CHAPTER 1

Introduction

CHAPTER 2

Background

CHAPTER 3

MIN

The 'native' notation for MIN is the graphical notation introduced in chapter 1. It is concrete, modelling the structure of programs visually, supporting a simple balls-and-strings metaphor for manipulating that structure, and executing by simple scissors-and-glue rewriting. This makes it easy to teach and use. It also makes it easy to see through the notation, to read a program written in it, and to reason about it. However, it does not make it easy to reason about MIN itself.

This chapter defines an alternative notation for MIN, based on a conventional textual grammar. It explains the relationship of the two notations, formally as far as possible, which is not quite all the way. It then uses the textual notation to prove the assertion made in chapter 1, that the labelled transition system is a sound and complete technique for proving that programs are or are not (barbed) equivalent. It is this result that makes an optimising MIN compiler a practical possibility.

3.1. The theorem

I claim that MIN makes a good intermediate language for a compiler. The sole reason I believe it to be better for this purpose than other languages, such as the many that resemble assembler, that are of similar complexity and capable of expressing the same programs, is that it is more tractable. Where other languages express in great detail what a program must do, MIN expresses only how a program must make its decisions, and how it must interact with other programs. The ways in which a compiler may choose to implement the program are limited only by its own ingenuity, and by a clause in the language definition.

The clause states the conditions under which one MIN program is a valid implementation of another MIN program. The intention (although several parts of the story remain untold) is that the compiler should take a program in a high-level language, encode it literally into a MIN program, transform it (using techniques founded on this thesis) into the MIN encoding of some machine code, and then execute that machine code. Although MIN has a concrete operational semantics, based on graph rewriting, a literal implementation of those semantics should only be considered an option of last resort.

The grand theorem of this chapter is that two different ways of defining whether or not an implementation of a program is valid coincide. The theorem is important because it connects a definition which is easy to motivate with one that is easy to apply. I hope it will leave future researchers in no doubt as to what is the simplest way to attain confidence in their optimisations.

3.1.1. Barbed coupled similarity. The definition of the 'can be implemented as' relation is based on the process calculus literature reviewed in chapter 2. In the context of that literature, it would be known as *barbed coupled similarity*. It is defined in terms of MIN's operational semantics, and in terms of the definition of linking. The story was told for CCS in chapter 2, but I will repeat the important parts. Here, these definitions are given slightly informally. They will be repeated again formally using the textual notation in section 3.8.1.

DEFINITION. (MIN+)

The definition of MIN includes a set of permissible constructor symbols. For the sake of concreteness, we may imagine that this is the set of strings in some alphabet. Let MIN+ be defined in exactly the same way, but with a set of constructor symbols that is larger by 1. The extra constructor symbol (which is different from all constructor symbols in MIN) will be called 'test'.

DEFINITION. (Well-formed relation)

Say a relation on programs is *well-formed* if it only relates programs with the same interface. An example of a well-formed relation is the rewrite relation. Barbed coupled similarity will be another example.

DEFINITION. (Context-closed)

Say a well-formed relation $\underline{S}$ is *context-closed* iff, for every element $P \ \underline{S} \ Q$ and for every program R with a compatible interface, we find $P \times R \ \underline{S} \ Q \times R$ (in which $\times$ represents the operation of linking programs).

DEFINITION. (Reduction-closed)

Say a well-formed relation $\underline{S}$ is *reduction-closed* iff, whenever $P \ \underline{S} \ Q$ and $Q \longrightarrow^* Q'$, there is a P' such that $P \longrightarrow^* P'$ and $P' \ \underline{S} \ Q'$ (in which $\longrightarrow$ is the rewrite relation).

DEFINITION. (Sound)

Say a well-formed relation $\underline{S}$ is *sound* iff, for every element $P \ \underline{S} \ Q$, the main graph of P contains 'test' iff the main graph of Q contains 'test'.

DEFINITION. (Coupled)

Say a well-formed relation $\underline{S}$ is *coupled* iff, whenever $P \ \underline{S} \ Q$, there is a P' such that $P \longrightarrow^* P'$ and $Q \ \underline{S} \ P'$.

DEFINITION. (Barbed coupled similarity)

Say a well-formed relation is a *barbed coupled simulation* iff it is context-closed, reduction-closed, sound and coupled.

Define *barbed coupled similarity*, written $\geq_C$, to be the restriction to MIN of the largest barbed coupled simulation on MIN+ programs.

The great attraction of barbed coupled similarity is that all the conditions imposed on it are easy to motivate. Context closure states that if the compiler claims to have implemented a program, it may not change its mind after seeing what it will be linked with. Reduction closure states that it may not change its mind after seeing what the program does when it is executed. Soundness states, when combined

with context closure, that a program linked with an implementation must not be able to discover that it is not linked with the original program (if it could, it could signal its discovery using a 'test' node). Coupling states, when combined with reduction closure, that an implementation must do at least one of the things that the original program could do, and in particular that it must not crash if the original program doesn't. All of these are basic sanity requirements.

3.1.2. Labelled coupled similarity. The great problem with barbed coupled similarity is that its definition only provides obvious ways of proving that an implementation of a program is *not* valid. There is no obvious way to make sure that one *is*. This models the real world well, but is rather inconvenient. We therefore also define *labelled coupled similarity* which is much easier to check (although it still includes the halting problem as a special case), but which cannot easily be motivated. The definition appears quite arbitrary. It isn't, however, because it coincides exactly with barbed coupled similarity.

Labelled coupled similarity is defined not in terms of MIN's operational semantics, but in terms of a *labelled transition system*. Elements of this new ternary relation are of the form $P \xrightarrow{\alpha} P'$ where P and P' are programs and α is an *action*. The labelled transition system agrees with the rewrite relation[1] in the sense that whenever $P \longrightarrow P'$ we also have $P \xrightarrow{\tau} P'$, where τ is the special *silent* action. However, there are also many more actions, interpreted as communications with other programs. It is these new actions that are difficult to motivate.

The definition of MIN's labelled transition system was given in the graphical notation in chapter 1, but I will repeat it.

