An introduction to the semantics of concurrent languages and systems

Davide Sangiorgi

INRIA Sophia Antipolis
2004 route des Lucioles, BP 93
06902 Sophia Antipolis CEDEX

Email: davide.sangiorgi@sophia.inria.fr
http://www-sop.inria.fr/meije/personnel/Davide.Sangiorgi.html

August 30, 2002

References

These slides: `http://www-sop.inria.fr/mimosa/personnel/Davide.Sangiorgi/DEA99.ps.gz`

The course, based on these slides, is (I hope) self-contained. For further reading you might consult:

1. Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.

   The course covers pieces of chapters 1-8 of the book. Some important differences: we use a simpler process calculus, which has no relabeling operators, and guarded summation in place of arbitrary summation; due to the use of guarded summation, for us weak bisimulation is a process congruence.

2. Robin Milner, *Operational and Algebraic Semantics of Concurrent Processes*, Chapter 19 of Handbook of Theoretical Computer Science, Elsevier, 1990.

   It collects the mains results from [1]; it has very few examples.

3. Robin Milner, *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press, 1999.

   The first part of the book (which in fact it is half of the book) is an excellent introduction to CCS; the course, by large, follows its structure (exception: part VII).

$\boxed{\text{Outline}}$

– Part I: From functions to processes.
– Part II: From language equivalence to bisimilarity.
– Part III: Induction and co-induction.
– Part IV: A process calculus: CCS.
– Part V: Algebraic and operational theory of processes.
– Part VI: Weak bisimilarity.
– Part VII: Value-passing, examples (including the semantics of a concurrent imperative language as translation into CCS)

Part I: From functions to processes

Denotation versus operational semantics, revisited

For the semantics of *sequential* programming languages, you have focused on *denotational*, rather than *operational*, semantics, for two main reasons:

1. We can think of sequential programs as mathematical objects, namely *functions*.
2. The operational definition of program "equivalence" requires quantification over contexts, such as:

$$\text{for all } C, \quad C[P] \Downarrow \text{ iff } C[Q] \Downarrow \tag{1}$$

For the semantics of *concurrent* programming languages, we focus on the operational approach, because:

1. A concurrent program does not (normally) behave as a function.

   Concurrent program are not functions, but *processes*. But what is a process?

   There is no universally-accepted mathematical answer to this. Hence we do not find in mathematics tools/concepts for the denotational semantics of concurrent languages, at least not as successful as those for the sequential ones.
2. Definition 1 of operational equivalence is not quite satisfactory in concurrency; we shall replace it with a different definition, that does not have quantification on contexts.

Processes are not functions

A sequential imperative language can be viewed as a function from states to states. Consider these two programs:

$$\mathtt{X} := 2 \qquad \text{and} \qquad \mathtt{X} := 1;\ \mathtt{X} := \ \mathtt{X} + 1$$

They denote the same function from states to states.

But now take a context with parallelism, such as $[\cdot] \mid \ \mathtt{X} := 2$. The program

$$\mathtt{X} := 2 \mid \ \mathtt{X} := 2$$

always terminates with $\mathtt{X} = 2$. This is not true (why?) for

$$(\ \mathtt{X} := 1;\ \mathtt{X} := \ \mathtt{X} + 1\ ) \mid \ \mathtt{X} := 2$$

Therefore: Viewing processes as functions gives us a notion of equivalence that is not a *congruence*. In other words, such a semantics of processes as functions would not be *compositional*.

Furthermore:

- A concurrent program may not terminate, and yet perform meaningful computations (examples: an operating system, the controllers of a nuclear station or of a railway system).
  In sequential languages programs that do not terminate are undesirable; they are "wrong".

- The behaviour of a concurrent program can be non-determistic. Example:

$$(\ \mathtt{X} := 1; \ \ \mathtt{X} := \ \ \mathtt{X} + 1 \ ) \mid \ \ \mathtt{X} := 2$$

  (In a functional approach, non-determinism can be dealt with using powersets and powerdomains; but there are limitations to their applicability).

What is a *processes*?
When are two processes behaviourally equivalent?

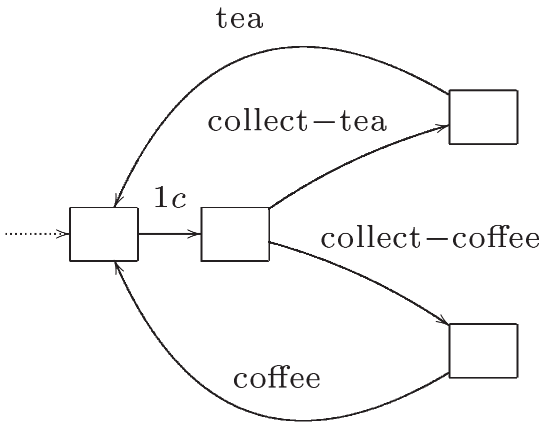
These are basic, fundamental questions; they have been at the core of the research in concurrency theory for the past 30 years. (They are still so today, although remarkable progress has been made)

We shall approach these questions from a simple case, in which interactions among processes are just synchronisations, without exchange of values.

Part II: From language equivalence to bisimilarity

Consider a vending machine, capable of dispensing tea or coffee.

The behaviour of the machine is what we can observe, by interacting with the machine. We can represent such a behaviour as a *transition diagram*:



(where ⇢ indicates the initial state)

This diagrams strongly reminds us of something very important in computer science: *automata*.

The only difference is that there is no final state. For now, we may think that all states are accepting. (We shall see however that this difference has important consequences.)

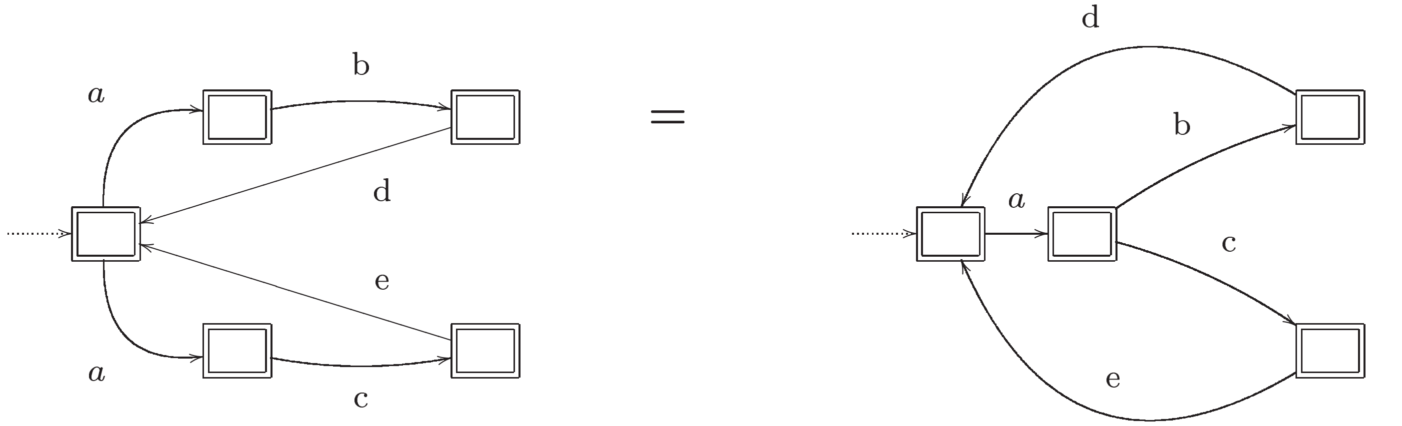
Important features of automata:

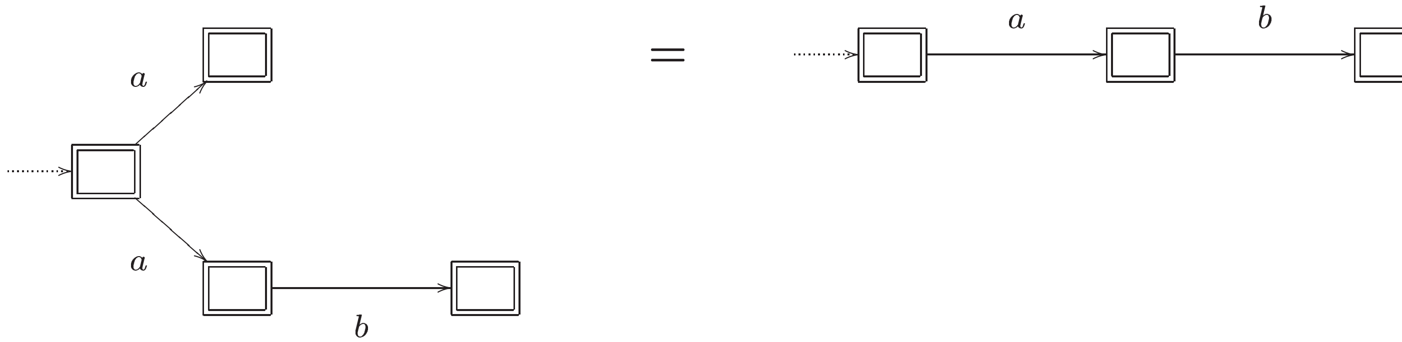
1. Behavioural equivalence is *language equivalence* (also called *trace* equivalence).
2. The language of an automata can be described as a regular expression.
3. A non-deterministic automata can be rewritten into a deterministic one (this is a consequence of the first point).

