Topics in Concurrency
Lecture Notes

Glynn Winskel

February 7, 2008

Syllabus for Topics in Concurrency

Lecturer: Professor G. Winskel (`gw104@cl.cam.ac.uk`)

No. of lectures: 12

Prerequisite courses: Semantics of Programming Languages

Aims

The aim of this course is to introduce fundamental concepts and techniques in the theory of concurrent processes. It will provide languages, models, logics and methods to formalise and reason about concurrent systems.

Lecture plan

- **Simple parallelism and nondeterminism.** Dijkstra's guarded commands. Communication by shared variables: A language of parallel commands. [1 lecture]
- **Communicating processes.** Milner's Calculus of Communicating Processes (CCS). Pure CCS. Labelled-transition-system semantics. Bisimulation and weak bisimulation equivalence. Equational consequences and examples. [3 lectures]
- **Specification and model-checking.** The modal μ-calculus. Its mathematical foundations in Tarski's fixed point theorem. Its relation with Temporal Logic. Introduction to model checking. Bisimulation checking. Examples. [3 lectures]
- **Introduction to Petri nets.** Petri nets, basic definitions and concepts. Petri-net semantics of CCS. [1 lecture]
- **Security protocols.** Security protocols informally. SPL, a language for security protocols. Its transition-system semantics. Its Petri-net semantics. Properties of security protocols: secrecy, authentication. Examples with proofs of correctness. [1 lecture]
- **Mobile computation.** An introduction to the higher-order process language HOPLA. Its properties and encodings of other languages such as CCS, higher-order CCS and the Ambient Calculus with public names. Research issues [3 lectures]

I will detail the examinable topics at the end of the course.

Objectives

At the end of the course students should

- know the basic theory of concurrent processes: nondeterministic and parallel commands, the process language CCS, its transition-system semantics, bisimulation, the modal μ-calculus, the temporal logic CTL, Petri nets, basic model checking, a process language for security protocols and its semantics, process languages for mobile computation.

- be able to formalise and to some extent analyse concurrent processes: establish bisimulation or its absence in simple cases, express and establish simple properties of transition systems in the modal μ-calculus, argue with respect to a process language semantics for secrecy or authentication properties of a small security protocol, exercise a basic understanding of process languages with process passing and name generation (as in SPL).

Reading guide

It's recommended that you skip Chapter 1, *apart from* the sections on well-founded induction and Tarski's fixed-point theorem. (You may find the rest of Chapter 1 useful for occasional reference for notation, or perhaps as a revision of the relevant parts from "Discrete Mathematics.")

Chapter 2 is important historically, though largely motivational. (This is not to exclude absolutely the possibility of Tripos questions on closely related topics.)

The bulk of the material is from chapter 3 on.

The notes contain many proofs, and you're not expected to memorise these. However the exam questions will assume familiarity with the various techniques outlined in "Objectives" above.

You are encouraged to do the exercises. Hard, or more peripheral exercises, are marked with a "$*$," and can be ignored.

Relevant past Tripos questions:

Those available from the Computer Laboratory's webpages under Topics in Concurrency, from 2001 on, together with

Communicating Automata and Pi Calculus:
1996 Paper 7 Question 12 (amended version)
1997 Paper 7 Question 12
1998 Paper 8 Question 15
1999 Paper 7 Question 13

Concurrency:
1993 Paper 9 Question 12
1994 Paper 7 Question 14
1994 Paper 8 Question 14

Additional reading:

Clarke, E., Grumberg, O., and Peled, D., (1999) *Model checking*. MITPress.

Milner, R., (1989). *Communication and Concurrency*. Prentice Hall.

Milner, R., (1999). *Communicating and mobile systems: the Pi-Calculus*. CUP.

Reisig, W., (1985) *Petri nets: an introduction*. EATCS Monographs on Theoretical Computer Science, Springer-Verlag.

Contents

Chapter 1

Mathematical Foundations

This chapter is meant largely as a review and for future reference. We will however be making heavy use of well-founded induction and the fixed point theorems for monotonic and continuous functions on powersets will be important for model checking.

1.1 Logical notation

We shall use some informal logical notation in order to stop our mathematical statements getting out of hand. For statements (or assertions) A and B, we shall commonly use abbreviations like:

- $A \mathbin{\&} B$ for (A and B), the conjunction of A and B,
- $A \Rightarrow B$ for (A implies B), which means (if A then B),
- $A \iff B$ to mean (A iff B), which abbreviates (A if and only if B), and expresses the logical equivalence of A and B.

We shall also make statements by forming disjunctions (A or B), with the self-evident meaning, and negations (not A), sometimes written $\neg A$, which is true iff A is false. There is a tradition to write for instance $7 \not< 5$ instead of $\neg(7 < 5)$, which reflects what we generally say: "7 is not less than 5" rather than "not 7 is less than 5."

The statements may contain variables (or unknowns, or place-holders), as in

$$(x \leq 3) \mathbin{\&} (y \leq 7)$$

which is true when the variables x and y over integers stand for integers less than or equal to 3 and 7 respectively, and false otherwise. A statement like $P(x, y)$, which involves variables x, y, is called a predicate (or property, or relation, or condition) and it only becomes true or false when the pair x, y stand for particular things.

We use logical quantifiers $\exists$, read "there exists", and $\forall$, read " for all". Then you can read assertions like

$$\exists x.\ P(x)$$

as abbreviating "for some x, $P(x)$" or "there exists x such that $P(x)$", and

$$\forall x.\ P(x)$$

as abbreviating " for all x, $P(x)$" or "for any x, $P(x)$". The statement

$$\exists x, y, \cdots, z.\ P(x, y, \cdots, z)$$

abbreviates

$$\exists x \exists y \cdots \exists z.\ P(x, y, \cdots, z),$$

and

$$\forall x, y, \cdots, z.\ P(x, y, \cdots, z)$$

abbreviates

$$\forall x \forall y \cdots \forall z.\ P(x, y, \cdots, z).$$

Later, we often wish to specify a set X over which a quantifier ranges. Then one writes $\forall x \in X.\ P(x)$ instead of $\forall x.\ x \in X \Rightarrow P(x)$, and $\exists x \in X.\ P(x)$ instead of $\exists x.\ x \in X \ \&\ P(x)$.

There is another useful notation associated with quantifiers. Occasionally one wants to say not just that there exists some x satisfying a property $P(x)$ but also that x is the *unique* object satisfying $P(x)$. It is traditional to write

$$\exists! x.\ P(x)$$

as an abbreviation for

$$(\exists x.\ P(x)) \ \&\ (\forall y, z.\ P(y) \ \&\ P(z) \Rightarrow y = z)$$

which means that there is some x satisfying the property P and also that if any y, z both satisfy the property P they are equal. This expresses that there exists a unique x satisfying $P(x)$.

1.2 Sets

Intuitively, a set is an (unordered) collection of objects, called its *elements* or *members*. We write $a \in X$ when a is an element of the set X. Sometimes we write *e.g.* $\{a, b, c, \cdots\}$ for the set of elements $a, b, c, \cdots$.
A set X is said to be a *subset* of a set Y, written $X \subseteq Y$, iff every element of X is an element of Y, *i.e.*

$$X \subseteq Y \iff \forall z \in X.\ z \in Y.$$

A set is determined solely by its elements in the sense that two sets are equal iff they have the same elements. So, sets X and Y are equal, written $X = Y$, iff every element of A is a element of B and *vice versa*. This furnishes a method for showing two sets X and Y are equal and, of course, is equivalent to showing $X \subseteq Y$ and $Y \subseteq X$.

1.2.1 Sets and properties

Sometimes a set is determined by a property, in the sense that the set has as elements precisely those which satisfy the property. Then we write

$$X = \{x \mid P(x)\},$$

meaning the set X has as elements precisely all those x for which $P(x)$ is true.

When set theory was being invented it was thought, first of all, that any property $P(x)$ determined a set

$$\{x \mid P(x)\}.$$

It came as a shock when Bertrand Russell realised that assuming the existence of certain sets described in this way gave rise to contradictions.

Russell's paradox is really the demonstration that a contradiction arises from the liberal way of constructing sets above. It proceeds as follows: consider the property

$$x \notin x$$

a way of writing "x is not an element of x". If we assume that properties determine sets, just as described, we can form the set

$$R = \{x \mid x \notin x\}.$$

Either $R \in R$ or not. If so, *i.e.* $R \in R$, then in order for R to qualify as an element of R, from the definition of R, we deduce $R \notin R$. So we end up asserting both something and is negation—a contradiction. If, on the other hand, $R \notin R$ then from the definition of R we see $R \in R$—a contradiction again. Either $R \in R$ or $R \notin R$ lands us in trouble.

We need to have some way which stops us from considering things like R as a sets. In general terms, the solution is to discipline the way in which sets are constructed, so that starting from certain given sets, new sets can only be formed when they are constructed by using particular, safe ways from old sets. We shall not be formal about it, but state those sets we assume to exist right from the start and methods we allow for constructing new sets. Provided these are followed we avoid trouble like Russell's paradox and at the same time have a rich enough world of sets to support most mathematics.

1.2.2 Some important sets

We take the existence of the empty set for granted, along with certain sets of basic elements.
Write $\emptyset$ for the *null*, or *empty* set, and
ω for the set of natural numbers $0, 1, 2, \cdots$.

We shall also take sets of symbols like

$$\{\text{“}a\text{”}, \text{“}b\text{”}, \text{“}c\text{”}, \text{“}d\text{”}, \text{“}e\text{”}, \cdots, \text{“}z\text{”}\}$$

for granted, although we could, alternatively have represented them as particular numbers, for example. The equality relation on a set of symbols is that given by syntactic identity; two symbols are equal iff they are the same.

1.2.3 Constructions on sets

We shall take for granted certain operations on sets which enable us to construct sets from given sets.

Comprehension:

If X is a set and $P(x)$ is a property, we can form the set

$$\{x \in X \mid P(x)\}$$

which is another way of writing

$$\{x \mid x \in X \ \& \ P(x)\}.$$

This is the subset of X consisting of all elements x of X which satisfy $P(x)$.

Sometimes we'll use a further abbreviation. Suppose $e(x_1, \ldots, x_n)$ is some expression which for particular elements $x_1 \in X_1, \cdots x_n \in X_n$ yields a particular element and $P(x_1, \ldots, x_n)$ is a property of such $x_1, \ldots, x_n$. We use

$$\{e(x_1, \ldots, x_n) \mid x_1 \in X_1 \ \& \ \cdots \& \ x_n \in X_n \ \& \ P(x_1, \ldots, x_n)\}$$

to abbreviate

$$\{y \mid \exists x_1 \in X_1, \cdots, x_n \in X_n. \ y = e(x_1, \ldots, x_n) \& \ P(x_1, \ldots, x_n)\}.$$

For example,

$$\{2m + 1 \mid m \in \omega \ \& \ m > 1\}$$

is the set of odd numbers greater than 3.

Powerset:

We can form a set consisting of the set of all subsets of a set, the so-called *powerset*:

$$\mathcal{P}ow(X) = \{Y \mid Y \subseteq X\}.$$

Indexed sets:

Suppose I is a set and that for any $i \in I$ there is a unique object x_i, maybe a set itself. Then

$$\{x_i \mid i \in I\}$$

is a set. The elements x_i are said to be *indexed* by the elements $i \in I$.

Union:

The set consisting of the *union* of two sets has as elements those elements which are either elements of one or the other set. It is written and described by:

$$X \cup Y = \{a \mid a \in X \text{ or } a \in Y\}.$$

Big union:

Let X be a set of sets. Their *union*

$$\bigcup X = \{a \mid \exists x \in X.\ a \in x\}$$

is a set. When $X = \{x_i \mid i \in I\}$ for some indexing set I we often write $\bigcup X$ as $\bigcup_{i \in I} x_i$.

Intersection:

Elements are in the *intersection* $X \cap Y$, of two sets X and Y, iff they are in both sets, *i.e.*

$$X \cap Y = \{a \mid a \in X \ \&\ a \in Y\}.$$

Big intersection:

Let X be a nonempty set of sets. Then

$$\bigcap X = \{a \mid \forall x \in X.\ a \in x\}$$

is a set called its *intersection*. When $X = \{x_i \mid i \in I\}$ for a nonempty indexing set I we often write $\bigcap X$ as $\bigcap_{i \in I} x_i$.

Product:

Given two elements a, b we can form a set (a, b) which is their ordered pair. To be definite we can take the ordered pair (a, b) to be the set $\{\{a\}, \{a, b\}\}$—this is one particular way of coding the idea of ordered pair as a *set*. As one would hope, two ordered pairs, represented in this way, are equal iff their first components are equal and their second components are equal too, *i.e.*

$$(a, b) = (a', b') \iff a = a' \ \&\ b = b'.$$

In proving properties of ordered pairs this property should be sufficient irrespective of the way in which we have represented ordered pairs as sets.

For sets X and Y, their *product* is the set

$$X \times Y = \{(a, b) \mid a \in X \ \&\ b \in Y\},$$

the set of ordered pairs of elements with the first from X and the second from Y.

A triple (a, b, c) is the set $(a, (b, c))$, and the product $X \times Y \times Z$ is the set of triples $\{(x, y, z) \mid x \in X \ \&\ y \in Y \ \&\ z \in Z\}$. More generally $X_1 \times X_2 \times \cdots \times X_n$ consists of the set of n-tuples $(x_1, x_2, \ldots, x_n) = (x_1, (x_2, (x_3, \cdots)))$.

Disjoint union:

Frequently we want to join sets together but, in a way which, unlike union, does not identify the same element when it comes from different sets. We do this by making copies of the elements so that when they are copies from different sets they are forced to be distinct.

$$X_0 \uplus X_1 \uplus \cdots \uplus X_n = (\{0\} \times X_0) \cup (\{1\} \times X_1) \cup \cdots \cup (\{n\} \times X_n).$$

In particular, for $X \uplus Y$ the copies $(\{0\} \times X)$ and $(\{1\} \times Y)$ have to be disjoint, in the sense that

$$(\{0\} \times X) \cap (\{1\} \times Y) = \emptyset,$$

because any common element would be a pair with first element both equal to 0 and 1, clearly impossible.

Set difference:

We can subtract one set Y from another X, an operation which removes all elements from X which are also in Y.

$$X \setminus Y = \{x \mid x \in X \;\&\; x \notin Y\}.$$

1.2.4 The axiom of foundation

A set is built-up starting from basic sets by using the constructions above. We remark that a property of sets, called the axiom of foundation, follows from our informal understanding of sets and how we can construct them. Consider an element b_1 of a set b_0. It is either a basic element, like an integer or a symbol, or a set. If b_1 is a set then it must have been constructed from sets which have themselves been constructed earlier. Intuitively, we expect any chain of memberships

$$\cdots b_n \in \cdots \in b_1 \in b_0$$

to end in some b_n which is some basic element or the empty set. The statement that any such descending chain of memberships must be finite is called the axiom of foundation, and is an assumption generally made in set theory. Notice the axiom implies that no set X can be a member of itself as, if this were so, we'd get the infinite descending chain

$$\cdots X \in \cdots \in X \in X,$$

—a contradiction.

1.3 Relations and functions

A *binary relation* between X and Y is an element of $\mathcal{P}ow(X \times Y)$, and so a subset of pairs in the relation. When R is a relation $R \subseteq X \times Y$ we shall often write xRy for $(x, y) \in R$.

A *partial function* from X to Y is a relation $f \subseteq X \times Y$ for which

$$\forall x, y, y'. \ (x, y) \in f \ \& \ (x, y') \in f \Rightarrow y = y'.$$

We use the notation $f(x) = y$ when there is a y such that $(x, y) \in f$ and then say $f(x)$ is *defined*, and otherwise say $f(x)$ is *undefined*. Sometimes we write $f : x \mapsto y$, or just $x \mapsto y$ when f is understood, for $y = f(x)$. Occasionally we write just fx, without the brackets, for $f(x)$.

A *(total) function* from X to Y is a partial function from X to Y such that for all $x \in X$ there is some $y \in Y$ such that $f(x) = y$. Although total functions are a special kind of partial function it is traditional to understand something described as simply a function to be a total function, so we always say explicitly when a function is partial.

Note that relations and functions are also sets.

To stress the fact that we are thinking of a partial function f from X to Y as taking an element of X and yielding an element of Y we generally write it as $f : X \rightharpoonup Y$. To indicate that a function f from X to Y is total we write $f : X \to Y$.

We write $(X \rightharpoonup Y)$ for the set of all partial functions from X to Y, and $(X \to Y)$ for the set of all total functions.

Exercise 1.1 * Why are we justified in calling $(X \rightharpoonup Y)$ and $(X \to Y)$ sets when X, Y are sets? $\Box$

1.3.1 Composing relations and functions

We compose relations, and so partial and total functions, R between X and Y and S between Y and Z by defining their *composition*, a relation between X and Z, by

$$S \circ R =_{def} \{(x, z) \in X \times Z \mid \exists y \in Y. \ (x, y) \in R \ \& \ (y, z) \in S\}.$$

Thus for functions $f : X \to Y$ and $g : Y \to Z$ their composition is the function $g \circ f : X \to Z$. Each set X is associated with an identity function Id_X where $Id_X = \{(x, x) \mid x \in X\}$.

Exercise 1.2 * Let $R \subseteq X \times Y$, $S \subseteq Y \times Z$ and $T \subseteq Z \times W$. Convince yourself that $T \circ (S \circ R) = (T \circ S) \circ R$ (*i.e.* composition is associative) and that $R \circ Id_X = Id_Y \circ R = R$ (*i.e.* identity functions act like identities with respect to composition). $\Box$

A function $f : X \to Y$ has an *inverse* $g : Y \to X$ iff $g(f(x)) = x$ for all $x \in X$, and $f(g(y)) = y$ for all $y \in Y$. Then the sets X and Y are said to be in *1-1 correspondence*. (Note a function with an inverse has to be total.)

Any set in 1-1 correspondence with a subset of natural numbers ω is said to be *countable*.

Exercise 1.3 * Let X and Y be sets. Show there is a 1-1 correspondence between the set of functions $(X \to \mathcal{P}ow(Y))$ and the set of relations $\mathcal{P}ow(X \times Y)$. $\square$

Cantor's diagonal argument

Late last century, Georg Cantor, one of the pioneers in set theory, invented a method of argument, the gist of which reappears frequently in the theory of computation. Cantor used a *diagonal argument* to show that X and $\mathcal{P}ow(X)$ are never in 1-1 correspondence for any set X. This fact is intuitively clear for finite sets but also holds for infinite sets. He argued by *reductio ad absurdum*, *i.e.*, by showing that supposing otherwise led to a contradiction:

Suppose a set X is in 1-1 correspondence with its powerset $\mathcal{P}ow(X)$. Let $\theta : X \to \mathcal{P}ow(X)$ be the 1-1 correspondence. Form the set

$$Y = \{x \in X \mid x \notin \theta(x)\}$$

which is clearly a subset of X and therefore in correspondence with an element $y \in X$. That is $\theta(y) = Y$. Either $y \in Y$ or $y \notin Y$. But both possibilities are absurd. For, if $y \in Y$ then $y \in \theta(y)$ so $y \notin Y$, while, if $y \notin Y$ then $y \notin \theta(y)$ so $y \in Y$. We conclude that our first supposition must be false, so there is no set in 1-1 correspondence with its powerset.

Cantor's argument is reminiscient of Russell's paradox. But whereas the contradiction in Russell's paradox arises out of a fundamental, mistaken assumption about how to construct sets, the contradiction in Cantor's argument comes from denying the fact one wishes to prove.

To see why it is called a diagonal argument, imagine that the set X, which we suppose is in 1-1 correspondence with $\mathcal{P}ow(X)$, can be enumerated as $x_0, x_1, x_2, \cdots, x_n, \cdots$. Imagine we draw a table to represent the 1-1 correspondence θ along the following lines. In the ith row and jth column is placed 1 if $x_i \in \theta(x_j)$ and 0 otherwise. The table below, for instance, represents a situation where $x_0 \notin \theta(x_0)$, $x_1 \in \theta(x_0)$ and $x_i \in \theta(x_j)$.

	$\theta(x_0)$	$\theta(x_1)$	$\theta(x_2)$	$\cdots$	$\theta(x_j)$	$\cdots$
x_0	0	1	1	$\cdots$	1	$\cdots$
x_1	1	1	1	$\cdots$	0	$\cdots$
x_2	0	0	1	$\cdots$	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	
x_i	0	1	0	$\cdots$	1	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	

The set Y which plays a key role in Cantor's argument is defined by running down the diagonal of the table interchanging 0's and 1's in the sense that x_n is put in the set iff the nth entry along the diagonal is a 0.

Exercise 1.4 * Show for any sets X and Y, with Y containing at least two elements, that there cannot be a 1-1 correspondence between X and the set of functions $(X \to Y)$. $\square$

1.3.2 Direct and inverse image of a relation

We extend relations, and thus partial and total functions, $R : X \times Y$ to functions on subsets by taking

$$RA = \{y \in Y \mid \exists x \in A.\ (x, y) \in R\}$$

for $A \subseteq X$. The set RA is called the *direct image* of A under R. We define

$$R^{-1}B = \{x \in X \mid \exists y \in B.\ (x, y) \in R\}$$

for $B \subseteq Y$. The set $R^{-1}B$ is called the *inverse image* of B under R. Of course, the same notions of direct and inverse image also apply in the special case where the relation is a function.

1.3.3 Equivalence relations

An *equivalence relation* is a relation $R \subseteq X \times X$ on a set X which is

- reflexive: $\forall x \in X.\ xRx$,
- symmetric: $\forall x, y \in X.\ xRy \Rightarrow yRx$ and
- transitive: $\forall x, y, z \in X.\ xRy \ \&\ yRz \Rightarrow xRz$.

If R is an equivalence relation on X then the *(R-)equivalence class* of an element $x \in X$ is the subset $\{x\}_R =_{def} \{y \in X \mid yRx\}$.

Exercise 1.5 * Let R be an equivalence relation on a set X. Show if $\{x\}_R \cap \{y\}_R \neq \emptyset$ then $\{x\}_R = \{y\}_R$, for any elements $x, y \in X$. □

Exercise 1.6 * Let xRy be a relation on a set of sets X which holds iff the sets x and y in X are in 1-1 correspondence. Show that R is an equivalence relation. □

Let R be a relation on a set X. Define $R^0 = Id_X$, the identity relation on the set X, and $R^1 = R$ and, assuming R^n is defined, define

$$R^{n+1} = R \circ R^n.$$

So, R^n is the relation $R \circ \cdots \circ R$, obtained by taking n compositions of R. Define the *transitive closure* of R to be the relation

$$R^+ = \bigcup_{n \in \omega} R^{n+1}.$$

Define the transitive, reflexive closure of a relation R on X to be the relation

$$R^* = \bigcup_{n \in \omega} R^n,$$

so $R^* = Id_X \cup R^+$.

Let R be a relation on a set X. Write R^{op} for the *opposite*, or *converse*, relation $R^{op} = \{(y, x) \mid (x, y) \in R\}$.

Exercise 1.7 * Show $(R \cup R^{op})^*$ is an equivalence relation. Show $R^* \cup (R^{op})^*$ need not be an equivalence relation. □

1.3.4 Relations as structure—partial orders

Definition: A *partial order* (p.o.) is a set P on which there is a binary relation $\sqsubseteq$, so described by $(P, \sqsubseteq)$, which is:
(i) reflexive: $\forall p \in P.\ p \sqsubseteq p$
(ii) transitive: $\forall p, q, r \in P.\ p \sqsubseteq q\ \&\ q \sqsubseteq r \Rightarrow p \sqsubseteq r$
(iii) antisymmetric: $\forall p, q \in P.\ p \sqsubseteq q\ \&\ q \sqsubseteq p \Rightarrow p = q$.

If we relax the definition of partial order and do not insist on (iii) antisymmetry, and only retain (i) reflexivity and (ii) transitivity, we have defined a *preorder* on a set.

Example: Let $\mathcal{S}$ be a set. Its powerset with the subset relation, $(\mathcal{P}ow(\mathcal{S}), \subseteq)$, is a partial order.

Often the partial order supports extra structure. For example, in a partial order $(P, \sqsubseteq)$, the *least upper bound* (*lub*, or *supremum*, or *join*) of a subset $X \subseteq P$ of a partial order is an element $\bigsqcup X \in P$ such that for all $p \in P$,

$$(\forall x \in X. x \sqsubseteq p) \Rightarrow \bigsqcup X \sqsubseteq p\ .$$

An element p such that $(\forall x \in X. x \sqsubseteq p)$ is called an *upper bound.* In a dual way, the *greatest lower bound* (*glb*, *infimum* or *meet*) of a subset $X \subseteq P$ is an element $\bigsqcap X \in P$ such that for all $p \in P$,

$$(\forall x \in X. p \sqsubseteq x) \Rightarrow p \sqsubseteq \bigsqcap X\ ,$$

and an element p such that $(\forall x \in X. p \sqsubseteq x)$ is called a *lower bound.* In the example of a partial order $(\mathcal{P}ow(\mathcal{S}), \subseteq)$, lubs are given by unions and glbs by intersections. A general partial order need not have all lubs and glbs. When it does it is called a *complete lattice.*

Exercise 1.8 Show that if a partial order has all lubs, then it necessarily also has all glbs and *vice versa.* □

1.4 Well-founded induction

Mathematical and structural induction are special cases of a general and powerful proof principle called well-founded induction. In essence structural induction works because breaking down an expression into subexpressions cannot go on forever, eventually it must lead to atomic expressions which cannot be broken down any further. If a property fails to hold of any expression then it must fail on some minimal expression which when it is broken down yields subexpressions, all of which satisfy the property. This observation justifies the principle of structural induction: to show a property holds of all expressions it is sufficient to show that a property holds of an arbitrary expression if it holds of all its subexpressions. Similarly with the natural numbers, if a property fails to hold of all natural numbers then there has to be a smallest natural number at which it fails. The essential feature shared by both the subexpression relation and the predecessor relation on natural numbers is that do not give rise to infinite descending chains. This is the feature required of a relation if it is to support well-founded induction.

Definition: A *well-founded relation* is a binary relation $\prec$ on a set A such that there are no infinite descending chains $\cdots \prec a_i \prec \cdots \prec a_1 \prec a_0$. When $a \prec b$ we say a is a *predecessor* of b.

Note a well-founded relation is necessarily *irreflexive i.e.*, for no a do we have $a \prec a$, as otherwise there would be the infinite decending chain $\cdots \prec a \prec \cdots \prec a \prec a$. We shall generally write $\preceq$ for the reflexive closure of the relation $\prec$, *i.e.*

$$a \preceq b \iff a = b \text{ or } a \prec b.$$

Sometimes one sees an alternative definition of well-founded relation, in terms of minimal elements.

Proposition 1.9 *Let $\prec$ be a binary relation on a set A. The relation $\prec$ is well-founded iff any nonempty subset Q of A has a minimal element, i.e. an element m such that*

$$m \in Q \;\&\; \forall b \prec m.\, b \notin Q.$$

Proof:
"if": Suppose every nonempty subset of A has a minimal element. If $\cdots \prec a_i \prec \cdots \prec a_1 \prec a_0$ were an infinite descending chain then the set $Q = \{a_i \mid i \in \omega\}$ would be nonempty without a minimal element, a contradiction. Hence $\prec$ is well-founded.
"only if": To see this, suppose Q is a nonempty subset of A. Construct a chain of elements as follows. Take a_0 to be any element of Q. Inductively, assume a chain of elements $a_n \prec \cdots \prec a_0$ has been constructed inside Q. Either there is some $b \prec a_n$ such that $b \in Q$ or there is not. If not stop the construction. Otherwise take $a_{n+1} = b$. As $\prec$ is well-founded the chain $\cdots \prec a_i \prec \cdots \prec a_1 \prec a_0$ cannot be infinite. Hence it is finite, of the form $a_n \prec \cdots \prec a_0$ with $\forall b \prec a_n.\, b \notin Q$. Take the required minimal element m to be a_n. □

Exercise 1.10 Let $\prec$ be a well-founded relation on a set B. Prove

1. its transitive closure $\prec^+$ is also well-founded,
2. its reflexive, transitive closure $\prec^*$ is a partial order.

□

The principle of well-founded induction.

Let $\prec$ be a well founded relation on a set A. Let P be a property. Then $\forall a \in A.\, P(a)$ iff

$$\forall a \in A.\, ([\forall b \prec a.\, P(b)] \Rightarrow P(a)).$$

The principle says that to prove a property holds of all elements of a well-founded set it suffices to show that if the property holds of all predecessors of an arbitrary element a then the property holds of a.

We now prove the principle. The proof rests on the observation that any nonempty subset Q of a set A with a well-founded relation $\prec$ has a minimal element. Clearly if $P(a)$ holds for all elements of A then $\forall a \in A.\ ([\forall b \prec a.\ P(b)] \Rightarrow P(a))$. To show the converse, we assume $\forall a \in A.\ ([\forall b \prec a.\ P(b)] \Rightarrow P(a))$ and produce a contradiction by supposing $\neg P(a)$ for some $a \in A$. Then, as we have observed, there must be a minimal element m of the set $\{a \in A \mid \neg P(a)\}$. But then $\neg P(m)$ and yet $\forall b \prec m.\ P(b)$, which contradicts the assumption.

Example: If we take the relation $\prec$ to be the predecessor relation

$$n \prec m \text{ iff } m = n + 1$$

on the non-negative integers the principle of well-founded induction specialises to mathematical induction. $\square$

Example: If we take $\prec$ to be the "strictly less than" relation $<$ on the non-negative integers, the principle specialises to course-of-values induction. $\square$

Example: If we take $\prec$ to be the relation between expressions such that $a \prec b$ holds iff a is an immediate subexpression of b we obtain the principle of structural induction as a special case of well-founded induction. $\square$

Proposition 1.9 provides an alternative to proofs by well-founded induction. Suppose A is a well-founded set. Instead of using well-founded induction to show every element of A satisfies a property P, we can consider the subset of A for which the property P fails, *i.e.* the subset F of counterexamples. By Proposition 1.9, to show F is $\emptyset$ it is sufficient to show that F cannot have a minimal element. This is done by obtaining a contradiction from the assumption that there is a minimal element in F. Whether to use this approach or the principle of well-founded induction is largely a matter of taste, though sometimes, depending on the problem, one approach can be more direct than the other.

Exercise 1.11 For suitable well-founded relation on strings, use the "no counterexample" approach described above to show there is no string u which satisfies $au = ub$ for two distinct symbols a and b. $\square$

Well-founded induction is the most important principle in proving the termination of programs. Uncertainties about termination arise because of loops or recursions in a program. If it can be shown that execution of a loop or recursion in a program decreases the value in a well-founded set then execution must eventually terminate.

1.5 Fixed points

Let $\mathcal{S}$ be a set. Then its powerset $\mathcal{P}ow(\mathcal{S}$ forms a partial order in which the order is that of inclusion $\subseteq$. We examine conditions under which functions $\varphi : \mathcal{P}ow(\mathcal{S}) \to \mathcal{P}ow(\mathcal{S})$ have canonical fixed points.

1.5.1 Tarski's fixed point theorem

We provide a proof of Tarski's fixed point theorem, specialised to powersets. This concerns fixed points of functions $\varphi : \mathcal{P}ow(\mathcal{S}) \to \mathcal{P}ow(\mathcal{S})$ which are *monotonic*, *i.e.* such that

$$S \subseteq S' \Rightarrow \varphi(S) \subseteq \varphi(S') \ ,$$

for $S, S' \in \mathcal{P}ow(\mathcal{S})$. Such monotonic functions have least (=minimum) and greatest (=maximum) fixed points.

Theorem 1.12 *(Tarski's theorem for minimum fixed points)*
Let $\mathcal{P}ow(\mathcal{S})$ be a powerset. Let $\varphi : \mathcal{P}ow(\mathcal{S}) \to \mathcal{P}ow(\mathcal{S})$ be a monotonic function. Define

$$m = \bigcap \{S \subseteq \mathcal{S} \mid \varphi(S) \subseteq S\}.$$

Then m is a fixed point of φ and the least prefixed point of φ, i.e. if $\varphi(S) \subseteq S$ then $m \subseteq S$. (When $\varphi(S) \subseteq S$ the set S is called a prefixed point *of φ.)*

Proof: Write $X = \{S \subseteq \mathcal{S} \mid \varphi(S) \subseteq S\}$. As above, define $m = \bigcap X$. Let $S \in X$. Certainly $m \subseteq S$. Hence $\varphi(m) \subseteq \varphi(S)$ by the monotonicity of φ. But $\varphi(S) \subseteq S$ because $S \in X$. So $\varphi(m) \subseteq S$ for any $S \in X$. It follows that $\varphi(m) \subseteq \bigcap X = m$. This makes m a prefixed point and, from its definition, it is clearly the least one. As $\varphi(m) \subseteq m$ we obtain $\varphi(\varphi(m)) \subseteq \varphi(m)$ from the monotonicity of φ. This ensures $\varphi(m) \in X$ which entails $m \subseteq \varphi(m)$. Thus $\varphi(m) = m$. We conclude that m is indeed a fixed point and is the least prefixed point of φ. $\square$

The proof of Tarski's theorem for minimum fixed points only makes use of the partial-order properties of the $\subseteq$ relation on $\mathcal{P}ow(\mathcal{S})$ and in particular that there is an intersection operation $\bigcap$. (In fact, Tarski's theorem applies equally well to complete lattice with an abstract partial order and greatest lower bound.) Replacing the roles of the order $\subseteq$ and intersection $\bigcap$ by the converse relation $\supseteq$ and union $\bigcup$ we obtain a proof of the dual result for maximum fixed points.

Theorem 1.13 *(Tarski's theorem for maximum fixed points)*
Let $\mathcal{P}ow(\mathcal{S})$ be a powerset. Let $\varphi : \mathcal{P}ow(\mathcal{S}) \to \mathcal{P}ow(\mathcal{S})$ be a monotonic function. Define

$$M = \bigcup \{S \subseteq \mathcal{S} \mid S \subseteq \varphi(S)\}.$$

Then M is a fixed point of φ and the greatest postfixed point of φ, i.e. if $S \subseteq \varphi(S)$ then $S \subseteq M$. (When $S \subseteq \varphi(S)$ the set S is called a postfixed point *of φ.)*

Notation: The minimum fixed point is traditionally written

$$\mu X.\varphi(X) \ ,$$

and the maximum fixed point as

$$\nu X.\varphi(X) \ .$$

Tarski's theorem for minimum fixed points provides another way to understand sets inductively defined by rules.

A *set of rule instances* R consists of elements which are pairs (X/y) where X is a set and y is an element. A pair (X/y) is called a *rule instance* with *premises* X and conclusion y.

We are more used to seeing rule instances (X/y) as

$$\frac{}{y} \quad \text{if } X = \emptyset, \text{ and as } \frac{x_1, \cdots, x_n}{y} \quad \text{if } X = \{x_1, \cdots, x_n\},$$

though here we aren't insisting on the set of premises X being finite.

Assuming that all the elements in the premises and conclusion lie within a set $\mathcal{S}$, we can turn R into a monotonic function $\varphi_R : \mathcal{P}ow(\mathcal{S}) \to \mathcal{P}ow(\mathcal{S})$: For $S \in \mathcal{P}ow(\mathcal{S})$, define

$$\varphi_R(S) = \{y \mid \exists X \subseteq S.\ (X/y) \in R\}\ .$$

The least fixed point of φ_R coincides with the set inductively defined by the rules R.

Sets defined as maximum fixed points are often called coinductively defined sets.

Exercise 1.14 Let $\mathbf{N}$ be the set of positive natural numbers. Let $\varphi : \mathcal{P}ow(\mathbf{N}) \to \mathcal{P}ow(\mathbf{N})$ be the function on its powerset given by:

$$\varphi(U) = \{3n/2 \mid n \in U\ \&\ n \text{ is even}\} \cup \{n \mid n \in U\ \&\ n \text{ is odd}\}\ .$$

(i) Show φ is monotonic with respect to $\subseteq$.

(ii) Suppose that $U \subseteq \varphi(U)$, *i.e.* U is a postfixed point of φ. Show that

$$n \in U\ \&\ n \text{ is even} \Rightarrow 2n/3 \in U\ .$$

Deduce that all members of U are odd. [Hint: Assume there is an even member of U, so a least even member of U, to derive a contradiction.]

(iii) Deduce that the maximum fixed point of φ is the set of all odd numbers.

(iv) Characterise the prefixed points of φ. What is the minimum fixed point of φ?

□

1.5.2 Continuous functions

Suppose $\varphi : \mathcal{P}ow(\mathcal{S}) \to \mathcal{P}ow(\mathcal{S})$ is monotonic. Then, starting from the empty set we can find a chain of approximations to the least fixed point. As the zeroth approximation take $\emptyset$ and as the first approximation $\varphi(\emptyset)$. Clearly,

$$\emptyset \subseteq \varphi(\emptyset)\ ,$$

and so, by monotonicity of φ,

$$\varphi(\emptyset) \subseteq \varphi^2(\emptyset) \ ,$$

and so on, inductively, to yield an infinite chain

$$\emptyset \subseteq \varphi(\emptyset) \subseteq \varphi^2(\emptyset) \subseteq \cdots \subseteq \varphi^n(\emptyset) \subseteq \varphi^{n+1}(\emptyset) \subseteq \cdots \ .$$

An easy induction establishes that

$$\varphi^n(\emptyset) \subseteq \mu X.\varphi(X) \ ,$$

for all $n \in \omega$, and it might be thought that the least fixed point was equal to the union

$$\bigcup_{n\in\omega} \varphi^n(\emptyset) \ .$$

But this is not true in general, and the union may be strictly below the least fixed point. However, when φ is $\bigcup$-continuous the least fixed point can be obtained in this simple way.

Definition: Let $\varphi : \mathcal{P}ow(\mathcal{S}) \to \mathcal{P}ow(\mathcal{S})$ be a monotonic function.

