.23. Explain why it is impossible to find a PDL formula that is true at the root of one of the graphs and false at the root of the other graph.

Bisimulation is intimately connected to modal logic and to PDL. Modal logic is a sublogic of PDL. It is given by restricting the set of programs to atomic programs.

$$\varphi \quad ::= \quad \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \langle a \rangle \varphi$$

Modal formulas can be used to define global properties of LTSs, as follows. Any modal formula φ can be viewed as a function that maps an LTS M to a subset of S_M, namely the set of those states where φ is true. Call this set φ_M. A global property φ is *invariant for bisimulation* if whenever C is a bisimulation between M and N with sCt, then $s \in \varphi_M$ iff $t \in \varphi_N$.

The notion of invariance for bisimulation generalises the invariance condition of bisimulations.

Exercise 6.47 Show that all modal formulas are invariant for bisimulation: If φ is a modal formula that is true of a state s, and s is bisimilar to t, then φ is true of t as well. (Hint: use induction on the structure of φ.)

Bisimulations are also intimately connected to PDL. Any PDL program α can be viewed as a global relation on LTSs, for α can be viewed as a function that maps an LTS M to a subset of $S_M \times S_M$, namely, the interpretation of α in M. Call this interpretation α_M. A global relation α is *safe for bisimulation* if whenever C is a bisimulation between M and N with sCt, then:

Zig: if $s\alpha_M s'$ for some $s' \in S_M$ then there is a $t' \in S_N$ with $t\alpha_N t'$ and $s'Ct'$,

Zag: vice versa: if $t\alpha_N t'$ for some $t' \in S_N$ then there is an $s' \in S_M$ with $s\alpha_M s'$ and $s'Ct'$.

The notion of safety for bisimulation generalises the zig and zag conditions of bisimulations.

Exercise 6.48 A modal action is a PDL program (action statement) that does not contain *. Use induction on the structure of α to show that all modal actions α are safe for bisimulation.

Summary of Things You Have Learnt in This Chapter *You have learnt how to look at action in a general way, and how to apply a general formal perspective to the analysis of action. You know what labelled transition systems (or: process graphs) are, and you are able to evaluate PDL formulas in states of LTSs. You understand how key programming concepts such as test, composition, choice, repetition, converse are handled in PDL, and how the familiar constructs 'skip','if-then-else', 'while-do', and 'repeat-until' can be expressed in terms of the PDL operations. You are able to check if a given program can be executed on a simple labelled transition system. Finally, you have an intuitive grasp of the notion of bisimulation, and you are able to check whether two states in a single process graph or in different process graphs are bisimilar.*

Further Reading An influential philosophy of action is sketched in [Dav67]. A classical logic of actions is PDL or propositional dynamic logic [Pra78, Pra80, KP81]. A textbook treatment of dynamic logic is presented in [HKT00].

Precise descriptions of how to perform given tasks are called algorithms. The logic of actions is closely connected to the theory of algorithm design. See [DH04]. Connections between logic and (functional) programming are treated in [DvE04].

Social actions are the topic of [EV09].