Example of equivalent automata:

(where indicates an accepting state)

More examples of equivalent automata:





The description of automata as regular languages is very important: regular languages have an algebra, which can be used for reasoning.

For instance, the reduction of non-determinististm to determinism in the examples above can be proved using the law

$$(a.\,P) + (a.\,Q) = a.\,(P + Q) \qquad (2)$$

But algebra is not the only means for reasoning on automata! (Example of another means: automata minimisation).

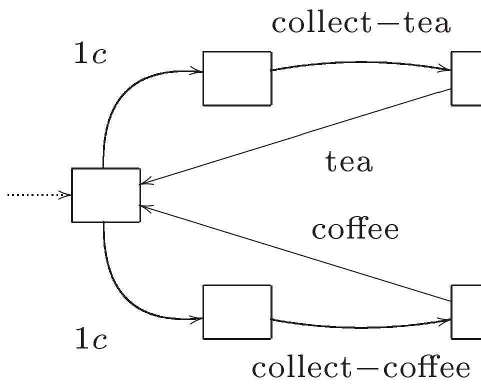
Similarly, in a *process calculus* algebra is very important; but it should not be the only tool (other tools: logics, induction, co-induction, etc.).

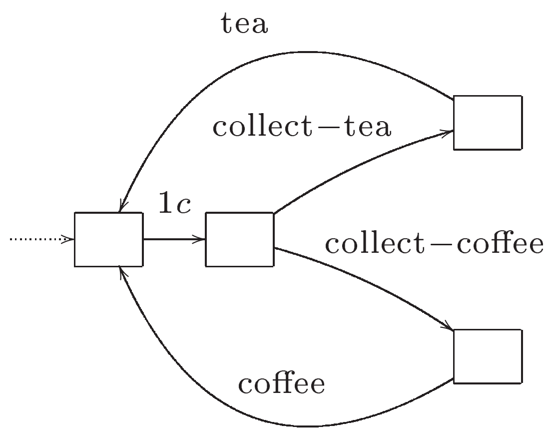
But first: is language equivalence acceptable as a notion of equivalence on processes?

Automata have final states. Hence they can still be viewed as functions (from strings to booleans).

For this reason equalities at page 13 are OK on automata.

But they are not acceptable on processes: in one case interacting with the machine can lead to deadlock! For instance, you would not consider these two vending machines equivalent:





At the same time, diagram isomorphism is too strong; cf.: the equality at page 12.

Note:

Later, when we will define a language for processes we shall see that language equivalence is the same as the contextual equivalence (1)

Language equivalence is still important in concurrency; for instance it is indeed satisfactory for confluent processes.

These examples suggest that the notion of equivalence we seek:

- should imply a tighter correspondence between transitions than language equivalence,
- should be based on the informations that the transitions convey, and not on the shape of the diagrams.

Intuitively, what does it mean for an observer that two machines are equivalent? if you do something on one machine, you must be able to the same on the other, and on the two states which the machines evolve to the same is again true. This is the idea of equivalence that we are going to formalise; it is called *bisimilarity*.

First, we formally define what these transition diagrams are.

Definition 1 *A* labeled transition system *(LTS) is a triple* $(\mathcal{P}, Act, \mathcal{T})$ *where*

- $\mathcal{P}$ *is the set of* states*, or* processes*;*
- Act *is the infinite set of* actions*;*
- $\mathcal{T} \subseteq (\mathcal{P}, Act, \mathcal{P})$ *is the* transition relation*.*

We write $P \xrightarrow{\mu} P'$ if $(P, \mu, P') \in \mathcal{T}$.

P' is a *derivative* of P if there are $P_1, \ldots, P_n, \mu_1, \ldots, \mu_n$ s.t. $P \xrightarrow{\mu_1} P_1 \ldots \xrightarrow{\mu_n} P_n$ and $P_n = P'$.

We define bisimulation on a single LTS, because: the union of two LTSs is an LTS; we will often want to compare derivatives of the same process.

Definition 2 (strong bisimulation) *A relation* $\mathcal{R}$ *on the states of an LTS is a* strong bisimulation *if whenever* $P \mathrel{\mathcal{R}} Q$*:*

1. *if* $P \xrightarrow{\mu} P'$*, then there is* Q' *such that* $Q \xrightarrow{\mu} Q'$ *and* $P' \mathrel{\mathcal{R}} Q'$*.*
2. *if* $Q \xrightarrow{\mu} Q'$*, then there is* P' *such that* $P \xrightarrow{\mu} P'$ *and* $P' \mathrel{\mathcal{R}} Q'$*.*

P *and* Q *are* strongly bisimilar*, written* $P \sim Q$*, if* $P \mathrel{\mathcal{R}} Q$*, for some strong bisimulation* $\mathcal{R}$*.*

Exercise 3 *Viewing the automata at page 12 as LTSs, prove that they are bisimilar. Are the automata at page 13 bisimilar?*

Proposition 4 *1. $\sim$ is an equivalence relation, i.e. the following hold:*

1.1. $p \sim p$ (reflexivity)
1.2. $p \sim q$ implies $q \sim p$ (symmetry)
1.3. $p \sim q$ and $q \sim r$ imply $p \sim r$ (transitivity);

2. $\sim$ itself is a strong bisimulation.

The second item of Proposition 4 suggests an alternative definition of $\sim$:

Proposition 5 *$\sim$ is the largest relation among the states of the LTS such that $P \sim Q$ implies:*

1. *if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim Q'$.*
2. *if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $P' \sim Q'$.*

Exercise 6 *Prove Propositions 4-5 (for 4.2 you have to show that $\cup\{\mathcal{R} : \mathcal{R}$ is a bisimulation $\}$ is a bisimulation).*

Exercises

We write $P \sim\mathcal{R}\sim Q$ if there are P', Q' s.t. $P \sim P'$, $P' \mathcal{R} Q'$, and $Q' \sim Q$ (and alike for similar notations).

Definition 7 (strong bisimulation up-to $\sim$) *A relation $\mathcal{R}$ on the states of an LTS is a* strong bisimulation up-to $\sim$ *if $P \mathcal{R} Q$ implies:*

1. *if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim\mathcal{R}\sim Q'$.*
2. *if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $P' \sim\mathcal{R}\sim Q'$.*

Exercise 8 *If $\mathcal{R}$ is a bisimulation up-to $\sim$ then $\mathcal{R} \subseteq \sim$. (Hint: prove that $\sim \mathcal{R} \sim$ is a bisimulation.)*

Definition 9 (simulation) *A relation $\mathcal{R}$ on the states of an LTS is a* strong simulation *if $P \mathcal{R} Q$ implies:*

1. *if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \mathcal{R} Q'$.*

P is strongly simulated by *Q, written $P < Q$, if $P \mathcal{R} Q$, for some strong simulation $\mathcal{R}$.*

Exercise* 10 *Does $P \sim Q$ imply $P < Q$ and $Q < P$? What about the converse? (Hint for the second point: think about the 2nd equality at page 13.)*

Bisimulation has been introduced by Park (1981) and made popular by Milner.

Bisimulation is a robust notion: characterisations of bisimulation have been given in terms of non-well-founded-sets (Aczel), modal logic (Hennessy, Milner), final coalgebras (Rutten, Turi), open maps in category theory (Joyal, Nielsen, Winskel).

Bisimulation has also been advocated outside concurrency theory; for instance for reasoning about elements of recursively defined domains, and data types (Fiore, Pitts) and for reasoning about equivalence in functional programs (Abramsky).

But the most important feature of bisimulation is the associated *co-inductive* proof technique.

Part III: Induction and co-induction

Co-inductive definitions and co-inductive proofs

Consider this definition of $\sim$ (Proposition 5):

$\sim$ is the largest relation among the states of the LTS such that $P \sim Q$ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim Q'$.
2. if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $P' \sim Q'$.

It is a circular definition; does it make sense?

We claimed that we can prove $(P, Q) \in \sim$ by showing that $(P, Q) \in \mathcal{R}$ and $\mathcal{R}$ is a *bisimulation relation*, that is a relation that satisfies the same clauses as $\sim$. Does such a proof technique make sense?

Contrast all this with the usual, familiar *inductive definitions* and *inductive proofs*.

The above definition of $\sim$, and the above proof technique for $\sim$ are examples of *co-inductive definition* and of *co-inductive proof technique*.

Bisimulation and co-induction: what are we talking about?
Has *co*-induction anything to do with induction?

An example of an inductive definition: reduction to a value in the λ-calculus

The set Λ of λ-terms (this is also an inductive def!)

$$e ::= x \;\Big|\; \lambda x.\, e \;\Big|\; e_1(e_2)$$

Consider the definition of $\Downarrow_n$ in λ-calculus (the same can be done for TFun, and extensions of it):

$$\frac{}{\lambda x.\, e \Downarrow_n \lambda x.\, e}$$

$$\frac{e_1 \Downarrow_n \lambda x.\, e_0 \qquad e_0\{e_2/x\} \Downarrow_n e'}{e_1(e_2) \Downarrow_n e'}$$

$\Downarrow_n$ is the *smallest* relation on λ-terms that is closed forwards under these rules; i.e., the smallest subset C of $\Lambda \times \Lambda$ s.t.

- $\lambda x.\, e C \lambda x.\, e$ for all abstractions,
- if $e_1\ C\ \lambda x.\, e_0$ and $e_0\{e_2/x\}\ C\ e'$ then also $e_1(e_2)\ C\ e'$.