Say φ is $\bigcup$-*continuous* iff for all increasing chains

$$X_0 \subseteq X_1 \subseteq \cdots \subseteq X_n \subseteq \cdots$$

in $\mathcal{P}ow(\mathcal{S})$ we have

$$\bigcup_{n\in\omega} \varphi(X_n) = \varphi(\bigcup_{n\in\omega} X_n).$$

Say φ is $\bigcap$-*continuous* iff for all decreasing chains

$$X_0 \supseteq X_1 \supseteq \cdots \supseteq X_n \supseteq \cdots$$

in $\mathcal{P}ow(\mathcal{S})$ we have

$$\bigcap_{n\in\omega} \varphi(X_n) = \varphi(\bigcap_{n\in\omega} X_n).$$

Theorem 1.15 *Let $\varphi : \mathcal{P}ow(\mathcal{S}) \to \mathcal{P}ow(\mathcal{S})$ be a monotonic function.*

If φ be $\bigcup$-continuous, then

$$\mu X.\varphi(X) = \bigcup_{n\in\omega} \varphi^n(\emptyset).$$

If φ be $\bigcap$-continuous, then

$$\nu X.\varphi(X) = \bigcap_{n\in\omega} \varphi^n(\mathcal{S}).$$

Proof: Assume φ is $\bigcup$-continuous. Write

$$\mathit{fix}\ \varphi = \bigcup_{n\in\omega} \varphi^n(\emptyset)\ .$$

Then,

$$\begin{aligned}\varphi(\mathit{fix}\ \varphi) =& \varphi(\bigcup_{n\in\omega} \varphi^n(\emptyset)) \\ =& \bigcup_{n\in\omega} \varphi^{n+1}(\emptyset) \qquad \text{by continuity,} \\ =& (\bigcup_{n\in\omega} \varphi^{n+1}(\emptyset)) \cup \{\emptyset\} \\ =& \bigcup_{n\in\omega} \varphi^n(\emptyset) \\ =& \mathit{fix}\ \varphi\ .\end{aligned}$$

Thus $\mathit{fix}\ \varphi$ is a fixed point. Suppose X is a prefixed point, *i.e.* $\varphi(X) \subseteq X$. Certainly $\emptyset \subseteq X$. By monotonicity $\varphi(\emptyset) \subseteq \varphi(X)$. But X is prefixed point, so $\varphi(\emptyset) \subseteq X$, and by induction $\varphi^n(\emptyset) \subseteq X$. Thus, $\mathit{fix}\ \varphi = \bigcup_{n\in\omega} \varphi^n(\emptyset) \subseteq X$.

As fixed points are certainly prefixed points, $\mathit{fix}\ \varphi$ is the least fixed point $\mu X.\varphi(X)$.

Analogously, we prove that the characterisation of maximum fixed points of $\bigcap$-continuous functions. $\square$

Exercise 1.16 Show that if a set of rules R is finitary, in each rule X/y the set of premises X is finite, then, the function φ_R is $\bigcup$-continuous.

Exhibit a set of rules (necessarily not finitary) such that φ_R is not $\bigcup$-continuous.

1.5.3 Fixed points in finite powersets

In the case where $\mathcal{S}$ is a finite set, any increasing chain

$$X_0 \subseteq X_1 \subseteq \cdots \subseteq X_n \subseteq \cdots$$

or any decreasing chain

$$X_0 \supseteq X_1 \supseteq \cdots \supseteq X_n \supseteq \cdots$$

in $\mathcal{P}ow(\mathcal{S})$ must be stationary, *i.e.* eventually constant; the number of strict increases/decreases along a chain can be at most the size of $\mathcal{S}$.

Consequently, when $\mathcal{S}$ is finite, any monotonic function $\varphi : \mathcal{P}ow(\mathcal{S}) \to \mathcal{P}ow(\mathcal{S})$ must be both $\bigcup$- and $\bigcap$-continuous.

Not only do we inherit from continuity the characterisations of least and greatest fixed points as limits of chains of approximations, but moreover we know, when the set $\mathcal{S}$ has size k, that we reach the fixed points by the k-th approximation.

Proposition 1.17 *Let $\mathcal{S}$ be a finite set of size k and $\varphi : \mathcal{P}ow(\mathcal{S}) \to \mathcal{P}ow(\mathcal{S})$ a monotonic function. Then,*

$$\begin{array}{lclcl} \mu X.\varphi(X) & = & \bigcup_{n\in w} \varphi^n(\emptyset) & = & \varphi^k(\emptyset) \\ \nu X.\varphi(X) & = & \bigcap_{n\in w} \varphi^n(\mathcal{S}) & = & \varphi^k(\mathcal{S}) \ . \end{array}$$

Chapter 2

Nondeterministic and parallel commands

This chapter is an introduction to nondeterministic and parallel (or concurrent) programs and systems and their semantics. It introduces communication via shared variables and Dijkstra's language of guarded commands and paves the way for languages of communicating processes in the next chapter.

2.1 Introduction

A simple way to introduce some basic issues in parallel programming languages is to extend the simple imperative language of while-programs by an operation of parallel composition.

$$c ::= \mathbf{skip} \mid X := a \mid c_0; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid c_0 \parallel c_1$$

where a ranges over arithmetic expressions, and b over boolean expressions.

For commands c_0, c_1 their parallel composition $c_0 \parallel c_1$ executes like c_0 and c_1 together, with no particular preference being given to either one. What happens, if, for instance, both c_0 and c_1 are in a position to assign to the same variable? One (and by that it is meant either one) will carry out its assignment, possibly followed by the other. It's plain that the assignment carried out by one can affect the state acted on later by the other. This means we cannot hope to accurately model the execution of commands in parallel using a relation between command configurations and final states. We must instead use a relation representing single uninterruptible steps in the execution relation and so allow for one command affecting the state of another with which it is set in parallel.

There is a choice as to what is regarded as a single uninterruptible step. This is determined by the rules written down for the execution of commands and, in turn, on the evaluation of expressions. But assuming that the evaluation rules have been done we can explain the execution of parallel commands by the following rules. (The set of *states* Σ consists of functions σ from locations to numbers.)

$$\frac{\langle c_0, \sigma \rangle \to_1 \sigma'}{\langle c_0 \parallel c_1, \sigma \rangle \to_1 \langle c_1, \sigma' \rangle} \qquad \frac{\langle c_0, \sigma \rangle \to_1 \langle c_0', \sigma' \rangle}{\langle c_0 \parallel c_1, \sigma \rangle \to_1 \langle c_0' \parallel c_1, \sigma' \rangle}$$

$$\frac{\langle c_1, \sigma \rangle \to_1 \sigma'}{\langle c_0 \parallel c_1, \sigma \rangle \to_1 \langle c_0, \sigma' \rangle} \qquad \frac{\langle c_1, \sigma \rangle \to_1 \langle c_1', \sigma' \rangle}{\langle c_0 \parallel c_1, \sigma \rangle \to_1 \langle c_0 \parallel c_1', \sigma' \rangle}$$

Look at the first two rules. They show how a single step in the execution of a command c_0 gives rise to a single step in the execution of $c_0 \parallel c_1$—these are two rules corresponding to the single step in the execution of c_0 completing the execution of c_0 or not. There are symmetric rules for the right-hand-side component of a parallel composition. If the two component commands c_0 and c_1 of a parallel composition have locations in common they are likely to influence each others' execution. They can be thought of as communicating by shared locations. Our parallel composition gives an example of what is often called *communication by shared variables.*

The symmetry in the rules for parallel composition introduces an unpredictability into the behaviour of commands. Consider for example the execution of the program $(X := 0 \parallel X := 1)$ from the initial state. This will terminate but with what value at X? More generally a program of the form

$$(X := 0 \parallel X := 1); \ \mathbf{if} \ \ X = 0 \ \ \mathbf{then} \ \ c_0 \ \ \mathbf{else} \ \ c_1$$

will execute either as c_0 or c_1, and we don't know which.

This unpredictability is called *nondeterminism.* The programs we have used to illustrate nondeterminism are artificial, perhaps giving the impression that it can be avoided. However it is a fact of life. People and computer systems do work in parallel leading to examples of nondeterministic behaviour, not so far removed from the silly programs we've just seen.

We note that an understanding of parallelism requires an understanding of nondeterminism, and that the interruptability of parallel commands means that we can't model a parallel command simply as a function from configurations to sets of possible end states. The interruptability of parallel commands also complicates considerably the Hoare logic for parallel commands.

Exercise 2.1 Complete the rules for the execution of parallel commands.

2.2 Guarded commands

Paradoxically a disciplined use of nondeterminism can lead to a more straightforward presentation of algorithms. This is because the achievement of a goal may not depend on which of several tasks is performed. In everyday life we might instruct someone to either do this or that and not care which. Dijkstra's language of guarded commands uses a nondeterministic construction to help free the programmer from overspecifying a method of solution. Dijkstra's language has arithmetic and boolean expressions $a \in \mathbf{Aexp}$ and $b \in \mathbf{Bexp}$ as well as two new syntactic sets that of commands (ranged over by c) and guarded commands (ranged over by gc). Their abstract syntax is given by these rules:

$$c ::= \mathbf{skip} \mid \mathbf{abort} \mid X := a \mid c_0; c_1 \mid \mathbf{if}\ gc\ \mathbf{fi} \mid \ \mathbf{do}\ gc\ \mathbf{od}$$

$$gc ::= b \to c \mid gc_0 [\!] gc_1$$

The constructor used to form guarded commands $gc_0 [\!] gc_1$ is called *alternative* (or "fatbar"). The guarded command typically has the form

$$(b_1 \to c_1) [\!] \ldots [\!] (b_n \to c_n).$$

In this context the boolean expressions are called *guards* – the execution of the command body c_i depends on the corresponding guard b_i evaluating to true. If no guard evaluates to true at a state the guarded command is said to *fail*, in which case the guarded command does not yield a final state. Otherwise the guarded command executes nondeterministically as one of the commands c_i whose associated guard b_i evaluates to true. The command syntax includes **skip**, a command which leaves the state unchanged, assignment and sequential composition. The new command **abort** does not yield a final state from any initial state. The command **if** gc **fi** executes as the guarded command gc, if gc does not fail, and otherwise acts like **abort**. The command **do** gc **od** executes repeatedly as the guarded command gc, while gc continues not to fail, and terminates when gc fails; it acts like **skip** if the guarded command fails initially.

We now capture these informal explanations in rules for the execution of commands and guarded commands. We assume evaluation relations for **Aexp** and **Bexp**. With an eye to the future section on an extension of the language to handle parallelism we describe one step in the execution of commands and guarded commands. A command configuration has the form $\langle c, \sigma \rangle$ or σ for commands c and states σ.

Initial configurations for guarded commands are pairs $\langle gc, \sigma \rangle$, for guarded commands gc and states σ, as is to be expected, but one step in their execution can lead to a command configuration or to a new kind of configuration called **fail**. Here are the rules for execution:

Rules for commands:

$$\langle \mathbf{skip}, \sigma\rangle \to \sigma$$

$$\frac{\langle a, \sigma\rangle \to n}{\langle X := a, \sigma\rangle \to \sigma[n/X]}$$

$$\frac{\langle c_0, \sigma\rangle \to \sigma'}{\langle c_0; c_1, \sigma\rangle \to \langle c_1, \sigma'\rangle} \qquad \frac{\langle c_0, \sigma\rangle \to \langle c_0', \sigma'\rangle}{\langle c_0; c_1, \sigma\rangle \to \langle c_0'; c_1, \sigma'\rangle}$$

$$\frac{\langle gc, \sigma\rangle \to \langle c, \sigma'\rangle}{\langle \mathbf{if}\ gc\ \mathbf{fi}, \sigma\rangle \to \langle c, \sigma'\rangle}$$

$$\frac{\langle gc, \sigma\rangle \to \mathbf{fail}}{\langle\ \mathbf{do}\ gc\ \mathbf{od}, \sigma\rangle \to \sigma} \qquad \frac{\langle gc, \sigma\rangle \to \langle c, \sigma'\rangle}{\langle\ \mathbf{do}\ gc\ \mathbf{od}, \sigma\rangle \to \langle c;\ \mathbf{do}\ gc\ \mathbf{od}, \sigma'\rangle}$$

Rules for guarded commands:

$$\frac{\langle b, \sigma\rangle \to \mathbf{true}}{\langle b \to c, \sigma\rangle \to \langle c, \sigma\rangle}$$

$$\frac{\langle gc_0, \sigma\rangle \to \langle c, \sigma'\rangle}{\langle gc_0 [\!] gc_1, \sigma\rangle \to \langle c, \sigma'\rangle} \qquad \frac{\langle gc_1, \sigma\rangle \to \langle c, \sigma'\rangle}{\langle gc_0 [\!] gc_1, \sigma\rangle \to \langle c, \sigma'\rangle}$$

$$\frac{\langle b, \sigma\rangle \to \mathbf{false}}{\langle b \to c, \sigma\rangle \to \mathbf{fail}} \qquad \frac{\langle gc_0, \sigma\rangle \to \mathbf{fail}\ \langle gc_1, \sigma\rangle \to \mathbf{fail}}{\langle gc_0 [\!] gc_1, \sigma\rangle \to \mathbf{fail}}$$

The rule for alternatives $gc_0 [\!] gc_1$ introduces nondeterminism—such a guarded command can execute like gc_0 or like gc_1. Notice the absence of rules for **abort** and for commands **if** gc **fi** in the case where the guarded command gc fails. In such situations the commands do not execute to produce a final state. Another possibility, not straying too far from Dijkstra's intentions in [4], would be to introduce a new command configuration **abortion** to make this improper termination explicit.[1]

As an example, here is a command which assigns the maximum value of two

[1]The reader may find one thing curious. As the syntax stands there is an unnecessary generality in the rules. From the rules for guarded commands it can be seen that in transitions $\langle gc, \sigma\rangle \to \langle c, \sigma'\rangle$ which can be derived the state is unchanged, *i.e.* $\sigma = \sigma'$. And thus in all rules whose premises are a transition $\langle gc, \sigma\rangle \to \langle c, \sigma'\rangle$ we could replace σ' by σ. Of course we lose nothing by this generality, but more importantly, the extra generality will be needed when later we extend the set of guards to allow them to have side effects.

locations X and Y to a location MAX:

$$\begin{array}{l} \textbf{if} \\ X \geq Y \to MAX := X \\ [\!] \\ Y \geq X \to MAX := Y \\ \textbf{fi} \end{array}$$

The symmetry between X and Y would be lost in a more traditional imperative program.

Euclid's algorithm for the greatest common divisor of two numbers is particularly striking in the language of guarded commands:

$$\begin{array}{l} \textbf{do} \\ X > Y \to X := X - Y \\ [\!] \\ Y > X \to Y := Y - X \\ \textbf{od} \end{array}$$

Compare this with the more clumsy program that would result through use of a conditional in language without $[\!]$, a clumsiness which is due to the asymmetry between the two branches of a conditional. See Dijkstra's book [4] for more examples of programs in his language of guarded commands.

Exercise 2.2 Explain informally why Euclid's algorithm terminates. $\square$

Exercise 2.3 Give an operational semantics for the language of guarded commands but where the rules determine transitions of the form $\langle c, \sigma \rangle \to \sigma'$ and $\langle gc, \sigma \rangle \to \sigma'$ between configurations and final states. $\square$

Exercise 2.4 Explain why this program terminates:

$$\textbf{do } (2|X \to X := (3 \times X)/2) [\!] (3|X \to X := (5 \times X)/3) \textbf{ od}$$

where *e.g.* $3|X$ means 3 divides X, and $(5 \times X)/3$ means $5 \times X$ divided by 3. $\square$

Exercise 2.5 A partial correctness assertion $\{A\}c\{B\}$, where c is a command or guarded command and A and B are assertions about states, is said to be valid if for any state at which A is true the execution of c, if it terminates, does so in a final state at which B is true. Write down sound proof rules for the partial correctness assertions of Dijktra's language. $\square$

Exercise 2.6 * Let the syntax of regular commands c be given as follows:

$$c := \textbf{skip} \mid X := e \mid b? \mid c; c \mid c + c \mid c^*$$

where X ranges over a set of locations, e is an integer expression and b is a boolean expression. States σ are taken to be functions from the set of locations to integers. It is assumed that the meaning of integer and boolean expressions are specified by semantic functions so $I[\![e]\!]\sigma$ is the integer which integer expression e evaluates to in state σ and $B[\![b]\!]\sigma$ is the boolean value given by b in state σ. The meaning of a regular command c is given by a relation of the form

$$\langle c, \sigma \rangle \to \sigma'$$

which expresses that the execution of c in state σ can lead to final state σ'. The relation is determined by the following rules:

$$\langle \mathbf{skip}, \sigma \rangle \to \sigma \qquad \frac{I[\![e]\!]\sigma = n}{\langle X := e, \sigma \rangle \to \sigma[n/X]}$$

$$\frac{B[\![b]\!]\sigma = true}{\langle b?, \sigma \rangle \to \sigma} \qquad \frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle c_0, \sigma \rangle \to \sigma'}{\langle c_0 + c_1, \sigma \rangle \to \sigma'} \qquad \frac{\langle c_1, \sigma \rangle \to \sigma'}{\langle c_0 + c_1, \sigma \rangle \to \sigma'}$$

$$\langle c^*, \sigma \rangle \to \sigma \qquad \frac{\langle c, \sigma \rangle \to \sigma'' \quad \langle c^*, \sigma'' \rangle \to \sigma'}{\langle c^*, \sigma \rangle \to \sigma'}$$

(i) Write down a regular command which has the same effect as the while loop

$$\mathbf{while}\ b\ \mathbf{do}\ c,$$

where b is a boolean expression and c is a regular command. Your command C should have the same effect as the while loop in the sense that

$$\langle C, \sigma \rangle \to \sigma' \ \text{ iff } \ \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma'.$$

(This assumes the obvious rules for while loops.)
(ii) For two regular commands c_0 and c_1 write $c_0 = c_1$ when $\langle c_0, \sigma \rangle \to \sigma'$ iff $\langle c_1, \sigma \rangle \to \sigma'$ for all states σ and σ'. Prove from the rules that

$$c^* = skip + c; c^*$$

for any regular command c.
(iii) Write down a denotational semantics of regular commands; the denotation of a regular command c should equal the relation

$$\{(\sigma, \sigma') | \langle c, \sigma \rangle \to \sigma'\}.$$

Describe briefly the strategy you would use to prove that this is indeed true of your semantics.
(iv) Suggest proof rules for partial correctness assertions of regular commands of the form $b?$, $c_0 + c_1$ and c^*. $\square$

Chapter 3

Communicating processes

This chapter introduces programming languages where communication is solely through the synchronised exchange of values. The first language, building on Dijkstra's guarded commands, is closely related to Occam and Hoare's CSP (Communicating Sequential Processes). The remainder of the chapter concentrates on Milner's CCS (Calculus of Communicating Systems), and shows how CCS with value passing can be understood in terms of a more basic, simple language, Pure CCS.

3.1 Synchronous communication

In the latter half of the seventies Hoare and Milner independently suggested the same novel communication primitive. It was clear that systems of processors, each with its own store, would become increasingly important. A communication primitive was sought which was independent of the medium used to communicate, the idea being that the medium, whether it be shared locations or something else, could itself be modelled as a process. Hoare and Milner settled on atomic actions of synchronisation, with the possible exchange of values, as the central primitive of communication.

Their formulations are slightly different. Here we will assume that a process communicates with other processes via channels. We will allow channels to be hidden so that communication along a particular channel can be made local to two or more processes. A process may be prepared to input or output at a channel. However it can only succeed in doing so if there is a companion process in its environment which performs the complementary action of output or input. There is no automatic buffering; an input or output communication is delayed until the other process is ready with the corresponding output or input. When successful the value output is then copied from the outputting to the inputting process.

We now present the syntax of a language of communicating processes. In addition to a set of locations $X \in \mathbf{Loc}$, boolean expressions $b \in \mathbf{Bexp}$ and arithmetic expressions $a \in \mathbf{Aexp}$, we assume:

Channel names	$\alpha, \beta, \gamma, \ldots \in \mathbf{Chan}$	
Input expressions	$\alpha?X$	where $X \in \mathbf{Loc}$
Output expressions	$\alpha!a$	where $a \in \mathbf{Aexp}$

Commands:

$$c \quad ::= \quad \mathbf{skip} \mid \mathbf{abort} \mid X := a \mid \alpha?X \mid \alpha!a \mid c_0; c_1 \mid \mathbf{if}\ gc\ \mathbf{fi} \mid\ \mathbf{do}\ gc\ \mathbf{od} \mid c_0 \parallel c_1 \mid c \setminus \alpha$$

Guarded commands:

$$gc \quad ::= \quad b \to c \mid b \wedge \alpha?X \to c \mid b \wedge \alpha!a \to c \mid gc_0 [\!] gc_1$$

Not all commands and guarded commands are well-formed. A *parallel composition* $c_0 \parallel c_1$ is only well-formed in case the commands c_0 and c_1 do not contain a common location. In general a command is well-formed if all its subcommands of the form $c_0 \parallel c_1$ are well-formed. A *restriction* $c \setminus \alpha$ hides the channel α, so that only communications internal to c can occur on it. [1]

How are we to formalise the intended behaviour of this language of communicating processes? As earlier, states will be functions from locations to the values they contain, and a command configuration will have the form $\langle c, \sigma \rangle$ or σ for a command c and state σ. We will try to formalise the idea of one step in the execution. Consider a particular command configuration of the form

$$\langle \alpha?X; c, \sigma \rangle.$$

This represents a command which is first prepared to receive a synchronised communication of a value for X along the channel α. Whether it does or not is, of course, contingent on whether or not the command is in parallel with another prepared to do a complementary action of outputting a value to the channel α. Its semantics should express this contingency on the environment. This we do in a way familiar from automata theory. We label the transitions. For the set of labels we take

$$\{\alpha?n \mid \alpha \in \mathbf{Chan}\ \&\ n \in \mathbf{Num}\} \cup \{\alpha!n \mid \alpha \in \mathbf{Chan}\ \&\ n \in \mathbf{Num}\}$$

Now, in particular, we expect our semantics to yield the labelled transition

$$\langle \alpha?X; c_0, \sigma \rangle \xrightarrow{\alpha?n} \langle c_0, \sigma[n/X] \rangle.$$

[1] In recent treatments of process algebra one often sees *new* $\alpha.c$, or $\nu\alpha.c$, instead of the restriction $c\backslash\alpha$. In *new* $\alpha.c$ the "new" operation is understood as a binder, binding α, treated as a variable, to a new, private channel name. Because the channel is private it cannot participate in any communication with the outside world, so *new* $\alpha.c$ has the same effect as restricting the channel α away. (In a more liberal regime where channel names can also be passed as values, as in the Pi-Calculus, the private name might be communicated, so allowing future communication along that channel; then a process *new* $\alpha.c$ may well behave differently than simple restriction.)

This expresses the fact that the command $\alpha?X; c_0$ can receive a value n at the channel α and store it in location X, and so modify the state. The labels of the form $\alpha!n$ represent the ability to output a value n at channel α. We expect the transition

$$\langle \alpha!e; c_1, \sigma\rangle \xrightarrow{\alpha!n} \langle c_1, \sigma\rangle$$

provided $\langle e, \sigma\rangle \to n$. Once we have these we would expect a possibility of communication when the two commands are set in parallel:

$$\langle(\alpha?X; c_0) \parallel (\alpha!e; c_1), \sigma\rangle \to \langle c_0 \parallel c_1, \sigma[n/X]\rangle$$

This time we don't label the transition because the communication capability of the two commands has been used up through an internal communication, with no contingency on the environment. We expect other transitions too. After all, there may be other processes in the environment prepared to send and receive values via the channel α. So as to not exclude those possibilities we had better also include transitions

$$\langle(\alpha?X; c_0) \parallel (\alpha!e; c_1), \sigma\rangle \xrightarrow{\alpha?n} \langle c_0 \parallel (\alpha!e; c_1), \sigma[n/X]\rangle$$

and

$$\langle(\alpha?X; c_0) \parallel (\alpha!e; c_1), \sigma\rangle \xrightarrow{\alpha!n} \langle(\alpha?X; c_0) \parallel c_1, \sigma[n/X]\rangle.$$

The former captures the possibility that the first component receives a value from the environment and not from the second component. In the latter the second component sends a value received by the environment, not by the first component.

Now we present the full semantics systematically using rules. We assume given the form of arithmetic and boolean expressions and their evaluation rules.

Guarded commands will be treated in a similar way to before, but allowing for communication in the guards. As earlier guarded commands can sometimes fail at a state.

To control the number of rules we shall adopt some conventions. To treat both labelled and unlabelled transitions in a uniform manner we shall use λ to range over labels like $\alpha?n$ and $\alpha!n$ as well as the empty label. The other convention aims to treat both kinds of command configurations $\langle c, \sigma\rangle$ and σ in the same way. We regard the configuration σ as configuration $\langle *, \sigma\rangle$ where $*$ is thought of as the *empty command*. As such $*$ satisfies the laws

$$*; c \;\equiv\; c; * \;\equiv\; * \parallel c \;\equiv\; c \parallel * \;\equiv\; c \quad \text{and} \quad *; * \;\equiv\; * \parallel * \;\equiv\; (* \setminus \alpha) \;\equiv\; *$$

which express, for instance, that $* \parallel c$ stands for the piece of syntax c. (Here and elsewhere we use $\equiv$ to mean equality of syntax.)

Rules for commands

$$\langle \mathbf{skip}, \sigma\rangle \to \sigma \qquad \frac{\langle a, \sigma\rangle \to n}{\langle X := a, \sigma\rangle \to \sigma[n/X]}$$

$$\langle \alpha?X, \sigma\rangle \xrightarrow{\alpha?n} \sigma[n/X] \qquad \frac{\langle a, \sigma\rangle \to n}{\langle \alpha!a, \sigma\rangle \xrightarrow{\alpha!n} \sigma}$$

$$\frac{\langle c_0, \sigma\rangle \xrightarrow{\lambda} \langle c_0', \sigma'\rangle}{\langle c_0; c_1, \sigma\rangle \xrightarrow{\lambda} \langle c_0'; c_1, \sigma'\rangle}$$

$$\frac{\langle gc, \sigma\rangle \xrightarrow{\lambda} \langle c, \sigma'\rangle}{\langle \mathbf{if}\ gc\ \mathbf{fi}, \sigma\rangle \xrightarrow{\lambda} \langle c, \sigma'\rangle}$$

$$\frac{\langle gc, \sigma\rangle \xrightarrow{\lambda} \langle c, \sigma'\rangle}{\langle\ \mathbf{do}\ gc\ \mathbf{od}, \sigma\rangle \xrightarrow{\lambda} \langle c;\ \mathbf{do}\ gc\ \mathbf{od}, \sigma'\rangle} \qquad \frac{\langle gc, \sigma\rangle \to \mathbf{fail}}{\langle\ \mathbf{do}\ gc\ \mathbf{od}, \sigma\rangle \to \sigma}$$

$$\frac{\langle c_0, \sigma\rangle \xrightarrow{\lambda} \langle c_0', \sigma'\rangle}{\langle c_0 \parallel c_1, \sigma\rangle \xrightarrow{\lambda} \langle c_0' \parallel c_1, \sigma'\rangle} \qquad \frac{\langle c_1, \sigma\rangle \xrightarrow{\lambda} \langle c_1', \sigma'\rangle}{\langle c_0 \parallel c_1, \sigma\rangle \xrightarrow{\lambda} \langle c_0 \parallel c_1', \sigma'\rangle}$$

$$\frac{\langle c_0, \sigma\rangle \xrightarrow{\alpha?n} \langle c_0', \sigma'\rangle \quad \langle c_1, \sigma\rangle \xrightarrow{\alpha!n} \langle c_1', \sigma\rangle}{\langle c_0 \parallel c_1, \sigma\rangle \to \langle c_0' \parallel c_1', \sigma'\rangle} \qquad \frac{\langle c_0, \sigma\rangle \xrightarrow{\alpha!n} \langle c_0', \sigma\rangle \quad \langle c_1, \sigma\rangle \xrightarrow{\alpha?n} \langle c_1', \sigma'\rangle}{\langle c_0 \parallel c_1, \sigma\rangle \to \langle c_0' \parallel c_1', \sigma'\rangle}$$

$$\frac{\langle c, \sigma\rangle \xrightarrow{\lambda} \langle c', \sigma'\rangle}{\langle c \setminus \alpha, \sigma\rangle \xrightarrow{\lambda} \langle c' \setminus \alpha, \sigma'\rangle} \text{ provided neither } \lambda \equiv \alpha?n \text{ nor } \lambda \equiv \alpha!n$$

Rules for guarded commands

$$\frac{\langle b,\sigma\rangle \to \mathbf{true}}{\langle b \to c,\sigma\rangle \to \langle c,\sigma\rangle} \qquad \frac{\langle b,\sigma\rangle \to \mathbf{false}}{\langle b \to c,\sigma\rangle \to \mathbf{fail}}$$

$$\frac{\langle b,\sigma\rangle \to \mathbf{false}}{\langle b \wedge \alpha?X \to c,\sigma\rangle \to \mathbf{fail}} \qquad \frac{\langle b,\sigma\rangle \to \mathbf{false}}{\langle b \wedge \alpha!a \to c,\sigma\rangle \to \mathbf{fail}}$$

$$\frac{\langle gc_0,\sigma\rangle \to \mathbf{fail} \quad \langle gc_1,\sigma\rangle \to \mathbf{fail}}{\langle gc_0 [\!] gc_1,\sigma\rangle \to \mathbf{fail}}$$

$$\frac{\langle b,\sigma\rangle \to \mathbf{true}}{\langle b \wedge \alpha?X \to c,\sigma\rangle \xrightarrow{\alpha?n} \langle c,\sigma[n/X]\rangle} \qquad \frac{\langle b,\sigma\rangle \to \mathbf{true} \quad \langle a,\sigma\rangle \to n}{\langle b \wedge \alpha!a \to c,\sigma\rangle \xrightarrow{\alpha!n} \langle c,\sigma\rangle}$$

$$\frac{\langle gc_0,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle}{\langle gc_0 [\!] gc_1,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle} \qquad \frac{\langle gc_1,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle}{\langle gc_0 [\!] gc_1,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle}$$

Example: The following illustrate various features of the language and the processes it can describe (several more can be found in Hoare's paper [7]):
A process which repeatedly receives a value from the α channel and transmits it on channel β:

$$\mathbf{do}\ (\mathbf{true} \wedge \alpha?X \to \beta!X)\ \mathbf{od}$$

A buffer with capacity 2 receiving on α and transmitting on γ:

$$(\ \mathbf{do}\ (\mathbf{true} \wedge \alpha?X \to \beta!X)\ \mathbf{od} \parallel\ \mathbf{do}\ (\mathbf{true} \wedge \beta?Y \to \gamma!Y)\ \mathbf{od}) \setminus \beta$$

Notice the use of restriction to make the β channel hidden so that all communications along it have to be internal.

One use of the alternative construction is to allow a process to "listen" to two channels simultaneously and read from one should a process in the environment wish to output there; in the case where it can receive values at either channel a nondeterministic choice is made between them:

$$\mathbf{if}\ (\mathbf{true} \wedge \alpha?X \to c_0) [\!] (\mathbf{true} \wedge \beta?Y \to c_1)\ \mathbf{fi}$$

Imagine this process in an environment offering values at the channels. Then it will not deadlock (*i.e.*, reach a state of improper termination) if neither c_0 nor c_1 can. On the other hand, the following process can deadlock:

$$\mathbf{if}\ (\mathbf{true} \to (\alpha?X; c_0)) [\!] (\mathbf{true} \to (\beta?Y; c_1))\ \mathbf{fi}$$

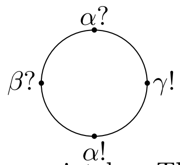
It autonomously chooses between being prepared to receive at the α or β channel. If, for example, it elects the right-hand branch and its environment is only able to output on the α channel there is deadlock. Deadlock can however arise in more subtle ways. The point of Dijkstra's example of the so-called "dining philosophers" is that deadlock can be caused by a complicated chain of circumstances often difficult to forsee (see *e.g.* [7]). □

The programming language we have just considered is closely related to Occam, the programming language of the transputer. It does not include all the features of Occam however, and for instance does not include the prialt operator which behaves like the alternative construction $[\!]$ except for giving priority to the execution of the guarded command on the left. On the other hand, it also allows outputs $\alpha!e$ in guards not allowed in Occam for efficiency reasons. Our language is also but a step away from Hoare's language of Communicating Sequential Processes (CSP) [7]. Essentially the only difference is that in CSP process names are used in place of names for channels; in CSP, $P?X$ is an instruction to receive a value from process P and put it in location X, while $P!5$ means output value 5 to process P.

3.2 Milner's CCS

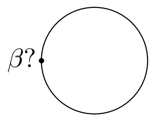
Robin Milner's work on a Calculus of Communicating Systems (CCS) has had an impact on the foundations of the study of parallelism. It is almost true that the language for his calculus, generally called CCS, can be derived by removing the imperative features from the language of the last section, the use of parameterised processes obviating the use of states. In fact, locations can be represented themselves as CCS processes.

A CCS process communicates with its environment via channels connected to its *ports*, in the same manner as we have seen. A process p which is prepared to input at the α and β channels and output at the channels α and γ can be visualised as



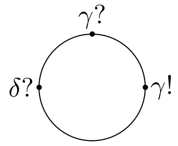
with its ports labelled appropriately. The parallel composition of p with a process q, a process able to input at α and output at β and δ can itself be thought of as a process $p \parallel q$ with ports $\alpha?, \alpha!, \beta?, \beta!, \gamma!, \delta!$.

The operation of restriction hides a specified set of ports. For example restricting away the ports specified by the set of labels $\{\alpha, \gamma\}$ from the process p results in a process $p \backslash \{\alpha, \gamma\}$ only capable of performing inputs from the channel β; it looks like:



Often it is useful to generate several copies of the same process but for a renaming of channels. A relabelling function is a function on channel names.

After relabelling by the function f with $f(\alpha) = \gamma,\ f(\beta) = \delta$ and $f(\gamma) = \gamma$ the process p becomes $p[f]$ with this interface with its environment:



In addition to communications $\alpha?n, \alpha!n$ at channels α we have an extra action τ which can do the duty of the earlier **skip**, as well as standing for actions of internal communication. Because we remove general assignments we will not need the states σ of earlier and can use variables $x, y, \ldots$ in place of locations. To name processes we have process identifiers $P, Q, \ldots$ in our syntax, in particular so we can define their behaviour recursively. Assume a syntax for arithmetic expressions a and boolean expressions b, with variables instead of locations. The syntax of processes p, $p_0, p_1, \ldots$ is:

$$
\begin{array}{rcl}
p & ::= & \mathbf{nil} \mid \\
 & & (\tau \to p) \mid (\alpha!a \to p) \mid (\alpha?x \to p) \mid (b \to p) \\
 & & p_0 + p_1 \mid p_0 \parallel p_1 \mid \\
 & & p \backslash L \mid p[f] \mid \\
 & & P(a_1, \cdots, a_k)
\end{array}
$$

where a and b range over arithmetic and boolean expressions respectively, x is a variable over values, L is a subset of channel names, f is a relabelling function, and P stands for a process with parameters $a_1, \cdots, a_k$—we write simply P when the list of parameters is empty.

Formally at least, $\alpha?x \to p$ is like a lambda abstraction on x, and any occurrences of the variable x in p will be bound by the $\alpha?x$ provided they are not present in subterms of the form $\beta?x \to q$. Variables which are not so bound will be said to be *free*. Process identifiers P are associated with definitions, written as

$$P(x_1, \cdots, x_k) \stackrel{\text{def}}{=} p$$

where all the free variables of p appear in the list $x_1, \cdots, x_k$ of distinct variables. The behaviour of a process will be defined with respect to such definitions for all the process identifiers it contains. Notice that definitions can be recursive in that p may mention P. Indeed there can be simultaneous recursive definitions, for example if

$$
\begin{aligned}
P(x_1, \cdots, x_k) &\stackrel{\text{def}}{=} p \\
Q(y_1, \cdots, y_l) &\stackrel{\text{def}}{=} q
\end{aligned}
$$

where p and q mention both P and Q.

In giving the operational semantics we shall only specify the transitions associated with processes which have no free variables. By making this assumption, we can dispense with the use of environments for variables in the operational semantics, and describe the evaluation of expressions without variables by relations $a \to n$ and $b \to t$. Beyond this, the operational semantics contains few surprises. We use λ to range over actions $\alpha?n, \alpha!n$, and τ.

nil process: has no rules.
Guarded processes:

$$(\tau \to p) \xrightarrow{\tau} p$$

$$\frac{a \to n}{(\alpha!a \to p) \xrightarrow{\alpha!n} p} \qquad \frac{}{(\alpha?x \to p) \xrightarrow{\alpha?n} p[n/x]}$$

$$\frac{b \to \mathbf{true} \quad p \xrightarrow{\lambda} p'}{(b \to p) \xrightarrow{\lambda} p'}$$

(By $p[n/x]$ we mean p with n substituted for the variable x. A more general substitution $p[a_1/x_1, \cdots, a_k/x_k]$, stands for a process term p in which arithmetic expressions a_i have replaced variables x_i.)
Sum:

$$\frac{p_0 \xrightarrow{\lambda} p_0'}{p_0 + p_1 \xrightarrow{\lambda} p_0'} \qquad \frac{p_1 \xrightarrow{\lambda} p_1'}{p_0 + p_1 \xrightarrow{\lambda} p_1'}$$

Composition:

$$\frac{p_0 \xrightarrow{\lambda} p_0'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0' \parallel p_1} \qquad \frac{p_0 \xrightarrow{\alpha?n} p_0' \quad p_1 \xrightarrow{\alpha!n} p_1'}{p_0 \parallel p_1 \xrightarrow{\tau} p_0' \parallel p_1'}$$

$$\frac{p_1 \xrightarrow{\lambda} p_1'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0 \parallel p_1'} \qquad \frac{p_0 \xrightarrow{\alpha!n} p_0' \quad p_1 \xrightarrow{\alpha?n} p_1'}{p_0 \parallel p_1 \xrightarrow{\tau} p_0' \parallel p_1'}$$

Restriction:

$$\frac{p \xrightarrow{\lambda} p'}{p \backslash L \xrightarrow{\lambda} p' \backslash L},$$

where if $\lambda \equiv \alpha?n$ or $\lambda \equiv \alpha!n$ then $\alpha \notin L$
Relabelling:

$$\frac{p \xrightarrow{\lambda} p'}{p[f] \xrightarrow{f(\lambda)} p'[f]}$$

Identifiers:

$$\frac{p[a_1/x_1, \cdots, a_k/x_k] \xrightarrow{\lambda} p'}{P(a_1, \cdots, a_k) \xrightarrow{\lambda} p'}$$

where $P(x_1, \cdots, x_k) \stackrel{\text{def}}{=} p$.

We expand on our claim that it is sufficient to consider processes without free variables and so dispense with environments in the operational semantics. Consider the process

$$(\alpha?x \to (\alpha!x \to \mathbf{nil})).$$

It receives a value n and outputs it at the channel α, as can be derived from the rules. From the rules we obtain directly that

$$(\alpha?x \to (\alpha!x \to \ \mathbf{nil})) \xrightarrow{\alpha?n} (\alpha!x \to \ \mathbf{nil})[n/x]$$

which is

$$(\alpha?x \to (\alpha!x \to \ \mathbf{nil})) \xrightarrow{\alpha?n} (\alpha!n \to \ \mathbf{nil}).$$

Then

$$(\alpha!n \to \ \mathbf{nil}) \xrightarrow{\alpha!n} \ \mathbf{nil}.$$

As can be seen here, when it comes to deriving the transitions of the subprocesses $(\alpha!x \ \to \ \ \mathbf{nil})$ the free variable x has previously been bound to a particular number n.

3.3 Pure CCS

Underlying Milner's work is a more basic calculus, which we will call *pure* CCS. Roughly it comes about by eliminating variables from CCS.