DEFINITION. (Labelled transition system)

There are three kinds of action:

- The silent action is called τ. Silent transitions are defined to match the rewrite relation, as previously explained.
- An *input* action is of the form $xc(\vec{y})$, and represents the receipt of a message c at the free port named x, replacing it with zero or more new free ports named $\vec{y}$. Graphically, $P \xrightarrow{xc(\vec{y})} P'$ represents the attachment of a constructor c to the main graph of P to obtain P'. The principal port of the constructor is joined to the free port named x, and the auxiliary ports remain free and are named $\vec{y}$. Such a transition is possible whenever P' is well typed.
- An *output* action is of the form $\overline{x}c(\vec{y})$, and represents the sending of a message c from the free port named x, replacing it with zero or more new free ports named $\vec{y}$. Graphically, $P \xrightarrow{\overline{x}c(\vec{y})} P'$ represents the removal of a constructor c from the main graph of P to obtain P'. The operation is easiest to explain in reverse: the auxiliary ports of the constructor are joined to the free ports of P' named $\vec{y}$, and the principal port remains free in P and is named x. Such a transition is possible whenever P has a constructor to remove.

[1] In the graphical notation. In the textual notation this statement must be weakened, because the textual notation makes more distinctions.

Figure [3G] shows the complete labelled transition system of the example program in figure [3F]. The program cannot perform any reactions on its own, but it could show some behaviour when linked with other programs. The input and output transitions provide a way to explore all this potential behaviour, without explicitly considering linking.

DEFINITION. (Labelled coupled similarity)

Say a well-formed relation $\underline{S}$ is a *labelled simulation* iff:

- Whenever $P \mathrel{\underline{S}} Q$ and $Q \xrightarrow{\tau}{}^{*} Q'$ there is a P' such that $P \xrightarrow{\tau}{}^{*} P'$ and $P' \mathrel{\underline{S}} Q'$.
- Whenever $P \mathrel{\underline{S}} Q$ and $Q \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}{}^{*} Q'$ there is a P' such that $P \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}{}^{*} P'$ and $P' \mathrel{\underline{S}} Q'$.

Define *labelled coupled similarity*, written $\gtrsim_C$, be the largest coupled labelled simulation.

THEOREM. *(Purpose of this chapter)*

$\geq_C$ and $\gtrsim_C$ coincide.

3.1.3. Outline of proof. To show that P labelled coupled simulates Q it is necessary and sufficient to demonstrate that they are related by some labelled coupled simulation $\underline{S}$. Similarly, to show that P barbed coupled simulates Q it is necessary and sufficient to find a barbed coupled simulation. Our task is therefore to construct a labelled coupled simulation containing $\geq_C$ and a barbed coupled simulation containing $\gtrsim_C$.

In fact, it will turn out that $\geq_C$ is already a labelled coupled simulation. It is certainly already coupled, but we have to check that if $P \geq_C Q$ and $Q \xrightarrow{\tau}{}^{*} Q'$ then $P \xrightarrow{\tau}{}^{*} P'$ and $P' \geq_C Q'$ (which is almost exactly the definition of reduction-closed) and a similar condition for non-silent actions. Input transitions are a special case of linking, so context-closure takes care of them. Output transitions require a little more work. We have to show that we can link any program with a test harness in such a way that if the original program could perform an output then the instrumented program can react so as to create a 'test' node. The result then follows from context-closure (to link with the test harness), reduction-closure (to detect the output) and soundness.

To turn $\gtrsim_C$ into a barbed coupled simulation we must simply extend it from MIN to MIN+. This involves linking with programs that may contain 'test'. Since 'test' nodes can only arise from the linking context, the resulting relation is sound. It remains coupled, and the close relationship of $\longrightarrow$ and $\xrightarrow{\tau}$ ensures that it is reduction-closed. However, it is far from obvious that it is context-closed, and a large fraction of the proof is dedicated to this part alone.

3.2. The textual notation

The best way to think of a textual program is as a series of steps in the construction of a graphical program. By the end of this section, you should be able, at an informal level at least, to read a textual program and construct and write down the

corresponding graphical program. You should also be able to reverse the process, although there are many constructions of any graphical program, so the textual program will not be unique.

The previous paragraph contains a small lie. The textual notation in this chapter is not capable of expressing any type information. This important short-coming will be fixed in chapter 4.

3.2.1. Environments and processes. Figure [3A] shows an example graphical program, and a corresponding textual program. It is the AND example from chapter 1. The large-scale syntax of the textual version is 'let E in p'. In this construction, the set E is called an *environment*, and corresponds to the rewrite rules of the graphical program, and p is called a *process*, and corresponds to the main graph. Although this construction deliberately looks a little like one found in functional languages, programs and processes are different syntactic classes; for example 'let E in let F in p' is not syntactically correct.

The node and type declarations of the graphical program have no counterpart in the textual one. In the graphical notation, declarations serve two purposes: they define the types, and they add plenty of redundancy that a compiler can use to spot errors. Since types will be handled separately, and since the textual notation is not intended for programming, declarations serve no useful purpose.

An environment is a set of *rewrites*, each of the form $p \longrightarrow p'$, in which p and p' are processes, with the same syntax as the main graph. This closely follows the graphical notation, in which rewrites consist of a pair of graphs. In fact, it is true in general that the textual notation uses processes wherever the graphical notation uses graphs. As in the graphical notation, the redex p is constrained to a special form, and must have the same free ports as the reduct p'. Unlike the graphical notation, it is necessary to name the free ports, although the choice of names is of no consequence.

3.2.2. Atoms and trees. It remains, then, to explain how a process represents a graph. This is, of course, the most interesting part. A process consists of one or more *atoms*, joined together using an operator $\times$. In figure [3A] we have been lucky in not needing the $\times$ operator at all, as all the graphs are atoms, but this is not true in general. Atoms are graphs of a special form, with the property that any graph can be broken in to atoms (in many ways).

The empty graph is an atom. Apart from this special case, atoms are made of *trees*. Trees are graphs of an even more special form. A graph is a tree iff it is singly-connected, it contains only constructors, and all of its internal wires join a principal port to an auxiliary port. Recalling that constructors must have exactly one principal port, a tree must therefore have exactly one free principal port, which is called its *root*, the other free ports being *leaves*. The remaining two kinds of atom are: two trees joined together at the root; or a destructor with a tree attached by its root to each port. All three kinds of atom must therefore be singly-connected. Figure [3B] shows some examples.

The textual notation for a tree is exactly what one would expect: one writes down a term with a parse tree that looks like the tree. The examples in part (a) of figure [3B] are therefore x, $c(u, v, w)$ and $a(b(c(u, v), w), c(x, y))$.