An example of a co-inductive definition: divergence in the λ-calculus

Consider the definition of $\Uparrow^n$ (divergence) in λ-calculus (the same can be done for TFun with recursive definitions):

$$\frac{e_1 \Uparrow^n}{e_1(e_2) \Uparrow^n}$$

$$\frac{e_1 \Downarrow_n \lambda x.\, e_0 \qquad e_0\{e_2/x\} \Uparrow^n}{e_1(e_2) \Uparrow^n}$$

$\Uparrow^n$ is the *largest* predicate on λ-terms that is closed backwards under these rules; i.e., the largest subset D of Λ s.t. if $e \in D$ then

- either $e = e_1(e_2)$ and $e_1 \in D$,
- or $e = e_1(e_2)$, $e_1 \Downarrow_n \lambda x.\, e_0$ and $e_0\{e_2/x\} \in D$.

Hence: to prove e is divergent it suffices to find $E \subseteq \Lambda$ that is closed backwards and with $e \in E$ (co-induction proof technique).

What is the smallest predicate closed backwards?

An example of an inductive definition: finite lists over a set A

$$\frac{}{\texttt{nil} \in \mathcal{L}} \qquad \frac{\ell \in \mathcal{L} \qquad a \in A}{\texttt{cons}(a, \ell) \in \mathcal{L}}$$

Finite lists: the set generated by these rules; i.e., the smallest set closed forwards under these rules.

Inductive proof technique for lists: Let $\mathcal{P}$ be a predicate (a property) on lists. To prove that $\mathcal{P}$ holds on all lists, prove that

- $\texttt{nil} \in \mathcal{P}$;
- $\ell \in \mathcal{P}$ implies $\texttt{cons}(a, \ell) \in \mathcal{P}$, for all $a \in A$.

An example of an co-inductive definition: finite and infinite lists over a set A

$$\frac{}{\texttt{nil} \in \mathcal{L}} \qquad \frac{\ell \in \mathcal{L} \qquad a \in A}{\texttt{cons}(a, \ell) \in \mathcal{L}}$$

Finite and infinite lists: the largest set closed backwards under these rules.

To explain finite and infinite lists as a set, we need *non-well-founded sets* (Forti-Honsell, Aczel).

- * An inductive definition tells us what are the *constructors* for generating all the elements (cf: closure forwards).
  * A co-inductive definition tells us what are the *destructors* for decomposing the elements (cf: closure backwards).
    The destructors show what we can *observe* of the elements (think of the elements as black boxes; the destructors tell us what we can do with them; this is clear in the case of infinite lists).
- When a definition is give by means of some rules:
  * if the definition is inductive, we look for the smallest universe in which such rules live.
  * if it is co-inductive, we look for the largest universe.

The duality

constructors	destructors
inductive defs	co-inductive defs
induction technique	co-inductive technique
congruence	bisimulation
least fixed-points	greatest fixed-points

(The dual of a *bisimulation* is a *congruence* because intuitively: a bisimulation is a relation that is "closed backwards", a congruence is a relation that is "closed forwards".)

- In what sense are $\Downarrow_n, \Uparrow^n, \sim$ fixed-points?
- What is the co-induction proof technique?
- In what sense is co-induction dual to the familiar induction technique?

What follows answers these questions. It is a simple application of fixed-point theory. It is not needed for the remainder of the course.

To make things simpler, we work on *powersets*. (It is possible to be more general, working with universal algebras or category theory.)

For a given set S, the powerset of S, written $\wp(S)$, is

$$\wp(S) \stackrel{\text{def}}{=} \{T \ : \ T \subseteq S\}$$

$\wp(S)$ is a *complete lattice*.

Complete lattices are "dualisable" structures: reverse the arrows and you get another complete lattice.

From Tarsky's theorem for complete lattices, we know that if $\mathcal{F} : \wp(S) \to \wp(S)$ is monotone, then $\mathcal{F}$ has a least fixed-point (lfp), namely:

$$\mathcal{F}_{\mathrm{lfp}} \stackrel{\mathsf{def}}{=} \bigcap \{A \;:\; \mathcal{F}(A) \subseteq A\}$$

As we are on a complete lattice, we can dualise the statement:

If $\mathcal{F} : \wp(S) \to \wp(S)$ is monotone, then $\mathcal{F}$ has a greatest fixed-point (gfp), namely:

$$\mathcal{F}^{\mathrm{gfp}} \stackrel{\mathsf{def}}{=} \bigcup \{A \;:\; A \subseteq \mathcal{F}(A)\}$$

These results give us proof techniques for $\mathcal{F}_{\mathrm{lfp}}$ and $\mathcal{F}^{\mathrm{gfp}}$:

$$\text{if } \mathcal{F}(A) \subseteq A \text{ then } \mathcal{F}_{\mathrm{lfp}} \subseteq A \tag{3}$$

$$\text{if } A \subseteq \mathcal{F}(A) \text{ then } A \subseteq \mathcal{F}^{\mathrm{gfp}} \tag{4}$$

– Inductive definitions give us lfp's (precisely: an inductive definition tells us how to construct the lfp). Co-inductive definitions give us gfp's.
– On inductively-defined sets (3) is the same as the familiar induction technique (cf: example of lists). (4) gives us the co-inductive proof technique.

$\Downarrow_n$ and $\Uparrow^n$ as fixed-points

A set R of rules on a set S give us a function $\mathcal{R}$: $\wp(S) \to \wp(S)$, so defined:

$$\mathcal{R}\ (A) \stackrel{\text{def}}{=} \{a \ : \quad \text{there are } a_1, \ldots, a_n \in A \quad \text{and a rule in } R \\ \text{so that using } a_1, \ldots, a_n \text{ as premises in the rule we can derive } a\}$$

$\mathcal{R}$ is monotone, and therefore (by Tarsky) has lfp and gfp.

In this way, the definitions of $\Downarrow_n$ and $\Uparrow^n$ can be formulated as lfp and gfp of functions.

For instance, in the case of $\Uparrow^n$, take $S = \Lambda$. Then

$$\mathcal{R}(A) = \{e_1(e_2) \ : \ e_1 \in A, \quad \text{or } e_1 \Downarrow_n \lambda x.\, e_0 \text{ and } e_0\{^{e_2}/_{x}\} \in A\}.$$

The co-induction proof technique for $\Uparrow^n$ mentioned at page 25 is just an instance of (4).

Bisimulation as a fixed-point

Let $(\mathcal{P}, Act, \mathcal{T})$ be an LTS. Consider the function $\mathcal{F} : \wp(\mathcal{P} \times \mathcal{P}) \to \wp(\mathcal{P} \times \mathcal{P})$ so defined.

$\mathcal{F}(R)$ is the set of all pairs (P, Q) s.t.:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $(P', Q') \in \mathcal{R}$.
2. if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $(P', Q') \in \mathcal{R}$.

Proposition 11 *1. $\mathcal{F}$ is monotone;*
2. $\sim = \mathcal{F}^{\mathrm{gfp}}$;
3. R is a bisimulation iff $R \subseteq \mathcal{F}(\mathcal{R})$.

The induction technique as a fixed-point technique: the example of finite lists

Let $\mathcal{F}$ be this function (from sets to sets):

$$\mathcal{F}(S) \stackrel{\text{def}}{=} \{\mathtt{nil}\} \cup \{\mathtt{cons}(a, s) \ : \ a \in A, s \in S\}$$

$\mathcal{F}$ is monotone, and $\mathrm{finLists} = \mathcal{F}_{\mathrm{lfp}}$. (i.e., $\mathrm{finLists}$ is the smallest set solution to the equation $\mathcal{L} = \mathtt{nil} + \mathtt{cons}(A, \mathcal{L})$).

From (3), we infer: Suppose $\mathcal{P} \subseteq \mathrm{finLists}$. If $\mathcal{F}(\mathcal{P}) \subseteq \mathcal{P}$ then $\mathcal{P} \subseteq \mathrm{finLists}$ (hence $\mathcal{P} = \mathrm{finLists}$).

Proving $\mathcal{F}(\mathcal{P}) \subseteq \mathcal{P}$ requires proving

- $\mathtt{nil} \in \mathcal{P}$;
- $\ell \in \mathrm{finLists} \cap \mathcal{P}$ implies $\mathtt{cons}(a, \ell) \in \mathcal{P}$, for all $a \in A$.

This is the same as the familiar induction technique for lists (page 26).

Note: $\mathcal{F}$ is defined the class of all sets, rather than on a powerset; the class of all sets is not a complete lattice (because of paradoxes such as Russel's), but the constructions that we have seen for lfp and gfp of monotone functions apply.

$\boxed{\text{Continuity}}$

Another important theorem of fixed-point theory: if $\mathcal{F} : \wp(S) \rightarrow \wp(S)$ is continuous, then

$$\mathcal{F}_{\mathrm{lfp}} = \bigcup_n \mathcal{F}^n(\bot)$$

This has, of course, a dual, for gfp (also the definition of continuity has to be dualised), but: the function $\mathcal{F}$ of which bisimilarity is the gfp may not be continuous! (This is usually the case for *weak* bisimilarity, that we shall introduce later.)

It is continuous only if the LTS is finite-branching, meaning that for all P the set $\{P' \ : \ P \xrightarrow{\mu} P', \text{ for some } \mu\}$ is finite.