We have assumed that the values communicated during synchronisations are numbers. We could, of course, instead have chosen expressions which denote values of some other type. But for the need to modify expressions, the development would have been the same. Suppose, for the moment, that the values lie in a finite set

$$V = \{n_1, \ldots, n_k\}.$$

Extend CCS to allow input actions $\alpha?n$ where α is a channel and $n \in V$. A process

$$(\alpha?n \to p)$$

first inputs the specific value n from channel α and then proceeds as process p; its behaviour can be described by the rule:

$$\frac{}{(\alpha?n \to p) \xrightarrow{\alpha?n} p}$$

It is not hard to see that under these assumptions the transitions of $\alpha?x \to p$ are the same as those of

$$(\alpha?n_1 \to p[n_1/x]) + \ldots + (\alpha?n_k \to p[n_k/x]).$$

The two processes behave in the same way. In this fashion we can eliminate variables from process terms. Numbers however form an infinite set and when the set of values is infinite, we cannot replace a term $\alpha?x \to p$ by a finite summation. However, this problem is quickly remedied by introducing arbitrary sums into the syntax of processes. For a set of process terms $\{p_i \mid i \in I\}$ indexed by a set I, assume we can form a term

$$\sum_{i \in I} p_i.$$

Then even when the values lie in the infinite set of numbers we can write

$$\sum_{m \in \mathbf{Num}} (\alpha?m \to p[m/x])$$

instead of $(\alpha?x \to p)$.

With the presence of variables x, there has existed a distinction between input and output of values. Once we eliminate variables the distinction is purely formal; input actions are written $\alpha?n$ as compared with $\alpha!n$ for output actions. Indeed in pure CCS the role of values can be subsumed under that of port names. It will be, for example, as if input of value n at port α described by $\alpha?n$ is regarded as a pure synchronisation, without the exchange of any value, at a "port" $\alpha?n$.

In pure CCS actions can carry three kinds of name. There are actions ℓ (corresponding to actions $\alpha?n$ or $\alpha!n$), complementary actions $\bar{\ell}$ (corresponding to $\alpha?n$ being complementary to $\alpha!n$, and *vice versa*) and internal actions τ. With our understanding of complementary actions it is natural to take $\bar{\bar{\ell}}$ to be the same as ℓ, which highlights the symmetry we will now have between input and output.

In the syntax of pure CCS we let λ range over actions of the form ℓ, $\bar{\ell}$ and τ where ℓ belongs to a given set of action labels. Terms for processes $p, p_0, p_1, p_i, \ldots$ of pure CCS take this form:

$$p ::= \mathbf{nil} \mid \lambda.p \mid \sum_{i \in I} p_i \mid (p_0 \parallel p_1) \mid p \backslash L \mid p[f] \mid P$$

The term $\lambda.p$ is simply a more convenient way of writing the guarded process $(\lambda \to p)$. The new general sum $\sum_{i \in I} p_i$ of indexed processes $\{p_i \mid i \in I\}$ has been introduced. We will write $p_0 + p_1$ in the case where $I = \{0, 1\}$. Above, L is to range over subsets of labels. We extend the complementation operation to such a set, taking $\bar{L} =_{\text{def}} \{\bar{\ell} \mid \ell \in L\}$. The symbol f stands for a *relabelling function* on actions. A relabelling function should obey the conditions that $f(\bar{\ell}) = \overline{f(\ell)}$ and $f(\tau) = \tau$. Again, P ranges over identifiers for processes. These are accompanied by definitions, typically of the form

$$P \stackrel{\text{def}}{=} p.$$

As before, they can support recursive and simultaneous recursive definitions.

The rules for the operational semantics of CCS are strikingly simple:

nil has no rules.

Guarded processes:

$$\lambda.p \xrightarrow{\lambda} p$$

Sums:

$$\frac{p_j \xrightarrow{\lambda} q}{\sum_{i \in I} p_i \xrightarrow{\lambda} q}\ j \in I$$

Composition:

$$\frac{p_0 \xrightarrow{\lambda} p_0'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0' \parallel p_1} \qquad \frac{p_1 \xrightarrow{\lambda} p_1'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0 \parallel p_1'}$$

$$\frac{p_0 \xrightarrow{l} p_0' \quad p_1 \xrightarrow{\bar{l}} p_1'}{p_0 \parallel p_1 \xrightarrow{\tau} p_0' \parallel p_1'}$$

Restriction:

$$\frac{p \xrightarrow{\lambda} q}{p \backslash L \xrightarrow{\lambda} q \backslash L}\ \lambda \notin L \cup \bar{L}$$

Relabelling:

$$\frac{p \xrightarrow{\lambda} q}{p[f] \xrightarrow{f(\lambda)} q[f]}$$

Identifiers:

$$\frac{p \xrightarrow{\lambda} q}{P \xrightarrow{\lambda} q}\ \text{ where } P \stackrel{\text{def}}{=} p.$$

We have motivated pure CCS as a basic language for processes into which the other languages we have seen can be translated. We now show, in the form of a table, how closed terms t of CCS can be translated to terms $\widehat{t}$ of pure CCS in a way which preserves their behaviour.

$(\tau \to p)$	$\tau.\widehat{p}$
$(\alpha!a \to p)$	$\overline{\alpha m}.\widehat{p}$ where a denotes the value m
$(\alpha?x \to p)$	$\sum_{m\in\mathbf{Num}}(\alpha m.\widehat{p[m/x]})$
$(b \to p)$	$\widehat{p}$ if b denotes **true** **nil** if b denotes $false$
$p_0 + p_1$	$\widehat{p_0} + \widehat{p_1}$
$p_0 \parallel p_1$	$\widehat{p_0} \parallel \widehat{p_1}$
$p\backslash L$	$\widehat{p}\backslash\{\alpha m \mid \alpha \in L \;\&\; m \in \mathbf{Num}\}$
$P(a_1, \cdots, a_k)$	$P_{m_1,\cdots,m_k}$ where $a_1, \cdots, a_k$ evaluate to $m_1, \cdots, m_k$.

To accompany a definition $P(x_1, \cdots, x_k) \stackrel{\text{def}}{=} p$ in CCS, where p has free variables $x_1, \ldots, x_k$, we have a collection of definitions in the pure calculus

$$P_{m_1,\ldots,m_k} \stackrel{\text{def}}{=} \widehat{p[m_1/x_1, \ldots, m_k/x_k]}$$

indexed by $m_1, \ldots, m_k \in \mathbf{Num}$.

Exercise 3.1 Justify the table above by showing that

$$p \xrightarrow{\lambda} q \text{ iff } \widehat{p} \xrightarrow{\hat{\lambda}} \widehat{q}$$

for closed process terms p, q, where

$$\widehat{\alpha?n} = \alpha n,\ \widehat{\alpha!n} = \overline{\alpha n}.$$

$\square$

Recursive definition:

In applications it is useful to use process identifiers and defining equations. However sometimes in the study of CCS it is more convenient to replace the use of defining equations by the explicit recursive definition of processes. Instead of defining equations such as $P \stackrel{\text{def}}{=} p$, we then use recursive definitions like

$$\mathrm{rec}(P = p).$$

The transitions of these additional terms are given by the rule:

$$\frac{p[\mathrm{rec}(P=p)/P] \xrightarrow{\lambda} q}{\mathrm{rec}(P=p) \xrightarrow{\lambda} q}$$

More generally we can have simultaneous recursive definitions of the form

$$\mathrm{rec}_j(P_i = p_i)_{i \in I}\ , \text{ also written } \mathrm{rec}_j(\vec{P} = \vec{p})\ ,$$

where $j \in I$, some indexing set, which informally stands for the j-th component of the family of processes defined recursively by equations $P_i = p_i$, for $i \in I$.

$$\frac{p_j[\mathrm{rec}(\vec{P}=\vec{p})/\vec{P}] \xrightarrow{\lambda} q}{\mathrm{rec}_j(\vec{P}=\vec{p}) \xrightarrow{\lambda} q}$$

where $\mathrm{rec}(\vec{P}=\vec{p})$ stands for the family $(\mathrm{rec}_k(\vec{P}=\vec{p}))_{k \in I}$.

Exercise 3.2 Use the operational semantics to derive the transition system reachable from the process term $\mathrm{rec}(P = a.b.P)$. $\square$

Exercise 3.3 * Let another language for processes have the following syntax:

$$p := 0 \mid a \mid p;p \mid p+p \mid p \times p \mid P \mid \mathrm{rec}(P=p)$$

where a is an action symbol drawn from a set Σ and P ranges over process variables used in recursively defined processes $\mathrm{rec}(P = p)$. Processes perform sequences of actions, precisely which being specified by an execution relation $p \to s$ between closed process terms and finite sequences $s \in \Sigma^*$; when $p \to s$ the process p can perform the sequence of actions s in a complete execution. Note the sequence s may be the empty sequence ϵ and we use st to represent the concatenation of strings s and t. The execution relation is given by the rules:

$$0 \to \epsilon \qquad a \to a \qquad \frac{p \to s \quad q \to t}{p;q \to st}$$

$$\frac{p \to s}{p+q \to s} \qquad \frac{q \to s}{p+q \to s}$$

$$\frac{p \to s \quad q \to s}{p \times q \to s} \qquad \frac{p[\mathrm{rec}(P=p)/P] \to s}{\mathrm{rec}(P=p) \to s}$$

The notation $p[q/P]$ is used to mean the term resulting from substituting q for all free occurrences of P in p.

Alternatively, we can give a denotational semantics to processes. Taking environments ρ to be functions from variables Var to subsets of sequences $P(\Sigma^*)$

ordered by inclusion, we define:

$$
\begin{aligned}
&[\![0]\!]\rho = \{\epsilon\} \qquad [\![a]\!]\rho = \{a\}\\
&[\![p;q]\!]\rho = \{st \mid s \in [\![p]\!]\rho \text{ and } t \in [\![q]\!]\rho\}\\
&[\![p+q]\!]\rho = [\![p]\!]\rho \cup [\![q]\!]\rho \qquad [\![p \times q]\!]\rho = [\![p]\!]\rho \cap [\![q]\!]\rho\\
&[\![X]\!]\rho = \rho(X)\\
&[\![\mathrm{rec}(P = p)]\!]\rho = \text{the least solution } S \text{ of } S = [\![p]\!]\rho[S/P]
\end{aligned}
$$

The notation $\rho[S/P]$ represents the environment ρ updated to take value S on P.
(i) Assuming a and b are action symbols, write down a closed process term with denotation the language $\{a,b\}^*$ in any environment.
(ii) Prove by structural induction that

$$[\![p[q/P]]\!]\rho = [\![p]\!]\rho[[\![q]\!]\rho/P]$$

for all process terms p and q, with q closed, and environments ρ.
(iii) Hence prove if $p \to s$ then $s \in [\![p]\!]\rho$, where p is a closed process term, $s \in \Sigma^*$ and ρ is any environment. Indicate clearly any induction principles you use. □

Chapter 4

Logics for processes

A specification language, the modal μ-calculus, consisting of a simple modal logic with recursion is motivated. Its relation with the temporal logic CTL is studied. An algorithm is derived for checking whether or not a finite-state process satisfies a specification. This begins a study of model-checking, an increasingly important area in verification.

4.1 A specification language

We turn to methods of reasoning about parallel processes. Historically, the earliest methods followed the line of Hoare logics. Instead Milner's development of CCS has been based on a notion of equivalence between processes with respect to which there are equational laws. These laws are sound in the sense that if any two processes are proved equal using the laws then, indeed, they are equivalent. They are also complete for finite-state processes. This means that if any two finite-state processes are equivalent then they can be proved so using the laws. The equational laws can be seen as constituting an algebra of processes. Different languages for processes and different equivalences lead to different process algebras.

Milner's equivalence is based on a notion of bisimulation between processes. Early on, in exploring the properties of bisimulation, Milner and Hennessy discovered a logical characterisation of this central equivalence. Two processes are bisimilar iff they satisfy precisely the same assertions in a little modal logic, that has come to be called *Hennessy-Milner logic*. The finitary version of this logic has a simple, if perhaps odd-looking syntax:

$$A ::= T \mid F \mid A_0 \wedge A_1 \mid A_0 \vee A_1 \mid \neg A \mid \langle\lambda\rangle A$$

The final assertion $\langle\lambda\rangle A$ is a *modal* assertion (pronounced "diamond λ A") which involves an action name λ. It will be satisfied by any process which can do a λ action to become a process satisfying A. To be specific, we will allow λ to be any action of pure CCS. The other ways of forming assertions are more usual. We use T for true, F for false and build more complicated assertions using conjunctions ($\wedge$), disjunctions ($\vee$) and negations ($\neg$). Thus $(\neg\langle a\rangle T) \wedge (\neg\langle b\rangle T)$ is satisfied by any process which can do neither an a nor a b action. We can define a dual modality in the logic. Take

$$[\lambda]A,$$

(pronounced "box λ A"), to abbreviate $\neg\langle\lambda\rangle\neg A$. Such an assertion is satisfied by any process which cannot do a λ action to become one failing to satisfy A. In other words, $[\lambda]A$ is satisfied by a process which whenever it does a λ action becomes one satisfying A. In particular, this assertion is satisfied by any process which cannot do any λ action at all. Notice $[c]F$ is satisfied by those processes which refuse to do a c action. In writing assertions we will assume that the modal operators $\langle a\rangle$ and $[a]$ bind more strongly than the boolean operations, so *e.g.* $([c]F \wedge [d]F)$ is the same assertion as $(([c]F) \wedge ([d]F))$. As another example,

$$\langle a\rangle\langle b\rangle([c]F \wedge [d]F)$$

is satisfied by any process which can do an a action followed by a b to become one which refuses to do either a c or a d action.

While Hennessy-Milner logic does serve to give a characterisation of bisimulation equivalence (see the exercise ending this section), central to Milner's approach, the finitary language above has obvious shortcomings as a language for writing down specifications of processes; a single assertion can only specify the behaviour of a process to a finite depth, and cannot express, for example, that a process can always perform an action throughout its possibly infinite course of behaviour. To draw out the improvements we can make we consider how one might express particular properties, of undeniable importance in analysing the behaviour of parallel processes.

Let us try to write down an assertion which is true precisely of those processes which can *deadlock*. A process might be said to be capable of deadlock if it can reach a state of improper termination. There are several possible interpretations of what this means, for example, depending on whether "improper termination" refers to the whole or part of the process. For simplicity let's assume the former and make the notion of "improper termination" precise. Assume we can describe those processes which are properly terminated with an assertion $terminal$. A reasonable definition of the characteristic function of this property would be the following, by structural induction on the presentation of pure CCS with explicit recursion:

$$terminal(\mathbf{nil}) = \mathbf{true}$$
$$terminal(\lambda.p) = \mathbf{false}$$
$$terminal(\sum_{i \in I} p_i) = \begin{cases} \mathbf{true} & \text{if } terminal(p_i) = \mathbf{true} \text{ for all } i \in I, \\ \mathbf{false} & \text{otherwise} \end{cases}$$
$$terminal(p_0 \parallel p_1) = terminal(p_0) \ \wedge_T \ terminal(p_1)$$
$$terminal(p \backslash L) = terminal(p)$$
$$terminal(p[f]) = terminal(p)$$
$$terminal(P) = \mathbf{false}$$
$$terminal(\mathrm{rec}(P = p)) = terminal(p)$$

This already highlights one way in which it is sensible to extend our logic, *viz.* by adding *constant* assertions to pick out special processes like the properly terminated ones. Now, reasonably, we can say a process represents an improper termination iff it is not properly terminated and moreover cannot do any actions. How are we to express this as an assertion? Certainly, for the particular action a, the assertion $[a]F$ is true precisely of those processes which cannot do a. Similarly, the assertion

$$[a_1]F \wedge \cdots \wedge [a_k]F$$

is satisfied by those which cannot do any action from the set $\{a_1, \cdots, a_k\}$. But without restricting ourselves to processes whose actions lie within a known finite set, we cannot write down an assertion true just of those processes which can (or cannot) do an arbitrary action. This prompts another extension to the assertions. A new assertion of the form

$$\langle . \rangle A$$

is true of precisely those processes which can do any action to become a process satisfying A. Dually we define the assertion

$$[.]A \equiv_{def} \neg \langle . \rangle \neg A$$

which is true precisely of those processes which become processes satisfying A whenever they perform an action. The assertion $[.]F$ is satisfied by the processes which cannot do any action. Now the property of *immediate deadlock* can be written as

$$Dead \equiv_{def} ([.]F \wedge \neg terminal) \ .$$

The assertion $Dead$ captures the notion of improper termination. A process can deadlock if by performing a sequence of actions it can reach a process satisfying $Dead$. It's tempting to express the possibility of deadlock as an *infinite* disjunction:

$$Dead \vee \langle . \rangle Dead \vee \langle . \rangle \langle . \rangle Dead \vee \langle . \rangle \langle . \rangle \langle . \rangle Dead \vee \cdots \vee (\langle . \rangle \cdots \langle . \rangle Dead) \vee \cdots$$

But, of course, this is not really an assertion because in forming assertions only finite disjunctions are permitted. Because there are processes which deadlock after arbitrarily many steps we cannot hope to reduce this to a finite disjunction, and so a real assertion. We want assertions which we can write down!

We need another primitive in our language of assertions. Rather than introducing extra primitives on an *ad hoc* basis as we encounter further properties we'd like to express, we choose one strong new method of defining assertions powerful enough to define the possibility of deadlock and many other properties. The infinite disjunction is reminiscient of the least upper bounds of chains one sees in characterising least fixed points of continuous functions, and indeed our extension to the language of assertions will be to allow the recursive definition of properties. The possibility of deadlock will be expressed by the least fixed point

$$\mu X.(Dead \vee \langle . \rangle X)$$

which intuitively unwinds to the infinite "assertion"

$$Dead \vee \langle . \rangle (Dead \vee \langle . \rangle (Dead \vee \langle . \rangle (\cdots$$

A little more generally, we can write

$$possibly(B) \equiv_{def} \mu X.(B \vee \langle . \rangle X)$$

true of those processes which can reach a process satisfying B through performing a sequence of actions. Other constructions on properties can be expressed too. We might well be interested in whether or not a process eventually becomes one satisfying assertion B no matter what sequence of actions it performs. This can be expressed by

$$eventually(B) \equiv_{def} \mu X.(B \vee (\langle . \rangle T \wedge [.]X)).$$

As this example indicates, it is not always clear how to capture properties as assertions. Even when we provide the mathematical justification for recursively defined properties in the next section, it will often be a nontrivial task to show that a particular assertion with recursion expresses a desired property. However this can be done once and for all for a batch of useful properties. Because they are all defined using the same recursive mechanism, it is here that the effort in establishing proof methods and tools can be focussed.

In fact, maximum (rather than minimum) fixed points will play the more dominant role in our subsequent work. With negation, one is definable in terms of the other. An assertion defined using maximum fixed points can be thought of as an infinite conjunction. The maximum fixed point $\nu X.(B \wedge [.]X)$ unwinds to

$$B \wedge [.](B \wedge [.](B \wedge [.](B \wedge \cdots$$

and is satisfied by those processes which, no matter what actions they perform, always satisfy B. In a similar way we can express that an assertion B is satisfied all the way along an infinite sequence of computation from a process:

$$\nu X.(B \wedge [.]X) \ .$$

Exercise 4.1 What is expressed by the following assertions?

(i) $\mu X.(\langle a\rangle T \vee [.]X)$

(ii) $\nu Y.(\langle a\rangle T \vee (\langle .\rangle T \wedge [.]Y))$

(Argue informally, by unwinding definitions. Later, will show how to prove that an assertion expresses a property, at least for finite-state processes.) □

4.2 The modal μ–calculus

We now provide the formal treatment of the specification language motivated in the previous Section 4.1. The language is called the *modal μ-calculus* [10].

Let $\mathcal{P}$ denote the set of processes in pure CCS. Assertions determine properties of processes. A property is either true or false of a process and so can be identified with the subset of processes $\mathcal{P}$ which satisfy it. In fact, we will understand assertions simply as a notation for describing subsets of processes. Assertions are built up using:

- *constants*: Any subset of processes $S \subseteq \mathcal{P}$ is regarded as a constant assertion taken to be true of a process it contains and false otherwise. (We can also use finite descriptions of them like *terminal* and *Dead* earlier. In our treatment we will identify such descriptions with the subset of processes satisfying them.)

- *logical connectives*: The special constants T, F stand for true and false respectively. If A and B are assertions then so are $\neg A$ ("not A"), $A \wedge B$ ("A and B"), $A \vee B$ ("A or B")

- *modalities*: If a is an action symbol and A is an assertion then $\langle a\rangle A$ is an assertion. If A is an assertion then so is $\langle .\rangle A$. (The box modalities $[a]A$ and $[.]A$ are abbreviations for $\neg\langle a\rangle\neg A$ and $\neg\langle .\rangle\neg A$, respectively.)

- *maximum fixed points*: If A is an assertion in which the variable X occurs positively (*i.e.* under an even number of negation symbols for every ocurrence) then $\nu X.A$ (the maximum fixed point of A) is an assertion. (The minimum fixed point $\mu X.A$ can be understood as an abbreviation for $\neg\nu X.\neg A[\neg X/X]$.)

In reasoning about assertions we shall often make use of their *size*. Precisely, the size of an assertion is defined by structural induction:

$$\begin{aligned}
&size(S) = size(T) = size(F) = 0 \quad \text{where } S \text{ is a constant}\\
&size(\neg A) = size(\langle a\rangle A) = size(\nu X.A) = 1 + size(A)\\
&size(A \wedge B) = size(A \vee B) = 1 + size(A) + size(B).
\end{aligned}$$

Assertions are a notation for describing subsets of processes. So for example, $A \wedge B$ should be satisfied by precisely those processes which satisfy A and satisfy

B, and thus can be taken to be the intersection $A \cap B$. Let's say what subsets of processes all the assertions stand for. In the following, an assertion on the left stands for the set on the right:

$$
\begin{array}{lcl}
S & = & S \quad \text{where } S \subseteq \mathcal{P} \\
T & = & \mathcal{P} \\
F & = & \emptyset \\
A \wedge B & = & A \cap B \\
A \vee B & = & A \cup B \\
\neg A & = & \mathcal{P} \setminus A \\
\langle a \rangle A & = & \{p \in \mathcal{P} \mid \exists q.p \xrightarrow{a} q \text{ and } q \in A\} \\
\langle . \rangle A & = & \{p \in \mathcal{P} \mid \exists a, q.p \xrightarrow{a} q \text{ and } q \in A\} \\
\nu X.A & = & \bigcup \{S \subseteq \mathcal{P} \mid S \subseteq A[S/X]\}
\end{array}
$$

Note, this is a good definition because the set associated with an assertion is defined in terms of sets associated with assertions of strictly smaller size. Most clauses of the definition are obvious; for example, $\neg A$ should be satisfied by all processes which do not satisfy A, explaining why it is taken to be the complement of A; the modality $\langle a \rangle A$ is satisfied by any process p capable of performing an a–transition leading to a process satisfying A. If X occurs only positively in A, it follows that the function

$$S \longmapsto A[S/X].$$

is monotonic on subsets of $\mathcal{P}$ ordered by $\subseteq$. Tarski's fixed-point Theorem (see the Appendix) characterises the maximum fixed point of this function as

$$\bigcup \{S \subseteq \mathcal{P} \mid S \subseteq A[S/X]\}$$

is the union of all postfixed points of the function $S \mapsto A[S/X]$. Above we see the use of an assertion $A[S/X]$ which has a form similar to A but with each occurrence of X replaced by the subset S of processes.

Proposition 4.2 *The minimum fixed point $\mu X.A$, where*

$$\mu X.A = \bigcap \{S \subseteq \mathcal{P} \mid A[S/X] \subseteq S\} \ ,$$

is equal to $\neg \nu X.\neg A[\neg X/X]$.

Proof: The operation of negation provides a 1-1 correspondence between prefixed points of the function $S \mapsto A[S/X]$ and postfixed points of the function $S \mapsto \neg A[\neg S/X]$:

Write $A(S)$ as an abbreviation for $A[S/X]$. Negation stands for complementation on subsets of processes. Consequently, $U \subseteq V \iff \neg V \subseteq \neg U$ and $\neg(\neg U) = U$, for subsets U, V. Hence,

$$A(S) \subseteq S \text{ iff } \neg S \subseteq \neg A(\neg(\neg S)) \ .$$

Thus the operation of negation gives a 1-1 correspondence between

$$Pre(A) = \{S \mid A(S) \subseteq S\} \ ,$$

the set of prefixed points of $S \mapsto A(S)$, and

$$Post(A) = \{U \mid U \subseteq \neg A(\neg U)\} \ ,$$

the set of postfixed points of $U \mapsto \neg A(\neg U)$. Notice that the 1-1 correspondence reverses the subset relation:

$$S' \subseteq S \ , \text{ where } S, S' \in Pre(A), \text{ iff } \neg S' \supseteq \neg S \ , \text{ where } \neg S, \neg S' \in Post(A) \ .$$

It follows that the least fixed prefixed point of $S \mapsto A(S)$ corresponds to the greatest postfixed point of $U \mapsto \neg A(\neg U)$; in other words, that $\neg \mu X.A = \nu X.\neg A[\neg X/X]$. □

Exercise 4.3 Regarding assertions as sets, show that

$$\begin{array}{ll} \langle a \rangle F = F \ , & \langle a \rangle (A \vee B) = \langle a \rangle A \vee \langle a \rangle B \ , \text{ and} \\ [a]T = T \ , & [a](A \wedge B) = [a] \wedge [a]B \ . \end{array}$$

Show that, although $\langle a \rangle (A \wedge B) \subseteq \langle a \rangle A \wedge \langle a \rangle B$, the converse inclusion need not hold. □

Exercise 4.4 Show $[a]A = \{p \in \mathcal{P} \mid \forall q \in \mathcal{P}.\ p \xrightarrow{a} q \Rightarrow q \in A\}$. By considering *e.g.*a process $\Sigma_{n \in \omega} a.p_n$ where the p_n, $n \in \omega$, are distinct, show that the function $S \mapsto [a]S$ is not continuous with respect to inclusion (it is monotonic). □

We can now specify what it means for a process p to satisfy an assertion A. We define the *satisfaction assertion* $p \models A$ to be **true** if $p \in A$, and **false** otherwise.

We have based the semantics of the modal μ-calculus on a particular transition system, that for pure CCS; the states of the transition system consist of pure CCS terms and form the set $\mathcal{P}$ and its transitions are given by the rules for the operational semantics. It should be clear by inspecting the clauses interpreting assertions of the modal μ-calculus as subsets of $\mathcal{P}$, that the same semantic definitions would make sense with respect to any transition system for which the transition actions match those of the modalities. Any such transition system can be used to interpret the modal μ-calculus. We shall especially concerned with *finite-state* transition systems, those for the set of states is finite. In the transition system for pure CCS, process terms do double duty: they stand for states of the transition system, but they also stand for transition systems themselves, *viz.* the transition system obtained as that forwards reachable from the process term—it is this localised transition system which represents the behaviour of the process. When the states forwards reachable from a process form a finite set we say the process is finite state. Although we shall often present results for finite-state processes, so working with particular transition systems built on pure CCS, it should be born in mind that the general results apply to any finite-state transition system interpreting the modal μ-calculus.

It is possible to check automatically whether or not a finite-state process p satisfies an assertion A. (One of the Concurrency-Workbench/TAV commands checks whether or not a process p satisfies an assertion A; it will not necessarily terminate for infinite-state processes though in principle, given enough time and space, it will for finite-state processes.) To see why this is feasible let p be a *finite-state* process. This means that the set of processes reachable from it

$$\mathcal{P}_p =_{def} \{q \in \mathcal{P} \mid p \xrightarrow{\cdot}^* q\}$$

is finite, where we use $p \xrightarrow{\cdot} q$ to mean $p \xrightarrow{a} q$ for some action a. In deciding whether or not p satisfies an assertion we need only consider properties of the reachable processes $\mathcal{P}_p$. We imitate what we did before but in the transition system based on $\mathcal{P}_p$ instead of $\mathcal{P}$. Again, the definition is by induction on the size of assertions. Define:

$$\begin{array}{lcll}
S\mid_p & = & S \cap \mathcal{P}_p & \text{where } S \subseteq \mathcal{P} \\
T\mid_p & = & \mathcal{P}_p & \\
F\mid_p & = & \emptyset & \\
A \wedge B\mid_p & = & A\mid_p \cap B\mid_p & \\
A \vee B\mid_p & = & A\mid_p \cup A\mid_p & \\
\neg A\mid_p & = & \mathcal{P}_p \setminus (A\mid_p) & \\
\langle a\rangle A\mid_p & = & \{r \in \mathcal{P}_p \mid \exists q \in \mathcal{P}_p . r \xrightarrow{a} q \text{ and } q \in A\mid_p\} & \\
\langle . \rangle A\mid_p & = & \{r \in \mathcal{P}_p \mid \exists a, q \in \mathcal{P}_p . r \xrightarrow{a} q \text{ and } q \in A\mid_p\} & \\
\nu X.A\mid_p & = & \bigcup\{S \subseteq \mathcal{P}_p \mid S \subseteq A[S/X]\mid_p\} &
\end{array}$$

As we would expect there is a simple relationship between the "global" and "local" meanings of assertions, expressed in the following lemma.

Lemma 4.5 *For all assertions A and processes p,*

$$A\mid_p = A \cap \mathcal{P}_p.$$

Proof: We first observe that:

$$A[S/X]|_p = A[S \cap \mathcal{P}_p/X]|_p.$$

This observation is easily shown by induction on the size of assertions A.

A further induction on the size of assertions yields the result. We consider the one slightly awkward case, that of maximum fixed points. We would like to show

$$\nu X.A|_p = (\nu X.A) \cap \mathcal{P}_p$$

assuming the property expressed by the lemma holds inductively for assertion A. Recall

$$\nu X.A = \bigcup\{S \subseteq \mathcal{P} \mid S \subseteq A[S/X]\} \quad \text{and}$$
$$\nu X.A|_p = \bigcup\{S' \subseteq \mathcal{P}_p \mid S' \subseteq A[S'/X]|_p\}.$$

Suppose $S \subseteq \mathcal{P}$ and $S \subseteq A[S/X]$. Then

$$\begin{aligned} S \cap \mathcal{P}_p &\subseteq A[S/X] \cap \mathcal{P}_p \\ &= A[S/X]|_p \quad \text{by induction} \\ &= A[S \cap \mathcal{P}_p/X]|_p \quad \text{by the observation.} \end{aligned}$$

Thus $S \cap \mathcal{P}_p$ is a postfixed point of $S' \mapsto A[S'/X]|_p$, so $S \cap \mathcal{P}_p \subseteq \nu X.A|_p$. Hence $\nu X.A \cap \mathcal{P}_p \subseteq \nu X.A|_p$.

To show the converse, suppose $S' \subseteq \mathcal{P}_p$ and $S' \subseteq A[S'/X]|_p$. Then, by induction, $S' \subseteq A[S'/X] \cap \mathcal{P}_p$. Thus certainly $S' \subseteq A[S'/X]$, making S' a postfixed point of $S \mapsto A[S/X]$ which ensures $S' \subseteq \nu X.A$. It follows that $\nu X.A|_p \subseteq \nu X.A$.

Whence we conclude $\nu X.A|_p = (\nu X.A) \cap \mathcal{P}_p$, as was required. □

One advantage in restricting to $\mathcal{P}_p$ is that, being a finite set of size n say, we know

$$\begin{aligned} \nu X.A\ |_p &= \bigcap_{0 \le i \le n} A^i[T/X]\ |_p \\ &= A^n[T/X] \cap \mathcal{P}_p \end{aligned}$$

where $A^\circ = T$, $A^{i+1} = A[A^i/X]$. This follows from the earlier results in Section 1.5 characterising the maximum fixed point of a $\bigcap$-continuous function on a powerset: The function $S \mapsto A[S/X]|_p$ is monotonic and so continuous on the the *finite* finite powerset $(\mathcal{P}ow(\mathcal{P}_p), \supseteq)$.

In this way maximum fixed points can be eliminated from an assertion A for which we wish to check $p \models A$. Supposing the result had the form $\langle a \rangle B$ we would then check if there was a process q with $p \xrightarrow{a} q$ and $q \models B$. If, on the other hand, it had the form of a conjunction $B \wedge C$ we would check $p \models B$ and $p \models C$. And no matter what the shape of the assertion, once maximum fixed points have been eliminated, we can reduce checking a process satisfies an assertion to checking processes satisfy strictly smaller assertions until ultimately we must settle whether or not processes satisfy constant assertions. Provided the constant assertions represent decidable properties, in this way we will eventually obtain an answer to our original question, whether or not $p \models A$. It is a costly method however; the elimination of maximum fixed points is only afforded through a possible blow-up in the size of the assertion. Nevertheless a similar idea, with clever optimisations, can form the basis of an efficient model-checking method, investigated by Emerson and Lei in [5].

We will soon provide another method, called "local model checking" by Stirling and Walker, which is more sensitive to the structure of the assertion being considered, and does not always involve finding the full, maximum-fixed-point set $\nu X.A\ |_p$.

4.3 CTL and other logics

Many important specification logics can be encoded within the modal μ-calculus. As an illustration we show how to encode CTL ("Computation Tree Logic").

This logic is widely used in model checking and is often introduced as a fragment of the more liberal logic CTL*, a logic obtained by combining certain state assertions and path assertions. A state assertion is similar to those we have seen in that it is either true or false of a state (or process). A path assertion is true or false of a path, where a path is understood to be a maximal sequence of states possible in the run of a process.[1]

CTL-assertions take the form:

$$A := At \mid A_0 \wedge A_1 \mid A_0 \vee A_1 \mid \neg A \mid \mathbf{EX}\ A \mid \mathbf{EG}\ A \mid \mathbf{E}[A_0\ \mathbf{U}\ A_1]$$

where At ranges over constant assertions.

Action names play no direct role in CTL, so we interpret CTL-assertions in a transition system with a single action called simply "$\cdot$". (In particular, we can interpret CTL in the transition system $\mathcal{P}$ using the transition relation $\stackrel{\cdot}{\rightarrow}$.) To each constant assertion is preassigned a set of states at which it is true. A *path* π in the transition system from a state π_0 is a *maximal* sequence of states $(\pi_0, \pi_1, \pi_2, \cdots)$ such that $\pi_i \stackrel{\cdot}{\rightarrow} \pi_{i+1}$ for all i; maximality means the path cannot be extended, so is either infinite or finite and with a final state s_n incapable of performing any action. Notice that the interpretation below involves quantifiers over paths and states in paths. In the broader logic CTL* the modalities **EX** , **EG** and $\mathbf{E}[-\ \mathbf{U}\ -]$ are explained as compound modalities involving a modality on paths (**E**) and modalities on states within a path (**X**, **G** and **U**)—thus the two-letter names for the CTL modalities.

Interpretation of CTL:

- A constant assertion At is associated with a set of states at which it is true, so we take $s \models At$ iff s is amongst those states.
- The boolean operations $A_0 \wedge A_1$, $A_0 \vee A_1$ and $\neg A$ are interpreted literally:
  $s \models A_0 \wedge A_1$ iff $s \models A_0$ and $s \models A_1$;
  $s \models A_0 \vee A_1$ iff $s \models A_0$ or $s \models A_1$;
  $s \models \neg A$ iff it is not the case that $s \models A$.
- $s \models \mathbf{EX}\ A$ iff for some path π from s we have, $\pi_1 \models A$. In other words, there **E**xists a path, starting at state s, whose ne**X**t state satisfies A.
- $s \models \mathbf{EG}\ A$ iff for some path π from s, we have $\pi_i \models A$, for all i. There **E**xists a path along which A holds **G**lobally.
- $s \models \mathbf{E}[A_0\ \mathbf{U}\ A_1]$ iff for some path π from s, there is j such that $\pi_j \models A_1$ and $\pi_i \models A_0$ for all $i < j$. There **E**xists a path on which A_0 **U**ntil A_1—note that A_1 must hold at some point on the path.

[1] Most often CTL* and CTL are interpreted with respect to infinite paths in transition-system models where states are never terminal, *i.e.* can always perform an transition. Maximal paths include such infinite paths but also paths ending in a terminal state. This added generality, more in keeping with the models used here, only requires a slight modification in the usual translation of the CTL-assertion **EG** A into the modal μ-calculus.

We can translate CTL into the modal μ-calculus. Define the translation function Tr by the following structural induction on CTL-assertions:

$Tr(At) = At$, standing for the set of states at which At holds,
$Tr(A_0 \wedge A_1) = Tr(A_0) \wedge Tr(A_1)$, $\quad Tr(A_0 \vee A_1) = Tr(A_0) \vee Tr(A_1)$,
$Tr(\neg A) = \neg Tr(A)$,
$Tr(\mathbf{EX}\ A) \equiv \langle . \rangle Tr(A)$,
$Tr(\mathbf{EG}\ A) \equiv \nu Y. Tr(A) \wedge ([\cdot]F \vee \langle . \rangle Y)$,
$Tr(\mathbf{E}[A\ \mathbf{U}\ B]) \equiv \mu Z. Tr(B) \vee (Tr(A) \wedge \langle . \rangle Z)$.

That the translation is correct hinges on Propositions 4.6, 4.8 below.

Proposition 4.6 *In a finite-state transition system, $s \models \nu Y. A \wedge ([\cdot]F \vee \langle . \rangle Y)$ iff there is some path π from s, such that $\pi_i \models A$, for all i.*

Proof: Let $\varphi(Y) = A \wedge ([\cdot]F \vee \langle . \rangle Y)$ for Y a subset of states. Then, there is a decreasing chain

$$T \supseteq \varphi(T) \supseteq \cdots \supseteq \varphi^n(T) \supseteq \cdots$$

such that

$$\nu Y. A \wedge ([\cdot]F \vee \langle . \rangle Y) = \bigcap_{n \in \omega} \varphi^n(T) \ .$$

Write $s \stackrel{.}{\rightarrow}$ to indicate that s can perform an action, and $s \stackrel{.}{\not\rightarrow}$ that it cannot. We show by induction on $n \geq 1$ that:

For all states s, we have $s \models \varphi^n(T)$ iff

(1) either $\exists m \leq n, s_1, \cdots, s_m$.

$$s = s_1 \stackrel{.}{\rightarrow} \cdots \stackrel{.}{\rightarrow} s_m \stackrel{.}{\not\rightarrow} \text{ and}$$
$$s_1 \models A \text{ and } \cdots \text{ and } s_m \models A$$

(*i.e.*, there is a finite (maximal) path from s of length $\leq n$ along which A always holds),

(2) or $\exists s_1, \cdots, s_n$.

$$s = s_1 \stackrel{.}{\rightarrow} \cdots \stackrel{.}{\rightarrow} s_n \stackrel{.}{\rightarrow} \text{ and}$$
$$s_1 \models A \text{ and } \cdots \text{ and } s_m \models A$$

(*i.e.*, there is a partial path from s of length n along which A always holds).

At the basis of the induction, when $n = 1$, $\varphi(T) = A \wedge ([\cdot]F \vee \langle \cdot \rangle T) = A$ which is satisfied by a state s precisely when (1) or (2) above hold with $n = 1$.