The three kinds of atom each have their own notation. The empty graph is written 1. The atom formed by joining the trees t and u at their root is written $\langle t, u\rangle$. The atom formed by joining trees t_1 to t_n to the n ports of a destructor with symbol d is written $d(t_1, \ldots, t_n)$. In the latter, it does not matter in what order the ports are listed, as long as the same order is used wherever d appears. Halt nodes are treated as destructors with symbol 0. The examples in part (b) of figure [3B] are therefore 1, $\langle y, x\rangle$, $\langle x, c(y, z)\rangle$, $d(c(w, z), x, y)$, $d(u, a(v, w), c(x, b(y, z)))$, $0(x, y, c(u, v, w))$.

The $\times$ operator joins two graphs together to make a new graph, in a manner that is commutative but not always associative. Of the many ways of joining together a given pair of graphs, one is chosen according to the names of their free ports. If a port of one graph has a name that does not appear on any port of the other, it remains free in the result and retains its name. Otherwise (that is, if a port of one graph has the same name as a port of the other) the two ports are joined together in the result, making an internal wire that has no name. You may recognise this operation as the part of the definition of linking that concerns the main graph. Figure [3C] shows an example, and figure [3D] shows a case in which $\times$ is not associative.

To write down the examples in part (c) of figure [3B], they must be broken into atoms. There are several ways of doing this. If we choose to do it as in figure [3E], they are written $\langle w, x\rangle \times \langle y, z\rangle$, $\langle u, c(w, x, y)\rangle \times \langle v, b(z, y)\rangle$, $\langle w, c(v, z)\rangle \times d(v, x, y)$, $d(w, v, z) \times e(v, x, y)$, $d(w, z, y) \times \langle w, z\rangle$, $\langle x, c(b(y, w), z)\rangle \times \langle w, z\rangle$ and $\langle x, y\rangle \times \langle x, y\rangle = (1 \times \langle x, y\rangle) \times \langle x, y\rangle = \langle x, y\rangle \times (\langle y, z\rangle \times \langle x, z\rangle)$.

3.2.3. Mathematical support. The correspondence of graphs to processes is one-to-many. Of the several differences between the two notations, this one, at least, can be handled formally in the textual notation. To do this, we define a structural congruence, written $\equiv$, which relates two processes iff they are constructions of the same graph.

It is easy to check each of the axioms of the structural congruence to verify that they do not change the graph, that is, that they are sound. It is rather harder to check that the axioms are complete, as this involves graphical reasoning that we do not know how to do. However, I believe them to be complete too.

Structural congruence impinges not only on the question of what a program is, but also of what it does. For example, it appears in the definition of the reaction relation, to ensure that constructions of the same graph have the same rewrites. It does not appear, however, in the definition of the labelled transition relation. Instead, we submit to a lengthy proof (culminating in theorem 3.7.11) that structurally congruent processes can perform the same actions, and that they remain structurally congruent. By doing this work just once, we earn the right to use a convenient definition of the labelled transition relation. The effort is rewarded in all subsequent proofs.

3.3. Definitions

MIN is defined in terms of a set ⟨name⟩ of *names*, a set ⟨constructor⟩ of *constructor symbols* and a set ⟨destructor⟩ of *destructor symbols*. The structure of these sets does not matter, but we need them to be countably infinite, and we need a way of

choosing fresh elements. It is also convenient to distinguish a special destructor symbol '0', which we will use to stand for the canonical inert process.

DEFINITION 3.3.1. (Grammar, and free names)

MIN's grammar is as follows:

$$\begin{array}{rcl}
\langle\text{program}\rangle & ::= & \text{let } \langle\text{environment}\rangle \text{ in } \langle\text{process}\rangle \\
\langle\text{environment}\rangle & ::= & \text{a set of } \langle\text{rewrite}\rangle\text{s} \\
\langle\text{rewrite}\rangle & ::= & \langle\text{destructor}\rangle\left(\langle\text{tree}\rangle, \ldots, \langle\text{tree}\rangle\right) \longrightarrow \langle\text{process}\rangle \\
\langle\text{process}\rangle & ::= & \langle\text{atom}\rangle \mid \langle\text{process}\rangle \times \langle\text{process}\rangle \\
\langle\text{atom}\rangle & ::= & \langle\langle\text{tree}\rangle, \langle\text{tree}\rangle\rangle \mid \langle\text{destructor}\rangle\left(\langle\text{tree}\rangle, \ldots, \langle\text{tree}\rangle\right) \\
\langle\text{tree}\rangle & ::= & \langle\text{name}\rangle \mid \langle\text{constructor}\rangle\left(\langle\text{tree}\rangle, \ldots, \langle\text{tree}\rangle\right)
\end{array}$$

with four side-conditions.

The first is that all ⟨name⟩s in an ⟨atom⟩ or ⟨tree⟩ must be different.

The second is that no ⟨rewrite⟩ $d(\vec{t}) \longrightarrow p$ may have $d = 0$.

The third is that in every ⟨rewrite⟩ $p \longrightarrow p'$ there must be exactly one ⟨constructor⟩ in p.

For the fourth, we must first define the function fn( ) which calculates the *free names* of a ⟨process⟩. If p is an ⟨atom⟩, define fn(p) to be the set of ⟨name⟩s that appear in p; otherwise define $\text{fn}(p) = \text{fn}(q) \vartriangle \text{fn}(r)$ where $p = q \times r$.[2] Also, extend fn( ) to ⟨tree⟩s in the obvious way, and extend it to ⟨program⟩s by fn(let E in p) = fn(p).

The fourth side-condition is that every rewrite $p \longrightarrow p'$ must have $\text{fn}(p) = \text{fn}(p')$.

The third side-condition is optional. Omitting it makes very little difference to the maths, and none to the expressive power. However, it is required for a tight correspondence with the graphical notation. You may like to work out what new kinds of graph would be allowed to be redexes if this side-condition were omitted.

For readability, I will usually choose variable names according to a convention. The convention serves no purpose besides readability. It is as follows:

- ⟨name⟩s will be called w, x, y or z.
- ⟨constructor⟩s will be called a, b or c.
- ⟨destructor⟩s will be called d, e or f.
- ⟨tree⟩s will be called t, u or v.
- ⟨process⟩es will be called p, q, r or s.
- ⟨environment⟩s will be called E or F.
- ⟨program⟩s will be called P, Q or R.
- Variables that stand for relations will be underlined.