On a finite branching LTS, it is indeed the case that

$$\sim = \bigcap_n \mathcal{F}^n(\mathcal{P} \times \mathcal{P})$$

where $\mathcal{P}$ is the set of all processes.

Exercise 12 *Let μ^+ range over non-empty sequences of actions. Consider the following definition of strong bisimulation:*

> *A relation $\mathcal{R}$ on the states of an LTS is a* strong bisimulation *if $P \ \mathcal{R} \ Q$ implies:*
>
> 1. *if $P \xrightarrow{\mu^+} P'$, then there is Q' such that $Q \xrightarrow{\mu^+} Q'$ and $P' \ \mathcal{R} \ Q'$;*
> 2. *if $Q \xrightarrow{\mu^+} Q'$, then there is P' such that $P \xrightarrow{\mu^+} P'$ and $P' \ \mathcal{R} \ Q'$.*

Prove that the definition above is the same as Definition 2.

Part IV: A process calculus: CCS

A language of sequential processes (or: finite-state LTSs)

P, Q, R range over *processes*; $a, b \ldots$ over *names*. $\overline{a}, \overline{b}$ are *co-names*. Names, co-names and τ form the set Act of *actions*. μ, λ range over Act. Actions a and $\overline{a}$ are complementary actions (think of a as an input, $\overline{a}$ as an output). We assume $\overline{\overline{a}} = a$, and τ different from any name or co-name.

Differences with the languages of automata (regular languages):

- iteration replaced by recursion;
- sequential composition between processes replaced by composition between an action and a process, which is simpler. Later, we shall see how to derive sequential composition between processes.

Summation: $\sum_{i \in I} \alpha_i . P_i$, where I is a finite indexing set, and $\alpha_i \in Act$.

$$\texttt{Sum}\frac{}{\sum_{i \in I} \alpha_i . P_i \xrightarrow{\alpha_i} P_i}$$

There are actually 2 operators in this construct: choice and prefixing. For technical reasons, it is convenient to combine them so (cf.: the congruence of weak bisimilarity, page 50).

Prefixing gives us sequential composition; sum gives us choice.

α is a *prefix*; in pure CCS prefixes and actions (i.e.,α and μ) coincide; later, when we add value-passing, they will be different.

We use M, N to range over summations (terms of the form $\sum_{i \in I} \alpha_i . P_i$).

Abbreviation: 0 if I is empty; $M + N$ for binary summation.

Constants (recursive process definitions): K

We assume an infinite set of *constants*, ranged over by K. Each constant that is used must have a defining equation of the form $K \stackrel{\circ}{=} P$.

$$\texttt{Const}\frac{P \xrightarrow{\mu} P'}{K \xrightarrow{\mu} P'} \text{ if } K \stackrel{\circ}{=} P$$

Communicating processes

Complex systems are composed of components, that interact *both* with each other *and* with the environment.

A component cam be a register, a bus, a memory, a program, a wire etc.

We need operators for composing processes in parallel, so that they can interact with each other. Such operators are missing in automata.

Parallel composition: $P_1 \mid P_2$

$$\texttt{ParL}\frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2}$$

$$\texttt{Com1}\frac{P_1 \xrightarrow{a} P_1' \qquad P_2 \xrightarrow{\overline{a}} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2'}$$

plus their symmetric versions `ParR` and `Com2`.

τ indicates an *internal activity* of the system; the system $P_1 \mid P_2$ is doing "some work".

> Communication is handshaking.
> Idea: everything is a process; including a memory, a communication medium such as a bus, a wire, a buffer.
> Several communication mechanisms (such as communication via shared variables, or buffered communications) can be modeled.

Restriction: $\boldsymbol{\nu} a\ P$

$$\texttt{Res}\frac{P \xrightarrow{\mu} P'}{\boldsymbol{\nu} a\ P \xrightarrow{\mu} \boldsymbol{\nu} a\ P'}\ a \text{ not in } \mu$$

Restriction gives us encapsulation. We need it, for instance, for modeling a printer that is private to a group of processes. In $\boldsymbol{\nu} a\ P$, the head is a binder for a, with scope P. Abbreviation: $(\boldsymbol{\nu} a_1, \ldots, a_n)P$ for $\boldsymbol{\nu} a_1 \ldots \boldsymbol{\nu} a_n\ P$.

The calculus CCS (Calculus of Communicating Systems)

$a, b, \ldots, \ldots$ *Names*

Processes

$$
\begin{array}{lcll}
P & ::= & \sum_{i \in I} \alpha_i . P_i & \text{summation} \\
 & \mid & P_1 \mid P_2 & \text{parallel composition} \\
 & \mid & \boldsymbol{\nu} a \, P & \text{restriction} \\
 & \mid & K & \text{constants (rec defs)}
\end{array}
$$

Actions

$$
\begin{array}{lcll}
\alpha & ::= & a & \text{input} \\
 & \mid & \overline{a} & \text{output} \\
 & \mid & \tau & \text{silent action}
\end{array}
$$

We abbreviate $\alpha . 0$ as α. Par. comp. and summation have the least syntactic precedence.

CCS is due to Robin Milner (end 70's). Similar motivations led to Tony Hoare's CSP.

Exercises

Exercise 13 *1. Which transitions can the process* $P \stackrel{def}{=} \boldsymbol{\nu} a\,((a + b) \mid \overline{a})$ *do?*

2. Find a process Q *in which there is no parallel composition and restriction and with* $P \sim Q$.

Exercise 14 *Draw a transition diagram for these processes:*

$$\begin{array}{rcl} K_1 & \stackrel{\circ}{=} & a.\,(\tau.\,K_1 + b) + \tau.\,a.\,K_1 \\ K_2 & \stackrel{\circ}{=} & \tau.\,\boldsymbol{\nu} a\,(a \mid (\overline{a} + b)) + c.\,K_3 \\ K_3 & \stackrel{\circ}{=} & d.\,K_3 \end{array}$$

Exercise 15 *Draw a transition diagram for the process* $\boldsymbol{\nu} c\,(K_1 \mid K_2)$, *where*

$$\begin{array}{rcl} K_1 & \stackrel{\circ}{=} & a.\,\overline{c}.\,K_1 \\ K_2 & \stackrel{\circ}{=} & b.\,c.\,K_2 \end{array}$$

Part V: Algebraic and operational theory of processes

Two important words for us: *congruence*, *bisimilarity*. We are interested in former because the process language is *inductively* defined; in the latter because behavioural equality is *co-inductively* defined.

The *process contexts* are defined by this grammar:

$$C ::= [\cdot] \;\Big|\; \alpha.\,C + M \;\Big|\; \boldsymbol{\nu} a\, C \;\Big|\; C \mid P \;\Big|\; P \mid C$$

A relation $\mathcal{R}$ on CCS processes is a *congruence* relation if it is an equivalence and $P \;\mathcal{R}\; Q$ implies $C[P] \;\mathcal{R}\; C[Q]$ for all process contexts C.

Theorem 16 *$\sim$ is a congruence relation, i.e., $P \sim Q$ implies $C[P] \sim C[Q]$, for all process contexts C.*

Proof: A useful exercise. ■

Some laws

$$
\begin{array}{rcll}
\sum_{i\in I} \alpha_i.\, P_i & \sim & \sum_{j \in J} \alpha_j.\, P_j & \text{if } J \text{ is a permutation of } I \\
N + M + M & \sim & N + M & \\
 & & & \\
P \mid Q & \sim & Q \mid P & \\
P \mid (Q \mid R) & \sim & (P \mid Q) \mid R & \\
P \mid 0 & \sim & P & \\
 & & & \\
\boldsymbol{\nu} a\, (P \mid Q) & \sim & (\boldsymbol{\nu} a\, P) \mid Q & \text{if } a \text{ not free in } Q \text{ (including the constants in } Q) \\
\boldsymbol{\nu} a\, \boldsymbol{\nu} b\, P & \sim & \boldsymbol{\nu} b\, \boldsymbol{\nu} a\, P & \\
\boldsymbol{\nu} a\, P & \sim & \boldsymbol{\nu} b\, (P\{b/a\}) & \text{if } b \text{ is fresh (cf.: alpha conversion)} \\
\boldsymbol{\nu} a\, (\alpha.\, P + M) & \sim & \boldsymbol{\nu} a\, M & \text{if } \alpha = a \text{ or } \alpha = \overline{a} \\
\boldsymbol{\nu} a\, (\sum_{i\in I} \alpha_i.\, P_i) & \sim & \sum_{i\in I} \alpha_i.\, \boldsymbol{\nu} a\, P_i & \text{if } a \text{ not in } \alpha_i, \text{ for all } i. \text{ (Hence } \boldsymbol{\nu} a\, 0 \sim 0.) \\
 & & & \\
K & \sim & P & \text{if } K \triangleq P
\end{array}
$$

The expansion law

Proposition 17 *For all* $n \geq 0$ *and* $P_1, \ldots, P_n$*:*

$$P_1 \mid \cdots \mid P_n \sim \left\{ \begin{array}{l} \sum\{\alpha.\,(P_1 \mid \cdots \mid P'_i \mid \cdots \mid P_n) \;:\; P_i \xrightarrow{\alpha} P'_i\} \\ + \sum\{\tau.\,(P_1 \mid \cdots \mid P'_i \mid \cdots \mid P'_j \mid \cdots \mid P_n) \;:\; \quad 1 \leq i < j \leq n, \\ \hfill P_i \xrightarrow{\mu} P'_i, P_j \xrightarrow{\overline{\mu}} P'_j \;\} \end{array} \right.$$