For the induction step:

$$
\begin{aligned}
s \models \varphi^{n+1}(T) \text{ iff } & s \models A \wedge ([\cdot]F \wedge \langle.\rangle\varphi^n(T)) \\
\text{iff } & s \models A \text{ and } (s \models [\cdot]F \text{ or } s \models \langle.\rangle\varphi^n(T)) \\
\text{iff } & s \models A \text{ and } (s \models [\cdot]F \text{ or } \exists s_1.\ s\ \dot{s}_1 \ \text{ and } s_1 \models \varphi^n(T)) \\
\text{iff } & (s \models A \text{ and } s \models [\cdot]F) \text{ or } (s \models A \text{ and } \exists s_1.\ s\ \dot{s}_1 \ \text{ and } s_1 \models \varphi^n(T)) \\
\text{iff } & \text{there is a maximal path, length} \leq n+1, \text{ or} \\
& \text{a partial path, length } n+1, \text{ from } s, \text{ along which } A \text{ holds.}
\end{aligned}
$$

Finally, as the transition system is finite-state, with say k states, the maximum fixed point of φ is $\varphi^k(T)$, so $\varphi(\varphi^k(T)) = \varphi^k(T)$, *i.e.*, $\varphi^{k+1}(T) = \varphi^k(T)$. Thus

$$s \models \nu Y.A \wedge ([\cdot]F \vee \langle.\rangle Y) \text{ iff } s \models \varphi^{k+1}(T) \ .$$

Hence, if there are no finite maximal paths from s along which A always holds, then from the meaning of $\varphi^{k+1}(T)$, there must be states $s_1, \cdots, s_{k+1}$ for which

$$
\begin{aligned}
& s = s_1 \stackrel{\cdot}{\to} \cdots \stackrel{\cdot}{\to} s_{k+1} \text{ and} \\
& s_1 \models A \text{ and } \cdots \text{ and } s_{k+1} \models A \ .
\end{aligned}
$$

But such a partial path must loop, and hence there is an infinite (so maximal) path along which A always holds. □

Exercise 4.7 Prove that the restriction to finite-state transition systems is unnecessary in Proposition 4.6.

(i) Suppose there is some path π from s, such that $\pi_i \models A$, for all i. Show that the set $\{\pi_0, \pi_1, \pi_2, \cdots\}$ is a postfixed point of the function $Y \mapsto A \wedge ([\cdot]F \vee \langle.\rangle Y)$. Deduce $s = \pi_0$ satisfies $\nu Y.A \wedge ([\cdot]F \vee \langle.\rangle Y)$.

(ii) Now show the converse. Suppose that $s \models \nu Y.A \wedge ([\cdot]F \vee \langle.\rangle Y)$. Suppose that there is no finite maximal path from s along which A always holds. By unwinding the recursive assertion, show how to construct by induction an infinite path from s along which A always holds.

□

Proposition 4.8 *In a transition system, $s \models \mu Z.B \vee (A \wedge \langle.\rangle Z)$ iff there is some path π from s, such that $\pi_j \models B$ and $\pi_i \models A$ for all $i < j$.*

Proof: Let $\varphi(Z) = B \vee (A \wedge \langle.\rangle Z)$, for Z a subset of states. The function φ is $\bigcup$-continuous (Exercise!). (In a finite-state transition system, the continuity of φ would be automatic.) So, there is a increasing chain

$$\emptyset \subseteq \varphi(\emptyset) \subseteq \cdots \subseteq \varphi^n(\emptyset) \subseteq \cdots$$

such that

$$\mu Z.B \vee (A \wedge \langle.\rangle Z) = \bigcup_{n \in \omega} \varphi^n(\emptyset) \ .$$

It is sufficient to show by induction on $n \geq 1$ that:

For all states s, we have $s \models \varphi^n(\emptyset)$ iff there are $m \leq n$ and states $s_1, \cdots, s_m$ such that

$$s = s_1 \xrightarrow{\cdot} \cdots \xrightarrow{\cdot} s_m \text{ and}$$
$$s_1 \models A \text{ and } \cdots \text{ and } s_{m-1} \models A \text{ and } s_m \models B \;;$$

in other words, there is a (partial) path from s such that B holds within n steps and along which A holds until B.

At the basis of the induction, $\varphi^1(\emptyset) = B$, so satisfying the induction hypothesis at $n = 1$.

For the induction step:

$$\begin{aligned} s \models \varphi^{n+1}(\emptyset) \text{ iff } & s \models B \vee (A \wedge \langle . \rangle \varphi^n(\emptyset)) \\ \text{iff } & s \models B \text{ or } (s \models A \text{ and } \exists s_1.\ s \xrightarrow{\cdot} s_1 \text{ and } s_1 \models \varphi^n(\emptyset)) \\ \text{iff } & s \models B \text{ or} \\ & (s \models A \text{ and } \exists m \leq n, s_1, \cdots, s_m. \\ & s \xrightarrow{\cdot} s_1 \xrightarrow{\cdot} \cdots s \xrightarrow{\cdot} s_{m-1} \xrightarrow{\cdot} s_m \text{ and} \\ & s_1 \models A \text{ and } \cdots \text{ and } s_{m-1} \models A \text{ and } s_m \models B)\ , \\ & \text{forming a path of length } \leq n+1 \text{for which } A \text{ holds until } B\ . \end{aligned}$$

$\square$

Exercise 4.9 Show the function φ taking Z, a subset of states of a transition system, to the subset $B \vee (A \wedge \langle . \rangle Z)$ is $\bigcup$-continuous. $\square$

The translation of CTL-assertions into the modal μ-calculus is correct:

Proposition 4.10 *For a state s in a finite-state transition system, and CTL-assertion A, $s \models A$ iff $s \models Tr(A)$.*

Proof: By a simple structural induction on CTL-assertions, using Propositions 4.6, 4.8 for the **EG** A and **E**$[A$ **U** $B]$ cases. $\square$

In the remaining exercises of this section we assume the processes are finite-state and consider other properties expressible in the modal μ-calculus.

Exercise 4.11 (i) Let p be a finite-state process. Prove p satisfies $\nu X.(\langle a \rangle X)$ iff p can perform an infinite chain of a-transitions.

What does $\mu X.(\langle a \rangle X)$ mean? Prove it.

In the remainder of this exercise assume the processes under consideration are finite-state (so that (i) is applicable). Recall a process p is finite-state iff the set $\mathcal{P}_p$ is finite, *i.e.* only finitely many processes are reachable from p.

(ii) Prove the assertion $\nu X.(A \wedge [.]X)$ is satisfied by those processes p which always satisfy an assertion A, *i.e.* q satisfies A, for all $q \in \mathcal{P}_p$.

(iii) How would you express in the modal μ-calculus the property true of precisely those processes which eventually arrive at a state satisfying an assertion A? Prove your claim.
(See the earlier text or Exercise 4.13 for a hint.)

□

Exercise 4.12

(i) A complex modal operator, often found in temporal logic, is the so-called until operator. Formulated in terms of transition systems for processes the until operator will have the following interpretation:

A process p satisfies A until B (where A and B are assertions) iff for all sequences of transitions

$$p = p_0 \stackrel{.}{\rightarrow} p_1 \stackrel{.}{\rightarrow} \ldots \stackrel{.}{\rightarrow} p_n$$

it holds that

$$\begin{array}{l} \forall i (0 \leq i \leq n).\ p_i \models A \\ \text{or } \exists i (0 \leq i \leq n).\ (p_i \models B\ \&\ \forall j (0 \leq j \leq i).\ p_j \models A). \end{array}$$

Formulate the until operator as a maximum fixed point assertion.
(See Exercise 4.13 for a hint.)

(ii) What does the following assertion (expressing so-called "strong-until") mean?

$$\mu X.(B \vee (A \wedge \langle . \rangle T \wedge [.]X))$$

□

Exercise 4.13 What do the following assertions mean? They involve assertions A and B.

(i) $inv(A) \equiv \nu X.(A \wedge [.]X)$

(ii) $ev(A) \equiv \mu X.(A \vee (\langle . \rangle T \wedge [.]X))$

(iii) $un(A, B) \equiv \nu X.(B \vee (A \wedge [.]X))$

□

Exercise 4.14 * For this exercise it will be useful to extend the modal μ-calculus with a modal operator $\langle -a \rangle A$, where a is an action, with

$$p \models \langle -a \rangle A \text{ iff } p \stackrel{b}{\rightarrow} q \text{ and } q \models A, \text{ for some } q \text{ and action } b \neq a.$$

A process p is said to be *unfair* with respect to an action a iff there is an infinite chain of transitions

$$p = p_0 \stackrel{a_0}{\rightarrow} p_1 \stackrel{a_1}{\rightarrow} \cdots \stackrel{a_{n-1}}{\rightarrow} p_n \stackrel{a_n}{\rightarrow} \cdots$$

such that

(a) $\exists q.\ p_i \xrightarrow{a} q$, for all $i \geq 0$, and

(b) $a_i \neq a$, for all $i \geq 0$.

Informally, there is an infinite chain of transitions in which a can always occur but never does.

(i) Express the property of a process being unfair as an assertion in the modal μ-calculus, and prove that any finite-state process p satisfies this assertion iff p is unfair with respect to a.

(ii) A process p is said to be *weakly unfair* with respect to an action a iff there is an infinite chain of transitions in which a can occur infinitely often but never does. Write down an assertion in the modal μ-calculus to express this property.

$\square$

4.4 Local model checking

We are interested in whether or not a finite-state process p satisfies a recursive modal assertion A, *i.e* in deciding the truth or falsity of $p \models A$. We shall give an algorithm for reducing such a satisfaction assertion to **true** or **false**. A key lemma, the Reduction Lemma, follows from Tarski's fixed point theorem.

Lemma 4.15 *(Reduction Lemma)*
Let φ be a monotonic function on a powerset $\mathcal{P}ow(\mathcal{S})$. For $S \subseteq \mathcal{S}$

$$S \subseteq \nu X.\varphi(X) \quad \Leftrightarrow \quad S \subseteq \varphi(\nu X.(S \cup \varphi(X))).$$

Proof:
"$\Rightarrow$" Assume $S \subseteq \nu X.\varphi(X)$. Then

$$S \cup \varphi(\nu X.\varphi(X)) = S \cup \nu X.\varphi(X) = \nu X.\varphi(X).$$

Therefore $\nu X.\varphi(X)$ is a postfixed point of $X \mapsto S \cup \varphi(X)$. As $\nu X.(S \cup \varphi(X))$ is the greatest such postfixed point,

$$\nu X.\varphi(X) \subseteq \nu X.(S \cup \varphi(X)).$$

By monotonicity,

$$\nu X.\varphi(X) = \varphi(\nu X.\varphi(X) \subseteq \varphi(\nu X.(S \cup \varphi(X))).$$

But $S \subseteq \nu X.\varphi(X)$ so $S \subseteq \varphi(\nu X(S \cup \varphi(X)))$, as required.
"$\Leftarrow$" Assume $S \subseteq \varphi(\nu X.(S \cup \varphi(X))$. As $\nu X.(S \cup \varphi(X))$ is a fixed point of $X \mapsto S \cup \varphi(X)$,

$$\nu X.(S \cup \varphi(X)) = S \cup \varphi(\nu X.(S \cup \varphi(X))).$$

Hence, by the assumption

$$\nu X.(S \cup \varphi(X)) = \varphi(\nu X.(S \cup \varphi(X)),$$

i.e. $\nu X.(S \cup \varphi(X))$ is a fixed point, and so a postfixed point of φ. Therefore

$$\nu X.(S \cup \varphi(X)) \subseteq \nu X.\varphi(X)$$

as $\nu X.\varphi(X)$ is the greatest postfixed point. Clearly $S \subseteq \nu X.(S \cup \varphi(X))$ so $S \subseteq \nu X.\varphi(X)$, as required. □

We are especially concerned with this lemma in the case where S is a singleton set $\{p\}$. In this case the lemma specialises to

$$p \in \nu X.\varphi(X) \Leftrightarrow p \in \varphi(\nu X.(\{p\} \cup \varphi(X))).$$

The equivalence says a process p satisfies a recursively defined property iff the process satisfies a certain kind of unfolding of the recursively defined property. The unfolding is unusual because into the body of the recursion we substitute not just the original recursive definition but instead a recursive definition in which the body is enlarged to contain p. As we shall see, there is a precise sense in which this small modification, $p \in \varphi(\nu X.(\{p\} \cup \varphi(X)))$, is easier to establish than $p \in \nu X.\varphi(X)$, thus providing a method for deciding the truth of recursively defined assertions at a process.

We allow processes to appear in assertions by extending their syntax to include a more general form of recursive assertion, ones in which finite sets of processes can tag binding occurrences of variables:
If A is an assertion in which the variable X occurs positively and $p_1, \cdots, p_n$ are processes, then $\nu X\{p_1, \cdots, p_n\}A$ is an assertion; it is to be understood as denoting the same property as $\nu X.(\{p_1, \cdots, p_n\} \vee A)$.
(The latter assertion is sensible because assertions can contain sets of processes as constants.)
We allow the set of processes $\{p_1, \cdots, p_n\}$ to be empty; in this case $\nu X\{\ \}A$ amounts simply to $\nu X.A$. In fact, from now on, when we write $\nu X.A$ it is to be understood as an abbreviation for $\nu X\{\ \}A$.

Exercise 4.16 Show $(p \models \nu X\{p_1, \cdots, p_n\}A) = \textbf{true}$ if $p \in \{p_1, \cdots, p_n\}$. □

With the help of these additional assertions we can present an algorithm for establishing whether a judgement $p \models A$ is **true** or **false**. We assume there are the usual boolean operations on truth values. Write $\neg_T$ for the operation of negation on truth values; thus $\neg_T$ (true) = false and $\neg_T$ (false) = true. Write $\wedge_T$ for the operation of binary conjunction on T; thus $t_0 \wedge_T t_1$ is true if both t_0 and t_1 are true and false otherwise. Write $\vee_T$ for the operation of binary disjunction; thus $t_0 \vee_T t_1$ is true if either t_0 or t_1 is true and false otherwise. More generally, we will use

$$t_1 \vee_T t_2 \vee_T \cdots \vee_T t_n$$

for the disjunction of the n truth values $t_1, \cdots, t_n$; this is true if one or more of the truth values is true, and false otherwise. An empty disjunction will be understood as false.

With the help of the Reduction Lemma we can see that the following equations hold:

$$
\begin{array}{rcl}
(p \models S) & = & \textbf{true} \quad \text{if } p \in S \\
(p \models S) & = & \textbf{false} \quad \text{if } p \notin S \\
(p \models T) & = & \textbf{true} \\
(p \models F) & = & \textbf{false} \\
(p \models \neg B) & = & \neg_T (p \models B) \\
(p \models A_0 \wedge A_1) & = & (p \models A_0) \wedge_T (p \models A_1) \\
(p \models A_0 \vee A_1) & = & (p \models A_0) \vee_T (p \models A_1) \\
(p \models \langle a \rangle B) & = & (q_1 \models B) \vee_T \cdots \vee_T (q_n \models B) \\
\text{where } \{q_1, \cdots, q_n\} & = & \{q | p \xrightarrow{a} q\} \\
(p \models \langle . \rangle B) & = & (q_1 \models B) \vee_T \cdots \vee_T (q_n \models B) \\
\text{where } \{q_1, \cdots, q_n\} & = & \{q | \exists a.p \xrightarrow{a} q\} \\
(p \models \nu X \{\vec{r}\} B) & = & \textbf{true} \quad \text{if } p \in \{\vec{r}\} \\
(p \models \nu X \{\vec{r}\} B) & = & (p \models B[\nu X \{p, \vec{r}\} B / X]) \quad \text{if } p \notin \{\vec{r}\}
\end{array}
$$

(In the cases where p has no derivatives, the disjunctions indexed by its derivatives are taken to be **false**.)
All but possibly the last two equations are obvious. The last equation is a special case of the Reduction Lemma, whereas the last but one follows by recalling the meaning of a "tagged" maximum fixed point (its proof is required by the exercise above).

The equations suggest reduction rules in which the left-hand-sides are replaced by the corresponding right-hand-sides, though at present we have no guarantee that this reduction does not go on forever. More precisely, the reduction rules should operate on boolean expressions built up using the boolean operations $\wedge, \vee, \neg$ from basic satisfaction expressions, the syntax of which has the form $p \vdash A$, for a process term p and an assertion A. The boolean expressions take the form:

$$b ::= p \vdash A \mid \textbf{true} \mid \textbf{false} \mid b_0 \wedge b_1 \mid b_0 \vee b_1 \mid \neg b$$

The syntax $p \vdash A$ is to be distinguished from the truth value $p \models A$.

To make the reduction precise we need to specify how to evaluate the boolean operations that can appear between satisfaction expressions as the reduction proceeds. Rather than commit ourselves to one particular method, to cover the range of different methods of evaluation of such boolean expressions we merely stipulate that the rules have the following properties:

For negations:

$$(b \to^* t \Leftrightarrow \neg b \to^* \neg_T t), \text{ for any truth value } t.$$

For conjunctions:
If $b_0 \rightarrow^* t_0$ and $b_1 \rightarrow^* t_1$ and $t_0, t_1 \in T$ then

$$(b_0 \wedge b_1) \rightarrow^* t \Leftrightarrow (t_0 \wedge_T t_1) = t, \text{ for any truth value } t.$$

For disjunctions:
If $b_0 \rightarrow^* t_0$ and $b_1 \rightarrow^* t_1$ and $t_0, t_1 \in T$ then

$$(b_0 \vee b_1) \rightarrow^* t \Leftrightarrow (t_0 \vee_T t_1) = t, \text{ for any truth value } t.$$

More generally, a disjunction $b_1 \vee b_2 \vee \cdots \vee b_n$ should reduce to **true** if, when all of $b_1, \cdots, b_n$ reduce to values, one of them is **true** and **false** if all of the values are **false**. As mentioned, an empty disjunction is understood as false.

Certainly, any sensible rules for the evaluation of boolean expressions will have the properties above, whether the evaluation proceeds in a left-to-right, right-to-left or parallel fashion. With the method of evaluation of boolean expressions assumed, the heart of the algorithm can now be presented in the form of reduction rules:

$$
\begin{array}{rcl}
(p \vdash S) & \rightarrow & \textbf{true} \quad \text{if } p \in S \\
(p \vdash S) & \rightarrow & \textbf{false} \quad \text{if } p \notin S \\
(p \vdash T) & \rightarrow & \textbf{true} \\
(p \vdash F) & \rightarrow & \textbf{false} \\
(p \vdash \neg B) & \rightarrow & \neg(p \vdash B) \\
(p \vdash A_0 \wedge A_1) & \rightarrow & (p \vdash A_0) \wedge (p \vdash A_1) \\
(p \vdash A_0 \vee A_1) & \rightarrow & (p \vdash A_0) \vee (p \vdash A_1) \\
(p \vdash \langle a \rangle B) & \rightarrow & (q_1 \vdash B) \vee \cdots \vee (q_n \vdash B) \\
\text{where } \{q_1, \cdots, q_n\} & = & \{q | p \xrightarrow{a} q\} \\
(p \vdash \langle . \rangle B) & \rightarrow & (q_1 \vdash B) \vee \cdots \vee (q_n \vdash B) \\
\text{where } \{q_1, \cdots, q_n\} & = & \{q | \exists a. p \xrightarrow{a} q\} \\
(p \vdash \nu X\{\vec{r}\}B) & \rightarrow & \textbf{true} \quad \text{if } p \in \{\vec{r}\} \\
(p \vdash \nu X\{\vec{r}\}B) & \rightarrow & (p \vdash B[\nu X\{p, \vec{r}\}B/X]) \quad \text{if } p \notin \{\vec{r}\}
\end{array}
$$

(Again, in the cases where p has no derivatives, the disjunctions indexed by its derivatives are taken to be **false**.)
The idea is that finding the truth value of the satisfaction assertion on the left is reduced to finding that of the expression on the right. In all rules but the last, it is clear that some progress is being made in passing from the left- to the right-hand-side; for these rules either the right-hand-side is a truth value, or concerns the satisfaction of strictly smaller assertions than that on the left. On the other hand, the last rule makes it at least thinkable that reduction may not terminate. In fact, we will prove it does terminate, with the correct answer. Roughly, the reason is that we are checking the satisfaction of assertions by

finite-state processes which will mean that we cannot go on extending the sets tagging the recursions forever.

Under the assumptions to do with the evaluation of boolean expressions the reduction rules are sound and complete in the sense of the theorem below. (Notice that the theorem implies the reduction terminates.)

Theorem 4.17 *Let $p \in \mathcal{P}$ be a finite-state process and A be a closed assertion. For any truth value $t \in T$,*

$$(p \vdash A) \to^* t \ \textit{ iff } (p \models A) = t.$$

Proof: Assume that p is a finite-state process. Say an assertion is a p-*assertion* if for all the recursive assertions $\nu X\{r_1, \cdots, r_k\}B$ within it $r_1, \cdots, r_k \in \mathcal{P}_p$, *i.e.* all the processes mentioned in the assertion are reachable by transitions from p. The proof proceeds by well-founded induction on p-assertions with the relation

$$\begin{aligned} A' \prec A \text{ iff } & A' \text{ is a proper subassertion of } A \\ & \text{or } A, A' \text{ have the form} \\ & A \equiv \nu X\{\vec{r}\}B \text{ and } A' \equiv \nu X\{p, \vec{r}\}B \text{ with } p \notin \{\vec{r}\}\ . \end{aligned}$$

As $\mathcal{P}_p$ is a finite set, the relation $\prec$ is well-founded.

We are interested in showing the property

$$Q(A) \Leftrightarrow_{def} \forall q \in \mathcal{P}_p\ \forall t \in T.\ [(q \vdash A) \to^* t \Leftrightarrow (q \models A) = t]$$

holds for all closed p-assertions A. The proof however requires us to extend the property Q to p-assertions A with free variables $FV(A)$, which we do in the following way:
For p-assertions A, define

$$\begin{aligned} Q^+(A) \Leftrightarrow_{def} & \forall \theta, \text{ a substitution from FV}(A) \text{ to closed } p\text{-assertions.} \\ & [(\forall X \in FV(A).\ Q(\theta(X))) \Rightarrow Q(A[\theta])]. \end{aligned}$$

Notice that when A is closed $Q^+(A)$ is logically equivalent to $Q(A)$. Here θ abbreviates a substitution like $B_1/X_1, \cdots, B_k/X_k$ and an expression such as $\theta(X_j)$ the corresponding assertion B_j.

We show $Q^+(A)$ holds for all p-assertions A by well-founded induction on $\prec$. To this end, let A be an p-assertion such that $Q^+(A')$ for all p-assertions $A' \prec A$. We are required to show it follows that $Q^+(A)$. So letting θ be a substitution from $FV(A)$ to closed p-assertions with $\forall X \in FV(A).\ Q(\theta(X))$, we are required to show $Q(A[\theta])$ for all the possible forms of A. We select a few cases:

$A \equiv A_0 \wedge A_1$: In this case $A[\theta] \equiv A_0[\theta] \wedge A_1[\theta]$. Let $q \in \mathcal{P}_p$. Let $(q \models A_0[\theta]) = t_0$ and $(q \models A_1[\theta]) = t_1$. As $A_0 \prec A$ and $A_1 \prec A$ we have $Q^+(A_0)$ and $Q^+(A_1)$. Thus $Q(A_0[\theta])$ and $Q(A_1[\theta])$, so $(q \vdash A_0[\theta]) \to^* t_0$ and $(q \vdash A_1[\theta]) \to^* t_1$. Now, for $t \in T$,

$$\begin{array}{rcl}(q \vdash A_0[\theta] \wedge A_1[\theta]) \to^* t & \Leftrightarrow & ((q \vdash A_0[\theta]) \wedge (q \vdash A_1[\theta])) \to^* t \\ & \Leftrightarrow & t_0 \wedge_T t_1 = t \\ & & \text{by the property assumed of evaln. of conjns.} \\ & \Leftrightarrow & (q \models A_0[\theta]) \wedge_T (q \models A_1[\theta]) = t \\ & \Leftrightarrow & (q \models A_0[\theta] \wedge A_1[\theta]) = t\end{array}$$

Hence $Q(A[\theta])$ in this case.

$A \equiv X$: In this case, when A is a variable, $Q(A[\theta])$ holds trivially by the assumption on θ.

$A \equiv \nu X\{\vec{r}\}B$: In this case $A[\theta] \equiv \nu X\{\vec{r}\}(B[\theta])$—recall θ is not defined on X because it is not a free variable of A. Let $q \in \mathcal{P}_p$. Either $q \in \{\vec{r}\}$ or not. If $q \in \{\vec{r}\}$ then it is easy to see

$$(q \vdash \nu X\{\vec{r}\}(B[\theta])) \to^* t \Leftrightarrow t = \mathbf{true}, \text{ for any } t \in T,$$

and that $(q \models \nu X\{\vec{r}\}(B[\theta])) = \mathbf{true}$. Hence $Q(A[\theta])$ when $q \in \{\vec{r}\}$ in this case. Otherwise $q \notin \{\vec{r}\}$. Then $\nu X\{q, \vec{r}\}B \prec A$, so $Q(\nu X\{q, \vec{r}\}(B[\theta]))$. Define a substitution θ' from $Y \in FV(B)$ to closed p-assertions by taking

$$\theta'(Y) = \begin{cases} \theta(Y) & \text{if } Y \not\equiv X \\ \nu X\{q, \vec{r}\}(B[\theta]) & \text{if } Y \equiv X \end{cases}$$

Certainly $Q(\theta'(Y))$, for all $Y \in FV(B)$. As $B \prec A$ we have $Q^+(B)$. Hence $Q(B[\theta'])$. But $B[\theta'] \equiv (B[\theta])[\nu X\{q, \vec{r}\}(B[\theta])/X]$. Thus from the reduction rules,

$$\begin{array}{rcl}(q \vdash \nu X\{\vec{r}\}(B[\theta])) \to^* t & \Leftrightarrow & (q \vdash (B[\theta])[\nu X\{q, \vec{r}\}(B[\theta])/X]) \to^* t \\ & \Leftrightarrow & (q \vdash B[\theta']) \to^* t \\ & \Leftrightarrow & (q \models B[\theta']) = t \quad \text{as } Q(B[\theta']) \\ & \Leftrightarrow & (q \models (B[\theta])[\nu X\{q, \vec{r}\}(B[\theta])/X]) = t \\ & \Leftrightarrow & (q \models \nu X\{\vec{r}\}(B[\theta])) = t \quad \text{by the Reduction Lemma.}\end{array}$$

Hence, whether $q \in \{\vec{r}\}$ or not, $Q(A[\theta])$ in this case.

For all the other possible forms of A it can be shown (Exercise!) that $Q(A[\theta])$. Using well-founded induction we conclude $Q^+(A)$ for all p-assertions A. In particular $Q(A)$ for all closed assertions A, which establishes the theorem. $\square$

Example: Consider the two element transition system given in CCS by

$$P \stackrel{def}{=} a.Q$$
$$Q \stackrel{def}{=} a.P$$

—it consists of two transitions $P \xrightarrow{a} Q$ and $Q \xrightarrow{a} P$. We show how the rewriting algorithm establishes the obviously true fact that P is able to do arbitrarily many a's, formally that $P \models \nu X.\langle a\rangle X$. Recalling that $\nu X.\langle a\rangle X$ stands for $\nu X\{\ \}\langle a\rangle X$, following the reductions of the model-checking algorithm we obtain:

$$\begin{aligned}
P \vdash \nu X\{\ \}\langle a\rangle X \ &\to P \vdash \langle a\rangle X[\nu X\{P\}\langle a\rangle X/X] \\
&i.e. P \vdash \langle a\rangle \nu X\{P\}\langle a\rangle X \\
&\to Q \vdash \nu X\{P\}\langle a\rangle X \\
&\to Q \vdash \langle a\rangle X[\nu X\{Q,P\}\langle a\rangle X/X] \\
&i.e. Q \vdash \langle a\rangle \nu X\{Q,P\}\langle a\rangle X \\
&\to P \vdash \nu X\{Q,P\}\langle a\rangle X \\
&\to \mathbf{true}.
\end{aligned}$$

□

Hence provided the constants of the assertion language are restricted to decidable properties the reduction rules give a method for deciding whether or not a process satisfies an assertion. We have concentrated on the correctness rather than the efficiency of an algorithm for local model checking. As it stands the algorithm can be very inefficient in the worst case because it does not exploit the potential for sharing data sufficiently (the same is true of several current implementations).

Exercise 4.18
(i) For the CCS process P defined by

$$P \stackrel{def}{=} a.P$$

show $p \vdash \nu X.\langle a\rangle T \wedge [a]X$ reduces to **true** under the algorithm above.
(ii) For the CCS definition

$$\begin{aligned}
P &\stackrel{def}{=} a.Q \\
Q &\stackrel{def}{=} a.P + a.\mathbf{nil}
\end{aligned}$$

show $P \vdash \mu X.[a]F \vee \langle a\rangle X$ reduces to **true**. □

Exercise 4.19 * (A project) Program a method to extract a transition system table for a finite-state process from the operational semantics in *e.g.* SML or Prolog. Program the model checking algorithm. Use it to investigate the following simple protocol. □

Exercise 4.20 * A simple communication protocol (from [17]) is described in CCS by:

$$\begin{array}{lcl}
\text{Sender} & = & a.\text{Sender'} \\[1ex]
\text{Sender'} & = & \overline{b}.(d.\text{Sender} + c.\text{Sender'}) \\[1ex]
\text{Medium} & = & b.(\overline{c}.\text{Medium} + \overline{e}.\text{Medium}) \\[1ex]
\text{Receiver} & = & e.f.\overline{d}.\text{Receiver} \\[1ex]
\text{Protocol} & = & (\text{Sender} \parallel \text{Medium} \parallel \text{Receiver})\backslash \ \{b, c, d, e\}
\end{array}$$

Use the tool developed in Exercise 4.19 (or the Concurrency Workbench or TAV system) to show the following:
The process Protocol does *not* satisfy $Inv([a](ev\langle f\rangle T))$.
Protocol does satisfy $Inv([f](ev\langle a\rangle T))$.
(Here $Inv(A) \equiv \nu X.(A \wedge [.]X)$ and $ev(A) \equiv \mu X.(A \vee (\langle . \rangle T \wedge [.]X))$, with $Inv(A)$ satisfied by precisely those processes which always satisfy A, and $ev(A)$ satisfied by precisely those processes which eventually satisfy A.) $\square$

Chapter 5

Process equivalence

The important process equivalence of strong bisimilarity is introduced, and related to Hennessy-Milner logic and the modal μ-calculus. Its equational laws are derived and its use in reasoning about processes indicated. Weak bisimilarity, to take account of the invisibility of τ-actions, is treated very briefly.

5.1 Strong bisimulation

In the absence of a canonical way to represent *the* behaviour of processes, equivalences on processes saying when processes have the same behaviour become important. Originally defined, by Milner and Park, between simple labelled transition systems as here, it is proving to have much broader currency, not just to much more general languages for concurrent processes, but also in reasoning about recursively-defined datatypes where there is no reference to concurrency.

Definition: A *(strong) bisimulation* is a binary relation R between CCS processes with the following property: If pRq then

$$\text{(i)}\forall\lambda, p'.\ p \xrightarrow{\lambda} p' \Rightarrow \exists q'.q \xrightarrow{\lambda} q' \ \&\ p'Rq' \ ,\ \text{and}$$

$$\text{(ii)}\forall\lambda, q'.\ q \xrightarrow{\lambda} q' \Rightarrow \exists p'.p \xrightarrow{\lambda} p' \ \&\ p'Rq' \ .$$

Write $p \sim q$, and say p and q are *(strongly) bisimilar* (or *strongly bisimulation equivalent*), iff there is a strong bisimulation R for which pRq.

For convenience we define bisimulation between CCS processes, making use of the transition system given by the operational semantics. But, as should be clear, the definition applies to any labelled transition system.

From the definition of bisimulation we see that to show two processes are bisimilar it suffices to exhibit a bisimulation relating them.

Techniques for building bisimulations:

Proposition 5.1 *Assume that R, S and R_i, for $i \in I$, are strong bisimulations. Then so are* *(i)* $Id_{\mathcal{P}}$, *the identity relation.*

(ii) R^{op}, *the opposite relation.*

(iii) $R \circ S$, *the composition, and*

(iv) $\bigcup_{i \in I} R_i$, *the union.*

Proof: Exercise! □

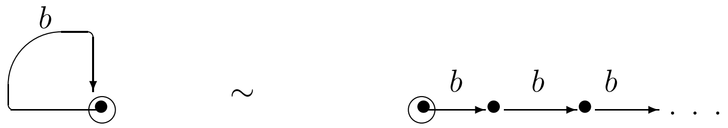
Exercise 5.2 Show the proposition above. □

It follows that:

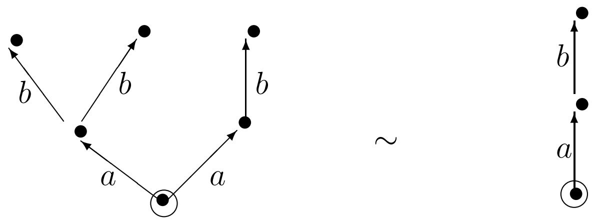
Proposition 5.3 *Strong bisimilarity* $\sim$ *is an equivalence relation.*

Proof: That $\sim$ is reflexive follows from the identity relation being a bisimulation. The symmetry of $\sim$ is a consequence of the converse of a bisimulation relation being a bisimulation, while its transitivity comes from the relation composition of bisimulations being a bisimulation. □

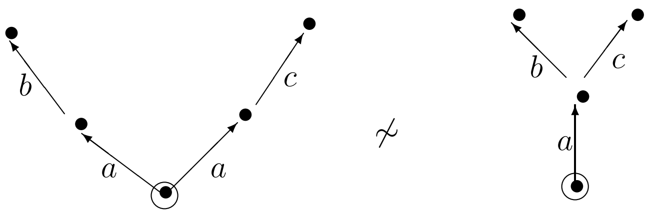
Example: (1) Bisimulation abstracts from looping:



(2) Bisimulation abstracts from inessential branching:



(3) But:



In the second process, after an a-action the process is prepared to do both a b- and a c-action (which may depend on the environment). The first process, however, is committed to either a b- or a c-action after an initial a-action (and might deadlock should the environment only be prepared to do one of the actions). Note, in particular, that bisimulation equivalence is not language equivalence (where two processes are language equivalent iff they gives rise to the same strings of actions).

5.2 Strong bisimilarity as a maximum fixed point

Given a relation R between CCS processes, define the relation $\varphi(R)$ to be such that:

$p\ \varphi(R)\ q$ iff

$$\begin{aligned}&\text{(i) } \forall a, p'.\ p \xrightarrow{a} p' \Rightarrow \exists q'.q \xrightarrow{a} q' \ \&\ p'Rq'\ ,\ \text{and}\\ &\text{(ii) } \forall a, q'.\ q \xrightarrow{a} q' \Rightarrow \exists p'.p \xrightarrow{a} p' \ \&\ p'Rq'\ .\end{aligned}$$

It is easily seen that φ is monotonic, *i.e.* if $R \subseteq S$, then $\varphi(R) \subseteq \varphi(S)$. We see that a binary relation R between processes is a strong bisimulation iff

$$R \subseteq \varphi(R)\ .$$

In other words, R is a bisimulation iff R is a postfixed point of φ regarded as function on $\mathcal{P}ow(\mathcal{P} \times \mathcal{P})$. Note that the relation $\sim$ can be described by

$$\sim = \bigcup\{R \mid R \text{ is a strong bisimulation}\}\ .$$

But, by Tarski's fixed point theorem, this relation is precisely $\nu R.\varphi(R)$, the maximum fixed point of φ. The relation $\sim$ is itself a bisimulation and moreover the largest bisimulation.

Exercise 5.4 (Bisimulation testing) Because strong bisimulation can be expressed as a maximum fixed point, the testing of bisimulation between two finite-state processes can be automated along the same lines as local model checking. Suggest how, and write a program, in *e.g.* SML or Prolog, to do it. □

5.3 Strong bisimilarity and logic

It turns out that the equivalence of strong bisimilarity is induced by Hennessy-Milner logic. This logic includes a possibly infinite conjunction, and its assertions A are given by

$$A ::= \bigwedge_{i \in I} A_i \mid \neg A \mid \langle a \rangle A$$

where I is a set, possibly empty, indexing a collection of asertions A_i, and a ranges over actions. The notion of a process p satisfying an assertion A is formalised in the relation $p \models A$ defined by structural induction on A:

$$\begin{aligned}&p \models \bigwedge_{i \in I} A_i \text{ iff } p \models A_i \text{ for all } i \in I,\\ &p \models \neg A \text{ iff not } p \models A,\\ &p \models \langle a \rangle A \text{ iff } p \xrightarrow{a} q \ \&\ q \models A \text{ for some } q.\end{aligned}$$

(An empty conjunction fulfils the role of **true** as it holds vacuously of all processes.)

Because minimum and maximum fixed points can be understood as (possibly infinite) disjunctions and conjunctions, Hennessy-Milner logic is as expressive as the modal μ-calculus.

Now we define $p \asymp q$ iff $(p \models A) \Leftrightarrow (q \models A)$ for all assertions A of Hennessy-Milner logic.

We show that $\asymp$ coincides with strong bisimulation, *i.e.* $\asymp = \sim$. So, for finite-state processes the equivalence $\asymp$ and strong bisimulation coincide with the equivalence induced by the modal μ-calculus.

Theorem 5.5 $\asymp = \sim$.

Proof: A routine structural induction on A shows that

$$\forall p, q.\ p \sim q \Rightarrow (p \models A \Leftrightarrow q \models A).$$

This shows $\asymp\ \supseteq\ \sim$.

From the definition of $\sim$, in order to show the converse inclusion, $\asymp \subseteq \sim$, it suffices to show that $\asymp$ is a strong bisimulation. This part is best proved by assuming that $\asymp$ is not a bisimulation, and deriving a contradiction. So, suppose $\asymp$ is not a bisimulation. This could only be through (i) or (ii) failing in the definition of strong bisimulation. By symmetry it is sufficient to consider one case, (i). So assume there are processes p, q with $p \asymp q$ for which $p \xrightarrow{\alpha} p'$ and yet,

$$\forall r.\ q \xrightarrow{\alpha} r \Rightarrow (p', r) \notin\ \asymp\ .$$

From the definition of $\asymp$, for any r such that $q \xrightarrow{\alpha} r$ there must be an assertion B_r such that

$$p' \models B_r \text{ and } r \not\models B_r$$

—because $(p', r) \notin\ \asymp$ the processes p', r must be distinguished by an assertion holding for one and not the other; using negation, if necessary, we can always find such a B_r. Now, take

$$A \equiv \langle\alpha\rangle(\bigwedge_{r \in I} B_r)$$

where

$$I = \{r \mid q \xrightarrow{\alpha} r\}\ .$$

Then

$$p \models A \text{ and } q \not\models A,$$

contradicting $(p, q) \in\ \asymp$. Hence $\asymp$ is a strong bisimulation. □

Corollary 5.6 *Bisimilar states satisfy the same modal μ-calculus assertions and the same CTL assertions.*

Exercise 5.7 Provide the structural induction establishing $\asymp\ \supseteq\ \sim$ omitted from the proof of the theorem above. □

5.4 Equational properties of bisimulation

A major purpose of process equivalences like bisimulation is to support equational reasoning about processes where congruence properties such as the following are useful.