[2] $\vartriangle$ is the symmetric difference operator, defined so that $X \vartriangle Y = (X \cup Y) - (X \cap Y) = (X - Y) \cup (Y - X)$. It occurs quite a lot in this chapter. It can be interpreted as pointwise addition modulo 2 of the characteristic functions of X and Y, or as pointwise XOR. It is commutative and associative, and the empty set is its unit. Furthermore, $X \vartriangle X = \{\}$ for any set X.

Two bits of syntactic sugar are useful. First, let 1 mean $0()$. We will use this to represent the empty graph. Second, given two tuples of trees $\vec{t}$ and $\vec{u}$ with $|\vec{t}| = |\vec{u}| = n$ and no name appearing more than once, let $\langle \vec{t}, \vec{u} \rangle$ mean $\langle t_1, u_1 \rangle \times (\ldots \times (\langle t_n, u_n \rangle \times 1) \ldots)$. This represents a bundle of parallel wires joining each t_i to the corresponding u_i.

DEFINITION 3.3.2. (Alpha-conversion)

Say an injection f on ⟨name⟩s is an *alpha-conversion for* p iff $f(x) = x$ for all $x \in \text{fn}(p)$.

Let *alpha-equivalence*, written $\equiv_\alpha$, be the smallest congruence such that $p = f(p)$ whenever f is an alpha-conversion for p.

Note that the identity function is an alpha-conversion for every ⟨process⟩, that the inverse of an alpha-conversion for p is another alpha-conversion for p, and that the composition of two alpha-conversions for p is another alpha-conversion for p. Thus, alpha-equivalence would be an equivalence relation even if it were not defined to be. However, it would not be a congruence.

We adopt the convention of *working up to alpha-conversion*. This is a completely standard technique that saves a lot of trouble. In the case of MIN it entails the following:

- Whenever we write a ⟨process⟩, or anything containing a ⟨process⟩, we understand that we mean its alpha-equivalence class. In particular, $q \times r$ means $\{p' \mid p' \equiv_\alpha q' \times r' \text{ and } q' \in q \text{ and } r' \in r\}$, which is equal to $\{p' \mid p' \equiv_\alpha q' \times r'\}$ for any $q' \in q$ and $r' \in r$.
- For any ⟨process⟩ p that is not an ⟨atom⟩, we may nominate any finite set X of ⟨name⟩s and then find q and r such that $\text{fn}(q) \cap \text{fn}(r) \cap X$ is empty and $p = q \times r$. This is justified by observing that the alpha-equivalence class certainly includes pairs $q \times r$ that do satisfy this property. However, we must be careful to choose X before using q or r; for example $\text{fn}(q)$ certainly overlaps with $\text{fn}(q)$, despite the fact that it is finite.

DEFINITION 3.3.3. (Substitution)

Say a function σ from ⟨name⟩ to ⟨tree⟩ is a *substitution* iff $\text{fn}(\sigma(x)) \cap \text{fn}(\sigma(y))$ is empty for all $x \neq y$, and $\sigma(x)$ is a ⟨name⟩ for all but finitely many x.

Define the result $p\sigma$ of applying a substitution σ to a process p in the obvious way if p is an ⟨atom⟩: replace each name x with $\sigma(x)$. Otherwise, use alpha-conversion to find q and r such that $p = q \times r$ and $\sigma(x)$ is a ⟨name⟩ for all $x \in \text{fn}(q) \cap \text{fn}(r)$, and define $p\sigma$ to be $q\sigma \times r\sigma$.

Given names $\vec{x}$ and trees $\vec{t}$ with $|\vec{x}| = |\vec{t}| = n$ and no name appearing more than once, and a ⟨process⟩ p with $\text{fn}(p) \cap \text{fn}(\langle \vec{x}, \vec{t} \rangle) = \{\vec{x}\}$, define $p[\vec{x} \mapsto \vec{t}]$ to be $p\sigma$ where σ is a substitution chosen so that $\sigma(x_i) = t_i$ for all $1 \leq i \leq n$ and $\sigma(y) = y$ for all $y \in \text{fn}(p) - \{\vec{x}\}$.

Note that $p\sigma$ and $p[\vec{x} \mapsto \vec{t}]$ are always defined uniquely up to alpha-conversion.

DEFINITION 3.3.4. (Structural congruence)

Define the *structural congruence*, written $\equiv$, to be the smallest congruence on processes satisfying the following axioms:

$$\frac{}{p \times 1 \equiv p}\text{STAR1} \qquad \frac{}{p \times q \equiv q \times p}\text{STAR2} \qquad \frac{\text{fn}(p) \cap \text{fn}(q) \cap \text{fn}(r) = \{\}}{p \times (q \times r) \equiv (p \times q) \times r}\text{STAR3}$$

$$\frac{}{\langle t, u\rangle \equiv \langle u, t\rangle}\text{WIRE} \qquad \frac{\text{fn}(p) \cap \text{fn}(\langle x, t\rangle) = \{x\}}{p[x \twoheadrightarrow t] \equiv p \times \langle x, t\rangle}\text{BUF}$$

Extend the structural congruence to ⟨program⟩s by let E in $p \equiv$ let E in q iff $p \equiv q$.

The structural congruence relates ⟨process⟩es that construct the same graph. The premise of STAR3 is necessary on account of examples like the one in figure [3D]. Note that if $p \equiv q$ then $\text{fn}(p) \equiv \text{fn}(q)$, that is, that the structural congruence is a well-formed relation.

DEFINITION 3.3.5. (Reaction relation)

Let the *reaction relation*, written $\longrightarrow$, be the smallest relation on ⟨program⟩s satisfying the following axioms, in which $E \vdash p \longrightarrow p'$ means let E in $p \longrightarrow$ let E in p':

$$\frac{(p \longrightarrow p') \in E}{E \vdash p \longrightarrow p'}\text{REACT} \qquad \frac{E \vdash p \longrightarrow p'}{E \vdash p \times q \longrightarrow p' \times q}\text{PAR} \qquad \frac{P \equiv \longrightarrow \equiv P'}{P \longrightarrow P'}\text{STRUCT}$$

I will often omit $E \vdash$, and just write $p \longrightarrow p'$.