Proof: Using induction on n and the bisimulation proof technique. ■

Proposition 18 *For all* $n \geq 0$ *and* $P_1, \ldots, P_n$*:*

$$\boldsymbol{\nu} a(P_1 \mid \cdots \mid P_n) \sim \left\{ \begin{array}{l} \sum\{\alpha.\,\boldsymbol{\nu} a\,(P_1 \mid \cdots \mid P'_i \mid \cdots \mid P_n) \;:\; P_i \xrightarrow{\alpha} P'_i \textit{ and } \alpha \neq a, \overline{a}\} \\ + \sum\{\tau.\,\boldsymbol{\nu} a\,(P_1 \mid \cdots \mid P'_i \mid \cdots \mid P'_j \mid \cdots \mid P_n) \;:\; \quad 1 \leq i < j \leq n, \\ \hfill P_i \xrightarrow{\mu} P'_i, P_j \xrightarrow{\overline{\mu}} P'_j \;\} \end{array} \right.$$

Proof: By Proposition 17 and the laws of restriction. ■

Exercises

Exercise 19 *Prove some of the laws in page 45.*

Exercise 20 *Is law (2) (at page 13) valid?*

Exercise 21 *Prove, algebraically:* $\boldsymbol{\nu} b\,(a.\,(b \mid c) + \tau.\,(b \mid \overline{b}.\,c)) \sim \tau.\,\tau.\,c + a.\,c$.

Exercise 22 *Let* $K \stackrel{\circ}{=} a.\,K$. *Prove that* $K \mid K \sim K$.

Exercise 23 *Use the laws of* $\sim$ *and the technique of "strong bisimulation up-to" to prove that* $\boldsymbol{\nu} c\,(K_1 \mid K_2) \sim H$ *for* K_1, K_2 *as defined in Exercise 15 and* $H \stackrel{\circ}{=} a.\,b.\,\tau.\,H + b.\,a.\,\tau.\,H$.

Exercise 24 (semaphores) *Here are the specifications of unary and binary semaphores:*

$$K_1 \stackrel{\circ}{=} p.\,v.\,K_1 \qquad\qquad \begin{array}{rcl} K_2 & \stackrel{\circ}{=} & p.\,K_2' \\ K_2' & \stackrel{\circ}{=} & p.\,v.\,K_2' + v.\,K_2 \end{array}$$

Prove that $K_1 \mid K_1 \sim K_2$

$\boxed{\text{Axiomatisation}}$

Let $\mathcal{A}$ be the expansion laws of Propositions 17 and 18 plus the laws that involve summation or restriction at page 45. Write $\mathcal{A} \vdash P \sim Q$ if $P \sim Q$ can be inferred from the laws of $\mathcal{A}$ by equational reasoning (equational reasoning means: you can use the laws that say that $\sim$ is a congruence relation).

Call a process *finite* if it does not contain constants.

Theorem 25 *Let P, Q be finite process. Then $P \sim Q$ iff $\mathcal{A} \vdash P \sim Q$.*

Proof: Implication from left to right: show that each law in $\mathcal{A}$ is sound for $\sim$.

Opposite direction: first show that each finite processes can be proved equal to a process in normal form, that is a process in which there is no constant, parallel composition and restriction. Then prove that if two normal forms are strongly bisimilar, they can be equated using the laws in $\mathcal{A}$. ■

Part VI: Weak bisimilarity

Consider the processes

$$\tau.\, \overline{a}.\, 0 \qquad \text{and} \qquad \overline{a}.\, 0$$

They are not strongly bisimilar.

But we do want to regard them as behaviourally equivalent! τ-transitions represent internal activities of processes, which are not visible.

(Analogy in functional languages: $(\lambda x.\, x)3$ and 3 are semantically the same.)

Internal work (τ-transitions) should be ignored in the bisimulation game. Define:

(i) $\Longrightarrow$ as the reflexive and transitive closure of $\xrightarrow{\tau}$.

(ii) $\stackrel{\mu}{\Longrightarrow}$ as $\Longrightarrow\xrightarrow{\mu}\Longrightarrow$ (relational composition).

(iii) $\stackrel{\widehat{\mu}}{\Longrightarrow}$ is $\Longrightarrow$ if $\mu = \tau$; it is $\stackrel{\mu}{\Longrightarrow}$ otherwise.

Definition 26 (weak bisimulation (or observation equivalence)) *A process relation $\mathcal{R}$ is a* weak bisimulation *if $P \mathcal{R} Q$ implies:*

1. *if $P \stackrel{\mu}{\Longrightarrow} P'$, then there is Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \mathcal{R} Q'$;*
2. *the converse of (1) on the actions from Q.*

P and Q are weakly bisimilar, *written $P \approx Q$, if $P \mathcal{R} Q$ for some weak bisimulation $\mathcal{R}$.*

Weak bisimulation will be for us *the* behavioural equivalence on processes.

Why did we study strong bisimulation?

- $\sim$ is simpler to work with, and $\sim \subseteq \approx$; (cf: exp. law)
- the theory of $\approx$ is in many aspects similar to that of $\sim$;
- the differences between $\sim$ and $\approx$ correspond to subtle points in the theory of $\approx$

If we want to ignore τ-transitions, why should we apply the clauses of Definition 26 also in the case $\mu = \tau$?

- processes can internally be non-deterministic. Without this requirement we would equate $\tau.\, 0 + \tau.\, \overline{a}.\, 0$ and $\overline{a}.\, 0$.
- without this requirement we lose congruence for parallel composition.

Examples of non-equivalence:

$$a + b \not\approx a + \tau.\, b \not\approx \tau.\, a + \tau.\, b \not\approx a + b$$

Examples of equivalence:

$$\tau.\, a \approx a \approx a + \tau.\, a$$

$$a.\, (b + \tau.\, c) \approx a.\, (b + \tau.\, c) + a.\, c$$

These are instances of useful algebraic laws, called the τ *laws*:

Lemma 27 *1.* $P \approx \tau.\, P$*;*
2. $\tau.\, N + N \approx N$*;*
3. $M + \alpha.\, (N + \tau.\, P) \approx M + \alpha.\, (N + \tau.\, P) + \alpha.\, P$*.*

Theorem 28 $\approx$ *is a process congruence.*

That is, $P \approx Q$ implies:

1. $\boldsymbol{\nu} a\, P \approx \boldsymbol{\nu} a\, Q$;
2. $P \mid R \approx Q \mid R$ and $R \mid P \approx R \mid Q$;
3. $\alpha.\, P + M \approx \alpha.\, Q + M$.

However, on summands, $\approx$ is not a congruence: $\tau.\, a \approx a$, but $\tau.\, a + b \not\approx a + b$. This is not very disturbing: we will always compare processes.

In the clauses of Definition 26, the use of $\stackrel{\mu}{\Longrightarrow}$ on the challenger side can be heavy. For instance, take $K \stackrel{\circ}{=} \tau.(a \mid K)$; for all n, we have $K \Longrightarrow (a \mid)^n \mid K$, and all these transitions have to be taken into account in the bisimulation game.

The following definition is much simpler to use (the challenger makes a single move):

Definition 29 *A process relation* $\mathcal{R}$ *is a* weak bisimulation *if* $P \mathcal{R} Q$ *implies:*

1. *if* $P \xrightarrow{\mu} P'$, *then there is* Q' *s.t.* $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ *and* $P' \mathcal{R} Q'$;
2. *the converse of (1) on the actions from* Q *(ie, the roles of* P *and* Q *are inverted).*

Proposition 30 *The definitions 26 and 29 of weak bisimulation coincide.*

Proof: A useful exercise. ■

Exercises

Exercise 31 *Prove these equivalences (all cases can be done purely algebraically or using the bisimulation technique):*

$$\begin{array}{rcll} (\tau.\, P) \mid Q & \approx & \tau.\, (P \mid Q) & \text{for all } P, Q \\ \boldsymbol{\nu} a\, (\tau.\, \overline{b} \mid c) & \approx & \overline{b} \mid c & \\ \boldsymbol{\nu} a\, (b.\, \overline{a} \mid a.\, c) & \approx & b.\, c & \end{array}$$

Exercise 32 *Explain why* $\tau.\, (\tau.\, a + b) + \tau.\, b \not\approx \tau.\, a + \tau.\, b$

Exercise 33 *Is it true that* $K \approx a$, *where* $K \stackrel{\circ}{=} \tau.\, K + a$*?*

Exercise 34 *Let* $K_1 \stackrel{\circ}{=} a.\, (b.\, K_1 \mid c)$, $K_1' \stackrel{\circ}{=} a.\, (b.\, c.\, K_1' + c.\, b.\, K_1')$, $K_2 \stackrel{\circ}{=} \overline{a}.\, K_2$, $H_1 \stackrel{\circ}{=} a.\, b.\, H_1$, $H_2 \stackrel{\circ}{=} \overline{a}.\, c.\, H_2$, $H \stackrel{\circ}{=} a.\, b.\, H + b.\, a.\, H$. *Explain why* $\boldsymbol{\nu} a\, (K_1 \mid K_2) \not\approx \boldsymbol{\nu} a\, (H_1 \mid H_2)$. *Prove that* $\boldsymbol{\nu} a\, (K_1' \mid K_2) \approx H \approx \boldsymbol{\nu} a\, (H_1 \mid H_2)$. *(Hint: for* $\approx$*, you may find Exercises 23 useful; then either use algebraic laws and unique solutions of equations (page 56), or define an appropriate weak bisimulation.)*