Proposition 5.8 *Sum and parallel are commutative and associative with respect to strong bisimilarity.*

Assume $p \sim q$. Then,

(1) $\lambda.p \sim \lambda.q$

(2) $p + r \sim q + r$

(3) $p \parallel r \sim q \parallel r$

(3) $p \setminus L \sim q \setminus L$

(4) $p[f] \sim q[f]$

Proof: All the claims rest on producing an appropriate bisimulation, for example to show the commutativity of $\parallel$ the bisimulation is

$$\{((p \parallel q), (q \parallel p)) \mid p, q \in \mathcal{P}\} \ .$$

We'll only do (3) in detail.

(3) Define

$$R = \{(p \parallel s, q \parallel s) \mid p \sim q \ \& \ s \in \mathcal{P}\} \ .$$

To show R is a bisimulation suppose $(p \parallel s)R(q \parallel s)$ and that $p \parallel s \xrightarrow{\alpha} t$. There are three cases:

Case $p \xrightarrow{\alpha} p'$ and $t \equiv p' \parallel s$.

Then, as $p \sim q$, we obtain $q \xrightarrow{\alpha} q'$, so $q \parallel s \xrightarrow{\alpha} q' \parallel s$ with $(p' \parallel s)R(q' \parallel s)$.

Case $s \xrightarrow{\alpha} s'$ and $t \equiv p \parallel s'$. Then, $q \parallel s \xrightarrow{\alpha} q \parallel s'$ and $(p \parallel s')R(q \parallel s')$.

Case $p \xrightarrow{l} p'$, $s \xrightarrow{\bar{l}} s'$, $\alpha = \tau$ and $t \equiv p' \parallel s'$. Then, as $p \sim q$, we obtain $q \xrightarrow{l} q'$ with $p' \sim q'$. So $q \parallel s \xrightarrow{\tau} q' \parallel s'$ and $(p' \parallel s')R(q' \parallel s')$.

The proofs of (4) and (5) are similar, while those of (1) and (2) are easy. They are left to the reader. □

Exercise 5.9 (i) Show

$$\mathrm{rec}(P = p) \sim p[\mathrm{rec}(P = p)/P] \ .$$

(ii) Show that

$$(p + q) \setminus L \sim p \setminus L + q \setminus L \ ,$$
$$(p + q)[f] \sim p[f] + q[f] \ ,$$

but that

$$(p + q) \parallel r \not\sim (p \parallel r) + (q \parallel r) \ .$$

[Hint: For the latter, taking $p \equiv a.\mathbf{nil}$, $q \equiv b.\mathbf{nil}$ and $r \equiv c.\mathbf{nil}$ suffices.] □

5.4.1 Expansion theorems

The parallel composition of processes behaves as if the actions of the processes were nondeterministically shuffled together, though with the possibility of synchronisation of complementary actions. So ultimately parallel composition is understood in terms of nondeterministic sum; certainly any finite process built up using parallel composition will be bisimilar to one not mentioning this operation. The role of the expansion theorem for parallel composition is to (partially) eliminate occurrences of parallel-composition operator in favour of nondeterministic sums.

Notice first that any CCS process is bisimilar to its expansion to a sum over its initial actions.

Proposition 5.10 *Let p be a pure CCS process. Then,*

$$p \sim \Sigma\{\lambda.p' \mid p \xrightarrow{\lambda} p'\} \ .$$

Proof: Clearly

$$\{(p, \Sigma\{\lambda.p' \mid p \xrightarrow{\lambda} p'\})\} \cup \{(r, r) \mid r \in \mathcal{P}\}$$

is a bisimulation. □

Because restriction and relabelling distribute through sum we immediately obtain the following proposition.

Proposition 5.11

(1) $(\Sigma_{i\in I}\alpha_i.p_i) \setminus L \sim \Sigma\{\alpha_i.(p_i \setminus L) \mid \alpha_i \notin L \cup \overline{L}\}$.

(2) $(\Sigma_{i\in I}\alpha_i.p_i)[f] \sim \Sigma_{i\in I} f(\alpha_i).(p_i[f])$.

The expansion theorem for parallel composition is more interesting. It expresses how the parallel composition of two processes allows the processes to proceed asynchronously, or through joint τ-action of synchronisation whenever their actions are complementary.

Theorem 5.12 *Suppose*

$$p \sim \Sigma_{i\in I}\alpha_i.p_i \text{ and } q \sim \Sigma_{j\in J}\beta_j.q_j \ .$$

Then,

$$\begin{aligned}(p \parallel q) \sim & \Sigma_{i\in I}\alpha_i(p_i \parallel q) + \Sigma_{j\in J}\beta_j(p \parallel q_j) + \\ & \Sigma\{\tau.(p_i \parallel q_j) \mid \alpha_i = \bar{\beta}_j\} \ .\end{aligned}$$

In practice, to save writing, it's often best to use a combination of the expansion laws for parallel composition, restriction and relabelling. For example, suppose

$$p \sim \Sigma_{i\in I}\alpha_i.p_i \text{ and } q \sim \Sigma_{j\in J}\beta_j.q_j \ .$$

Then,

$$\begin{aligned}(p \parallel q) \setminus L \sim & \Sigma\{\alpha_i(p_i \parallel q) \setminus L \mid \alpha_i \notin L\} + \Sigma\{\beta_j(p \parallel q_j \setminus L) \mid \beta_j \notin L\} + \\ & \Sigma\{\tau.(p_i \parallel q_j) \setminus L \mid \alpha_i = \bar{\beta}_j\} .\end{aligned}$$

Exercise 5.13 Write down an expansion law for three components set in parallel. □

As an example of the expansion laws in use we study a binary semaphore in parallel with two processes which are imagined to get the semaphore when they wish to access their critical region. Define

$$\begin{aligned} Sem &\stackrel{\text{def}}{=} get.\ put.\ Sem\ , \\ P_1 &\stackrel{\text{def}}{=} \bar{get}.\ a_1.\ b_1.\ \bar{put}.P_1\ , \\ P_2 &\stackrel{\text{def}}{=} \bar{get}.\ a_2.\ b_2.\ \bar{put}.P_2\ . \end{aligned}$$

Combining them together, we obtain

$$SYS \equiv (P_1 \parallel P_2 \parallel Sem) \setminus \{get, put\} .$$

To understand the behaviour of SYS, we apply the expansion laws and derive:

$$\begin{aligned}
& SYS \\
& \sim (\bar{get}.a_1.b_1.\bar{put}.P_1 \parallel \bar{get}.a_2.b_2.\bar{put}.P_2 \parallel get.put.Sem) \setminus \{get, put\} \\
& \sim \tau.((a_1.b_1.\bar{put}.P_1 \parallel P_2 \parallel put.Sem) \setminus \{get, put\}) + \tau.((P_1 \parallel a_2.b_2.\bar{put}.P_2 \parallel put.Sem) \setminus \{get, put\}) \\
& \sim \tau.a_1.b_1.((\bar{put}.P_1 \parallel P_2 \parallel put.Sem) \setminus \{get, put\}) + \tau.a_2.b_2.((P_1 \parallel \bar{put}.P_2 \parallel put.Sem) \setminus \{get, put\}) \\
& \sim \tau.a_1.b_1.\tau.((P_1 \parallel P_2 \parallel Sem) \setminus \{get, put\}) + \tau.a_2.b_2.\tau.((P_1 \parallel P_2 \parallel Sem) \setminus \{get, put\}) \\
& \sim \tau.a_1.b_1.\tau.SYS + \tau.a_2.b_2.\tau.SYS .
\end{aligned}$$

Exercise 5.14 A semaphore with capacity 2 in its start state can be defined as the process $Twosem0$ in the definition

$$\begin{aligned} Twosem0 &\stackrel{\text{def}}{=} get.Twosem1\ , \\ Twosem1 &\stackrel{\text{def}}{=} get.Twosem2 + put.Twosem0\ , \\ Twosem2 &\stackrel{\text{def}}{=} put.Twosem1\ . \end{aligned}$$

Exhibit a bisimulation showing that

$$Twosem0 \sim Sem \parallel Sem\ ,$$

where Sem is the unary semaphore above. □

5.5 Weak bisimulation and observation congruence

Strong bisimilarity discriminates between, for example, the three CCS processes **nil**, $\tau.\mathbf{nil}$ and $\tau.\tau.\mathbf{nil}$, though their only difference is in the number of invisible τ-actions they perform. The τ-actions are invisible in the sense that no process can interact with a τ-action, although they can have a marked effect on the course a computation follows as in the process $a.p + \tau.q$, where the τ-action can make a preemptive choice.

In order to take better account of the invisibility of τ-actions, Hennessy and Milner developed *weak bisimilarity.* In fact weak bisimilarity can be viewed as strong bisimilarity on a transition system modified to make the number of τ-actions invisible.

For CCS processes, define

$$p \overset{\tau}{\Rightarrow} q \text{ iff } p(\overset{\tau}{\rightarrow})^* q \ ,$$

i.e. p can do several, possibly zero, τ-actions, to become q. Thus $p \overset{\tau}{\Rightarrow} p$, for any CCS process p. For any non-τ action l, define

$$p \overset{l}{\Rightarrow} q \text{ iff } \exists r, r'.\ p \overset{\tau}{\Rightarrow} r \ \&\ r \overset{l}{\rightarrow} r' \ \&\ r' \overset{\tau}{\Rightarrow} q \ .$$

A *weak bisimulation* is a binary relation R between CCS processes with the following property: If pRq then

$$\text{(i)}\forall \lambda, p'.\ p \overset{\lambda}{\Rightarrow} p' \Rightarrow \exists q'.q \overset{\lambda}{\Rightarrow} q' \ \&\ p'Rq' \ , \text{ and}$$
$$\text{(ii)}\forall \lambda, q'.\ q \overset{\lambda}{\Rightarrow} q' \Rightarrow \exists p'.p \overset{\lambda}{\Rightarrow} p' \ \&\ p'Rq' \ .$$

Two processes p and q are *weakly bisimilar*, written $p \approx q$, iff there is a weak bisimulation relating them.

So, weak bisimilarity is simply strong bisimilarity but based on $\overset{\lambda}{\Rightarrow}$- rather than $\overset{\lambda}{\rightarrow}$-transitions.

The following exercise guides you through the key properties of weak bisimulation.

Exercise 5.15 Show that if $p \sim q$, then $p \approx q$.
Show $p \approx \tau.p$.
Explain why, if $p \approx q$, then

$$\alpha.p \approx \alpha.q \ , \quad p \parallel r \approx q \parallel r \ , \quad p \setminus L \approx q \setminus L \ , \quad p[f] \approx q[f] \ .$$

Explain why, though $\mathbf{nil} \approx \tau.\mathbf{nil}$, it is *not* the case that $\mathbf{nil}+a.\mathbf{nil} \approx \tau.\mathbf{nil}+a.\mathbf{nil}$. □

As the exercise above shows, it is possible for $p \approx q$ and yet $p + r \not\approx q + r$. The failure of this congruence property led Milner to refine weak bisimulation to observation congruence which takes more careful account of initial actions.

Processes p, q are *observation congruent* iff

$$p \xrightarrow{\alpha} p' \Rightarrow \exists q'.\ q \stackrel{\tau}{\Rightarrow}\xrightarrow{\alpha}\stackrel{\tau}{\Rightarrow} q' \ \&\ p' \stackrel{\sim}{\approx} q' \ , \text{ and}$$
$$q \xrightarrow{\alpha} q' \Rightarrow \exists p'.\ p \stackrel{\tau}{\Rightarrow}\xrightarrow{\alpha}\stackrel{\tau}{\Rightarrow} p' \ \&\ p' \stackrel{\sim}{\approx} q' \ .$$

Consequently, if processes p, q are strongly bisimilar, then they are observation congruent.

Exercise 5.16 Writing $=$ for observation congruence, show that if $p = q$, then $p+r = q+r$. (In fact, all the expected congruence properties hold of observation congruence—see [11] ch.7.) □

5.6 On interleaving models

The two CCS processes $a.b.\mathbf{nil}+b.a.\mathbf{nil}$ and $a.\mathbf{nil} \parallel b.\mathbf{nil}$ have isomorphic transition systems according to the operational semantics, and so are certainly bisimilar, despite one being a parallel composition and the other not even containing a parallel composition. The presentation of parallelism here has, in effect, treated parallel composition by regarding it as a shorthand for nondeterministic interleaving of atomic actions of the components. This is despite the possibility that the a- and b-actions in $a.\mathbf{nil} \parallel b.\mathbf{nil}$ might be completely separate and independent of each other. Consider too the parallel composition $a.\mathbf{nil} \parallel \mathrm{rec}(P = \tau.P)$. This process might perform an infinite sequence of τ-actions without performing the a-action, even though the a-action might in reality be completely independent of all the τ-actions; a biproduct of an interleaving model is that processes compete for execution time. If one were interested in the independence of actions the transition semantics is too abstract. We turn next to a model designed to make such independence explicit.

Chapter 6

Petri nets

This chapter provides a quick introduction to Petri nets, probably the best known example of a model which makes explicit the causal independence of events. Nets will be applied in the next chapter in order to give an event-based semantics and analysis of security protocols.

6.1 Preliminaries on multisets

The explanation of general Petri nets involves a little algebra of multisets (or bags), which are like sets but where multiplicities of elements matters. Its convenient to also allow infinite multiplicities, so we adjoin an extra element ∞ to the natural numbers.

Extend the natural numbers ω by the new element ∞ and write $\omega^\infty = \omega \cup \{\infty\}$. Extend addition on integers to the element ∞ by defining $\infty + n = n + \infty = \infty$, for all $n \in \omega^\infty$. We can also extend subtraction to an operation between ω^∞, on the left of the minus, and ω, on the right, by taking

$$\infty - n = \infty \ ,$$

for all $n \in \omega$. In this way we avoid $\infty - \infty$! We also extend the order on numbers by setting $n \leq \infty$ for any $n \in \omega^\infty$.

A ∞-*multiset* over a set X is a function $f : X \to \omega^\infty$, associating a nonnegative number or ∞ with each $x \in X$; here it is usual to write f_x instead of $f(x)$. Write $\mathbf{m}^\infty X$ for the set of ∞-multisets over X. (It is helpful to think of a ∞-multiset f over X as a kind of vector in a space $\mathbf{m}^\infty X$ with basis X serving to index its components, or entries, f_x.) We call *multisets* those ∞-multisets whose entries are never ∞. Write $\mathbf{m}X$ for the space of multisets over X. In particular, the *null* multiset over X is the function $x \mapsto 0$ for any $x \in X$. We can identify subsets of X with those multisets of $f \in \mathbf{m}X$ such that $f_x \leq 1$ for all $x \in X$.

Some operations on multisets

Useful operations and relations on ∞-multisets are induced pointwise by operations and relations on integers, though some care must be taken to avoid negative entries and operations with ∞.

Let $f, g \in \mathbf{m}^\infty X$. Define

$$f \leq g \iff \forall x \in X.\ f_x \leq g_x.$$

Clearly ∞-multisets are closed under $+$ but not, in general, under $-$. Define

$$(f+g)_x = f_x + g_x\ ,$$

for all $x \in X$. If $g \leq f$, for ∞-multiset f and multiset g, then their difference $f - g$ is a multiset with

$$(f-g)_x = f_x - g_x\ .$$

6.2 General Petri nets

Petri nets are a well-known model of parallel computation. They come in several variants. Roughly, a Petri net can be thought of as a transition system where, instead of a transition occurring from a single global state, an occurrence of an event is imagined to affect only the conditions in its neighbourhood.

We start with the definition of general nets, where multisets play an explicit role. Later we will specialise to two subclasses of nets for the understanding of which only sets need appear explicitly; the implicit use of multisets does however help explain the nets' behaviour.

A *general Petri net* (often called a *place-transition system*) consists of

- a set of *conditions* (or *places*), P,
- a set of *events* (or *transitions*), T,
- a *precondition map* $pre : T \to \mathbf{m}P$, which to each $t \in T$ assigns a multiset of conditions $pre(t)$. It is traditional to write ${}^{\cdot}t$ for $pre(t)$.
- a *postcondition* map $post : T \to \mathbf{m}^\infty P$ which to each $t \in T$ assigns a ∞-multiset of conditions $post(t)$, traditionally written $t^{\cdot}$.
- a *capacity* function $Cap \in \mathbf{m}^\infty P$ which to each $p \in P$ assigns a nonnegative number or ∞, bounding the multiplicity to which a condition can hold; a capacity of ∞ means the capacity is unbounded.

A state of a Petri net consists of a *marking* which is an ∞-multiset $\mathcal{M}$ over P bounded by the capacity function, *i.e.*

$$\mathcal{M} \leq Cap\ .$$

A marking captures a notion of distributed, global state.

6.2.1 The token game for general nets

Markings can change as events occur, precisely how being expressed by the transitions

$$\mathcal{M} \xrightarrow{t} \mathcal{M}'$$

events t determine between markings $\mathcal{M}$ and $\mathcal{M}'$.

For markings $\mathcal{M}$, $\mathcal{M}'$ and $t \in T$, define

$$\mathcal{M} \xrightarrow{t} \mathcal{M}' \text{ iff } {}^{\cdot}t \leq \mathcal{M} \text{ and } \mathcal{M}' = \mathcal{M} - {}^{\cdot}t + t^{\cdot}\ .$$

An event t is said to have *concession* (or be *enabled*) at a marking $\mathcal{M}$ iff its preconditions are met by the marking and its occurrence would lead to a legitimate marking, *i.e.* iff

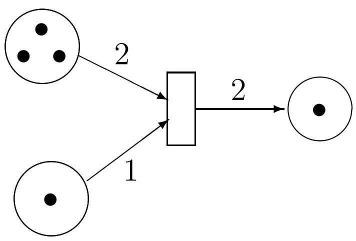
$${}^{\cdot}t \leq \mathcal{M} \text{ and } \mathcal{M} - {}^{\cdot}t + t^{\cdot} \leq Cap\ .$$

There is a widely-used graphical notation for nets in which events are represented by squares, conditions by circles and the pre- and postcondition maps by directed arcs weighted by nonzero numbers or ∞. A marking is represented by the presence of tokens on a condition, the number of tokens representing the multiplicity to which the condition holds. So, for example, a marking $\mathcal{M}$ in which a

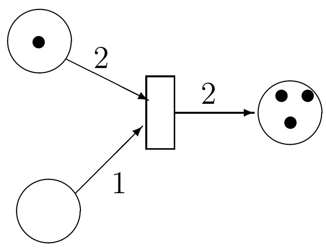
$$\mathcal{M}_p = 2,\ \mathcal{M}_q = 5 \text{ and } \mathcal{M}_r = \infty,$$

would be represented by 2 tokens residing on the condition p, a number of 5 tokens on condition q and countably infinitely many tokens residing on r. As an event t occurs for each condition p it removes $({}^{\cdot}t)_p$ tokens from p and sets $(t^{\cdot})_p$ tokens onto p—for the event to have concession it must be possible to do this without violating the capacity bounds. Note that it is possible for a condition p to be both a precondition and postcondition of the same event t in the sense that both $({}^{\cdot}t)_p \neq 0$ and $(t^{\cdot})_p \neq 0$; then there would be arcs in both directions between p and t.

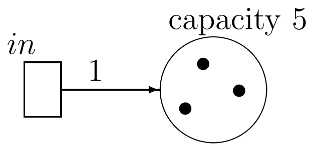
Example:
(1) Assuming that no capacities are exceeded, for example if the conditions have unbounded capacities, the occurrence of the event affects the marking in the way shown:



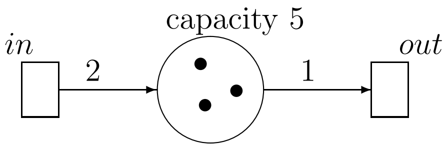
goes to



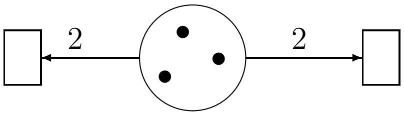
(2) The following simple net represents, for example, a buffer which can store up to five identical items, into which an item is put on each occurrence of the event *in*. Initially it contains three items. The event *in* has no preconditions so can occur provided in so doing it does not cause the capacity to be exceeded. Consequently, the event *in* can only occur twice.



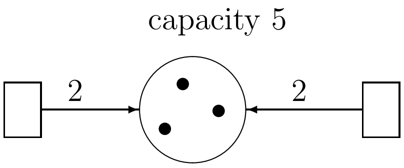
The following simple net represents a buffer which can store up to five identical items, into which 2 items are put on each occurrence of the event *in*, and out of which one item is taken on each occurrence of the event *out*. Initially it contains three items. If the event *out* does not occur, then the event *in* can only occur once, but can occur additionally as the buffer is emptied through occurrences of *out*.



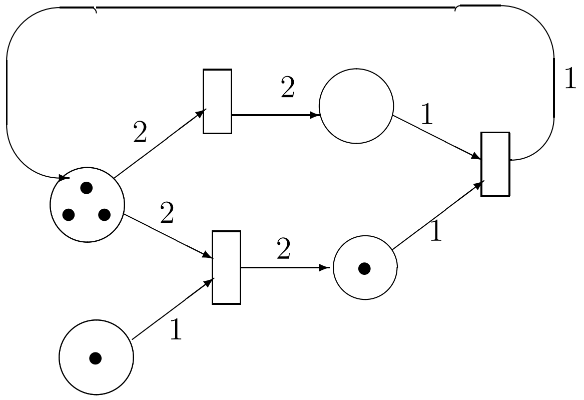
(3) Events can be in conflict, in the sense that the occurrence of one excludes the occurrence of the other, either through competing at their preconditions, as in:



or through competing at their postconditions if the occurrence of both would violate the capacity bounds, as in:



This shows how nondeterminism can be represented in a net. (4) The behaviour of general nets can be quite complicated, even with unbounded capacities, as is assumed of the net below:



[In fact, the use of finite capacities does not lead to any richer behaviour over nets without any capacity constraints; for the conditions with a finite capacity there is a way to adjoin "complementary" conditions so that all capacities may be set to ∞ without upsetting the net's behaviour—a special case of this construction is given in Exercise 6.4 below.]

Exercise 6.1 Explain in words the behaviour of the net drawn in (4) of the example above.

We won't make use of it here, but more generally one can talk of a multiset of events having concession, and whose joint, or concurrent, occurrence leads from one marking to another. The concurrent, or independent, occurrence of events is more easily understood for simpler nets of the kind we consider next.

6.3 Basic nets

We instantiate the definition of general Petri nets to a case where in all the multisets the multiplicities are either 0 or 1, and so can be regarded as sets. In particular, we take the capacity function to assign 1 to every condition, so that markings become simply subsets of conditions. The general definition now specialises to the following.

A *basic Petri net* consists of

- a set of *conditions*, B,
- a set of *events*, E, and
- two maps: a *precondition* map $pre : E \rightarrow \mathcal{P}ow(B)$, and a *postcondition* map $post : E \rightarrow \mathcal{P}ow(B)$. We can still write ${}^{\cdot}e$ for the preconditions and $e^{\cdot}$ for the postconditions of $e \in E$.
  [Note that it is possible for a condition to be both a precondition and a postcondition of the same event.]

Now a *marking* consists of a subset of conditions, specifying those conditions which hold.

6.3.1 The token game for basic nets

Markings can change as events occur, precisely how being expressed by the transitions

$$\mathcal{M} \xrightarrow{e} \mathcal{M}'$$

events e determine between markings $\mathcal{M}, \mathcal{M}'$.

For $\mathcal{M}, \mathcal{M}' \subseteq B$ and $e \in E$, define

$$\begin{aligned} \mathcal{M} \xrightarrow{e} \mathcal{M}' \text{ iff } & (1)\ {}^{\cdot}e \subseteq \mathcal{M}\ \&\ (\mathcal{M} \setminus {}^{\cdot}e) \cap e^{\cdot} = \emptyset \text{ (Concession), and} \\ & (2)\ \mathcal{M}' = (\mathcal{M} \setminus {}^{\cdot}e) \cup e^{\cdot}\ . \end{aligned}$$

Property (1) expresses that the event e has *concession* at the marking $\mathcal{M}$. Returning to the definition of concession for general nets, of which it is an instance, it ensures that the event does not load another token on a condition that is already marked. Property (2) expresses in terms of sets the marking that results from the occurrence of an event. So, an occurrence of the event ends the holding of its preconditions and begins the holding of its postconditions.

There is an alternative characterisation of the transitions between markings induced by event occurrences:

Proposition 6.2 *Let $\mathcal{M}, \mathcal{M}'$ be markings and e an event of a basic net. Then*

$$\mathcal{M} \xrightarrow{e} \mathcal{M}' \text{ iff } {}^{\cdot}e \subseteq \mathcal{M}\ \&\ e^{\cdot} \subseteq \mathcal{M}'\ \&\ \mathcal{M} \setminus {}^{\cdot}e = \mathcal{M}' \setminus e^{\cdot}.$$

Exercise 6.3 Prove the proposition above. □

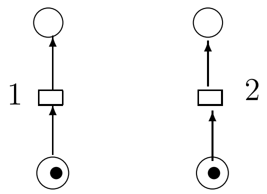
For basic nets, it is simple to express when two events are independent of each other; two events are independent if their neighbourhoods of conditions are disjoint. Events e_0, e_1 are *independent* iff

$$({}^{\cdot}e_0 \cup e_0^{\cdot}) \cap ({}^{\cdot}e_1 \cup e_1^{\cdot}) = \emptyset\ .$$

We illustrate by means of a few small examples how basic nets can be used to model nondeterminism and concurrency. We can still make use of the commonly accepted graphical notations for nets, though now conditions either hold or don't hold in a marking and the directed arcs always implicitly carry multiplicity 1. The holding of a condition is represented by marking it by a single "token". The distribution of tokens changes as the "token game" takes place; when an event occurs the tokens are removed from its preconditions and placed on its postconditions.

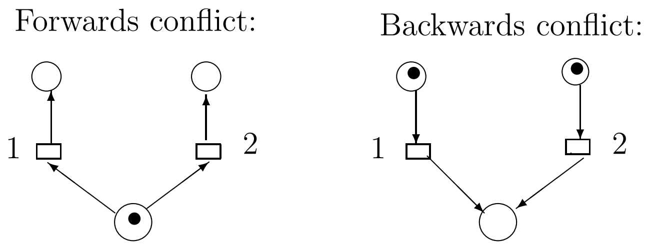
Example:

(1) Concurrency:



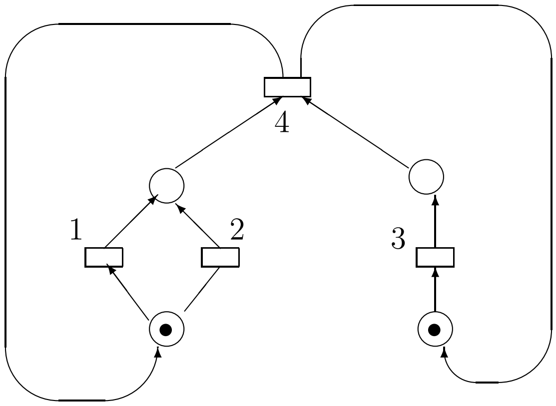
The events 1 and 2 can occur concurrently, in the sense that they both have concession and are independent in not having any pre or post conditions in common.

(2)

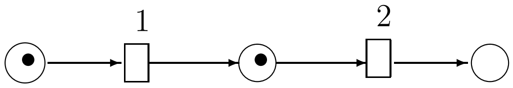


Either one of events 1 and 2 can occur, but not both. This shows how nondeterminism can be represented in a basic net.

(3) Basic nets are generally more complicated of course, and may possess looping behaviour. In the net below, initially, at the marking shown, the events 1, 2 and 3 all have concession. The events 1 and 3 can occur concurrently as can the events 2 and 3. The events 1 and 2 are however in conflict with each other, and only one of them can occur. Once either (1 and 3) or (2 and 3) have occurred the event 4 has concession, and its occurrence will the restore the net back to its initial marking.



(4) Contact:



The event 2 has concession. The event 1 does not—its post condition holds—and it can only occur after 2.

Example (4) above illustrates contact. In general, there is *contact* at a marking $\mathcal{M}$ when for some event e

$${}^{\cdot}e \subseteq \mathcal{M} \ \& \ (\mathcal{M} \setminus {}^{\cdot}e) \cap e^{\cdot} \neq \emptyset.$$

Contact has a paradoxical feel to it; the occurrence of an event is blocked through conditions, which it should begin, holding already. However blocking through contact is perfectly consistent with the understanding that the occurrence of an event should end the holding of its preconditions and begin the holding of its postconditions; if the postconditions already hold, and are not also preconditions of the event, then they cannot begin to hold on the occurrence of the event. Often contact is avoided at the markings which can arise in the behaviour of nets. For example, often nets come equipped with an initial marking from which all future markings are reachable through the occurrences of events. Such nets are called *safe* when contact never occurs at any reachable marking, and many constructions on nets preserve safeness. In fact, any net can be turned into a safe net with essentially the same behaviour.

Exercise 6.4 Define the *complement* of a condition b in a net to be the condition $\tilde{b}$ such that

$$\forall \text{ events } e.\ \tilde{b} \in {}^{\cdot}e \text{ iff } b \in e^{\cdot} \ \&\ b \notin {}^{\cdot}e\ ,$$
$$\tilde{b} \in e^{\cdot} \text{ iff } b \in {}^{\cdot}e \ \&\ b \notin e^{\cdot}$$

—so the starting and ending events of $\tilde{b}$ are the reverse of those of b.

Here's a way to make a non-safe net (so a net with an initial marking) into a safe net with the same behaviour: adjoin, in addition to the existing conditions b all their complements, extending the pre- and postcondition maps accordingly, and take $\tilde{b}$ to be in the initial marking iff b is not in the initial marking.

Perform this operation on the net with contact exampled above. Why is the result safe? □

One important use of Petri nets and related independence models has been in "partial-order" model-checking, where the independence of events is exploited in exploring the reachable state space of a process. A net marking $\mathcal{M}$ is reachable if there is a sequence of events

$$e_1, \cdots e_i, e_{i+1} \cdots e_k$$

such that

$$\mathcal{M}_0 \xrightarrow{e_1} \cdots \xrightarrow{e_i} \mathcal{M}_i \xrightarrow{e_{i+1}} \mathcal{M}_{i+1} \cdots \xrightarrow{e_k} \mathcal{M}_k\ ,$$

where $\mathcal{M}_0$ is the initial marking and $\mathcal{M}_k = \mathcal{M}$. If two consecutive events e_i, e_{i+1} are independent, then the sequence

$$e_1, \cdots e_{i+1}, e_i \cdots e_k\ ,$$

with the two events interchanged, also leads to $\mathcal{M}$. Two such sequences are said to be *trace-equivalent*. Clearly in seeking out the reachable markings it suffices to follow only one event sequence up to trace equivalence. This fact is the corner stone of partial-order model-checking, the term "partial order" referring to a partial-order of event occurrences that can be extracted via trace equivalence.

Exercise 6.5 Check that, in a net, if

$$\mathcal{M} \xrightarrow{e_1} \mathcal{M}_1 \xrightarrow{e_2} \mathcal{M}'$$

where the events e_1, e_2 are independent, then

$$\mathcal{M} \xrightarrow{e_2} \mathcal{M}_2 \xrightarrow{e_1} \mathcal{M}'$$

for some marking $\mathcal{M}_2$. □

Exercise 6.6 Try to describe the operations of prefix, sum and parallel composition of pure CCS in terms of (graphical) operations on Petri nets. Assume that the nets' events carry labels. (This will be illustrated in the lectures.) □

6.4 Nets with persistent conditions

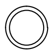
Sometimes we have use for conditions which once established continue to hold and can be used repeatedly. This is true of assertions in traditional logic, for example, where once an assertion is established to be true it can be used again and again in the proof of further assertions. Similarly, if we are to use net events to represent rules of the kind we find in inductive definitions, we need conditions that persist.

Persistent conditions can be understood as an abbreviation for conditions within general nets which once they hold, do so with infinite multiplicity. Consequently any number of events can make use of them as preconditions but without their ever ceasing to hold. Such conditions, having infinite capacity, can be postconditions of several events without there being conflict. Let us be more precise.

We modify the definition of basic net given above by allowing certain conditions to be *persistent*. A net with persistent conditions will still consist of events and conditions related by pre- and postcondition maps which to an event will assign a set of preconditions and a set of postconditions. But, now amongst the conditions are the persistent conditions forming a subset P. A marking of a net with persistent conditions will be simply a subset of conditions, of which some may be persistent.

We can understand a net with persistent conditions as a general net with the same sets for conditions and events. (It is this interpretation that leads to the token game for nets with persistent conditions.) The general net's capacity function will be either 1 or ∞ on a condition, being ∞ precisely on the persistent conditions. When p is persistent, $p \in e^{\cdot}$ is interpreted in the general net as $(e^{\cdot})_p = \infty$, and $p \in {}^{\cdot}e$ as $({}^{\cdot}e)_p = 1$. A marking of a net with persistent conditions will correspond to a marking in the general Petri net in which those persistent conditions which hold do so with infinite multiplicity.

Graphically, we'll distinguish persistent conditions by drawing them as double circles:



6.4.1 Token game for nets with persistent conditions

The token game is modified to account for the subset of conditions P being persistent. Let $\mathcal{M}$ and $\mathcal{M}'$ be markings (*i.e.* subsets of conditions), and e an event. Define

$$\mathcal{M} \xrightarrow{e} \mathcal{M}' \text{ iff } {}^{\cdot}e \subseteq \mathcal{M} \ \& \ (\mathcal{M} \setminus (P \cup {}^{\cdot}e)) \cap e^{\cdot} = \emptyset \ (e \text{ has concession}), \text{ and}$$
$$\mathcal{M}' = (\mathcal{M} \setminus {}^{\cdot}e) \cup e^{\cdot} \cup (\mathcal{M} \cap P) \ .$$

The token game of a net with persistent conditions fits our understanding of persistency, and specifically it matches the token game in the interpretation as a general net.

We will find nets with persistent conditions useful in modelling and analysing security protocols.

6.5 Other independence models

Petri nets are an example of an independence model, in the sense that they make explicit the independence of events. Other examples of independence models are Mazurkiewicz trace languages (languages subject to trace equivalence determined by the independence of actions), event structures (sets of events with extra relations of causality and conflict), pomset languages (languages of labelled partial orders of events) and transition systems with an extra relation of independence on transitions or actions. Despite their superficial differences, independence models, including nets, are closely related, and there are translations between the different models (see [21]). For example, just as one can unfold a transition system to a tree, so can one unfold a net to an occurrence net, a trace language, or an event structure. Trace languages give rise to event structures with a partial order of causal dependence on events (the reason for the term "partial order" model checking). Not only do Petri nets determine transition systems with independence (with the markings as states), but conversely transition systems with independence give rise to Petri nets (the conditions being built out of certain subsets of states and transitions).

Chapter 7

Security protocols

Security protocols raise new issues of correctness. A process language **SPL** (Security Protocol Language) in which to formalise security protocols is introduced. Its Petri-net semantics supports a form of event-based reasoning in the analysis of security protocols (similar to that used in strand spaces and Paulson's inductive method). Several reasoning principles are extracted from the semantics and applied in proofs of secrecy and authentication.[1]

7.1 Introduction

Security protocols are concerned with exchanging messages between agents via an untrusted medium or network. The protocols aim at providing guarantees such as confidentiality of transmitted data, user authentication, anonymity etc. A protocol is often described as a sequence of messages, and usually encryption is used to achieve security goals.

As an example consider the Needham-Schröder-Lowe (NSL) protocol:

$$\begin{array}{ll}
(1) & A \longrightarrow B : \{m, A\}_{Pub(B)} \\
(2) & B \longrightarrow A : \{m, n, B\}_{Pub(A)} \\
(3) & A \longrightarrow B : \{n\}_{Pub(B)}
\end{array}$$

This protocol, like many others of its kind, has two roles: one for the initiator, here A, and one for the responder, here B. It is a public-key protocol that assumes an underlying public-key infrastructure, such as RSA. Both A and B have their own, secret private keys. Public keys in contrast are known to all participants in the protocol. In addition, the NSL protocol makes use of *nonces*, m, n. One can think of them as newly generated, unguessable numbers whose purpose is to ensure the freshness of messages.

Suppose A and B are agent names standing say for agents Alice and Bob. The protocol describes an interaction between the initiator Alice and the responder Bob as following: Alice sends to Bob a new nonce m together with her

[1]This chapter is based on joint work with Federico Crazzolara.

own agent name A both encrypted with Bob's public key. When the message is received by Bob, he decrypts it with his secret private key. Once decrypted, Bob prepares an encrypted message for Alice that contains a new nonce together with the nonce received from Alice and his name B. Acting as responder, Bob sends it to Alice, who recovers the clear text using her private key. Alice convinces herself that this message really comes from Bob, by checking whether she got back the same nonce sent out in the first message. If that is the case, she acknowledges Bob by returning his nonce. He will do a similar test.

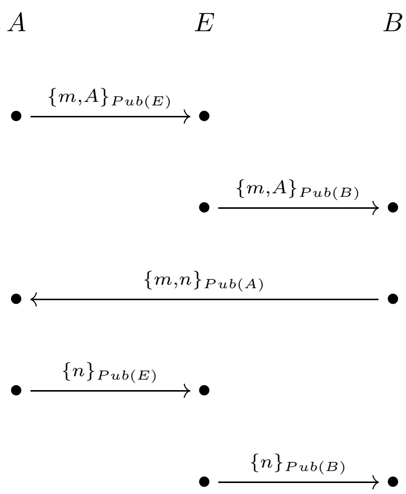
The NSL protocol aims at distributing nonces m and n in a secure way, allowing no one but the initiator and the responder to know them (*secrecy*). Another aim of the protocol is *authentication*: Bob should be guaranteed that m is indeed the nonce sent by Alice.

The protocol should be thought of as part of a longer message-exchange sequence. After initiator and responder complete a protocol exchange, they will continue communication, possibly using the exchanged nonces to establish a session key.

Even if protocols are designed carefully, following established criteria, they may contain flaws. Protocols involve many concurrent runs among a set of distributed users. Then, the NSL protocol is prone to a "middle-man" attack if the name B is not included in the second message. This attack on the original protocol of Needham and Schröder was discovered by Gavin Lowe. B is not included in the second message. Consider the original protocol of Needham and Schröder:

$$\begin{array}{ll} (1) & A \longrightarrow B : \{m, A\}_{Pub(B)} \\ (2) & B \longrightarrow A : \{m, n\}_{Pub(A)} \\ (3) & A \longrightarrow B : \{n\}_{Pub(B)} \end{array}$$

The following is a sequence of message exchanges, following the above protocol and leading to an attack.