The STRUCT rule is extremely powerful, and makes it rather difficult to prove anything useful about the reaction relation. This is the textual counterpart of the problem of proving anything about graphs. Note that if $P \longrightarrow P'$ then $\text{fn}(P) = \text{fn}(P')$, that is, that the reaction relation is well-formed.

DEFINITION 3.3.6. (Actions)

Let ⟨action⟩ be defined by the following grammar:

$$\begin{array}{rcl} \langle\text{action}\rangle & ::= & \tau \\ & | & \langle\text{name}\rangle\,\langle\text{constructor}\rangle\,(\langle\text{name}\rangle, \ldots, \langle\text{name}\rangle) \\ & | & \overline{\langle\text{name}\rangle}\,\langle\text{constructor}\rangle\,(\langle\text{name}\rangle, \ldots, \langle\text{name}\rangle) \end{array}$$

with the side-condition that no ⟨name⟩ appears more than once in an ⟨action⟩.

Extend fn( ) to ⟨action⟩ so that $\text{fn}(\alpha)$ is the set of all names in α.

I will usually use α to stand for an action, and when I want to be more specific I will use λ and $\overline{\lambda}$ to stand for the two kinds of non-silent action.

DEFINITION 3.3.7. (Labelled transition relation)

Let the *labelled transition relation* , written $P \xrightarrow{\alpha} P'$, be the smallest ternary relation on ⟨program⟩ × ⟨action⟩ × ⟨program⟩ satisfying the following axioms, in which $E \vdash p \xrightarrow{\alpha} p'$ means let E in $p \xrightarrow{\alpha}$ let E in p':

$$\frac{\vec{y} \notin \text{fn}(\langle x, c(\vec{t})\rangle)}{E \vdash \langle x, c(\vec{t})\rangle \xrightarrow{\overline{x}c(\vec{y})} \langle \vec{y}, \vec{t}\rangle}\text{OUT}_1 \qquad \frac{E \vdash p \xrightarrow{\alpha} p' \quad \text{fn}(\alpha) \cap \text{fn}(q) = \{\}}{E \vdash p \times q \xrightarrow{\alpha} p' \times q}\text{PAR}_1$$

$$\frac{\vec{y} \notin \mathrm{fn}(\langle c(\vec{t}), x \rangle)}{E \vdash \langle c(\vec{t}), x \rangle \xrightarrow{\overline{x}c(\vec{y})} \langle \vec{t}, \vec{y} \rangle}\mathrm{OUT}_2 \qquad \frac{\mathrm{fn}(p) \cap \mathrm{fn}(\alpha) = \{\} \quad E \vdash q \xrightarrow{\alpha} q'}{E \vdash p \times q \xrightarrow{\alpha} p \times q'}\mathrm{PAR}_2$$

$$\frac{x \in \mathrm{fn}(p) \quad \vec{y} \notin \mathrm{fn}(p)}{E \vdash p \xrightarrow{xc(\vec{y})} p[x \rightarrowtail c(\vec{y})]}\mathrm{IN} \qquad \frac{E \vdash p \xrightarrow{\overline{\lambda}} p' \quad E \vdash q \xrightarrow{\lambda} q' \quad E \vdash p' \times q' \xrightarrow{\alpha} r'}{E \vdash p \times q \xrightarrow{\alpha} r'}\mathrm{COM}_1$$

$$\frac{p \longrightarrow p' \in E}{E \vdash p\sigma \xrightarrow{\tau} p'\sigma}\mathrm{REACT} \qquad \frac{E \vdash p \xrightarrow{\lambda} p' \quad E \vdash q \xrightarrow{\overline{\lambda}} q' \quad E \vdash p' \times q' \xrightarrow{\alpha} r'}{E \vdash p \times q \xrightarrow{\alpha} r'}\mathrm{COM}_2$$

I will often omit $E \vdash$ and just write $p \xrightarrow{\alpha} p'$.

This definition has a quite different style from that of the reaction relation. One would expect it to be a little more complicated, given that it is defining a more complicated relation. However, at least part of the complexity goes towards avoiding a rule like STRUCT. This definition does not mention the structural congruence at all. Furthermore, the labelled transitions of programs with non-atomic main graphs are defined entirely in terms of the labelled transitions of programs with smaller main graphs. This property is very valuable, and is impossible to achieve for the reaction relation.

Later, I will prove (theorem 3.7.11) that labelled transitions obey a rule like STRUCT anyway, even though no such rule appears explicitly in their definition. I will also prove (proposition 3.7.13) that this definition is equivalent to the one given informally in chapter 1 and repeated in section 3.1.2.

DEFINITION 3.3.8. (Linking)

Fix two injections i and j on ⟨destructor⟩s, with disjoint ranges, and extend them to ⟨process⟩es and ⟨environment⟩s in the obvious way.

Define the result $P \times Q$ of *linking* two programs $P = \text{let } E \text{ in } p$ and $Q = \text{let } F \text{ in } q$ to be let $i(E) \cup j(F)$ in $i(p) \times j(q)$.

I hope it is clear that this definition is simply a formalisation of the linking procedure described for the graphical notation in chapter 1. The treatment of the main graph can be delegated to the primitive $\times$ operator on ⟨process⟩es, so the only thing to worry about here is the behaviour of the ⟨constructor⟩ and ⟨destructor⟩ symbols. The ⟨constructor⟩ symbols of the two programs P and Q belong to the same global namespace, and so are not renamed. Constructors even appear in the labelled transition relation. In contrast, we want each program to have its own private namespace for ⟨destructor⟩ symbols, and we achieve this by renaming them to avoid capture.

I have chosen to use the same symbol $\times$ for linking as for constructing graphs. There should be no risk of confusion, since one operates on ⟨program⟩s and the other on ⟨process⟩es. Later I will prove (proposition 3.4.7) that the labelled transitions of $P \times Q$ can be derived from those of P and Q. It turns out that ⟨program⟩s obey rules analogous to the PAR and COM rules that are used to derive the transitions of ⟨process⟩es. Historically, I invented the rules for ⟨program⟩s first, before deciding how to represent their internal structure.

3.3.1. Canonical derivations. It is sometimes useful to be able to assume that the IN rule is only used in a canonical way.

PROPOSITION 3.3.9. *(Canonical inputs)*

Any input transition $p \xrightarrow{\alpha} p'$ can be derived applying the IN rule only to $\langle atom \rangle$s.