Weak bisimulations "up-to"

Definition 35 (weak bisimulation up-to $\sim$) *A process relation $\mathcal{R}$ is a* weak bisimulation up-to $\sim$ *if $P \; \mathcal{R} \; Q$ implies:*

1. *if $P \xrightarrow{\mu} P'$, then there is Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \sim\mathcal{R}\sim Q'$;*
2. *the converse of (1) on the actions from Q.*

Exercise 36 *If $\mathcal{R}$ is a weak bisimulation up-to $\sim$ then $\mathcal{R} \subseteq \approx$.*

Definition 37 (weak bisimulation up-to $\approx$) *A process relation $\mathcal{R}$ is a* weak bisimulation up-to $\approx$ *if $P \; \mathcal{R} \; Q$ implies:*

1. *if $P \stackrel{\mu}{\Longrightarrow} P'$, then there is Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \approx\mathcal{R}\approx Q'$;*
2. *the converse of (1) on the actions from Q.*

Exercise 38 *If $\mathcal{R}$ is a weak bisimulation up-to $\approx$ then $\mathcal{R} \subseteq \approx$.*

Unique solutions of equations

Theorem 39 *Let $X_1, X_2, \ldots$ be a (possibly infinite) sequence of process variables. Consider the following system of equations:*

$$\begin{array}{lll} X_1 & \approx & \alpha_{11}.\, X_{f(11)} + \ldots + \alpha_{1n_1}.\, X_{f(1n_1)} \\ X_2 & \approx & \alpha_{21}.\, X_{f(21)} + \ldots + \alpha_{2n_2}.\, X_{f(2n_2)} \\ \ldots & & \ldots \end{array}$$

where $\alpha_{ij} \neq \tau$ for all i, j, and f is a function from pairs of integers to integers. There is, modulo $\approx$, a unique sequence of processes $P_1, P_2 \ldots$ that is solution to this system of equations.

Of course there is at least one solution! Take constants $K_1, K_2, \ldots$ and define them thus, for all i:

$$K_i \quad \stackrel{\circ}{=} \quad \alpha_{i1}.\, K_{f(i1)} + \ldots + \alpha_{in_i}.\, K_{f(in_i)}$$

There can be more solutions if some of the prefixes α_{ij} is τ. Example: every process is a solution to $X \approx \tau.\, X$.

There is an analogous theorem for strong bisimulation (replace all $\approx$ with $\sim$). In this case, the hypothesis $\alpha_{ij} \neq \tau$ is not needed.

Proof: [of Theorem 39] Suppose $P_1, P_2, \ldots$ and $Q_1, Q_2, \ldots$ are solutions to that system. Then

$$\mathcal{R} \stackrel{\mathsf{def}}{=} \{(P_i, Q_i) \;:\; i = 1, 2, \ldots\}$$

is a weak bisimulation up-to $\approx$. ■

Exercises

Exercise 40 *Let* $K_1 \stackrel{\circ}{=} f\,.\,a\,.\,\overline{d}\,.\,K_1$, $K_2 \stackrel{\circ}{=} d\,.\,b\,.\,\overline{e}\,.\,K_2$, $K_3 \stackrel{\circ}{=} \overline{f}\,.\,e\,.\,c\,.\,K_3$, $H \stackrel{\circ}{=} a\,.\,b\,.\,c\,.\,H$. *Prove that* $(\boldsymbol{\nu} d, e, f\,)(K_1 \mid K_2 \mid K_3) \approx H$.

Exercise 41 *Let* $H_1 \stackrel{\circ}{=} a\,.\,\overline{c_1}\,.\,e_1\,.\,d$, $H_2 \stackrel{\circ}{=} b\,.\,\overline{c_2}$, $\mathrm{Sync} \stackrel{\circ}{=} c_1\,.\,c_2\,.\,\overline{e_1}$. *Prove that*
$(\boldsymbol{\nu} c_1, c_2, e_1\,)(H_1 \mid H_2 \mid \mathrm{Sync}) \approx a\,.\,b\,.\,d + b\,.\,a\,.\,d$.

Exercise* 42 *Let* $H_1 \stackrel{\circ}{=} a\,.\,\overline{c_1}\,.\,e_1\,.\,H_1$, $H_2 \stackrel{\circ}{=} b\,.\,\overline{c_2}\,.\,e_2\,.\,H_2$, $\mathrm{Sync} \stackrel{\circ}{=} c_1\,.\,c_2\,.\,\overline{e_1}\,.\,\overline{e_2}\,.\,\mathrm{Sync}$. *Prove that*

$$(\boldsymbol{\nu} c_1, c_2, e_1, e_2\,)(H_1 \mid H_2 \mid \mathrm{Sync}) \approx H$$

for $H \stackrel{\circ}{=} a\,.\,b\,.\,H + b\,.\,a\,.\,H$.

Note: process Sync is used to synchronise 2 processes. Similarly synchronisers for $n > 2$ processes can be defined.

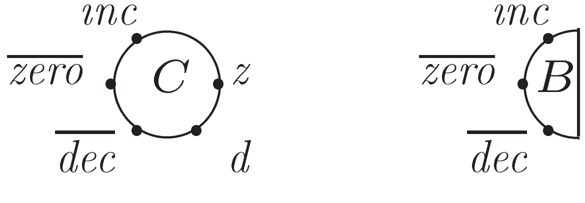
Exercise* 43 *In Exercise 12 we proved that strong bisimulation can also be defined on sequences of actions. Prove something analogous for weak bisimulation. (The crux is to find the right definition of weak bisimulation on sequences of actions).*

An example: the specification and an implementation of a counter

The specification:

$$
\begin{array}{rcl}
Count_0 & \stackrel{\circ}{=} & inc.\ Count_1 + \overline{zero}.\ Count_0 \\
Count_{n+1} & \stackrel{\circ}{=} & inc.\ Count_{n+2} + \overline{dec}.\ Count_n
\end{array}
$$

We implement $Count_n$ it as a chain of simple communicating cells: n cells C followed by a barrier cell B.



For the definition of the cell C we use an auxiliary constant D.

$$\begin{array}{lcl} B & \stackrel{\circ}{=} & inc.\,(C \frown B) + \overline{zero}.\,B \\ C & \stackrel{\circ}{=} & inc.\,(C \frown C) + \overline{dec}.\,D \\ D & \stackrel{\circ}{=} & d.\,C + z.\,B \end{array}$$

where $\frown$ is this abbreviation:

$$P \frown Q \stackrel{\text{def}}{=} (\boldsymbol{\nu} i', z', d')(P\{^{i', z', d'}/_{i, z, d}\} \mid Q\{^{i', z', d'}/_{inc, zero, dec}\})$$

Lemma 44 *$\frown$ is associative, i.e., $P \frown (Q \frown R) \sim (P \frown Q) \frown R$.*

Lemma 45 *$D \frown C \approx C \frown D$, and $D \frown B \approx B$.*

Theorem 46 *$Count_n \approx \overbrace{C \frown \cdots \frown C}^{n \text{ times}} \frown B$.*

Proof: Use Lemma 44 and 45 to prove that $\overbrace{C \frown \cdots \frown C}^{n \text{ times}} \frown B$ satisfies the defining equations of $Count_n$, for all n. ■

Similarly to counters, we can write structures like stacks, or queues (however, for this the value-passing CCS, that will be introduced soon, is more appropriate). A Turing Machine can be simulated by 2 stacks and a finite-state automata. Therefore: Turing Machine can be simulated in CCS. Hence $\approx$ is undecidable (in the same way one can prove that also $\sim$ is undecidable).

Part VII: Value-passing, examples, ...

The value-passing calculus: examples

A one-place buffer: $Buf_1 \stackrel{\circ}{=} \mathtt{in}(x).\, \overline{\mathtt{out}}\langle x\rangle.\, Buf_1$

A two-place buffer:

$$\begin{array}{rcl} Buf_2 & \stackrel{\circ}{=} & \mathtt{in}(x).\, B_2\langle x\rangle \\ B_2 & \stackrel{\circ}{=} & (x).\, (\mathtt{in}(y).\, \overline{\mathtt{out}}\langle x\rangle.\, B_2\langle y\rangle + \overline{\mathtt{out}}\langle x\rangle.\, Buf_2) \end{array}$$

Another two-place buffer:

$$ImpBuf_2 \stackrel{\circ}{=} \boldsymbol{\nu c}\, (Buf_1\{{}^{c}\!/\mathtt{out}\} \mid Buf_1\{{}^{c}\!/\mathtt{in}\})$$

We have: $Buf_2 \approx ImpBuf_2$.

A memory register:

$$Reg \stackrel{\circ}{=} (x).\, \overline{\mathtt{read}}\langle x\rangle.\, Reg\langle x\rangle + \mathtt{write}(y).\, Reg\langle y\rangle$$

Value-passing CCS: syntax and transition rules

A *data expressions* e is build out of variables $x, y..$, values v, w (such as $0, 1, 2, \mathtt{true}, \mathtt{false}$), using any operator we wish.