The agent E not only gets to know the "secret" nonce n but also convinces B that he is talking to A, when A is instead talking to E. It is both an attack on secrecy and on the desired authentication guarantee for B.

Exercise 7.1 Suppose Alice wishes to send a secret message to Bob by an untrustworthy messager. Alice has her own padlock, the key to which only she possesses. Similarly, Bob has his own padlock and key. Alice also has a casket which is impregnable when padlocked. How can Alice send her secret message to Bob without revealing it to the untrustworthy messenger? □

7.1.1 Security properties

When we talk about secrecy we mean:

> "A message M is secret if it never appears unprotected on the network."

A common definition of authentication is an agreement property defined for instance by Lowe:

> "An initiator A agrees with a responder B on same message M if whenever A completes a run of the protocol as initiator, using M apparently with B, then there exists a run of the protocol where B acts as responder using M apparently with A."

7.2 SPL—a language for security protocols

In order to be more explicit about the activities of participants in a protocol and those of a possible attacker we design a little process language for the purpose. The language **SPL**(Security Protocol Language) is close to an asynchronous Pi-Calculus of Milner and is similar to the Spi-calculus of Abadi and Gordon.

7.2.1 The syntax of SPL

We start by giving the syntactic sets of the language:

- An infinite set of **Names**, with elements $n, m, \cdots, A, B, \cdots$.
- Variables over names $x, y, \cdots, X, Y, \cdots$.
- Variables over messages $\psi, \psi', \psi_1, \cdots$.
- Indices $i \in$ **Indices** with which to index components of parallel compositions.

The other syntactic sets of the language are described by the grammar shown in Figure 7.1. Note we use "vector" notation; for example, the "vector" $\vec{x}$ abbreviates some possibly empty list $x_1, \cdots, x_l$.

Name expressions	v	$::=$	$n, A, \cdots \mid x, X, \cdots$
Key expressions	k	$::=$	$Pub(v) \mid Priv(v) \mid Key(v_1, v_2)$
Messages	M	$::=$	$v \mid k \mid M_1, M_2 \mid \{M\}_k \mid \psi$
Processes	p	$::=$	$out\ new\vec{x}M.p \mid$
			$in\ pat\vec{x}\vec{\psi}M.p \mid$
			$\Vert_{i\in I}p_i$

Figure 7.1: Syntax of **SPL**

We take $fv(M)$, the free variables of a a message M, to be the set of variables which appear in M, and define the free variables of process terms by:

$$\begin{aligned} fv(out\ new\vec{x}M.p) &= (fv(p) \cup fv(M))\backslash\{\vec{x}\} \\ fv(in\ pat\vec{x}\vec{\psi}M.p) &= (fv(p) \cup fv(M))\backslash\{\vec{x}, \vec{\psi}\}) \\ fv(\Vert_{i\in I}p_i) &= \textstyle\bigcup_{i\in I} fv(p_i) \end{aligned}$$

As usual, we say that a process without free variables is closed, as is a message without variables. We shall use standard notation for substitution into the free variables of an expression, though we will only be concerned with the substitution of names or closed (variable-free) messages, obviating the problems of variable capture.

We use $Pub(v)$, $Priv(v)$ for the public, private keys of v, and $Key(v_1, v_2)$ for the symmetric key of v_1 and v_2. Keys can be used in building up encrypted messages. Messages may consist of a name or a key, be the composition of two messages (M_1, M_2), or an encrypted message $\{M\}_k$ representing the message M encrypted using the key k.

An informal explanation of the language:

$out\ new\vec{x}M.p$ This process chooses fresh, distinct names $\vec{n}$ and binds them to the variables $\vec{x}$. The message $M[\vec{n}/\vec{x}]$ is output to the network and the process resumes as $p[\vec{n}/\vec{x}]$. The communication is *asynchronous* in the sense that the action of output does not await input.

The *new* construct abstracts out an important property of a value chosen randomly from some large set; such a value is likely to be new.

$in\ pat\vec{x}\vec{\psi}M.p$ This process awaits an input that matches the pattern M for some binding of the pattern variables $\vec{x}\vec{\psi}$ and resumes as p under this binding. All the pattern variables $\vec{x}\vec{\psi}$ must appear in the pattern M.

$\Vert_{i\in I}p_i$ This process is the parallel composition of all components p_i for i in the indexing set I. The set I is a subset of **Indices**. Indices will help us distinguish in what agent and what run a particular action occurs. The nil process, written *nil*, abbreviates the empty parallel composition (where the indexing set is empty).

Convention 7.2 It simplifies the writing of process expressions if we adopt some conventions.

First, we simply write

$$out\ M.p$$

when the list of "*new*" variables is empty.

Secondly, and more significantly, we allow ourselves to write

$$\cdots in\ M.p \cdots$$

in an expression, to be understood as meaning the expression

$$\cdots in\ pat \vec{x}\vec{\psi}M.p \cdots$$

where the pattern variables $\vec{x}, \vec{\psi}$ are precisely those variables left free in M by the surrounding expression. For example, we can describe a responder in NSL as the process

$$\begin{array}{rcl} Resp(B) & \equiv & in\{x, Z\}_{Pub(B)}. \\ & & out\ new\ y\ \{x, y, B\}_{Pub(Z)}. \\ & & in\ \{y\}_{Pub(B)}. \\ & & nil \end{array}$$

For the first input, the variables x, Z in $\{x, Z\}_{Pub(B)}$ are free in the whole expression, so by convention are pattern variables, and we could instead write

$$in\ pat\ x, Z\ \{x, Z\}_{Pub(B)}.\cdots.$$

On the other hand, in the second input the variable y in $\{y\}_{Pub(B)}$ is bound by the outer *new* y $\cdots$ and so by the convention is not a pattern variable, and has to be that value sent out earlier.

Often we won't write the *nil* process explicitly, so, for example, omitting its mention at the end of the responder code above.

A parallel composition can be written in infix form via the notation

$$p_1 \| p_2 \cdots \| p_r \equiv \|_{i \in \{1, \cdots, r\}} p_i\ .$$

Replication of a process, $!p$, abbreviates $\|_{i \in \omega} p$, consisting of a countably infinite copies of p set in parallel.

The set of names of a process is defined to be all the names that appear in its subterms and submessages (even under encryption).

Note that the language permits the coding of privileged agents such as key servers, whose power is usually greater than one would anticipate of an attacker. As an extreme, here is code for a key-server gone mad, which repeatedly outputs private keys unprotected onto the network on receiving a name as request

$$!(in\ X.\ out\ Priv(X).nil)\ .$$

$$
\begin{aligned}
Init(A,B) \quad &\equiv \quad out\ new\ x\{x,A\}_{Pub(B)}. \\
&\qquad in\ \{x,y,B\}_{Pub(A)}. \\
&\qquad out\{y\}_{Pub(B)}. \\
&\qquad nil \\[1em]
Resp(B) \quad &\equiv \quad in\{x,Z\}_{Pub(B)}. \\
&\qquad out\ new\ y\ \{x,y,B\}_{Pub(Z)}. \\
&\qquad in\ \{y\}_{Pub(B)}. \\
&\qquad nil
\end{aligned}
$$

Figure 7.2: Initiator and responder code

$$
\begin{array}{lcll}
Spy_1 & \equiv & in\ \psi_1.in\ \psi_2.out(\psi_1,\psi_2).nil & \text{(composing)} \\
Spy_2 & \equiv & in(\psi_1,\psi_2).out\ \psi_1.out\ \psi_2.nil & \text{(decomposing)} \\
Spy_3 & \equiv & in\ x.in\ \psi.out\ \{\psi\}_{Pub(x)}.nil & \text{(encryption)} \\
Spy_4 & \equiv & in\ Priv(x).in\ \{\psi\}_{Pub(x)}.out\ \psi.nil & \text{(decryption)} \\
 & & & \\
Spy & \equiv & \|_{i\in\{1,\ldots,4\}} Spy_i &
\end{array}
$$

Figure 7.3: Attacker code

7.2.2 NSL as a process

We can program the NSL protocol in the language **SPL**, and so formalise the introductory description given in the Section 7. We assume given a set of agent names, **Agents**, of agents participating in the protocol. The agents participating in the NSL protocol play two roles, as initiator and responder with any other agent. Abbreviate by $Init(A,B)$ the program of initiator $A \in$ **Agents** communicating with $B \in$ **Agents** and by $Resp(B)$ the program of responder $B \in$ **Agents**. The code of both an arbitrary initiator and an arbitrary responder is given in Figure 7.2. In programming the protocol we are forced to formalise aspects that are implicit in the informal description, such as the creation of new nonces, the decryption of messages and the matching of nonces.

We can model the attacker by directly programming it as a process. Figure 7.3, shows a general, active attacker or "spy". The spy has the capability of composing eavesdropped messages, decomposing composite message, and using cryptography whenever the appropriate keys are available; the available keys are all the public keys and the leaked private keys. By choosing a different program for the spy we can restrict or augment its power, *e.g.*, to passive eavesdropping or active falsification.

The whole system is obtained by putting all components in parallel. Components are replicated, to model multiple concurrent runs of the protocol. The system is described in Figure 7.4.

$$
\begin{array}{lcl}
P_{init} & \equiv & \|_{A,B} \; ! \; Init(A,B) \\
P_{resp} & \equiv & \|_{A} \; ! \; Resp(A) \\
P_{spy} & \equiv & ! \; Spy \\
 & & \\
NSL & \equiv & \|_{i \in \{resp,init,spy\}} \, P_i
\end{array}
$$

Figure 7.4: The NSL process

Induction on size

Often definitions and proofs by structural induction won't suffice, and we would like to proceed by induction on the "size" of closed process expressions, *i.e.*, on the number of prefix and parallel composition operations in the process expression. But because of infinite parallel compositions, expressions may not contain just a finite number of operations, so we rely on the following well-founded relation.

Definition: Define the *size relation* on closed process terms:

$p[\vec{n}/\vec{x}] \prec (out\ new\vec{x}M.p)$,

for any substitution of names $\vec{n}/\vec{x}$.

$p[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}] \prec (in\ pat\vec{x}\vec{\psi}M.p)$,

for any substitution of names $\vec{n}/\vec{x}$, and closed messages $\vec{L}/\vec{\psi}$.

$p_j \prec (\|_{i \in I} p_i)$, for any index $j \in I$.

Proposition 7.3 *The size relation $\prec$ is well-founded.*

Proof: Let $\sqsubset_1$ denote the subexpression relation between process expressions; so $p' \sqsubset_1 p$ iff p' is an immediate subexpression of p. Then

$$q' \prec q \iff q' \equiv p_0[\sigma] \ \&\ p_0 \sqsubset_1 q$$

for some process expression p_0 and some substitution σ making q' a closed substitution instance of p_0. A simple structural induction on q shows that

$$p \sqsubset_1 q[\sigma] \Rightarrow \exists q' \sqsubset_1 q.\ p \equiv q'[\sigma] \ .$$

Hence any infinite descending $\prec$-chain would induce an infinite descending $\sqsubset_1$-chain, and so a contradiction. $\square$

As we go down with respect to the relation $\prec$ so does the "size" of the closed term in the sense that less prefix and parallel operations appear in the closed term. This cannot go on infinitely. We shall often do proofs by well-founded induction on the relation $\prec$ which we will call "induction on the size" of closed terms.[2]

[2]Alternatively, using ordinals, we could define the size measure $size(p)$ of a process term p to be an ordinal measuring the height of the number of process operations in a term, *e.g.*

$$size(out\ new\vec{x}M.p) = 1 + size(p), \quad size((\|_{i \in I} p_i) = 1 + sup_{i \in I} size(p_i) \ .$$

(output) Provided the names $\vec{n}$ are all distinct and not in s,

$$\langle out\ new\ \vec{x}M.p, s, t\rangle \stackrel{out\ new\ \vec{n}M[\vec{n}/\vec{x}]}{\longrightarrow} \langle p[\vec{n}/\vec{x}], s \cup \{\vec{n}\}, t \cup \{M[\vec{n}/\vec{x}]\}\rangle$$

(input) Provided $M[\vec{n}/\vec{x}, \vec{N}/\vec{\psi}] \in t$,

$$\langle in\ pat\vec{x}\vec{\psi}M.p, s, t\rangle \stackrel{in\ M[\vec{n}/\vec{x}, \vec{N}/\vec{\psi}]}{\longrightarrow} \langle p[\vec{n}/\vec{x}, \vec{N}/\vec{\psi}], s, t\rangle$$

(par)

$$\frac{\langle p_j, s, t\rangle \stackrel{\alpha}{\longrightarrow} \langle p'_j, s', t'\rangle}{\langle \|_{i\in I} p_i, s, t\rangle \stackrel{j:\alpha}{\longrightarrow} \langle \|_{i\in I} p_i[p'_j/j], s', t'\rangle}\ j \in I$$

Figure 7.5: Transition semantics

7.2.3 A transition semantics

A *configuration* consists of a triple

$$\langle p, s, t\rangle$$

where p is a closed process term, s is a subset of the set of names **Names**, and t is a subset of closed (*i.e.*, variable-free) messages. We say the configuration is *proper* iff the names in p and t are included in s. The idea is that a closed process p acts in the context of the set of names s that have been used so far, and the set of messages t which have been output, to input a message or to generate new names before outputting a message.

Actions α may be inputs or new-outputs, possibly tagged by indices to show at which parallel component they occur:

$$\alpha ::= out\ new\ \vec{n}.M \mid in\ M \mid i : \alpha$$

where M is a closed message, $\vec{n}$ are names and i is an index drawn from **Indices**. We write *out* M for an output action, outputting a message M, where no new names are generated.

How configurations evolve is expressed by transitions

$$\langle p, s, t\rangle \stackrel{\alpha}{\longrightarrow} \langle p', s', t'\rangle\ ,$$

given by the rules displayed in Figure 7.5.

The transition semantics allows us to state formally many security properties. It does not support directly local reasoning of the kind one might wish to apply in the analysis of security protocols. To give an idea of the difficulty, imagine we wished to establish that the nonce generated by B as responder in

NSL was never revealed as an open message on the network. More exactly:

Secrecy of responder's nonce:

> Suppose $Priv(A), Priv(B)$ do not appear as the contents of any message in t_0. For all runs
>
> $$\langle NSL, s_0, t_0\rangle \xrightarrow{\alpha_1} \cdots \langle p_{r-1}, s_{r-1}, t_{r-1}\rangle \xrightarrow{\alpha_r} \cdots ,$$
>
> where $\langle NSL, s_0, t_0\rangle$ is proper, if α_r has the form $resp : B : j : out\ new\ n\ \{m, n, B\}_{Pub(A)}$, then $n \notin t_l$, for all $l \in \omega$.

A reasonable way to prove such a property is to find a stronger invariant, a property which can be shown to be preserved by all the reachable actions of the process. Equivalently, one can assume that there is an earliest action α_l in a run which violates the invariant, and derive a contradiction by showing that this action must depend on a previous action, which itself violates the invariant.

An action might depend on another action through being, for example, an input depending on a previous output, or simply through occurring at a later control point in a process. A problem with the transition semantics is that it masks such local dependency, and even the underlying process events on which the dependency rests. The wish to support arguments based on local dependency leads to an event-based semantics.

7.3 A net from SPL

In obtaining an event-based semantics, we follow the lead from Petri nets, and define events in terms of how they affect conditions. We can discern three kinds of conditions: *control*, *output* and *name* conditions.

The set of *control conditions* **C** consists of output or input processes, perhaps tagged by indices, and is given by the grammar

$$b ::= out\ new\ \vec{x}M.p \mid in\ pat\vec{x}\vec{\psi}M.p \mid i : b$$

where $i \in$ **Indices**. A condition in **C** stands for the point of control in a process. When C is a subset of control conditions we will write $i : C$ to mean $\{i : b \mid b \in C\}$.

The set of *network conditions* **O** consists of closed message expressions. An individual condition M in **O** stands for the message M having been output on the network. Network conditions are assumed to be *persistent*; once they are made to hold they continue to hold forever.

The set of *name conditions* is precisely the set of names **Names**. A condition n in **S** stands for the name n being in use.

We define the *initial conditions* of a closed process term p, to be the subset

$Ic(p)$ of $\mathbf{C}$, given by the following induction on the size of p:

$$Ic(out\ new\ \vec{x}M.p) = \{out\ new\ \vec{x}M.p\}$$
$$Ic(in\ pat\vec{x}\vec{\psi}M.p) = \{in\ pat\vec{x}\vec{\psi}M.p\}$$
$$Ic(\|_{i\in I}p_i) = \bigcup_{i\in I} i : Ic(p_i)$$

where the last case also includes the base case nil, when the indexing set is empty.

We will shortly define the set of *events* **Events** as a subset of

$$\mathcal{P}ow(\mathbf{C}) \times \mathcal{P}ow(\mathbf{O}) \times \mathcal{P}ow(\mathbf{S}) \times \mathcal{P}ow(\mathbf{C}) \times \mathcal{P}ow(\mathbf{O}) \times \mathcal{P}ow(\mathbf{S})\ .$$

So an individual event $e \in$ **Events** is a tuple

$$e = ({}^c e, {}^o e, {}^s e, e^c, e^o, e^s)$$

where ${}^c e$ is the set of $\mathbf{C}$-preconditions of e, e^c is the set of $\mathbf{C}$-postconditions of e, etc. Write ${}^\cdot e$ for ${}^c e \cup {}^o e \cup {}^s e$, all preconditions of e, and $e^\cdot$ for all postconditions $e^c \cup e^o \cup e^s$. Thus an event will be determined by its effect on conditions.

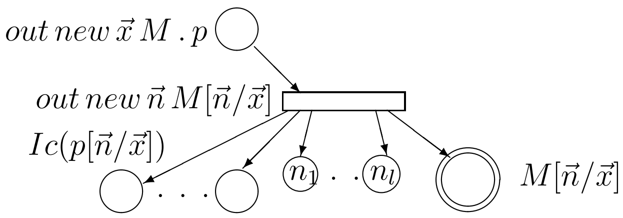
Earlier in the transition semantics we used actions α to specify the nature of transitions. An event e is associated with a unique action $act(e)$, though carry more information.

The set of events associated with **SPL** is given by an inductive definition. Define **Events** to be the smallest set which includes all

- *output events* $\mathbf{Out}(out\ new\ \vec{x}M.p; \vec{n})$, where $\vec{n} = n_1, \cdots, n_l$ are *distinct* names to match the variables $\vec{x} = x_1, \cdots, x_l$, consists of an event e with these pre- and postconditions:

  $${}^c e = \{out\ new\ \vec{x}M.p\}\ , \quad {}^o e = \emptyset\ , \quad {}^s e = \emptyset\ ,$$
  $$e^c = Ic(p[\vec{n}/\vec{x}])\ , \quad e^o = \{M[\vec{n}/\vec{x}]\} \quad e^s = \{n_1, \cdots, n_l\}\ .$$

  The *action* of an output event $act(\mathbf{Out}(out\ new\ \vec{x}M.p; \vec{n}))$ is $out\ new\ \vec{n}\ M[\vec{n}/\vec{x}]$.



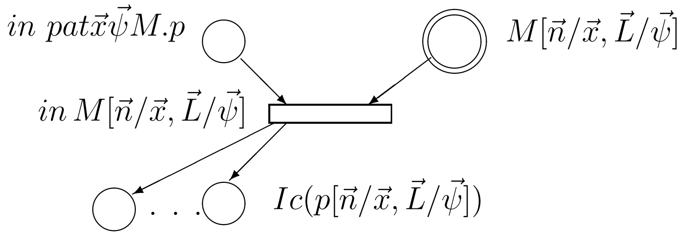
  An occurrence of the output event $\mathbf{Out}(out\ new\ \vec{x}M.p; \vec{n})$ affects the control conditions and puts the new names $n_1, \cdots, n_l$ into use, necessarily for the first time as according to the token game the event occurrence must avoid contact with names already in use.

The definition includes the special case when $\vec{x}$ and $\vec{n}$ are empty lists, and we write $\mathbf{Out}(out\ M.p)$ for the output event with no name conditions and action $out\ M$.

- *input events* $\mathbf{In}(in\ pat\vec{x}\vec{\psi}M.p; \vec{n}, \vec{L})$, where $\vec{n}$ is a list of names to match $\vec{x}$ and $\vec{L}$ is a list of closed messages to match $\vec{\psi}$, consists of an event e with these pre- and postconditions:

$$
\begin{aligned}
&{}^{c}e = \{in\ pat\vec{x}\vec{\psi}M.p\}\ , \quad {}^{o}e = \{M[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}]\}\ , \quad {}^{s}e = \emptyset\ , \\
&e^{c} = Ic(p[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}])\ , \quad e^{o} = \emptyset\ , \quad e^{s} = \emptyset\ .
\end{aligned}
$$

The *action* of an input event $act(\mathbf{In}(in\ pat\vec{x}\vec{\psi}M.p; \vec{n}, \vec{L}))$ is $in\ M[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}]$.



- *indexed events* $i : e$ where $e \in \mathbf{Events}$, where $i \in \mathbf{Indices}$ and

$$
\begin{aligned}
&{}^{c}(i:e) = i :{}^{c} e\ , \quad {}^{o}(i:e) = {}^{o} e\ , \quad {}^{s}(i:e) = {}^{s} e\ , \\
&(i:e)^{c} = i : e^{c}\ , \quad (i:e)^{o} = e^{o}\ , \quad (i:e)^{s} = e^{s}\ .
\end{aligned}
$$

The *action* of an indexed event $act(i:e)$ is $i:\alpha$, where α is the action of e.

When E is a subset of events we will generally use $i : E$ to mean $\{i : e \mid e \in E\}$.

Having specified its conditions and events, we have now defined a (rather large) net from the syntax of **SPL**. Its behaviour is closely related to the earlier transition semantics.

7.4 Relating the net and transition semantics

The **SPL**-net has conditions $\mathbf{C} \cup \mathbf{O} \cup \mathbf{S}$ and events $\mathbf{Events}$. Its markings $\mathcal{M}$ will be subsets of conditions and so of the form

$$\mathcal{M} = c \cup s \cup t$$

where $c \subseteq \mathbf{C}$, $s \subseteq \mathbf{S}$, and $t \subseteq \mathbf{O}$. The set of conditions $\mathbf{O}$ are persistent and determine the following token game.

Letting $c \cup s \cup t$ and $c' \cup s' \cup t'$ be two markings, $c \cup s \cup t \xrightarrow{e} c' \cup s' \cup t'$ iff

$$\begin{array}{l} \text{the event } e \text{ has concession,} \\ {}^{\cdot}e \subseteq c \cup s \cup t \ \& \ e^c \cap c = \emptyset \ \& \ e^s \cap s = \emptyset \ , \\ \text{and} \\ c' = (c \setminus {}^c e) \cup e^c \ \& \ s' = s \cup e^s \ \& \ t' = t \cup e^o \ . \end{array}$$

In particular, the occurrence of e begins the holding of its name postconditions e^s—these names have to be distinct from those already in use to avoid contact.

It turns out that all the markings reached in the behaviour of processes will have the form

$$\mathcal{M} = Ic(p) \cup s \cup t$$

for some closed process term p, names s and network conditions and t. There will be no contact at control conditions, throughout the reachable behaviour of the net, by the following.

Proposition 7.4 *Let p be a closed process term. Let $e \in$ **Events**. Then,*

$${}^c e \subseteq Ic(p) \Rightarrow e^c \cap Ic(p) = \emptyset \ .$$

Proof: By induction on the size of p. □

The behaviour of the **SPL**-net is closely related to the transition semantics given earlier.

Theorem 7.5

(1) If

$$\langle p, s, t \rangle \xrightarrow{\alpha} \langle p', s', t' \rangle \ ,$$

then

$$Ic(p) \cup s \cup t \xrightarrow{e} Ic(p') \cup s' \cup t'$$

*in the **SPL**-net, for some $e \in$ **Events** with $act(e) = \alpha$.*

(2) If

$$Ic(p) \cup s \cup t \xrightarrow{e} \mathcal{M}' \ ,$$

then

$$\langle p, s, t \rangle \xrightarrow{act(e)} \langle p', s', t' \rangle \quad \textit{and} \quad \mathcal{M}' = Ic(p') \cup s' \cup t' \ ,$$

*for some closed process p', $s' \subseteq$ **S** and $t' \subseteq$ **O**.*

Proof: Both (1) and (2) are proved by induction on the size of p.

(1)

Consider the possible forms of the closed process term p.

Case $p \equiv out\ new\ \vec{x}M.q$:

Assuming $\langle p, s, t\rangle \xrightarrow{\alpha} \langle p', s', t'\rangle$, there must be distinct names $\vec{n} = n_1, \cdots, n_l$, not in s, for which $\alpha = out\ new\ \vec{n}.M[\vec{n}/\vec{x}]$ and $p' \equiv q[\vec{n}/\vec{x}]$.

The initial conditions $Ic(p)$ form the singleton set $\{p\}$. The output event

$$e = \mathbf{Out}(out\ new\ \vec{x}M.q; \vec{n})$$

is enabled at the marking $\{p\} \cup s \cup t$, its action is α, and

$$Ic(p) \cup s \cup t \xrightarrow{e} Ic(q[\vec{n}/\vec{x}]) \cup s' \cup t'\ .$$

Case $p \equiv in\ pat\vec{x}\vec{\psi}M.q$:

The argument is very like that above for the output case.

Case $p \equiv \|_{i\in I} p_i$:

Assuming $\langle p, s, t\rangle \xrightarrow{\alpha} \langle p', s', t'\rangle$, there must be $\langle p_j, s, t\rangle \xrightarrow{\beta} \langle p'_j, s', t'\rangle$, with $\alpha = j : \beta$ and $p' \equiv \|_{i\in I} p'_i$, where $p'_i = p_i$ whenever $i \neq j$. Inductively,

$$Ic(p_j) \cup s \cup t \xrightarrow{e} Ic(p'_j) \cup s' \cup t'\ ,$$

for some event e such that $act(e) = \beta$. It is now easy to check that

$$Ic(p) \cup s \cup t \xrightarrow{j:e} Ic(\|_{i\in I} p'_i) \cup s' \cup t'\ .$$

(2)

Consider the possible forms of the closed process term p.

Case $p \equiv out\ new\ \vec{x}M.q$:

Assume that

$$Ic(p) \cup s \cup t \xrightarrow{e} \mathcal{M}'.$$

Note that $Ic(p) = \{p\}$. By the definition of **Events**, the only possible events with concession at $\{p\} \cup s \cup t$, are ones of the form

$$e = \mathbf{Out}(out\ new\ \vec{x}M.q; \vec{n})\ ,$$

for some choice of distinct names $\vec{n}$ not in s. The occurrence of e would make $s' = s \cup \{\vec{n}\}$ and $t' = t \cup \{M[\vec{n}/\vec{x}]\}$. Clearly, from the transition semantics,

$$\langle p, s, t\rangle \xrightarrow{act(e)} \langle q[\vec{n}/\vec{x}], s', t'\rangle\ .$$

Case $p \equiv in\ pat\vec{x}\vec{\psi}M.q$: The argument is like that for the output case.

Case $p \equiv \|_{i\in I} p_i$:

Assume that

$$Ic(p) \cup s \cup t \xrightarrow{e} c' \cup s' \cup t' \ ,$$

for $c' \subseteq \mathbf{C}$, $s' \subseteq \mathbf{S}$, and $t' \subseteq \mathbf{O}$.

From the token game and by the definition of **Events**, the event e can only have the form $e = j : e'$, where

$$Ic(p_j) \cup s \cup t \xrightarrow{e'} c'_j \cup s' \cup t'$$

and

$$c' = \bigcup_{i \neq j} Ic(p_i) \cup j : c'_j \ .$$

Inductively,

$$\langle p_j, s, t\rangle \xrightarrow{act(e')} \langle p'_j, s', t'\rangle \ \text{ and } \ c'_j = Ic(p'_j) \ ,$$

for some closed process p'_j. Thus, according to the transition semantics,

$$\langle p, s, t\rangle \xrightarrow{act(e)} \langle \|_{i\in I} p'_i, s', t'\rangle \ ,$$

where $p'_i = p_i$ whenever $i \neq j$. Hence, $c' = Ic(\|_{i\in I} p'_i)$. □

Definition: Let $e \in$ **Events**. Let p be a closed process term, $s \subseteq \mathbf{S}$, and $t \subseteq \mathbf{O}$. Write

$$\langle p,\ s,\ t\rangle \xrightarrow{e} \langle p',\ s',\ t'\rangle$$

iff

$$Ic(p) \cup s \cup t \xrightarrow{e} Ic(p') \cup s' \cup t'$$

in the **SPL**-net.

7.5 The net of a process

The **SPL**-net is awfully big of course. Generally for a process p only a small subset of the events **Events** can ever come into play. For this reason it's useful to restrict the events to those reachable in the behaviour of a process.

The events $Ev(p)$ of a closed process term p are defined by induction on size:

$$
\begin{aligned}
Ev(out\ new\ \vec{x}M.p) = &\{\mathbf{Out}(out\ new\ \vec{x}M.p; \vec{n}) \mid \vec{n} \text{ distinct names}\} \\
&\cup \bigcup \{Ev(p[\vec{n}/\vec{x}]) \mid \vec{n} \text{ distinct names}\}
\end{aligned}
$$

$$
\begin{aligned}
Ev(in\ pat\vec{x}\vec{\psi}M.p) = &\{\mathbf{In}(in\ pat\vec{x}\vec{\psi}M.p; \vec{n}, \vec{L}) \mid \vec{n} \text{ names}, \vec{L} \text{ closed messages}\} \\
&\cup \bigcup \{Ev(p[\vec{n}/\vec{x}, \vec{L}/\vec{\psi}]) \mid \vec{n} \text{ names}, \vec{L} \text{ closed messages}\}
\end{aligned}
$$

$$
Ev(\|_{i\in I} p_i) = \bigcup_{i\in I} i : Ev(p_i) \ .
$$

A closed process term p denotes a net $Net(p)$ consisting of the global set of conditions $\mathbf{C}\cup\mathbf{O}\cup\mathbf{S}$ built from **SPL**, events $Ev(p)$ and initial control conditions $Ic(p)$.

The net $Net(p)$ is open to the environment at its **O**- and **S**-conditions; the occurrence of events is also contingent on the output and name conditions that hold in a marking. The net of a closed process term $\|_{i\in I} p_i$ is the net with initial conditions $\bigcup_{i\in I} i : Ic(p_i)$ and events $\bigcup_{i\in I} i : Ev(p_i)$. We can view this as a parallel composition of the nets for p_i, $i \in I$; only the control conditions of different components are made disjoint so the components' events affect one another through the name and output conditions that they have in common. We can define the token game on the net $Net(p)$ exactly as we did earlier for the **SPL**-net, but this time events are restricted to being in the set $Ev(p)$.

It's clear that if an event transition is possible in the restricted net $Net(p)$ then so is it in the **SPL**-net. The converse also holds provided one starts from a marking whose control conditions either belong to $Ic(p)$ or are conditions of events in $Ev(p)$.

Definition: Let p be a closed process term. Define the *control-conditions of* p to be

$$
p^c = Ic(p) \cup \bigcup \{e^c \mid e \in Ev(p)\} \ .
$$

Proposition 7.6 *Let p be a closed process term and $e \in$ **Events**. If ${}^c e \subseteq p^c$, then $e \in Ev(p)$.*

Proof: By induction on the size of p. □

Lemma 7.7 *Let $\mathcal{M} \cap \mathbf{C} \subseteq p^c$. Let $e \in$ **Events**. Then,*

$$
\begin{aligned}
&\mathcal{M} \xrightarrow{e} \mathcal{M}' \ \textit{in the } \mathbf{SPL}\textit{-net} \\
&\textit{iff} \\
&e \in Ev(p) \ \&\ \mathcal{M} \xrightarrow{e} \mathcal{M}' \ \textit{in } Net(p) \ .
\end{aligned}
$$

Proof: *"if"*: Clear. *"only if"*: Clear by Proposition 7.6. □

Consequently, in analysing those sequences of event transitions

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots ,$$

a closed process p can perform, or correspondingly those of the transition semantics, it suffices to study the behaviour of $Net(p)$ with its restricted set of events $Ev(p)$. This simplification is especially useful in proving invariance properties which amount to an argument by cases on the form of events possible in the process.

Recall that we say a configuration $\langle p, s, t\rangle$ is *proper* iff the names in p and t are included in s.

Proposition 7.8 *Let $e \in$ **Events**. Suppose that $\langle p, s, t\rangle$ and $\langle p', s', t'\rangle$ are configurations, and that $\langle p, s, t\rangle$ is proper. If*

$$\langle p, s, t\rangle \xrightarrow{e} \langle p', s', t'\rangle ,$$

then $\langle p', s', t'\rangle$ is also proper.

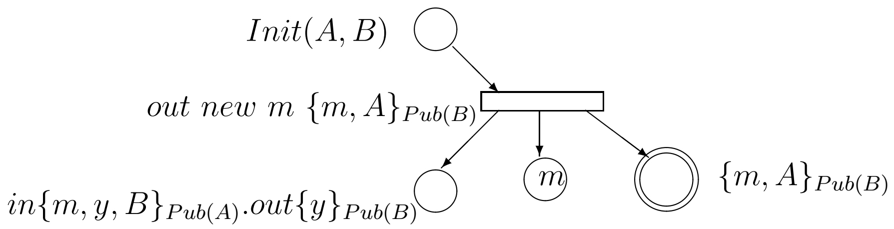
Proof: By considering the form of e. Input and output events are easily seen to preserve properness, and consequently indexed events do too. $\square$

Important convention: From now on we assume that all configurations $\langle p, s, t\rangle$ are proper. Notice, in particular, that in a proper configuration $\langle NSL, s, t\rangle$ the set s will include all agent names because all agent names are mentioned in NSL, the process describing the Needham-Schröder-Lowe protocol.

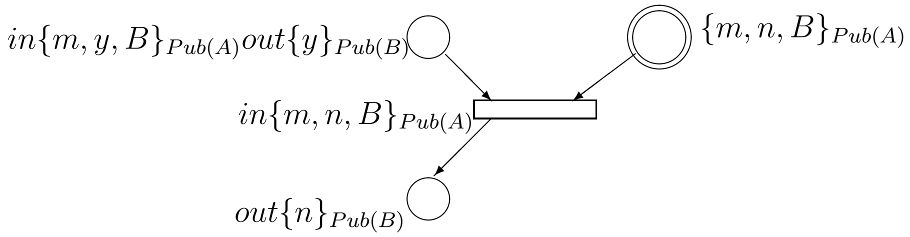
7.6 The events of NSL

We can classify the events $Ev(NSL)$ involved in the NSL protocol.
Initiator events:

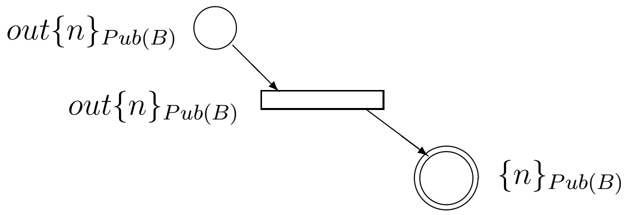
Out$(Init(A, B); m)$:



In$(in\{m, y, B\}_{Pub(A)}.out\{y\}_{Pub(B)}; n)$:

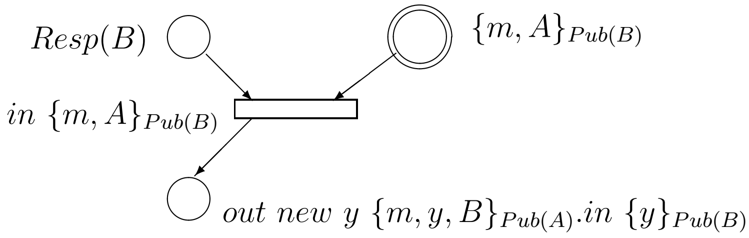


Out($out\{n\}_{Pub(B)}$):

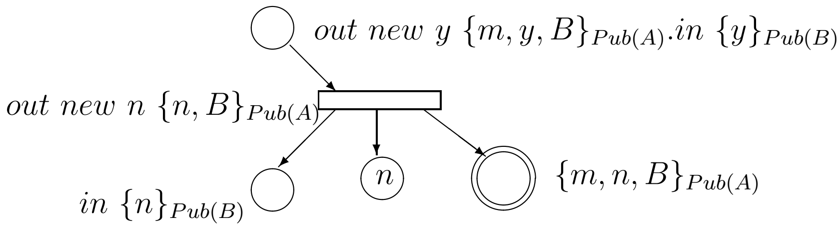


Responder events:

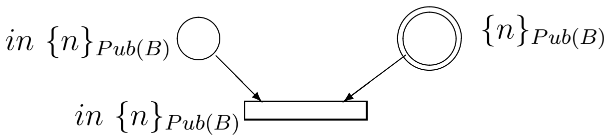
In($Resp(B); m, A$):



Out($out\ new\ y\ \{m, y, B\}_{Pub(A)}.in\ \{y\}_{Pub(B)}; n$):

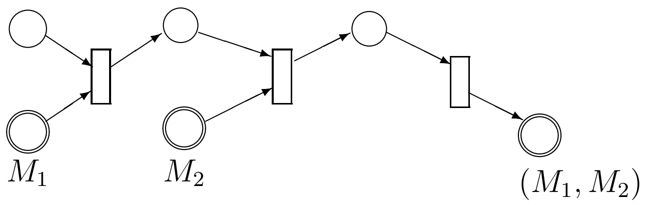


In($in\ \{n\}_{Pub(B)}$):

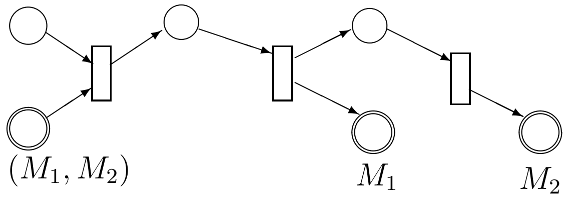


Spy events:

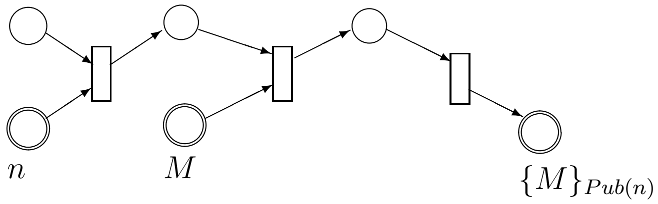
Composition, $Spy_1 \equiv in\ \psi_1.in\ \psi_2.out(\psi_1, \psi_2)$:



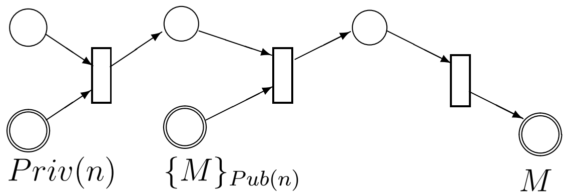
Decomposition, $Spy_2 \equiv in(\psi_1, \psi_2).out\ \psi_1.out\ \psi_2$:



Encryption, $Spy_3 \equiv in\ x.in\ \psi.out\ \{\psi\}_{Pub(x)}$:



Decryption, $Spy_4 \equiv in\ Priv(x).in\ \{\psi\}_{Pub(x)}.out\ \psi$:



7.7 Security properties for NSL

In this section we apply our framework to prove authentication and secrecy guarantees for the responder part of the NSL protocol.