PROOF. Induction on the structure of p. The interesting case is when the transition is derived using the IN rule but $p = q \times r$. Then $\alpha = xc(\vec{y})$ and $x \in \mathrm{fn}(p)$ and $\vec{y} \notin \mathrm{fn}(p)$ and $p' = p[x \mapsto c(\vec{y})]$. By alpha-conversion, we may assume $\mathrm{fn}(q) \cap \mathrm{fn}(r) \cap \mathrm{fn}(\alpha) = \{\}$. Then either $x \in \mathrm{fn}(q)$ or $x \in \mathrm{fn}(r)$. Without loss of generality, suppose the former. By the IN rule, $q \xrightarrow{x\,c(\vec{y})} q[x \!\!\rightarrowtail c(\vec{y})]$, and by the inductive hypothesis the transition can be derived applying the IN rule only to $\langle$atom$\rangle$s. Therefore, by the PAR_1 rule, so can $q \times r \xrightarrow{x\,c(\vec{y})} q[x \!\!\rightarrowtail c(\vec{y})] \times r = (q \times r)[x \mapsto c(\vec{y})]$ as required. □

3.4. Basic properties

I have now defined six largely separate concepts regarding MIN processes and programs: free names, substitution, the structural congruence, the reaction relation, the labelled transition relation, and linking. Each of these is of independent interest. In this section, I will add a seventh concept: the rearrangement relation, which is of mainly technical interest. I will then make a start on the long list of relationships between almost every pair of these seven concepts.

The following table may help you navigate the maths. It gives, for each pair of concepts, the number of the principal result (or definition) that relates them.

		σ	$\times$	$\equiv$	$\longrightarrow$	$\xrightarrow{\alpha}$	$\triangleright$
Free names	fn( )	3.4.1	3.4.1	3.4.1	3.4.1	3.4.1	3.5.3
Substitution	σ	3.4.3		3.4.4	3.4.4	3.4.4	3.5.3
Linking	$\times$			3.4.6	3.4.6	3.4.7	3.5.4
Structural congruence	$\equiv$			(3.3.4)	(3.3.5)	3.7.11	3.5.7
Reaction	$\longrightarrow$					3.7.12	3.5.8
Labelled transition	$\xrightarrow{\alpha}$					3.5.17	3.5.16
Rearrangement	$\triangleright$						3.5.14

3.4.1. Behaviour of free names. There are several places in the definitions where a side-condition depends on the free names of a process. In order to check these side-conditions during proofs, it is necessary to be able to calculate the free names of a program or process, starting only from its relationship with another program or process. This subsection proves the relevant theorems.

Unlike non-linear calculi, such as the π-calculus, the free names of a MIN program follow a very rigid discipline. For example, in the π-calculus, from the reaction $p \longrightarrow p'$ it is only possible to deduce $\mathrm{fn}(p) \supseteq \mathrm{fn}(p')$, whereas in MIN we have the stronger condition $\mathrm{fn}(p) = \mathrm{fn}(p')$. Though not particularly important, little bonuses like this are one of the attractions of a linear calculus.

PROPOSITION 3.4.1. *(Behaviour of free names)*

(1) *For any ⟨process⟩ p, ⟨name⟩s $\vec{x}$ and ⟨tree⟩s $\vec{t}$ such that the substitution is defined,* $fn(p[\vec{x} \twoheadrightarrow \vec{t}]) = \quad fn(p) \vartriangle fn(\langle \vec{x}, \vec{t} \rangle)$.
(2) *For any ⟨program⟩s P and Q,* $fn(P \times Q) = fn(P) \vartriangle fn(Q)$.
(3) *If* $p \equiv q$ *then* $fn(p) = fn(q)$.
(4) *If* $p \longrightarrow p'$ *then* $fn(p) = fn(p')$.
(5) *If* $p \xrightarrow{\alpha} p'$ *then* $fn(p) \vartriangle fn(\alpha) \vartriangle fn(p') = \{\}$.

PROOF. 1 can be proved by induction on the structure of p. 2 is immediate from definitions 3.3.1 and 3.3.8. 3 can be proved by induction on the derivation of $p \equiv q$, using 1 to handle the BUF rule. 4 can be proved by induction on the derivation of $p \longrightarrow p'$, using 3 to handle the STRUCT rule. 5 can be proved by induction on the derivation of $p \xrightarrow{\alpha} p'$, using 1 to handle the IN rule. □

PROPOSITION 3.4.2. *(Inputs and outputs use a channel)*

If $p \xrightarrow{x c(\vec{y})} p'$ *or* $p \xrightarrow{\overline{x} c(\vec{y})} p'$ *then* $x \in fn(p)$ *(so* $x \notin fn(p')$*) and* $\vec{y} \in fn(p')$ *(so* $\vec{y} \notin fn(p)$*).*

PROOF. Induction on the derivation of the transition. □

3.4.2. Behaviour of substitution. Substitutions compose and commute nicely.

PROPOSITION 3.4.3. *(Composition of substitutions)*

Extend substitutions to ⟨tree⟩s in the obvious way: $\sigma(t)$ *is* t *with every name* x *replaced by* $\sigma(x)$.

Given two substitutions σ_1 *and* σ_2*, define their composition* σ *by* $\sigma(x) = \sigma_2(\sigma_1(x))$. *Then* $p\sigma_1\sigma_2 = p\sigma$.

If $fn(\langle \vec{x}, \vec{t} \rangle) \cap fn(\langle \vec{y}, \vec{u} \rangle) = \{\}$ *then* $p[\vec{x} \twoheadrightarrow \vec{t}][\vec{y} \mapsto \vec{u}] = p[\vec{x}, \vec{y} \mapsto \vec{t}, \vec{u}] = p[\vec{y} \mapsto \vec{u}][\vec{x} \twoheadrightarrow \vec{t}]$.

PROOF. The second result follows from the first, which follows immediately from definition 3.3.3. □

PROPOSITION 3.4.4. *(Behaviour of substitution)*

(1) *If* $p \equiv q$ *then* $p\sigma \equiv q\sigma$.
(2) *If* $p \longrightarrow p'$ *then* $p\sigma \longrightarrow p'\sigma$.
(3) *If* $p \xrightarrow{\alpha} p'$ *and* $\sigma(x) = x$ *for all* $x \in fn(\alpha)$ *then* $p\sigma \xrightarrow{\alpha} p'\sigma$.