The syntax of prefixes and actions is now:

$$
\begin{array}{rcl}
\alpha & ::= & a(x) \;\Big|\; \overline{a}\langle e\rangle \;\Big|\; \tau \\
\mu & ::= & a\langle v\rangle \;\Big|\; \overline{a}\langle v\rangle \;\Big|\; \tau
\end{array}
$$

where $a\langle v\rangle$ and $\overline{a}\langle v\rangle$ are the complementary actions. Add to the process syntax:

$$
P ::= K\langle\widetilde{e}\rangle \;\Big|\; \mathtt{if}\ e_b\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2
$$

where e_b is a boolean expressions; and K has a defining equation $(\widetilde{x}).\,P$. Expression $(\widetilde{a}).\,P$ is an *abstraction*; the head $(\widetilde{x})$ is a binder for $\widetilde{x}$, with scope P.

Channel and constants have a type, that says what value the channel may carry, and what parameters the constant may take. Typing here is obvious; we shall not comment on it any further; we shall just assume that all expressions are well-typed.

For e closed (i.e., no free variables), write $\mathit{Val}(e)$ for the value which e evaluates to.

Replace the rules for sum and constants of pure CCS with these:

$$\texttt{Vinp}\,\frac{v \text{ and } x \text{ of the same type}}{a(x).\,P \xrightarrow{a\langle v\rangle} P\{v/x\}} \qquad \texttt{Vout}\,\frac{\mathit{Val}(e) = v}{\overline{a}\langle e\rangle.\,P \xrightarrow{\overline{a}\langle v\rangle} P}$$

$$\texttt{Vtau}\,\frac{}{\tau.\,P \xrightarrow{\tau} P} \qquad \texttt{Vsum}\,\frac{M_i \xrightarrow{\mu} M'}{\sum_{i\in I} M_i \xrightarrow{\mu} M'}$$

$$\texttt{Viftrue}\,\frac{\mathit{Val}(e) = \texttt{true} \qquad P_1 \xrightarrow{\mu} P'}{\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \xrightarrow{\mu} P'} \quad \texttt{Viffalse}\,\frac{\mathit{Val}(e) = \texttt{false} \qquad P_2 \xrightarrow{\mu} P'}{\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \xrightarrow{\mu} P'}$$

$$\texttt{Vconst}\,\frac{\mathit{Val}(\widetilde{e}) = \widetilde{v} \qquad P\{\widetilde{v}/\widetilde{x}\} \xrightarrow{\mu} P'}{K\langle\widetilde{e}\rangle \xrightarrow{\mu} P'} \text{ if } K \triangleq (\widetilde{x}).\,P$$

(NB: We are adopting a call-by-value evaluation strategy for expressions)

All the theory we have seen for the pure CCS extends very smoothly, and in the expected way, to the value-passing calculus (for instance the definitions of bisimulation and bisimilarity do not require any change, neither in the strong not in the weak case).

We still write $a.\,P, \overline{a}.\,P, K$, in case the argument does not matter (think that the argument is `unit`)

Exercise 47 *Let* $K_1 \stackrel{\circ}{=} a(x).\, \overline{c}\langle x\rangle.\, d.\, K_1$, $K_2 \stackrel{\circ}{=} c(x).\, \overline{b}\langle x\rangle.\, \overline{d}.\, K_2$, $H \stackrel{\circ}{=} a(x).\, \overline{b}\langle x\rangle.\, H$. *Prove that* $(\boldsymbol{\nu} c, d\,)(K_1 \mid K_2) \approx H$.

Exercise 48 (bags) *A* bag *holds elements of a certain type; elements can be added and removed; the order in which they are removed is arbitrary. Here is a specification SPECbag and an implementation IMPbag of a bag of integers (thus* X *is a multiset of integer, and* x, y *integer variables):*

$$\begin{array}{rcl} \mathit{SPECbag} & \stackrel{\circ}{=} & (X).\, (a(x).\, \mathit{SPECbag}\langle X \cup \{x\}\rangle + \sum_{y\in X} \overline{b}\langle y\rangle.\, \mathit{SPECbag}\langle X - \{y\}\rangle) \\ \mathit{IMPbag} & \stackrel{\circ}{=} & a(x).\, (\overline{b}\langle x\rangle \mid \mathit{IMPbag}) \end{array}$$

Prove that $\mathit{SPECbag}\langle\emptyset\rangle \sim \mathit{IMPbag}$.

Exercise* 49 (a slot machine, Bradfield-Stirling) *A* slot machine *take coins in input. Each coin is a play; the player either gets a loss, or a win; in the latter case, the money won may not exceed what the slot machine has collected so far. Here is a specification SPEC of the machine; x is an integer variable:*

$$SPEC \quad \hat{=} \quad (x).\, \mathtt{slot}.\, (\tau.\, \overline{\mathtt{loss}}.\, SPEC\langle x+1\rangle + \tau.\, \textstyle\sum_{0\leq i\leq x} \overline{\mathtt{win}}\langle i+1\rangle.\, SPEC\langle x-i\rangle)$$

Here is an implementation SM of the slot machine. It uses 3 components: IO that handles the money transactions with the players; a bank B that holds the money, and a component D that decides the result of each play.

$$\begin{array}{rcl}
IO & \hat{=} & \mathtt{slot}.\, \overline{\mathtt{bank}}.\, (\mathtt{lost}.\, \overline{\mathtt{loss}}.\, IO + \mathtt{release}(y).\, \overline{\mathtt{win}}\langle y\rangle.\, IO) \\
B & \hat{=} & (x).\, \mathtt{bank}.\, \overline{\mathtt{max}}\langle x+1\rangle.\, \mathtt{left}(y).\, B\langle y\rangle \\
D & \hat{=} & \mathtt{max}(z).\, (\overline{\mathtt{lost}}.\, \overline{\mathtt{left}}\langle z\rangle.\, D + \sum_{1\leq y\leq z} \overline{\mathtt{release}}\langle y\rangle.\, \overline{\mathtt{left}}\langle z-y\rangle.\, D) \\
SM & \hat{=} & (x).\, (\boldsymbol{\nu}\mathtt{bank}, \mathtt{max}, \mathtt{left}, \mathtt{lost}, \mathtt{release}\,)(IO \mid B\langle x\rangle \mid D)
\end{array}$$

Prove that $SPEC\langle n\rangle \approx SM\langle n\rangle$, for any n.

Semantics of a concurrent imperative language

The source language

X	$::=$	$\mathtt{X} \mid \mathtt{Y}$	*Variables*
F	$::=$	$+ \mid -$	*Function symbols*
	$\mid$	$\cdots \mid 0 \mid 1 \cdots$	
			Commands
C	$::=$	$X := E$	assignment
	$\mid$	$C\ ;\ C'$	sequential composition
	$\mid$	$\mathtt{if}\ E\ \mathtt{then}\ C\ \mathtt{else}\ C'$	conditional
	$\mid$	$\mathtt{while}\ E\ \mathtt{do}\ C$	while
	$\mid$	$\mathtt{begin}\ \ \mathtt{var}\ X\ ;\ C$	block
	$\mid$	$C\ \mathtt{par}\ C'$	parallel composition
	$\mid$	$\mathtt{skip}$	skip
	$\mid$	$\mathtt{input}\ X$	input
	$\mid$	$\mathtt{output}\ E$	output
			Expressions
E	$::=$	X	variable
	$\mid$	$F(E_1, \ldots, E_n)$	function application

- Challenging features of the language:

– communication by shared variables
– sequential composition of commands
– loops
– local variables

- To be noted about the translation:

– the granularity in parallel composition of commands
– the use of tranlation to *prove* properties of the language

- Limitations of the language and of the translation:

– limited form of function application
– no procedure declarations
– call-by-name

$\boxed{\text{Translation of variables}}$

$$[\![X]\!] \stackrel{\text{def}}{=} \texttt{Loc}_X$$

where

$$\begin{aligned}
\texttt{Loc}_X &\doteq \texttt{put}_X(x).\,\texttt{Reg}_X\langle x\rangle \\
\texttt{Reg}_X &\doteq (x).\,\texttt{put}_X(y).\,\texttt{Reg}_X\langle y\rangle + \overline{\texttt{get}_X}\langle x\rangle.\,\texttt{Reg}_X\langle x\rangle
\end{aligned}$$

Translation of expressions

$$[\![F]\!]_{\mathtt{arg}_1,\ldots,\mathtt{arg}_n,\mathtt{res}} \stackrel{\mathsf{def}}{=} \mathtt{arg}_1(x_1).\cdots\mathtt{arg}_n(x_n).\,\overline{\mathtt{res}}\langle f(x_1,\ldots,x_n)\rangle$$

if F stands for f and has arity n.

$$[\![X]\!]_{\mathtt{res}} \stackrel{\mathsf{def}}{=} \mathtt{get}_X(x).\,\overline{\mathtt{res}}\langle x\rangle$$

$$[\![F(E_1,\ldots,E_n)]\!]_{\mathtt{res}} \stackrel{\mathsf{def}}{=} (\boldsymbol{\nu}\mathtt{arg}_1,\ldots,\mathtt{arg}_n)(\; [\![E_1]\!]_{\mathtt{arg}_1} \mid \ldots \mid [\![E_n]\!]_{\mathtt{arg}_n} \mid [\![F]\!]_{\mathtt{arg}_1,\ldots,\mathtt{arg}_n,\mathtt{res}})$$

Granularity of parallel composition

What should this program do?

$$\begin{array}{c} \mathtt{X} := 0 \ ; \\ (\ \mathtt{X} := \ \mathtt{X} + 1 \ \mathtt{par} \ \ \mathtt{X} := \ \mathtt{X} + 1 \ ) \end{array}$$

Can it terminate with $\mathtt{X} = 1$?