7.7.1 Principles

Some principles are useful in proving authentication and secrecy of security protocols. Write $M \sqsubset M'$ to mean message M in a subexpression of message M'. More precisely, $\sqsubset$ is the smallest binary relation on messages such that

$$
\begin{aligned}
&M \sqsubset M \ , \\
&M \sqsubset N \Rightarrow M \sqsubset (N, N') \ \&\ M \sqsubset (N', N) \ , \\
&M \sqsubset N \Rightarrow M \sqsubset \{N\}_k \ .
\end{aligned}
$$

We also write $M \sqsubset t$ iff $\exists M' .\, M \sqsubset M'\ \&\ M' \in t$, for a set of messages t.

Proposition 7.9 *(Well-foundedness) Given a property $\mathcal{P}$ on configurations, if a run*

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots ,$$

contains a configurations such that $\mathcal{P}(p_0, s_0, t_0)$ and $\neg\mathcal{P}(p_j, s_j, t_j)$, then there is an event e_h, $0 < h \leq j$, such that $\mathcal{P}(p_i, s_i, t_i)$ for all $i < h$ and $\neg\mathcal{P}(p_h, s_h, t_h)$.

We say that a name $m \in$ **Names** is fresh on an event e if $m \in e^s$ and we write $Fresh(m, e)$.

Proposition 7.10 *(Freshness) Within a run*

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots ,$$

the following properties hold:

1. *If $n \in s_i$ then either $n \in s_0$ or there is a previous event e_j such that $Fresh(n, e_j)$.*
2. *Given a name n there exists at most one event e_i such that $Fresh(n, e_i)$.*
3. *If $Fresh(n, e_i)$ then for all $j < i$ the name n does not appear in $\langle p_j, s_j, t_j\rangle$.*

Proposition 7.11 *(Control precedence) Within a run*

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots ,$$

if $b \in {}^c e_i$ either $b \in Ic(p_0)$ or there is an earlier event e_j, $j < i$, such that $b \in {e_j}^c$.

Proposition 7.12 *(Output-input precedence) In a run*

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots ,$$

if $M \in {}^o e_i$ either $M \in t_0$ or there is an earlier event e_j, $j < i$, such that $M \in {e_j}^o$.

7.7.2 Secrecy

The following lemma is an essential prerequisite in proving secrecy. In fact, it is a secrecy result in its own right.

Lemma 7.13 *Consider a run*

$$\langle NSL, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots ,$$

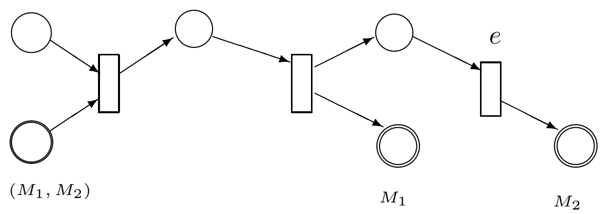
and consider an agent A_0. If $Priv(A_0) \not\sqsubset t_0$, then $Priv(A_0) \not\sqsubset t_l$ at all stages l.

Proof: Assume $Priv(A_0) \not\sqsubset t_0$. Suppose there were a stage in the run at which $Priv(A_0) \sqsubset t_l$. By well-foundedness, there would be an earliest event $e = e_r$ in the run at which $Priv(A_0) \not\sqsubset t_{r-1}$ and $Priv(A_0) \sqsubset t_r$.

We can show that such an earliest event e cannot exist by considering all the possible forms it might take. We need only consider indexed output events since only for such events do we have $e^o \neq \emptyset$.

An easy check shows that e cannot be an initiator or responder event.

It remains to consider spy events. We consider only one case—the other cases follow in a similar fashion. Suppose that the event e has the form $spy : i : 2 : \mathbf{Out}(out\, M_2)$ for some run index i with $Priv(A_0) \sqsubset M_2$. Then, by control precedence, there must be a preceding event which has as precondition the network condition (M_1, M_2). Clearly, $Priv(A_0) \sqsubset (M_1, M_2)$. As $Priv(A_0) \not\sqsubset t_0$, by output-input precedence, there must be an even earlier event than e that marked the condition (M_1, M_2).



□

Exercise 7.14 Complete the proof of Lemma 7.13, by considering the remaining cases of spy events. □

The following theorem shows that the nonce of a responder in the NSL protocol remains secret.

Theorem 7.15 *Consider a run*

$$\langle NSL, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r \rangle \xrightarrow{e_{r+1}} \cdots .$$

Suppose it contains an event e_r with

$$act(e_r) = resp : B_0 : j_0 : out\, new\, n_0\, \{m_0, n_0, B_0\}_{Pub(A_0)} \ ,$$

where j_0 is an index. Suppose that $Priv(A_0) \not\sqsubset t_0$ and $Priv(B_0) \not\sqsubset t_0$. Then, at all stages $n_0 \notin t_l$.

Proof: We prove a stronger invariant: At all stages l,

> for all messages $M \in t_l$, if $n_0 \sqsubset M$ then
> either $\{m_0, n_0, B_0\}_{Pub(A_0)} \sqsubset M$ or $\{n_0\}_{Pub(B_0)} \sqsubset M$.

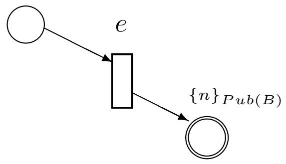
Because $Fresh(n_0, e_r)$, by freshness (Proposition 7.10) all configurations $\langle p_l, s_l, t_l \rangle$ where $l < r$ satisfy this invariant. In particular so does $\langle NSL, s_0, t_0 \rangle$. The proof is based on the well-foundedness principle. Suppose the invariant fails to hold at some stage. Then there is an earliest event e in the run that violates the invariant through having a network condition M as postcondition for which

$$n_0 \sqsubset M \ \& \ \{m_0, n_0, B_0\}_{Pub(A_0)} \not\sqsubset M \ \& \ \{n_0\}_{Pub(B_0)} \not\sqsubset M \ .$$

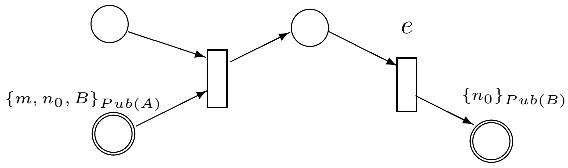
Assume there is such an earliest event e. We shall obtain a contradiction no matter what the form of e. Since indexed input events leave the network conditions unchanged, they cannot violate the invariant. It remains to consider indexed output events.

Initiator events. There are two cases:

Case 1. Assume $e = init : (A, B) : i : \mathbf{Out}(out\,\{n\}_{Pub(B)})$ for some index i:

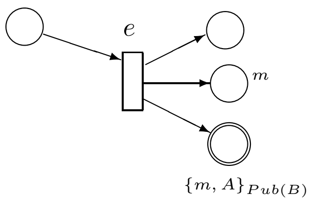


Since the event e is assumed to violate the invariant it must be the case that $n = n_0$ and $B_0 \neq B$. There exists a preceding event that marked e's control precondition. This condition determines the form of the preceding event:



Consider now the network condition $\{m, n_0, B\}_{Pub(A)}$. We know there exists an even earlier event that marked it, which thus violates the invariant (remember $B \neq B_0$). But this contradicts e being the earliest event to violate the invariant.[3]

Case 2. Assume the event $e = init : (A, B) : i : \mathbf{Out}(Init(A, B); m)$ for some index i. This event has the form:



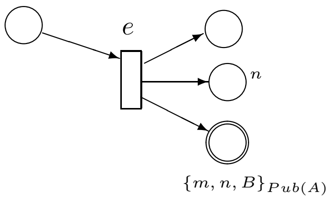
Since e violates the invariant, $n_0 \sqsubset \{m, A\}_{Pub(B)}$, so:

[3]At this point in the proof, the ingredient $B \neq B_0$ is crucial in showing that there is an earlier event violating the secrecy invariant. An analogous proof attempt for the original protocol of Needham and Schröder would fail here. For the original protocol we might try to establish the modified invariant: For all l, for all messages $M \in t_l$, if $n_0 \sqsubset M$ then either $\{m_0, n_0\}_{Pub(A_0)} \sqsubset M$ or $\{n_0\}_{Pub(B_0)} \sqsubset M$. However, at this point, we would be unable to deduce that there was an earlier event which violated this modified invariant.

- Either $m = n_0$. So $Fresh(e, n_0)$. Since e is an initiator event it is distinct from e_r and $Fresh(e_r, n_0)$, this contradicts freshness.

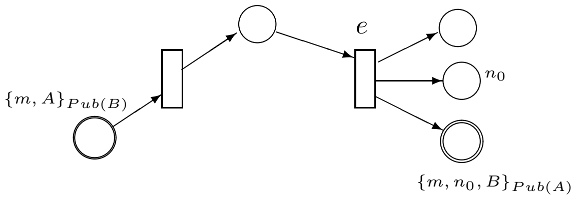
- Or $A = n_0$. But in this case, by the properness of the initial configuration $n_0 \in s_0$, which again contradicts freshness.

Responder events. There is only one form of event to consider. Assume $e = resp : B : i : \mathbf{Out}(out\, new\, y\, \{m, y, B\}_{Pub(A)}.in\{y\}_{Pub(B)}; n)$:



Since e violates the invariant, $n_0 \sqsubset \{m, n, B\}_{Pub(A)}$, so one of the following:

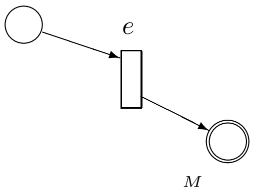
- $m = n_0$. There must then be an earlier event that marked the network condition $\{m, A\}_{Pub(B)}$ and thus violates the invariant. This contradicts the assumption that e is the earliest event to violate the invariant.



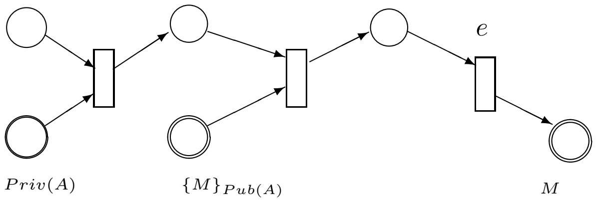
- $n = n_0$. Then since e violates the invariant, we must have $e \neq e_r$. We have $Fresh(e, n_0)$ and $Fresh(e_r, n_0)$ which contradicts freshness.

- The case $B = n_0$ is excluded because n_0 is fresh on e_r and so cannot be an agent name in t_0.

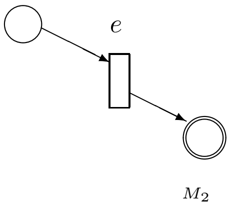
Spy events. We consider only some cases, the others follow in a similar way. Case 1. Assume the event $e = spy : 4 : i : \mathbf{Out}(outM)$ for some index i:



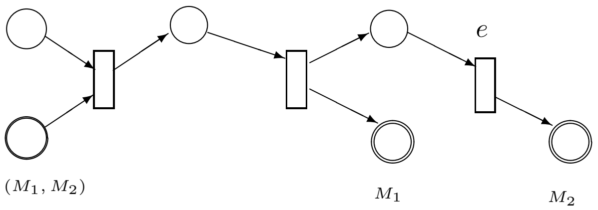
By precedence there is an earlier event that marked $\{M\}_{Pub(A)}$.



Lemma 7.13 guarantees that $Priv(A) \neq Priv(A_0)$ and $Priv(A) \neq Priv(B_0)$, so that $A \neq A_0$ and $A \neq B_0$. Therefore because e violates the invariant, $n_0 \sqsubset M$ with $\{m_0, n_0, B_0\}_{Pub(A_0)} \not\sqsubset \{M\}_{Pub(A)}$ and $\{n_0\}_{Pub(B_0)} \not\sqsubset \{M\}_{Pub(A)}$. This contradicts e being the earliest event to violate the invariant.
Case 2. Assume the event $e = spy : 2 : i : \mathbf{Out}(out\, M_2)$ for some index i:



By precedence there is an earlier event marking the persistent condition (M_1, M_2).



Even though e violates the invariant, it may still be that $\{m_0, n_0, B_0\}_{Pub(A_0)} \sqsubset M_1$ or $\{n_0\}_{Pub(B_0)} \sqsubset M_1$ (and this prevents us from immediately deducing that there is an even earlier event which violates the invariant by virtue of having (M_1, M_2) as a postcondition). However, in this situation only an earlier spy event can have marked a network condition with (M_1, M_2) as submessage. Consider the earliest such spy event e' in the sequence of transitions. A case analysis (Exercise!) of the possible events which precede e' always yields an earlier event which either violates the invariant or outputs a message with (M_1, M_2) as submessage. $\square$

Exercise 7.16 Complete the proof of secrecy, Theorem 7.15, by finishing off the case analysis marked "Exercise!" in the proof. $\square$

Exercise 7.17 (Big exercise: Secrecy of initiator's nonce)
Consider a run

$$\langle NSL, s_0, t_0 \rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r \rangle \xrightarrow{e_{r+1}} \cdots ,$$

containing the initiator event e_r where

$$act(e_r) = init : (A_0, B_0) : j_0 : out\, new\, m_0\, \{m_0, A_0\}_{Pub(B_0)}\ ,$$

with j_0 an index, and such that $Priv(A_0) \not\sqsubset t_0$ and $Priv(B_0) \not\sqsubset t_0$. Show that at all stages $m_0 \notin t_i$.
[Use the invariant: For all r and for all messages $M \in t_r$, if $m_0 \sqsubset M$ then either $\{m_0, A_0\}_{Pub(B_0)} \sqsubset M$ or $\{m_0, n_0, B_0\}_{Pub(A_0)} \sqsubset M$.] $\Box$

Simplifying the attacker events

We have described the possible events of an attacker as those of an **SPL** process *Spy*. We could, however, reduce the number of cases to consider and simplify proofs by describing the attacker events directly as having the following form—note all the conditions involved are network conditions and so persistent:

- Composition events: an event with two network conditions M_1 and M_2 as preconditions, and the network condition (M_1, M_2) as postcondition.

- Decomposition events: an event with an network condition network condition (M_1, M_2) as precondition, and the two network conditions M_1 and M_2 as postconditions.

- Encryption events: an event with network conditions M and a name n as preconditions, and the network condition $\{M\}_{Pub(n)}$ as postcondition.

- Decryption events: an event with the two network conditions $\{M\}_{Pub(n)}$ and a key $Priv(n)$ as preconditions, and the output condition M as postcondition.

Then, we can show security properties of an **SPL** process p by establishing properties of the net obtained by setting $Net(p)$ in parallel with all the attacker events.

7.7.3 Authentication

We will prove authentication for a responder in an NSL protocol in the sense that: To any complete session of agent B_0 as responder, apparently with agent A_0, there corresponds a complete session of agent A_0 as initiator.

In the proof it's helpful to make use of a form of diagrammatic reasoning which captures the precedence of events. When the run

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots$$

is understood we draw

$$e \longrightarrow e'$$

when e precedes e' in the run, allowing $e = e'$.

Theorem 7.18 (Authentication) *If a run of NSL*

$$\langle NSL, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots \ ,$$

contains the responder events b_1, b_2, b_3, with actions

$$\begin{aligned} act(b_1) &= resp : B_0 : i : \ in\, \{m_0, A_0\}_{Pub(B_0)} \ , \\ act(b_2) &= resp : B_0 : i : \ out\, new\, n_0\, \{m_0, n_0, B_0\}_{Pub(A_0)} \ , \\ act(b_3) &= resp : B_0 : i : \ in\{n_0\}_{Pub(B_0)} \ , \end{aligned}$$

for an index i, and $Priv(A_0) \not\sqsubseteq t_0$, then the run contains initiator events a_1, a_2, a_3 with $a_3 \longrightarrow b_3$, where, for some index j,

$$\begin{aligned} act(a_1) &= init : (A_0, B_0) : j : \ out\, new\, m_0\, \{m_0, A_0\}_{Pub(B_0)} \ , \\ act(a_2) &= init : (A_0, B_0) : j : \ in\{m_0, n_0, B_0\}_{Pub(A_0)} \ , \\ act(a_3) &= init : (A_0, B_0) : j : \ out\{n_0\}_{Pub(B_0)} \ . \end{aligned}$$

Proof: By control precedence we know that

$$b_1 \longrightarrow b_2 \longrightarrow b_3 \ .$$

Consider the property of configurations

$$Q(p, s, t) \Leftrightarrow \forall M \in t.\ n_0 \sqsubset M \Rightarrow \{m_0, n_0, B_0\}_{Pub(A_0)} \sqsubset M \ .$$

By freshness, the property Q holds immediately after b_2, but clearly not immediately before b_3. By well-foundedness there is a earliest event following b_2 but preceding b_3 that violates Q. Let e be such an event.

$$\begin{tikzcd} b_1 \arrow[r] & b_2 \arrow[rr] \arrow[dr] & & b_3 \\ & & e \arrow[ur] & \end{tikzcd}$$

Inspecting the events of the NSL protocol, in a similar way to the proof of secrecy, using the assumption that $Priv(A_0) \not\sqsubseteq t_0$, one can show (Exercise!) that e can only be an initiator event a'_3 with action

$$act(a'_3) = init : (A, B_0) : j : out\{n_0\}_{Pub(B_0)}$$

for some index j and agent A. There must also be preceding events a'_1, a'_2 with actions

$$\begin{aligned} act(a'_1) &= init : (A, B_0) : j : out\, new\, m\, \{m, A\}_{Pub(B_0)} \ , \\ act(a'_2) &= init : (A, B_0) : j : in\{m, n_0, B_0\}_{Pub(A)} \ . \end{aligned}$$

$$\begin{tikzcd} b_1 \arrow[r] & b_2 \arrow[rr] \arrow[dr] & & b_3 \\ a'_1 \arrow[r] & a'_2 \arrow[r] & a'_3 \arrow[ur] & \end{tikzcd}$$

Since $Fresh(b_2, n_0)$, the event b_2 must precede a'_2. The property Q holds on configurations up to a'_3 and, in particular, on the configuration immediately before a'_2. From this we conclude that $m = m_0$ and $A = A_0$. Hence $a'_3 = a_3$, $a'_2 = a_2$, and $a'_1 = a_1$ as described below.

$$\begin{tikzcd} b_1 \arrow[r] & b_2 \arrow[r] \arrow[d] & b_3 \\ a_1 \arrow[r] & a_2 \arrow[r] & a_3 \arrow[u] \end{tikzcd}$$

(Since $Fresh(a_1, m_0)$, the event a_1 precedes b_1.)

Exercise 7.19 Complete the proof of Theorem 7.18 at the point marked "Exercise!" in the proof. $\square$

As the proof of Theorem 7.18 suggests, authentication for a responder in NSL can be summarised in a diagram showing the dependency of the key events. For all NSL-runs in which events b_1, b_2, b_3 occur with actions

$$\begin{array}{l} act(b_1) = resp : B_0 : i : \ in\, \{m_0, A_0\}_{Pub(B_0)}\,, \\ act(b_2) = resp : B_0 : i : \ out\, new\, n_0\, \{m_0, n_0, B_0\}_{Pub(A_0)}\,, \\ act(b_3) = resp : B_0 : i : \ in\{n_0\}_{Pub(B_0)}\,, \end{array}$$

for an index i, and where $Priv(A_0)$ is not a submessage of any initial output message, there are events a_1, a_2, a_3 with actions

$$\begin{array}{l} act(a_1) = init : (A_0, B_0) : j : \ out\, new\, m_0\, \{m_0, A_0\}_{Pub(B_0)}\,, \\ act(a_2) = init : (A_0, B_0) : j : \ in\{m_0, n_0, B_0\}_{Pub(A_0)}\,, \\ act(a_3) = init : (A_0, B_0) : j : \ out\{n_0\}_{Pub(B_0)}\,, \end{array}$$

such that

$$\begin{tikzcd} b_1 \arrow[r] & b_2 \arrow[r] \arrow[d] & b_3 \\ a_1 \arrow[r] \arrow[u] & a_2 \arrow[r] & a_3. \arrow[u] \end{tikzcd}$$

In particular, the occurrence of the event b_3 depends on the previous occurrence of an event a_3 with action having the form above. Drawing such an event-dependency diagram, expressing the intended event dependencies in a protocol, can be a good preparation for a proof of authentication.

Exercise 7.20 (Big exercise: Authentication guarantee for initiator)
Consider a run of NSL

$$\langle NSL, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_r} \langle p_r, s_r, t_r\rangle \xrightarrow{e_{r+1}} \cdots\,,$$

containing the initiator events a_1, a_2, a_3 where

$$\begin{array}{lcl} act(a_1) & = & init : (A_0, B_0) : j : out\, new\, m_0\, \{m_0, A_0\}_{Pub(B_0)}\,, \\ act(a_2) & = & init : (A_0, B_0) : j : in\{m_0, n_0, B_0\}_{Pub(A_0)}\,, \\ act(a_3) & = & init : (A_0, B_0) : j : out\{n_0\}_{Pub(B_0)}\,, \end{array}$$

for an index j. Assume $Priv(A_0) \not\sqsubset t_0$ and $Priv(B_0) \not\sqsubset t_0$. Show the run contains responder events b_1, b_2 with $b_2 \longrightarrow a_3$, where, for some index i,

$$\begin{array}{rcl} act(b_1) & = & resp : B_0 : i : in\, \{m_0, A_0\}_{Pub(B_0)}\,, \\ act(b_2) & = & resp : B_0 : i : out\, new\, n_0\, \{m_0, n_0, B_0\}_{Pub(A_0)}\,. \end{array}$$

In addition, show that if $Priv(A_0) \in t_0$, then there is an attack violating the above authentication. $\square$

Chapter 8

Mobile processes

This chapter introduces a small but powerful language for higher-order nondeterministic processes. It is used to explain and unify a range of process languages. The language, HOPLA (Higher Order Process LAnguage), can be viewed as an extension of the lambda calculus with a prefix operation and nondeterministic sums, in which types express the form of computation path of which a process is capable. Its operational semantics, bisimulation, congruence properties and expressive power are investigated. The meaning and consequences of linearity, where a process argument is executed exactly once, are explored. It is shown how HOPLA can directly encode process languages such as CCS, CCS with value passing, process passing, and mobile ambients with public names. Finally it is indicated how HOPLA may be extended to include name generation.[1]

8.1 Introduction

We broaden our study to more general processes. A topic of current research is that of mobile processes. Mobility concerns the relocation of processes.

One form of mobility is achieved through treating a process as encapsulated like a value which moves from one context to another, perhaps executing as it moves. This idea is central to the Calculus of Mobile Ambients [1]. This kind of mobility can lead to abrupt changes in the process's environment. Such forms of mobility can perhaps always be realized through a mechanism for passing processes.

In another form of mobility the relocation of process is more virtual, and achieved through a change of contiguity of the process, by incrementally amending its access to processes in its environment. This is the form of mobility manifest in the Pi-Calculus [12]. Though often described as a calculus of mobile processes, it is perhaps more informative to view the the Pi-Calculus as one in which the communication links are mobile; the movement of a process is

[1]This chapter is based on joint work with Mikkel Nygaard.

achieved only through a change in the set of channel names it knows, so altering the processes it can access.

The field of calculi for mobility is very alive and unsettled and for this reason hard to summarise. However, a case can be made that almost all current process calculi centre on two mechanisms:[2]

- *Name generation*: the ability of processes to generate new names. The new names can be transmitted in a restricted way and so made into private channels of communication. The language SPL allows this indirectly through the generation of new keys. The Pi-Calculus is built directly around the dynamic creation of new channels to effect a change of process contiguity.
- *Higher-order processes:* the ability of processes to pass and receive processes as values. Higher-order processes are akin to terms of the lambda-calculus, but where processes rather than functions appear as arguments. A higher-order process can receive a process as an argument and yield a process as result. In this way a process can be passed from one context to another. Process passing is sometimes implicit as in the Ambient Calculus.

In this chapter we'll chiefly study higher-order processes and the kind of mobility they permit. To do this systematically we will use a single higher-order process language, HOPLA. Processes will have first-class status and will receive a type which describes the nature of their computations. It's possible to add types and terms to support name generation to HOPLA, to get "a language to do everything", but the result is harder to understand—the subject of current research!

8.2 A Higher-Order Process Language

The types of the language HOPLA are given by the grammar

$$\mathbb{P}, \mathbb{Q} ::= .\mathbb{P} \mid \mathbb{P} \to \mathbb{Q} \mid \Sigma_{a\in A} a\mathbb{P}_a \mid P \mid \mu_j \vec{P}\,\vec{\mathbb{P}} \ .$$

P is drawn from a set of type variables used in defining recursive types; $\mu_j \vec{P}.\vec{\mathbb{P}}$ abbreviates $\mu_j P_1, \ldots, P_k.(\mathbb{P}_1, \ldots, \mathbb{P}_k)$ and is interpreted as the j-component, for $1 \le j \le k$, of the recursive type $P_1 = \mathbb{P}_1, \ldots, P_k = \mathbb{P}_k$, in which the type expressions $\mathbb{P}_1, \ldots, \mathbb{P}_k$ may contain the P_j's. We shall write $\mu\vec{P}.\vec{\mathbb{P}}$ as an abbreviation for the k-tuple with j-component $\mu_j \vec{P}.\vec{\mathbb{P}}$. In a sum type $\Sigma_{a\in A} a\mathbb{P}_a$ the indexing set A may be an arbitrary countable set. When the set A is finite, so $A = \{a_1, \cdots, a_k\}$ we shall most often write the sum type as $a_1\mathbb{P}_{a_1} + \cdots + a_k\mathbb{P}_{a_k}$. Write $\mathbb{O}$ for the empty sum type. Henceforth by a process type we mean a closed type expression.

[2]Current research in "domain theory for concurrency" suggests however that a better understanding will uncover other fundamental mechanisms, some related to independence models.

The types $\mathbb{P}$ describe the computations processes can perform. A process of prefix type $.\mathbb{P}$ can do an action $.$ and then resume as a process doing computations of type $\mathbb{P}$; a process of function type $\mathbb{P} \to \mathbb{Q}$ is a higher-order process which given a process of type $\mathbb{P}$ as input yields a process of type $\mathbb{Q}$ as output; a process of sum type $\Sigma_{a\in A} a\mathbb{P}_a$ can perform computations in any of the types $\mathbb{P}_a$ but with the difference that, when it does so, its initial action is tagged by the component label a; a process of recursive type $\mu_j \vec{P}.\vec{\mathbb{P}}$ can do the same computations as the type obtained as the unfolding $\mathbb{P}_j[\mu_j \vec{P}.\vec{\mathbb{P}}/\vec{P}]$.

The raw syntax of terms:

$$t, u ::= x \mid rec\, x\, t \mid \Sigma_{i\in I} t_i \mid .t \mid [u > .x \Rightarrow t] \mid \lambda x\, t \mid t\ u \mid a\, t \mid \pi_a(t)$$

Here x is a process variable, which can be captured within a recursive definition $rec\, x\, t$, an abstraction $\lambda x\, t$, or a match term. The variable x in the prefix match term $[u > .x \Rightarrow t]$ is a binding occurrence and so binds later occurrences of the variable in the body t; intuitively, the variable x in the pattern $.x$ is matched with any possible resumption of u after doing an action $.$. In a nondeterministic sum, $\Sigma_{i\in I} t_i$, the indexing set I may be an arbitrary set; we write $t_1 + \cdots + t_k$ for a typical finite sum, and *nil* when I is empty. A prefix term $.t$ describes a process able to do the action $.$ and resume as the process t. A term $a\, t$ represents the injection of t into the a-component of a sum type $\Sigma_{a\in A} a\mathbb{P}_a$, while $\pi_a(t)$ is the projection of a term t of sum type to the a-component.

Let $\mathbb{P}_1, \ldots, \mathbb{P}_k, \mathbb{Q}$ be closed type expressions and assume that the variables $x_1, \ldots, x_k$ are distinct. A syntactic judgement $x_1 : \mathbb{P}_1, \ldots, x_k : \mathbb{P}_k \vdash t : \mathbb{Q}$ means that with processes of the alloted types assigned to the free variables, the term t performs computations of type $\mathbb{Q}$. We let Γ range over environment lists $x_1 : \mathbb{P}_1, \ldots, x_k : \mathbb{P}_k$, which we may treat as finite partial functions from variables to closed type expressions. The term formation rules are:

$$\frac{\Gamma(x) = \mathbb{P}}{\Gamma \vdash x : \mathbb{P}} \qquad \frac{\Gamma, x : \mathbb{P} \vdash t : \mathbb{P}}{\Gamma \vdash rec\, x\, t : \mathbb{P}} \qquad \frac{\Gamma \vdash t_j : \mathbb{P} \quad \text{all } j \in I}{\Gamma \vdash \Sigma_{i\in I} t_i : \mathbb{P}}$$

$$\frac{\Gamma \vdash t : \mathbb{P}}{\Gamma \vdash .t : .\mathbb{P}} \qquad \frac{\Gamma \vdash u : .\mathbb{P} \quad \Gamma, x : \mathbb{P} \vdash t : \mathbb{Q}}{\Gamma \vdash [u > .x \Rightarrow t] : \mathbb{Q}}$$

$$\frac{\Gamma, x : \mathbb{P} \vdash t : \mathbb{Q}}{\Gamma \vdash \lambda x\, t : \mathbb{P} \to \mathbb{Q}} \qquad \frac{\Gamma \vdash t : \mathbb{P} \to \mathbb{Q} \quad \Gamma \vdash u : \mathbb{P}}{\Gamma \vdash t\ u : \mathbb{Q}}$$

$$\frac{\Gamma \vdash t : \mathbb{P}_b \qquad b \in A}{\Gamma \vdash b\, t : \Sigma_{a\in A} a\mathbb{P}_a} \qquad \frac{\Gamma \vdash t : \Sigma_{a\in A} a\mathbb{P}_a \quad b \in A}{\Gamma \vdash \pi_b(t) : \mathbb{P}_b}$$

$$\frac{\Gamma \vdash t : \mathbb{P}_j[\mu \vec{P}\, \vec{\mathbb{P}}/\vec{P}]}{\Gamma \vdash t : \mu_j \vec{P}\, \vec{\mathbb{P}}} \qquad \frac{\Gamma \vdash t : \mu_j \vec{P}\, \vec{\mathbb{P}}}{\Gamma \vdash t : \mathbb{P}_j[\mu \vec{P}\, \vec{\mathbb{P}}/\vec{P}]}$$

We write $t : \mathbb{P}$ when $\emptyset \vdash t : \mathbb{P}$.

We have a syntactic substitution lemma:

Lemma 8.1 *Suppose* $\Gamma, x : \mathbb{P} \vdash t : \mathbb{Q}$ *and* $\Gamma \vdash u : \mathbb{P}$. *Then* $\Gamma \vdash t[u/x] : \mathbb{Q}$.

Proof: By induction on the derivation of $\Gamma, x : \mathbb{P} \vdash t : \mathbb{Q}$. $\square$

8.3 Transition Semantics

Every type is associated with actions processes of that type may do. Actions are given by the grammar

$$p ::= \boldsymbol{.} \mid u \mapsto p \mid a\,p \ ,$$

where u is a closed term and a is a sum index. Processes of type $\boldsymbol{.}\mathbb{P}$ may do an action $\boldsymbol{.}$. An action of type $\mathbb{P} \to \mathbb{Q}$ comprises $u \mapsto p$, where $u : \mathbb{P}$ and p is an action of type $\mathbb{Q}$; a process $t : \mathbb{P} \to \mathbb{Q}$ can do the action $u \mapsto p$ precisely when $t\,u$ can do p. Processes of sum type $\Sigma_{a\in A} a\mathbb{P}_a$ may do actions $a\,p$ where $a \in A$ and p is an action of processes of type $\mathbb{P}_a$.

It is useful in typing actions not just to specify the type of processes that can do an action—the type of process at the beginning of the action, but also say which type of process results—the type of process at the end of the action. We type actions with their begin and end types:

$$\boldsymbol{.}\mathbb{P} : \boldsymbol{.} : \mathbb{P} \qquad \frac{u : \mathbb{P} \qquad \mathbb{Q} : q : \mathbb{Q}'}{\mathbb{P} \to \mathbb{Q} : (u \mapsto q) : \mathbb{Q}'} \qquad \frac{\mathbb{P}_a : p : \mathbb{P}'}{\Sigma_{a\in A} a\mathbb{P}_a : a\,p : \mathbb{P}'} \qquad \frac{\mathbb{P}_j[\mu\vec{P}\,\vec{\mathbb{P}}/\vec{P}] : p : \mathbb{P}'}{\mu_j\vec{P}\,\vec{\mathbb{P}} : p : \mathbb{P}'}$$

HOPLA has transitions $\mathbb{P} : t \xrightarrow{p} t'$ between a closed term $t : \mathbb{P}$, an action $\mathbb{P} : p : \mathbb{P}'$ and a closed term t', the resumption of t after action p, given by the following rules; as a consequence we will obtain $t' : \mathbb{P}'$.

$$\frac{\mathbb{P} : t[rec\,x\,t/x] \xrightarrow{p} t'}{\mathbb{P} : rec\,x\,t \xrightarrow{p} t'} \qquad \frac{\mathbb{P} : t_j \xrightarrow{p} t'}{\mathbb{P} : \Sigma_{i\in I} t_i \xrightarrow{p} t'}\ j \in I$$

$$\frac{}{\boldsymbol{.}\mathbb{P} : \boldsymbol{.}t \xrightarrow{\boldsymbol{.}} t} \qquad \frac{\boldsymbol{.}\mathbb{P} : u \xrightarrow{\boldsymbol{.}} u' \quad \mathbb{Q} : t[u'/x] \xrightarrow{q} t'}{\mathbb{Q} : [u > \boldsymbol{.}x \Rightarrow t] \xrightarrow{q} t'}$$

$$\frac{\mathbb{Q} : t[u/x] \xrightarrow{p} t'}{\mathbb{P} \to \mathbb{Q} : \lambda x\,t \xrightarrow{u \mapsto p} t'} \qquad \frac{\mathbb{P} \to \mathbb{Q} : t \xrightarrow{u \mapsto p} t'}{\mathbb{Q} : t\,u \xrightarrow{p} t'}$$

$$\frac{\mathbb{P}_a : t \xrightarrow{p} t'}{\Sigma_{a\in A} a\mathbb{P}_a : a\,t \xrightarrow{a\,p} t'} \qquad \frac{\Sigma_{a\in A} a\mathbb{P}_a : t \xrightarrow{a\,p} t'}{\mathbb{P}_a : \pi_a(t) \xrightarrow{p} t'}$$

$$\frac{\mathbb{P}_j[\mu\vec{P}\,\vec{\mathbb{P}}/\vec{P}] : t \xrightarrow{p} t'}{\mu_j\vec{P}\,\vec{\mathbb{P}} : t \xrightarrow{p} t'}$$

Proposition 8.2 *Suppose* $\mathbb{P} : t \xrightarrow{p} t'$ *where* $\mathbb{P} : p : \mathbb{P}'$. *Then* $t' : \mathbb{P}'$.

Proof: By a simple induction on derivations. □

Definition: From the proposition above it makes sense to write

$$\mathbb{P} : t \xrightarrow{p} t' : \mathbb{P}'$$

when $\mathbb{P} : t \xrightarrow{p} t'$ and $\mathbb{P} : p : \mathbb{P}'$ (though I'll omit types whenever I think I can get away with it—then it should be a simple matter to infer them).

Exercise 8.3 By considering their form of derivations, argue that for $t : \Sigma_{a\in A} a\mathbb{P}_a$

$$t \xrightarrow{a\,p} t' \text{ iff } \pi_a(t) \xrightarrow{p} t' \ ,$$

and for $\mathbb{P} \to \mathbb{Q}$ and $u : \mathbb{P}$ that

$$t \xrightarrow{u\mapsto p} t' \text{ iff } t\,u \xrightarrow{p} t' \ .$$

For an action p and a term t, define

$$\mathbf{.}^*(t) \equiv t \ , \quad (u \mapsto p)^*(t) \equiv p^*(t\,u) \ , \quad (a\,p)^*(t) \equiv p^*(\pi_a(t)) \ .$$

Show by structural induction on p, that

$$t \xrightarrow{p} t' \text{ iff } p^*(t) \xrightarrow{\mathbf{.}} t'$$

$\Box$

Exercise 8.4 * The HOPLA type $\mathbb{B} \equiv true\mathbf{.}\mathbb{O} + false\mathbf{.}\mathbb{O}$ represents the truth values. The HOPLA type $\mathbb{N} \equiv \mu P\,(0\mathbf{.}\mathbb{O} + Suc\,P)$ represents the natural numbers. With respect to these types, write down HOPLA terms to represent zero, perform successor, test for zero, sum and multiplication. $\Box$

8.3.1 Abbreviations

Products

We can define the product type $\mathbb{P} \,\&\, \mathbb{Q}$ of types $\mathbb{P}$, $\mathbb{Q}$ to be $1\mathbb{P} + 2\mathbb{Q}$. A *pair* (t, u) of terms t of type $\mathbb{P}$ and u of type $\mathbb{Q}$ is given by

$$(t, u) \equiv 1\,t + 2\,u \ .$$

Projections are given by $\mathit{fst}\ (v) \equiv \pi_1(v)$ and $\mathit{snd}\ (v) \equiv \pi_2(v)$, for v a term of type $\mathbb{P} \,\&\, \mathbb{Q}$. For actions of type $\mathbb{P} \,\&\, \mathbb{Q}$, write $(p, -) \equiv 1p$, where p is an action of type $\mathbb{P}$, and $(-, q) \equiv 2q$, where q is an action of type $\mathbb{Q}$.