PROOF. 1 can be proved by induction on the derivation of $p \equiv q$, using proposition 3.4.3 for the BUF rule. 2 can be proved by induction on the derivation of $p \longrightarrow p'$, using 1 for the STRUCT rule. 3 can be proved by induction on the derivation of $p \xrightarrow{\alpha} p'$, using proposition 3.4.3 for the IN and REACT rules. □

The converses do not hold.

3.4.3. Behaviour of linking. I want to digress for one proposition to illustrate the sense in which the ⟨destructor⟩s of a program are private. The proposition provides a way of alpha-converting ⟨destructor⟩s. It is not needed much, but symmetry operations are almost always worth noting for their own sake. In particular, note that $d(x)$ and $d(x, y)$ are quite unrelated destructors which can be independently renamed, despite the fact that they are both called d, because they have different arity.

PROPOSITION 3.4.5. *(Change of destructors)*

Given a sequence $f_0, f_1, \ldots$ of injections on ⟨destructor⟩, define a function $\vec{f}$ on ⟨program⟩s. The result $\vec{f}(P)$ is formed from P by replacing all occurrences of $d(t_1, \ldots, t_n)$ by $f_n(d)(t_1, \ldots, t_n)$. This includes all occurrences in the ⟨environment⟩ as well as in the main graph.

Then, $P \xrightarrow{\alpha} P'$ iff $\vec{f}(P) \xrightarrow{\alpha} \vec{f}(P')$.

PROOF. An easy induction on the derivation of the transition. □

Now back to the main story. Linking behaves very similarly to the primitive × operator on ⟨process⟩es. In fact, the best viewpoint is that the primitive × operator is the special case of linking in which the two programs to be linked have the same ⟨environment⟩. It is a pity that the definition must be the other way around. The results of this section patch up the difference.

PROPOSITION 3.4.6. *(Behaviour of linking)*

If $P_1 \equiv Q_1$ and $P_2 \equiv Q_2$ then $P_1 \times P_2 \equiv Q_1 \times Q_2$.

If $P \longrightarrow P'$ then $P \times Q \longrightarrow P' \times Q$ and $Q \times P \longrightarrow Q \times P'$.

PROOF. Definition 3.3.8 immediately reduces both parts to corresponding statements about ⟨process⟩es. □

The converses do not hold. However, the next proposition holds in both directions.

PROPOSITION 3.4.7. *(Labelled transitions of linked programs)*

The labelled transition relation satisfies the following rules:

$$\frac{P \xrightarrow{\alpha} P' \quad fn(\alpha) \cap fn(Q) = \{\}}{P \times Q \xrightarrow{\alpha} P' \times Q} PAR'_1 \qquad \frac{P \xrightarrow{\overline{\lambda}} P' \quad Q \xrightarrow{\lambda} Q' \quad P' \times Q' \xrightarrow{\alpha} R'}{P \times Q \xrightarrow{\alpha} R'} COM'_1$$

$$\frac{fn(P) \cap fn(\alpha) = \{\} \quad Q \xrightarrow{\alpha} Q'}{P \times Q \xrightarrow{\alpha} P \times Q'} PAR'_2 \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q' \quad P' \times Q' \xrightarrow{\alpha} R'}{P \times Q \xrightarrow{\alpha} R'} COM'_2$$

Furthermore, any labelled transition of $P \times Q$ can be derived using these rules.

PROOF. An easy induction, similar to proposition 3.4.5, shows that adding new ⟨rewrite⟩s to a program's ⟨environment⟩ does not affect its labelled transitions, provided the ⟨destructor⟩s mentioned in the new rules are not mentioned anywhere in the original program. This result, along with proposition 3.4.5 and definition 3.3.8, allow any derivation using the rules of this proposition to be converted into one using the corresponding rules of definition 3.3.7, and *vice versa*. □

3.5. Rearrangements

If you tick off the results we have seen so far in the table at the start of section 3.4, you will see that there are only three interesting ones left, all of which concern the labelled transition relation. Unfortunately, we are not yet in a position to prove them. The tool we require is the rearrangement relation, written $\triangleright$, and we have a whole column of results about it to fill in before we can go any further.

3.5.1. Communication in MIN. The problem that the rearrangement relation is designed to solve is the rather unusual form of the COM rules. Compare one of the communication rules of CCS:

$$\frac{P \xrightarrow{\overline{\lambda}} P' \quad Q \xrightarrow{\lambda} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

with MIN's COM_1 rule:

$$\frac{p \xrightarrow{\overline{\lambda}} p' \quad q \xrightarrow{\lambda} q' \quad p' \times q' \xrightarrow{\alpha} r'}{p \times q \xrightarrow{\alpha} r'}\mathrm{COM}_1$$

What is going on here? What is that third premise for? I will answer using two examples. The first (figure [3H]) illustrates that the CCS rule is quite wrong for MIN, by showing what it means graphically. In fact, it is arguably wrong for any asynchronous calculus. The second (figure [3I]) shows an instance of a rather complicated rule that does work, and which clearly must be derivable somehow.

The important points to note in figure [3I] are that a sequence of several transitions of p and q individually go towards a single transition of $p \times q$, and that the conclusion need not be a silent transition. It should be clear that examples can be concocted in which the sequences are arbitrarily long, and that the transitions of p and q are complementary all the way along, apart from the very last transition which becomes the conclusion.

The third premise allows the COM rules to be stacked into a list. Each item in the list is a transition of p and a complementary transition of q. The list is terminated with a transition derived in some other way, which contributes the action for the conclusion. For example, the transition in figure [3I] can be derived like this (distracting details are suppressed):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\overline{x}c()} \quad \dfrac{}{xc()} \qquad \dfrac{\dfrac{}{yc()} \quad \dfrac{}{\overline{y}c()} \qquad \dfrac{}{\overline{z}c()}}{(1 \times \langle c(), z\rangle) \times 1 \xrightarrow{\overline{z}c()} (1 \times 1) \times 1}\mathrm{PAR}_1}{(1 \times \langle y, z\rangle) \times \langle c(), y\rangle \xrightarrow{\overline{z}c()} (1 \times 1) \times 1}\mathrm{COM}_2}{(\langle c(), x\rangle \times \langle y, z\rangle) \times \langle x, y\rangle \xrightarrow{\overline{z}c()} (1 \times 1) \times 1}\mathrm{COM}_1$$

3.5.2. Rearrangements. The purpose of the rearrangement relation is to represent just one of the steps in a stack of COM rules; that is, an output by one

program and a complementary input by the other. The goal is to be able to derive the communication in figure [3I] as follows:

$$\begin{array}{lll}(\langle c(), x\rangle \times \langle y, z\rangle) \times \langle x, y\rangle & \triangleright & (1 \times \langle y, z\rangle) \times \langle c(), y\rangle \\ & \triangleright & (1 \times \langle c(), z\rangle) \times 1 \\ & \xrightarrow{\overline{z}c()} & (1 \times 1) \times 1\end{array}$$

Each of the rearrangement steps is derived using a rule similar to the simple, tractable one that is familiar from CCS. They are called rearrangements because their effect on a ⟨process⟩ is only to turn it into another that constructs the same graph. In fact, we will find (proposition 3.5.7) that rearrangements are a subrelation of the structural congruence.