We shall assume that the answer is yes. Later, we shall discuss how to model atomic execution of assignments.

On the sequential composition of commands

In the language we are translating, sequential composition is on *commands*; but in CCS sequential composition is on *actions*, not on *processes*. We can derive a form of sequential composition on processes, as follows.

A process P is well-terminating on $\mathtt{d}$ if $\mathtt{d}$ is only used to signal the termination of the execution of P.

Definition 50 *A process P is* well-terminating on $\mathtt{d}$ *if for all sequences $\mu_1 \dots \mu_n$ of actions, and process P', if $P \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_n} P'$, then*

- *either $\mathtt{d}$ does not appear in $\{\mu_1. \dots, \mu_n\}$,*
- *or μ_n is an output at $\mathtt{d}$ and $P' \approx 0$.*

A co-inductive definition of well-terminating:

The set of processes *well-terminating on* $\mathtt{d}$ is the largest set of processes $\mathcal{WT}_{\mathtt{d}}$ such that for all $P \in \mathcal{WT}_{\mathtt{d}}$, whenever $P \xrightarrow{\mu} P'$, then

- either $\mathtt{d}$ does not appear in μ and $P' \in \mathcal{WT}_{\mathtt{d}}$,
- or μ is an output at $\mathtt{d}$ and $P' \approx 0$.

We define sequential composition between well-terminating processes.

$$P_1 \ \mathit{Before}_{\mathtt{d}} \ P_2 \quad \stackrel{\mathsf{def}}{=} \quad \boldsymbol{\nu}\mathtt{d}\,(P_1 \mid \mathtt{d}.\,P_2)$$

Lemma 51 *Suppose that for $i, j = 1, 2$, $i \neq j$, we have:*

- P_i *is well-terminating on* $\mathtt{d}_i$*;*
- $\mathtt{d}_i$ *is fresh for* P_j

Then P_1 *Before*$_{\mathtt{d}_1}$ P_2 *is terminatinig on* $\mathtt{d}_2$*.*

Lemma 52 *Suppose that for $i, j = 1, 2, 3$, $i \neq j$, we have:*

- P_i *is terminating on* $\mathtt{d}_i$*;*
- $\mathtt{d}_i$ *is fresh for* P_j*.*

Then $P_1 \ \mathit{Before}_{\mathtt{d}_1} \ (P_2 \ \mathit{Before}_{\mathtt{d}_2} \ P_3) \sim (P_1 \ \mathit{Before}_{\mathtt{d}_1} \ P_2) \ \mathit{Before}_{\mathtt{d}_2} \ P_3$*.*

The translation $[\![C]\!]_d$ of a command C will be a well-terminating process.

Lemma 53 *For all commands C, and fresh name* $\mathtt{d}$*, it holds that* $[\![C]\!]_{\mathtt{d}}$ *is well-terminating on* $\mathtt{d}$*.*

Assume $\mathtt{done}, \mathtt{d}, r$ fresh:

$$
\begin{array}{rcl}
[\![X := E]\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & \boldsymbol{\nu} r\,([\![E]\!]_r \mid r(x).\,\overline{\mathtt{put}_X}\langle x\rangle.\,\overline{\mathtt{done}}) \\
[\![C \;\; ; \;\; C']\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & [\![C]\!]_{\mathtt{d}}\; \mathit{Before}_{\mathtt{d}}\; [\![C']\!]_{\mathtt{done}} \\
[\![\; \mathtt{if}\;\; E\;\; \mathtt{then}\;\; C\;\; \mathtt{else}\;\; C']\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & \boldsymbol{\nu} r\,([\![E]\!]_r \mid r(x).\;\; \mathtt{if}\;\; x\;\; \mathtt{then}\;\; [\![C]\!]_{\mathtt{done}}\;\; \mathtt{else}\;\; [\![C']\!]_{\mathtt{done}}) \\
[\![\; \mathtt{begin}\;\;\; \mathtt{var}\;\; X\;\; ;\;\; C]\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & (\boldsymbol{\nu}\mathtt{put}_X, \mathtt{get}_X\,)(\mathtt{Loc}_X \mid [\![C]\!]_{\mathtt{done}}) \\
[\![C\;\; \mathtt{par}\;\; C']\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & \boldsymbol{\nu}\mathtt{d}\,([\![C]\!]_{\mathtt{d}} \mid [\![C']\!]_{\mathtt{d}} \mid \mathtt{d}.\,\mathtt{d}.\,\overline{\mathtt{done}}) \\
[\![\; \mathtt{skip}\;\;]\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & \overline{\mathtt{done}} \\
[\![\; \mathtt{input}\;\;\; X]\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & \mathtt{inp}(x).\,\overline{\mathtt{put}_X}\langle x\rangle.\,\overline{\mathtt{done}} \\
[\![\; \mathtt{output}\;\;\; E]\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & \boldsymbol{\nu} r\,([\![E]\!]_r \mid r(x).\,\overline{\mathtt{out}}\langle x\rangle.\,\overline{\mathtt{done}}) \\
[\![\; \mathtt{while}\;\; E\;\; \mathtt{do}\;\; C]\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & W_{\mathtt{done}} \qquad\qquad\qquad \text{where}
\end{array}
$$

$$
W_{\mathtt{done}} \stackrel{\circ}{=} \boldsymbol{\nu} r\,([\![E]\!]_r \mid r(x).\;\; \mathtt{if}\;\; x\;\; \mathtt{then}\;\; ([\![C]\!]_{\mathtt{d}}\; \mathit{Before}_{\mathtt{d}}\; W_{\mathtt{done}})\;\; \mathtt{else}\;\; \overline{\mathtt{done}})
$$

Granularity, continued

Suppose we do not want

$$\mathtt{X} := 0 \;\; ;$$
$$\mathtt{X} := \;\; \mathtt{X} + 1 \;\; \mathtt{par} \;\;\; \mathtt{X} := \;\; \mathtt{X} + 1$$

to terminate with $\mathtt{X} = 1$. We can avoid using semaphores (cf.: Exercise 24): Define:

$$\mathrm{Sem}_X \stackrel{\circ}{=} p_X . \, v_X . \, \mathrm{Sem}_X$$

Then change the translation as follows:

$$\begin{array}{rcl}
[\![X]\!] & \stackrel{\mathrm{def}}{=} & \mathtt{Loc}_X \mid \mathrm{Sem}_X \\
[\![\; \mathtt{begin} \;\; \mathtt{var} \; X \; ; \; C]\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & (\boldsymbol{\nu}\mathtt{put}_X, \mathtt{get}_X, p_X, v_X \;)([\![X]\!] \mid [\![C]\!]_{\mathtt{done}}) \\
[\![X := E]\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & \overline{p_X} . \, \boldsymbol{\nu} r \; ([\![E]\!]_r \mid r(x) . \, \overline{\mathtt{put}_X} \langle x \rangle . \, \overline{v_X} . \, \overline{\mathtt{done}}) \\
[\![\; \mathtt{input} \;\; X]\!]_{\mathtt{done}} & \stackrel{\mathrm{def}}{=} & \overline{p_X} . \, \mathtt{inp}(x) . \, \overline{\mathtt{put}_X} \langle x \rangle . \, \overline{v_X} . \, \overline{\mathtt{done}}
\end{array}$$

Using the translation

We can use the translation to prove proprieties of the source language. Here are some examples, writing $C \approx C'$ for $[\![C]\!]_{\mathtt{done}} \approx [\![C']\!]_{\mathtt{done}}$ (for `done` fresh):

a) C `par` $C' \approx C'$ `par` C

b) $C\ ;\ (C'\ ;\ C'') \approx (C\ ;\ C')\ ;\ C''$

c) $C\ ;$ `skip` $\approx C$

d) if variable X does not appear in C, then:

$$\texttt{begin var } X \ ; \ C \quad \approx \quad C$$

$$\texttt{begin var } X \ ; \ (C \texttt{ par } C') \quad \approx \quad C \texttt{ par (begin var } X \ ; \ C')$$

e) $\texttt{while } E \texttt{ do } C \approx \texttt{ if } E \texttt{ then } (C \ ; \ \texttt{while } E \texttt{ do } C) \texttt{ else skip}$

Exercise 54 *Prove the properties above.*

Exercise 55 *Is* $C\ ;\ (C' \texttt{ par } C'') \approx (C\ ;\ C') \texttt{ par } C''$*? And assuming that* C *does not perform any input, output, and does not access any variables?*

Name and process passing

With CCS, it becomes complicated, and sometimes unfeasable, to handle richer languages, with features such as: general forms of function application, local declarations of procedures, and call-by-name evaluation of arguments.

For translating these languages, we need a further extension to CCS, in which values exchanged in communications may be channels. This extension adds a remarkable expressiveness to the calculus (for instance it becomes possible to give simple translation of call-by-name and call-by-value lambda calculi). The extension needs however some non-trivial modifications to the operational and algebraic theory of CCS. The best known of such extensions is the π-*calculus*.