We can derive the following type rules for product terms and actions:

$$\frac{\Gamma \vdash t : \mathbb{P} \quad \Gamma \vdash u : \mathbb{Q}}{\Gamma \vdash (t, u) : \mathbb{P} \,\&\, \mathbb{Q}} \qquad \frac{\Gamma \vdash v : \mathbb{P} \,\&\, \mathbb{Q}}{\Gamma \vdash \mathit{fst}\ (v) : \mathbb{P}} \qquad \frac{\Gamma \vdash v : \mathbb{P} \,\&\, \mathbb{Q}}{\Gamma \vdash \mathit{snd}\ (v) : \mathbb{Q}}$$

$$\frac{\mathbb{P} : p : \mathbb{P}'}{\mathbb{P} \,\&\, \mathbb{Q} : (p, -) : \mathbb{P}'} \qquad \frac{\mathbb{Q} : q : \mathbb{Q}'}{\mathbb{P} \,\&\, \mathbb{Q} : (-, q) : \mathbb{Q}'}$$

The rules are derivable in the sense that from the existing rules we can build a derivation starting with their premises and ending with their conclusion. Transitions of product terms are associated with the following rules, derivable from the existing transition rules:

$$\frac{\mathbb{P} : t \xrightarrow{p} t'}{\mathbb{P} \,\&\, \mathbb{Q} : (t, u) \xrightarrow{(p,-)} t'} \qquad \frac{\mathbb{Q} : u \xrightarrow{p} u'}{\mathbb{P} \,\&\, \mathbb{Q} : (t, u) \xrightarrow{(-,q)} u'}$$

$$\frac{\mathbb{P} \,\&\, \mathbb{Q} : v \xrightarrow{(p,-)} v'}{\mathbb{P} : \mathit{fst}\ (v) \xrightarrow{p} v'} \qquad \frac{\mathbb{P} \,\&\, \mathbb{Q} : v \xrightarrow{(-,q)} v'}{\mathbb{Q} : \mathit{snd}\ (v) \xrightarrow{q} v'}$$

Exercise 8.5 Derive the above typing and transition rules for products. $\Box$

General patterns

Patterns in match expression can be extended from the basic form $\boldsymbol{.}x$, where x is a variable, to general patterns

$$p ::= \boldsymbol{.}x \mid u \mapsto p \mid a\, p$$

where a is an index and u is a term. We adopt the following abbreviations

$$[u > a\, p \Rightarrow t] \equiv [\pi_a(u) > p \Rightarrow t]\ , \qquad [v > (u \mapsto p) \Rightarrow t] \equiv [(vu) > p \Rightarrow t]\ .$$

The extended patterns p are each associated with a unique variable x and with a derivable rule of the form:

$$\frac{\mathbb{P} : u \xrightarrow{p} u' \quad \mathbb{Q} : t[u'/x] \xrightarrow{q} t'}{\mathbb{Q} : [u > p \Rightarrow t] \xrightarrow{q} t'}$$

By convention we elide the variable mentioned in the pattern p when it appears as an action on a transition. Again, the rule is derivable in the sense that from the existing rules we can build a derivation starting with its premises and ending with its conclusion.

Exercise 8.6 Show by induction on the structure of the pattern p, assumed to have variable x, that the transition rule for $[u > \mathbb{Q} : p \Rightarrow t]$ above is derivable.

Show, by induction on the structure of the pattern p, that the typing rule

$$\frac{\Gamma \vdash u : \mathbb{P} \quad \Gamma, x : \mathbb{R} \vdash t : \mathbb{Q}}{\Gamma \vdash [u > p \Rightarrow t] : \mathbb{Q}}\ ,$$

where $\mathbb{P} : p : \mathbb{R}$ (eliding the unique variable x in p), is derivable. $\square$

8.4 Bisimulation

A *type-respecting* relation R on closed process terms is a collection of relations $R_{\mathbb{P}}$, indexed by types $\mathbb{P}$, such that if $t_1\ R_{\mathbb{P}}\ t_2$, then $t_1 : \mathbb{P}$ and $t_2 : \mathbb{P}$.

A type-respecting relation R on closed process terms is a *bisimulation* if, whenever $t_1\ R_{\mathbb{P}}\ t_2$, then

1. if $\mathbb{P} : t_1 \xrightarrow{p} t_1' : \mathbb{Q}$, then $\mathbb{P} : t_2 \xrightarrow{p} t_2' : \mathbb{Q}$ for some t_2' such that $t_1'\ R_{\mathbb{Q}}\ t_2'$;
2. if $\mathbb{P} : t_2 \xrightarrow{p} t_2' : \mathbb{Q}$, then $\mathbb{P} : t_1 \xrightarrow{p} t_1' : \mathbb{Q}$ for some t_1' such that $t_1'\ R_{\mathbb{Q}}\ t_2'$.

Bisimilarity, $\sim$, is the largest bisimulation.
(Again, I'll often drop type annotations when it's easy to restore them.)

Proposition 8.7 *The following pairs of closed, well-formed terms are bisimilar:*

$$
\begin{array}{rrcl}
1. & rec\ x\, t & \sim & t[rec\ x\, t/x] \\
2. & [.u > .x \Rightarrow t] & \sim & t[u/x] \\
3. & [\Sigma_{i\in I} u_i > .x \Rightarrow t] & \sim & \Sigma_{i\in I}[u_i > .x \Rightarrow t] \\
4. & [u > .x \Rightarrow \Sigma_{i\in I} t_i] & \sim & \Sigma_{i\in I}[u > .x \Rightarrow t_i] \\
5. & [u > .x \Rightarrow .x] & \sim & u \\
6. & (\lambda x\, t)\ u & \sim & t[u/x] \\
7. & \lambda x\, (t\ x) & \sim & t \\
8. & \lambda x\, (\Sigma_{i\in I} t_i) & \sim & \Sigma_{i\in I}(\lambda x\, t_i) \\
9. & (\Sigma_{i\in I} t_i)\ u & \sim & \Sigma_{i\in I}(t_i\ u) \\
10. & \pi_a(a\ t) & \sim & t \\
11. & \pi_a(b\ t) & \sim & nil \qquad \text{if } a \neq b \\
12. & t & \sim & \Sigma_{a\in A} a\ \pi_a(t) \\
13. & \pi_a(\Sigma_{i\in I} t_i) & \sim & \Sigma_{i\in I}\pi_a(t_i)
\end{array}
$$

Proof: In each case $t_1 \sim t_2$, we can prove that the identity relation extended by the pair (t_1, t_2) is a bisimulation, so the correspondence between transitions is very close. □

Exercise 8.8 Check the bisimilarities above. Justify that the following bisimilarities involving derived terms:

$$
\begin{array}{rrcl}
1. & [a.u > a.x \Rightarrow t] & \sim & t[u/x] \\
2. & [b.u > a.x \Rightarrow t] & \sim & nil \quad \text{if } a \neq b \\
3. & [\Sigma_{i\in I} u_i > a.x \Rightarrow t] & \sim & \Sigma_{i\in I}[u_i > a.x \Rightarrow t] \\
4. & [u > a.x \Rightarrow \Sigma_{i\in I} t_i] & \sim & \Sigma_{i\in I}[u > a.x \Rightarrow t_i] \\
5. & fst\ (t, u) & \sim & t \\
6. & snd\ (t, u) & \sim & u \\
7. & t & \sim & (fst\ t, snd\ t) \\
8. & (\Sigma_{i\in I} t_i, \Sigma_{i\in I} u_i) & \sim & \Sigma_{i\in I}(t_i, u_i) \\
9. & fst\ (\Sigma_{i\in I} t_i) & \sim & \Sigma_{i\in I}(fst\ t_i) \\
10. & snd\ (\Sigma_{i\in I} t_i) & \sim & \Sigma_{i\in I}(snd\ t_i)
\end{array}
$$

Devise and prove correct bisimiliarities like 1-4 above, but for a match $[u > p \Rightarrow t]$ associated with a general pattern p (with unique variable x). □

Theorem 8.9 *Bisimilarity is a congruence.*

Proof: [Non-examinable] We use Howe's method [9]. It depends on first extending the definition of bisimulation to open terms. When $\Gamma \vdash t_1 : \mathbb{P}$ and $\Gamma \vdash t_2 : \mathbb{P}$, we write $t_1 \sim^\circ t_2$ iff for all type-respecting substitutions σ for Γ by closed terms, $t_1[\sigma] \sim t_2[\sigma]$.

In summary, we define a precongruence candidate $\hat{\sim}$ by the rules:

$$\frac{x \sim^\circ w}{x \mathrel{\hat{\sim}} w} \qquad \frac{t \mathrel{\hat{\sim}} t' \quad rec\, x\, t' \sim^\circ w}{rec\, x\, t \mathrel{\hat{\sim}} w} \qquad \frac{t_j \mathrel{\hat{\sim}} t'_j \quad \text{all } j \in I \quad \Sigma_{i\in I} t'_i \sim^\circ w}{\Sigma_{i\in I} t_i \mathrel{\hat{\sim}} w}$$

$$\frac{t \mathrel{\hat{\sim}} t' \quad \boldsymbol{.}t' \sim^\circ w}{\boldsymbol{.}t \mathrel{\hat{\sim}} w} \qquad \frac{t \mathrel{\hat{\sim}} t' \quad u \mathrel{\hat{\sim}} u' \quad [u' > \boldsymbol{.}x \Rightarrow t'] \sim^\circ w}{[u > \boldsymbol{.}x \Rightarrow t] \mathrel{\hat{\sim}} w}$$

$$\frac{t \mathrel{\hat{\sim}} t' \quad \lambda x\, t' \sim^\circ w}{\lambda x\, t \mathrel{\hat{\sim}} w} \qquad \frac{t \mathrel{\hat{\sim}} t' \quad u \mathrel{\hat{\sim}} u' \quad t'\, u' \sim^\circ w}{t\, u \mathrel{\hat{\sim}} w}$$

$$\frac{t \mathrel{\hat{\sim}} t' \quad a\, t' \sim^\circ w}{a\, t \mathrel{\hat{\sim}} w} \qquad \frac{t \mathrel{\hat{\sim}} t' \quad \pi_a t' \sim^\circ w}{\pi_a t \mathrel{\hat{\sim}} w}$$

By induction on derivations, we can now show that: (i) $\hat{\sim}$ is reflexive; (ii) $\hat{\sim}$ is operator-respecting; (iii) $\sim^\circ \subseteq \hat{\sim}$; (iv) if $t \mathrel{\hat{\sim}} t'$ and $t' \sim^\circ w$ then $t \mathrel{\hat{\sim}} w$; (v) if $t \mathrel{\hat{\sim}} t'$ and $u \mathrel{\hat{\sim}} u'$ then $t[u/x] \mathrel{\hat{\sim}} t'[u'/x]$ whenever the substitutions are well-formed; (vi) since $\sim$ is an equivalence relation, the transitive closure $\hat{\sim}^+$ of $\hat{\sim}$ is symmetric, and therefore, so is $\hat{\sim}_{\mathsf{c}}^+$.

Now we just need to show that $\hat{\sim}_{\mathsf{c}}$ is a simulation (*i.e.*, it satisfies the first clause in the definition of bisimulation), because then $\hat{\sim}_{\mathsf{c}}^+$ is a bisimulation by (vi), and so $\hat{\sim}_{\mathsf{c}}^+ \subseteq \sim$. In particular, $\hat{\sim}_{\mathsf{c}} \subseteq \sim$. By (i) and (v), it follows that $\hat{\sim} \subseteq \sim^\circ$, and so by (iii), $\hat{\sim} = \sim^\circ$. Hence, $\sim$ is a congruence because it is an equivalence relation and by (ii) it is operator respecting.

We prove that $\hat{\sim}_{\mathsf{c}}$ is a simulation by induction on the derivations of the operational semantics and using (iv-v). In fact, we need an induction hypothesis slightly stronger than one might expect. It involves extending $\hat{\sim}_{\mathsf{c}}$ to actions by the rules:

$$\frac{}{\boldsymbol{.} \mathrel{\hat{\sim}_{\mathsf{c}}} \boldsymbol{.}} \qquad \frac{u_1 \mathrel{\hat{\sim}_{\mathsf{c}}} u_2 \quad p_1 \mathrel{\hat{\sim}_{\mathsf{c}}} p_2}{(u_1 \mapsto p_1) \mathrel{\hat{\sim}_{\mathsf{c}}} (u_2 \mapsto p_2)} \qquad \frac{p_1 \mathrel{\hat{\sim}_{\mathsf{c}}} p_2}{a\, p_1 \mathrel{\hat{\sim}_{\mathsf{c}}} a\, p_2}$$

Now by induction on the derivation of $t_1 \xrightarrow{p_1} t'_1$ it can be shown that:

> if $t_1 \mathrel{\hat{\sim}_{\mathsf{c}}} t_2$ and $t_1 \xrightarrow{p_1} t'_1$, then for all p_2 with $p_1 \mathrel{\hat{\sim}_{\mathsf{c}}} p_2$ we have $t_2 \xrightarrow{p_2} t'_2$ for some t'_2 such that $t'_1 \mathrel{\hat{\sim}_{\mathsf{c}}} t'_2$.

By (i), $p_1 \mathrel{\hat{\sim}_{\mathsf{c}}} p_1$ for all actions p_1, from which it follows that $\hat{\sim}_{\mathsf{c}}$ is a simulation. $\square$

We now know that $\sim$ supports an equational style of reasoning in which bisimilar terms of the same type may substituted for each other while maintaining bisimilarity.

Proposition 8.10 *Let t_1, t_2 be closed terms of type $\mathbb{P} \to \mathbb{Q}$. The following are equivalent:*

1. $t_1 \sim t_2$*;*
2. $t_1\, u \sim t_2\, u$ *for all closed terms u of type $\mathbb{P}$;*

3. $t_1\, u_1 \sim t_2\, u_2$ *for all closed terms* $u_1 \sim u_2$ *of type* $\mathbb{P}$*.*

Proof: 1. implies 2. and 3. as $\sim$ is a congruence. 3. clearly implies 2. as $\sim$ is reflexive. From the transition semantics it's easily seen that

$$t \xrightarrow{u \mapsto p} t' \text{ iff } t\, u \xrightarrow{p} t' \ .$$

Consequently, 2. implies 1. by:

$$\begin{aligned} t_1 \xrightarrow{u \mapsto p} t_1' \quad &\Rightarrow t_1\, u \xrightarrow{p} t_1' \\ &\Rightarrow t_2\, u \xrightarrow{p} t_2' \ \&\ t_1' \sim t_2' \\ &\Rightarrow t_2 \xrightarrow{u \mapsto p} t_2' \ \&\ t_1' \sim t_2' \ , \end{aligned}$$

and vice versa. □

8.5 Linearity

Consider a process term t with a single free variable x, associated with the typing

$$x : \mathbb{P} \vdash t : \mathbb{Q} \ .$$

We can regard the term t as an operation (or function) from processes of type $\mathbb{P}$ to processes of type $\mathbb{Q}$; given a process $u : \mathbb{P}$ as argument, we obtain a process $t[u/x]$ as result.

Imagine $\mathbb{Q} : t[u/x] \xrightarrow{q} t'$ and that every time u is executed to get a transition that capability is used up. We can ask how many copies of the process u were needed to produce the transition. In distributed computation the copying of processes is often limited. For this reason many naturally-occurring operations on processes require at most one copy of the process they take as argument. Informally, an operation is *linear* when any resulting transition always requires precisely one copy of any input process. There are two ways in which an operation on processes can fail to be linear: either the operation does not need to execute the argument process, or it needs to execute strictly more than one copy.[3]

Consider the term $\boldsymbol{.}x$ where

$$x : \mathbb{P} \vdash \boldsymbol{.}x : \boldsymbol{.}\mathbb{P} \ .$$

Certainly

$$\mathbb{P} : \boldsymbol{.}u \xrightarrow{\bullet} u \ .$$

But the process u was not executed in producing the transition—the *nil* process would have served as well. The operation determined by $\boldsymbol{.}x$ is not linear because it required no copy of the argument process.

[3]There's much more to linearity than appears here. The language HOPLA was in fact discovered through a form of domain theory [14, 15, 16] which is also a model of a resource-conscious logic called Linear Logic, discovered and shown to underlie traditional logic by Jean-Yves Girard in the mid 1980's.

Consider now the term

$$t \equiv [x > a.y \Rightarrow [x > b.z \Rightarrow c.nil]]$$

with typing

$$x : a.\mathbb{O} + b.\mathbb{O} \vdash t : c.\mathbb{O} \ .$$

In order to produce a transition

$$c.\mathbb{O} : t[u/x] \xrightarrow{c.} nil$$

one copy of u is executed to give a $a.$-transition and another copy to give a $b.$-transition.

Linearity and its absence have mathematical consequences. We have explained linearity informally. Its mathematical counterpart is reflected in the preservation of nondeterministic sums. This turns on the fact that a single execution of a sum $\Sigma_{i\in I} u_i$ amounts to a single execution of a component. Let t be a term of type $\mathbb{Q}$ with one free variable x of type $\mathbb{P}$. Say t is *linear in* x iff for any sum of closed terms $\Sigma_{i\in I} u_i$ of type $\mathbb{Q}$

$$t[\Sigma_{i\in I} u_i/x] \sim_{\mathbb{P}} \Sigma_{i\in I} t[u_i/x] \ .$$

It is easy to check that the prefix term $.x$ is not linear in x (Exercise!). Nor is the term

$$t \equiv [x > a.y \Rightarrow [x > b.z \Rightarrow c.nil]]$$

linear in x. It is easy to see that

$$t[a.nil + b.nil/x] \not\sim t[a.nil/x] + t[b.nil/x]$$

because

$$t[a.nil + b.nil/x] \xrightarrow{c} nil$$

whereas

$$t[a.nil/x] + t[b.nil/x]$$

is incapable of any transitions.

However many operations are linear:

Exercise 8.11 Show that $.x$ is not linear in x (assume that x has any type that is convenient). Show from the transition semantics, referring to previously done exercises if helpful, that the following terms, assumed well-typed and to have only x as free variable, are all linear in x:

$$a\,x \qquad \pi_a(x) \qquad x\,u \qquad \lambda y(x\,y) \qquad [x > .y \Rightarrow u] \qquad (x \text{ is not free in } u).$$

Show that $x + x$ is always linear in x.
Let u be a term of type $\mathbb{P}$ with no free variables; then $x : \mathbb{Q} \vdash u : \mathbb{P}$ for any type $\mathbb{Q}$. Show that u is linear in x iff $u \sim nil$. [Hint: Consider empty sums.] $\Box$

Exercise 8.12 *Is there a typable term which does not involve prefix or prefix match which is not linear. $\Box$

8.6 Examples

In the examples, for readability we'll depart from the strict syntax of HOPLA and write recursive definitions of types and terms as equations; it is routine to convert such definitions into HOPLA.

8.6.1 CCS

As in CCS [11], let N be a set of names and $\bar{N}$ the set of complemented names $\{\bar{n} \mid n \in N\}$. Let l range over labels $L = N \cup \bar{N}$, with complementation extended to L by taking $\bar{\bar{n}} = n$, and let τ be a distinct label. We can then specify a type $\mathbb{P}$ as

$$\mathbb{P} = \tau.\mathbb{P} + \Sigma_{n \in N} n.\mathbb{P} + \Sigma_{n \in N} \bar{n}.\mathbb{P} \ .$$

Below, we let α range over $L \cup \{\tau\}$. The terms of CCS can be translated into HOPLA as the following terms of type $\mathbb{P}$:

$$\begin{array}{rlrl} [\![x]\!] &\equiv x & [\![rec\ x\, P]\!] &\equiv rec\ x\, [\![P]\!] \\ [\![\alpha.P]\!] &\equiv \alpha.[\![P]\!] & [\![\Sigma_{i\in I} P_i]\!] &\equiv \Sigma_{i \in I}[\![P_i]\!] \\ [\![P|Q]\!] &\equiv Par\ [\![P]\!]\ [\![Q]\!] & [\![P \setminus S]\!] &\equiv Res_S\ [\![P]\!] \\ [\![P[f]]\!] &\equiv Rel_f\ [\![P]\!] & & \end{array}$$

Here, $Par : \mathbb{P} \to (\mathbb{P} \to \mathbb{P})$, $Res_S : \mathbb{P} \to \mathbb{P}$, and $Rel_f : \mathbb{P} \to \mathbb{P}$ are abbreviations for the following recursively defined processes:

$$\begin{array}{rl} Par \equiv rec\ p.\lambda x\, \lambda y. & \Sigma_\alpha[x > \alpha.x' \Rightarrow \alpha.(p\ x'\ y)] + \\ & \Sigma_\alpha[y > \alpha.y' \Rightarrow \alpha.(p\ x\ y')] + \\ & \Sigma_l[x > l.x' \Rightarrow [y > \bar{l}.y' \Rightarrow \tau.(p\ x'\ y')]] \\ Res_S \equiv rec\ r.\lambda x & \Sigma_{\alpha \notin (S \cup \bar{S})}[x > \alpha.x' \Rightarrow \alpha.(r\ x')] \\ Rel_f \equiv rec\ r.\lambda x & \Sigma_\alpha[x > \alpha.x' \Rightarrow f(\alpha).(r\ x')] \end{array}$$

Proposition 8.13 *If $P \xrightarrow{\alpha} P'$ in CCS then $[\![P]\!] \xrightarrow{\alpha} [\![P']\!]$ in HOPLA.*

Conversely, if $[\![P]\!] \xrightarrow{a} t'$ in the higher-order language, then $a \equiv \alpha$ and $t' \equiv [\![P']\!]$ for some α, P' such that $P \xrightarrow{\alpha} P'$ according to CCS.

Proof: For each HOPLA definition of the CCS operations, the corresponding CCS rules can be derived in HOPLA (Exercise!). □

It follows that the translations of two CCS terms are bisimilar in the general language iff they are strongly bisimilar in CCS.

We can recover the expansion law for general reasons. Write $P|Q$ for the application $Par\ P\ Q$, where P and Q are terms of type $\mathbb{P}$. Suppose

$$P \sim \Sigma_\alpha \Sigma_{i \in I(\alpha)} \alpha.P_i \quad \text{and} \quad Q \sim \Sigma_\alpha \Sigma_{j \in J(\alpha)} \alpha.Q_j .$$

Using especially the derived properties of Exercise 8.8 1-4 (plus some basic bisimilarities of Proposition 8.7), we get

$$\begin{aligned} P|Q \sim\ & \Sigma_\alpha[P > \alpha.x' \Rightarrow \alpha.(x'|Q)] + \\ & \Sigma_\alpha[Q > \alpha.y' \Rightarrow \alpha.(P|y')] + \\ & \Sigma_l[P > l.x' \Rightarrow [Q > \bar{l}.y' \Rightarrow \tau.(x'|y')]] \\ \sim\ & \Sigma_\alpha \Sigma_{i \in I(\alpha)} \alpha.(P_i|Q) + \\ & \Sigma_\alpha \Sigma_{j \in J(\alpha)} \alpha.(P|Q_j) + \\ & \Sigma_l \Sigma_{i \in I(l), j \in J(\bar{l})} \tau.(P_i|Q_j)\ . \end{aligned}$$

Exercise 8.14 Spell out the details in the derivation of the expansion law above. □

8.6.2 CCS with value passing

It's easy to encode CCS with value passing of the kind introduced early in the course. Assume values form the set V. Define

$$\mathbb{P} = \tau.\mathbb{P} + \Sigma_{n \in N} n?\Sigma_{v \in V} v.\mathbb{P} + \Sigma_{n \in N} n!\Sigma_{v \in V} v.\mathbb{P}\ .$$

Exercise 8.15 Give a translation of the earlier CCS with value passing. To avoid using the datatype of numbers (itself requiring a recusive definition), cut down from general boolean tests to just tests for equality. You'll need first to invent HOPLA operators for parallel composition and restriction for CCS with value passing. □

In fact there is a choice in how one gives semantics to CCS with value passing. That we've used is called *early* value passing and is essentially that originally given by Milner. There is an alternative *late* value passing. The idea here is that $a?x.t$ represents a process which first synchronises on the a-channel and then resumes as a function $v \mapsto t[v/x]$. We can achieve late value passing through the type:

$$\mathbb{P} = \tau.\mathbb{P} + \Sigma_{n \in N} n?.(\Sigma_{v \in V} v\mathbb{P}) + \Sigma_{n \in N} n!\Sigma_{v \in V} v.\mathbb{P}\ .$$

The sum $(\Sigma_{v \in V} v\mathbb{P})$ is like a function space; for $f : (\Sigma_{v \in V} v\mathbb{P})$, a given value v selects a process $\pi_v(f)$.

Exercise 8.16 Describe the bisimilarity relations inherited for the two value-passing calculi, paying special attention to when two input processes are bisimilar. Exhibit two CCS terms which are bisimilar with respect to the early semantics and yet not bisimilar with respect to the late semantics. [Hint: Let one term have the form $\alpha?X.t$ and the other the form $\alpha?X.t_1 + \alpha?X.t_2$.] □

8.6.3 Higher-Order CCS

In [6], Hennessy considers a language like CCS but where processes are passed at channels C. For a translation into our language, we follow Hennessy in defining types that satisfy the equations

$$\mathbb{P} = \tau.\mathbb{P} + \Sigma_{c \in C} c!.\mathbb{C} + \Sigma_{c \in C} c?.\mathbb{F} \qquad \mathbb{C} = \mathbb{P} \,\&\, \mathbb{P} \qquad \mathbb{F} = \mathbb{P} \to \mathbb{P}\ .$$

We are chiefly interested in the parallel composition of processes, $Par_{\mathbb{P},\mathbb{P}}$ of type $\mathbb{P} \,\&\, \mathbb{P} \to \mathbb{P}$. But parallel composition is really a family of mutually dependent operations also including components such as $Par_{\mathbb{F},\mathbb{C}}$ of type $\mathbb{F} \,\&\, \mathbb{C} \to \mathbb{P}$ to say how abstractions compose in parallel with concretions *etc.* All these components can be tupled together in a product and parallel composition defined as a simultaneous recursive definition whose component at $\mathbb{P} \,\&\, \mathbb{P} \to \mathbb{P}$ satisfies

$$\begin{aligned} P|Q = {} & \Sigma_\alpha[P > \alpha.x \Rightarrow \alpha.(x|Q)] + \\ & \Sigma_\alpha[Q > \alpha.y \Rightarrow \alpha.(P|y)] + \\ & \Sigma_c[P > c?.f \Rightarrow [Q > c!.z \Rightarrow \tau.((f\ \mathit{fst}\ z)|\mathit{snd}\ z)]] + \\ & \Sigma_c[P > c!.z \Rightarrow [Q > c?.f \Rightarrow \tau.(\mathit{snd}\ z|(f\ \mathit{fst}\ z))]]\ , \end{aligned}$$

where, *e.g.*, $P|Q$ abbreviates $Par_{\mathbb{P},\mathbb{P}}\ (P, Q)$. In the summations $c \in C$ and α ranges over $c!, c?, \tau$.

Exercise 8.17 Describe bisimilarity inherited for the process-passing calculus. (Make use of Proposition 8.10.) □

8.6.4 Mobile Ambients with Public Names

The Calculus of Mobile Ambients [1] invented by Luca Cardelli and Andy Gordon in the late 1990's has been very influential as it sought to cover an important aspect of distributed compution, not previously addressed well by process calculi. This was the aspect of administrative domains. For example, the internet is partitioned into administrative domains by firewalls which isolate domains except for strictly contolled pathways.

Roughly an ambient $n[P]$ consists of a process term P enclosed within an ambient named n. The process term has capabilities which specify the enclosing ambient's movement into and out of other ambients, as well as the powerful capability of opening an ambient by removing its boundary.

The simplest way to explain the behaviour of ambients is via reduction rules showing how compound ambient terms execute. The central capabilities of ambients are given by the following reductions:
An ambient entering another ambient:

$$n[\mathit{in}\ m.P|Q]|m[R] \ \to\ m[n[P|Q]|R]$$

An ambient exiting another ambient:

$$m[n[\mathit{out}\ m.P|Q]|R] \ \to\ n[P|Q]|m[R]$$

A process opening an ambient:

$$\mathit{open}\ m!.P|m[Q] \ \to\ P|Q$$

Note the reduction rules do not give a compositional account of ambient behaviour.

The full Ambient Calculus allows name generation in the manner of the Pi-Calculus. We can translate the Ambient Calculus with public names [2] into the higher-order language, following similar lines to the process-passing language above. This is a way to give an compositional semantics to Ambient Calculus. Assume a fixed set of ambient names $n, m, \ldots \in N$. The syntax of ambients is extended beyond processes (P) to include concretions (C) and abstractions (F):

$$\begin{aligned} P &::= \mathit{nil} \mid P|P \mid \mathit{rep} P \mid n[P] \mid \mathit{in}\ n.P \mid \mathit{out}\ n.P \mid \mathit{open}\ n!.P \mid \\ &\quad \tau.P \mid \mathit{mvin}\ n!.C \mid \mathit{mvout}\ n!.C \mid \mathit{open}\ n?.P \mid \mathit{mvin}\ n?.F \mid x \\ C &::= (P, P) \qquad F \ ::= \ \lambda x\, P \ . \end{aligned}$$

The notation for actions departs a little from that of [1]. Here some actions are marked with ! and others with ?—active (or inceptive) actions are marked by ! and passive (or receptive) actions by ?. We say actions α and β are *complementary* iff one has the form *open* $n!$ or *mvin* $n!$ while the other is *open* $n?$ or *mvin* $n?$ respectively. Complementary actions can synchronise together to form a τ-action. We adopt a slightly different notation for concretions ((P, R) instead of $\langle P \rangle R$) and abstractions ($\lambda x\, P$ instead of $(x)P$) to make their translation into the higher-order language clear.

Types for ambients are given recursively by (n ranges over N):

$$\begin{aligned} \mathbb{P} &= \tau.\mathbb{P} + \Sigma_n \mathit{in}\ n.\mathbb{P} + \Sigma_n \mathit{out}\ n.\mathbb{P} + \Sigma_n \mathit{open}\ n!.\mathbb{P} + \Sigma_n \mathit{mvin}\ n!.\mathbb{C} + \\ &\quad \Sigma_n \mathit{mvout}\ n!.\mathbb{C} + \Sigma_n \mathit{open}\ n?.\mathbb{P} + \Sigma_n \mathit{mvin}\ n?.\mathbb{F} \\ \mathbb{C} &= \mathbb{P} \,\&\, \mathbb{P} \qquad \mathbb{F} \ = \ \mathbb{P} \to \mathbb{P} \ . \end{aligned}$$

The eight components of the prefixed sum in the equation for $\mathbb{P}$ correspond to the eight forms of ambient actions τ, *in* n, *out* n, *open* $n!$, *mvin* $n!$, *mvout* $n!$, *open* $n?$ and *mvin* $n?$. We obtain the prefixing operations as injections into the appropriate component of the prefixed sum $\mathbb{P}$.

Parallel composition is really a family of operations, one of which is a binary operation between processes but where in addition there are parallel compositions of abstractions with concretions, and even abstractions with processes and concretions with processes. The family of operations

$$\begin{aligned} &(-|-) : \mathbb{F} \,\&\, \mathbb{C} \to \mathbb{P}, \quad (-|-) : \mathbb{F} \,\&\, \mathbb{P} \to \mathbb{F}, \quad (-|-) : \mathbb{C} \,\&\, \mathbb{P} \to \mathbb{C}, \\ &(-|-) : \mathbb{C} \,\&\, \mathbb{F} \to \mathbb{P}, \quad (-|-) : \mathbb{P} \,\&\, \mathbb{F} \to \mathbb{F}, \quad (-|-) : \mathbb{P} \,\&\, \mathbb{C} \to \mathbb{C} \end{aligned}$$

are defined in a simultaneous recursive definition:

Processes in parallel with processes:

$$\begin{aligned} P|Q = {} & \Sigma_\alpha [P > \alpha.x \Rightarrow \alpha.(x|Q)] + \Sigma_\alpha [Q > \alpha.y \Rightarrow \alpha.(P|y)] + \\ & \Sigma_n [P > \mathit{open}\ n!.x \Rightarrow [Q > \mathit{open}\ n?.y \Rightarrow \tau.(x|y)]] + \\ & \Sigma_n [P > \mathit{open}\ n?.x \Rightarrow [Q > \mathit{open}\ n!.y \Rightarrow \tau.(x|y)]] + \\ & \Sigma_n [P > \mathit{mvin}\ n?.f \Rightarrow [Q > \mathit{mvin}\ n!.z \Rightarrow \tau.((f\ \mathit{fst}\ z)|\mathit{snd}\ z)]] + \\ & \Sigma_n [P > \mathit{mvin}\ n!.z \Rightarrow [Q > \mathit{mvin}\ n?.f \Rightarrow \tau.(\mathit{snd}\ z|(f\ \mathit{fst}\ z))]] \ . \end{aligned}$$

Abstractions in parallel with concretions: $F|C = (F\ (\mathit{fst}\ C))|(\mathit{snd}\ C)$.

Abstractions in parallel with processes: $F|P = \lambda x\,((F\ x)|P)$.

Concretions in parallel with processes: $C|P = (\mathit{fst}\ C, (\mathit{snd}\ C)|P)$.

The remaining cases are given symmetrically. Processes P, Q of type $\mathbb{P}$ will—up to bisimilarity—be sums of prefixed terms, and by Proposition 8.7, their parallel composition satisfies the obvious expansion law.

Ambient creation can be defined recursively in the higher-order language:

$$\begin{aligned} m[P] = {} & [P > \tau.x \Rightarrow \tau.m[x]] + \\ & \Sigma_n[P > \mathit{in}\ n.x \Rightarrow \mathit{mvin}\ n!.(m[x], \mathit{nil})] + \\ & \Sigma_n[P > \mathit{out}\ n.x \Rightarrow \mathit{mvout}\ n!.(m[x], \mathit{nil})] + \\ & [P > \mathit{mvout}\ m!.y \Rightarrow \tau.(\mathit{fst}\ y|m[\mathit{snd}\ y])] + \\ & \mathit{open}\ m?.P + \mathit{mvin}\ m?.\lambda y.m[P|y] \ \ . \end{aligned}$$

The denotations of ambients are determined by their capabilities: an ambient $m[P]$ can perform the internal (τ) actions of P, enter a parallel ambient ($\mathit{mvin}\ n!$) if called upon to do so by an $\mathit{in}\ n$-action of P, exit an ambient n ($\mathit{mvout}\ n!$) if P so requests through an $\mathit{out}\ n$-action, be exited if P so requests through an $\mathit{mvout}\ m!$-action, be opened ($\mathit{open}\ m?$), or be entered by an ambient ($\mathit{mvin}\ m?$); initial actions of other forms are restricted away. Ambient creation is at least as complicated as parallel composition. This should not be surprising given that ambient creation corresponds intuitively to putting a process behind (so in parallel with) a wall or membrane which if unopened mediates in the communications the process can do, converting some actions to others and restricting some others away. The tree-containment structure of ambients is captured in the chain of $\mathit{open}\ m?$'s that they can perform.

By the properties of prefix match (Exercise 8.8, items 1-4), there is an expansion theorem for ambient creation. For a process P with $P \sim \Sigma_\alpha \Sigma_{i \in I(\alpha)} \alpha.P_i$, where α ranges over atomic actions of ambients,

$$\begin{aligned} m[P] \sim {} & \Sigma_{i \in I(\tau)} \tau.m[P_i] + \\ & \Sigma_n \Sigma_{i \in I(\mathit{in}\ n)} \mathit{mvin}\ n!.(m[P_i], \mathit{nil}) + \\ & \Sigma_n \Sigma_{i \in I(\mathit{out}\ n)} \mathit{mvout}\ n!.(m[P_i], \mathit{nil}) + \\ & \Sigma_{i \in I(\mathit{mvout}\ m!)} \tau.(\mathit{fst}\ P_i|m[\mathit{snd}\ P_i]) + \\ & \mathit{open}\ m?.P + \mathit{mvin}\ m?.(\lambda y.m[P|y]) \ \ . \end{aligned}$$

8.6.5 Message Passing

HOPLA is a new language and the range of its expressivity is not yet fully explored. Presumably it can express message passing as found in SPL and LINDA. But this has not been done yet.

Exercise 8.18 ** Simplify SPL to a simple message-passing language, so it has a fixed set of names, no new-name generation, and messages which do not involve encryption. Can you encode this simplified SPL in HOPLA? [I'd be interested in seeing any (partial) successes.]* □

8.7 Name generation

Process languages often follow the pioneering work on the Pi-Calculus and allow name generation. HOPLA can be extended to encompass such languages—work with Francesco Zappa Nardelli on new-HOPLA; the extensions are to add a type of names $\mathcal{N}$, a function space $\mathcal{N} \to \mathbb{P}$ as well as a type $\delta\mathbb{P}$ supporting new-name generation through the abstraction $new\, x\, t$. The denotational semantics of the extension to name generation of HOPLA seems best carried out in the Nominal Sets of Andrew Pitts or the broader framework of Fraenkel-Mostowski Set Theory—work with Dave Turner on "Nominal HOPLA". An extension of the operational semantics is more easily accessible; it is like that of HOPLA but given at stages indexed by the current set of names.

Bibliography

[1] Cardelli, L., and Gordon,A. D. "Mobile ambients." Proc. ETAPS, 1998.

[2] Cardelli, L., and Gordon,A. D. "Anytime, anywhere. Modal logics for mobile ambients." In *Proc. POPL'00*.

[3] Clarke, E., Grumberg, O., and Peled, D., "**Model checking**." MITPress, 1999.

[4] Dijkstra, E.W., "**A discipline of programming**." Prentice-Hall, 1976.

[5] Emerson, A. and Lei, C., "Efficient model checking in fragments of the propositional mu-calculus." Proc. of Symposium on Logic in Computer Science, 1986.

[6] Hennessy, M. "A fully abstract denotational model for higher-order processes." *Information and Computation*, 112(1):55–95, 1994.

[7] Hoare, C.A.R., "Communicating sequential processes." CACM, vol.21, No.8, 1978.

[8] Hoare, C.A.R., "**Communicating sequential processes.**" Prentice-Hall, 1985.

[9] Howe, D.J. "Proving congruence of bisimulation in functional programming languages." *Information and Computation*, 124(2):103–112, 1996.

[10] Kozen, D., "Results on the propositional mu-calculus," Theoretical Computer Science 27, 1983.

[11] Milner, A.J.R.G., "**Communication and concurrency.**" Prentice Hall, 1989.

[12] Milner, A.J.R.G., "**Communicating and mobile systems: the Pi-Calculus.**" Cambridge University Press, 1999.

[13] inmos, "**Occam programming manual**." Prentice Hall, 1984.

[14] Nygaard, M., and Winskel, G. "Linearity in process languages." Proc. of LICS'02, 2002.

[15] Nygaard, M., and Winskel, G. "HOPLA—A Higher Order Process Language." Proc. of CONCUR'02, 2002.

[16] Nygaard, M., and Winskel, G., "Domain theory for concurrency." Theoretical Computer Science special edition on the occasion of Dana Scott's 70th birthday, Theoretical Computer Science, Volume 316, Issues 1-3, pp. 153-190, 2004.

[17] Parrow, J., "Fairness properties in process algebra." PhD thesis, Uppsala University, Sweden, 1985.

[18] Reisig, W., "**Petri nets: an introduction**." EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985.

[19] Stirling, C. and Walker D., "Local model checking the modal mu-calculus." Proc.of TAPSOFT, 1989.

[20] Tarski, A., "A lattice-theoretical fixpoint theorem and its applications." Pacific Journal of Mathematics, 5, 1955.

[21] Winskel, G., and Nielsen, M., "Models for concurrency." A chapter in vol.IV of the **Handbook of Logic and the Foundations of Computer Science**, Oxford University Press, 1995.