DEFINITION 3.5.1. (One-step transitions)

Let the *one-step labelled transition relation*, with elements written $P \xrightarrow{\alpha}_1 P'$, be the part of the labelled transition relation that can be derived without using the COM rules.

DEFINITION 3.5.2. (Rearrangements)

Let the *rearrangement relation*, written $\triangleright$, be the smallest relation satisfying the following rules:

$$\frac{E \vdash p \triangleright p'}{E \vdash p \times q \triangleright p' \times q}\mathrm{PAR}_1^{\triangleright} \qquad \frac{E \vdash p \xrightarrow{\overline{\lambda}}_1 p' \quad E \vdash q \xrightarrow{\lambda}_1 q'}{E \vdash p \times q \triangleright p' \times q'}\mathrm{COM}_1^{\triangleright}$$

$$\frac{E \vdash q \triangleright q'}{E \vdash p \times q \triangleright p \times q'}\mathrm{PAR}_2^{\triangleright} \qquad \frac{E \vdash p \xrightarrow{\lambda}_1 p' \quad E \vdash q \xrightarrow{\overline{\lambda}}_1 q'}{E \vdash p \times q \triangleright p' \times q'}\mathrm{COM}_2^{\triangleright}$$

I will often omit $E \vdash$, and just write $p \triangleright p'$.

PROPOSITION 3.5.3. *(Rearrangements respect free names and substitutions)*

If $p \triangleright p'$ *then* $fn(p) = fn(p')$.

If $p \triangleright p'$ *then* $p\sigma \triangleright p'\sigma$.

PROOF. Simple inductions on the derivation of $\triangleright$, using propositions 3.4.1 and 3.4.4 for the base cases. □

PROPOSITION 3.5.4. *(Rearrangements of linked programs)*

The rearrangement relation satisfies the following rules:

$$\frac{P \triangleright P'}{P \times Q \triangleright P' \times Q} \qquad \frac{P \xrightarrow{\overline{\lambda}}_1 P' \quad Q \xrightarrow{\lambda}_1 Q'}{P \times Q \triangleright P' \times Q'}$$

$$\frac{Q \triangleright Q'}{P \times Q \triangleright P \times Q'} \qquad \frac{P \xrightarrow{\lambda}_1 P' \quad Q \xrightarrow{\overline{\lambda}}_1 Q'}{P \times Q \triangleright P' \times Q'}$$

Furthermore, any rearrangement of $P \times Q$ *can be derived using these rules.*

PROOF. Reason as for proposition 3.4.7. □

3.5.3. Sound and complete. The following propositions relate rearrangements to the COM rules of definition 3.3.7.

PROPOSITION 3.5.5. *(Completeness of rearrangements)*

$$\xrightarrow{\alpha} \quad \subseteq \quad \overset{*}{\triangleright} \xrightarrow{\alpha}_1$$

PROOF. Induction on the derivation of $\xrightarrow{\alpha}$. If it was derived using any rule other than the PAR or COM rules, then let the derivation stand unchanged. If it was derived using the PAR_1 rule (the PAR_2 rule is similar), apply the inductive hypothesis to the premise, then use the $\mathrm{PAR}_1^{\triangleright}$ rule several times and the PAR_1 rule once. If $p \times q \xrightarrow{\alpha} r'$ was derived using the COM_1 rule (the COM_2 rule is similar) from $p \xrightarrow{\overline{\lambda}} p'$ and $q \xrightarrow{\lambda} q'$ and $p' \times q' \xrightarrow{\alpha} r'$ then apply the inductive hypothesis to the first two premises to obtain $p \ \triangleright^* \xrightarrow{\overline{\lambda}} \ p'$ and $q \ \triangleright^* \xrightarrow{\lambda} \ q'$. Then use the $\mathrm{PAR}^{\triangleright}$ rules several times then then the $\mathrm{COM}_1^{\triangleright}$ rule to deduce $p \times q \ \triangleright^* \triangleright^* \triangleright \ p' \times q'$. Finally, use the inductive hypothesis again to obtain $p' \times q' \ \triangleright^* \xrightarrow{\alpha} \ r'$. □

PROPOSITION 3.5.6. *(Soundness of rearrangements)*

$$\overset{*}{\triangleright} \xrightarrow{\alpha} \quad \subseteq \quad \xrightarrow{\alpha} \overset{*}{\triangleright}$$

PROOF. In fact, if $p \ \triangleright \xrightarrow{\alpha} \ p'$ then either $p \xrightarrow{\alpha} p'$ or $p \ \xrightarrow{\alpha} \triangleright \ p'$. This can be proved by induction on the derivation of $\triangleright$. If it was derived using a $\mathrm{COM}^{\triangleright}$ rule, then it can be absorbed into the transition using the corresponding COM rule, to obtain the first of the alternative conclusions. If it was derived using the PAR_1 rule (the PAR_2 rule is similar) then $p = q \times r$ and $q \ \triangleright \ q'$ and $q' \times r \xrightarrow{\alpha} p'$. Distinguish cases according to the rule used to derive $\xrightarrow{\alpha}$. If it was PAR_1, then use the inductive hypothesis. Similarly, if it was COM_1 (COM_2 is similar) then apply the inductive hypothesis to the first premise, then use the COM_1 rule and, if necessary, the $\mathrm{PAR}_1^{\triangleright}$ rule. If it was PAR_2, then $r \xrightarrow{\alpha} r'$ and $p' = q' \times r'$, so use PAR_2 then $\mathrm{PAR}_1^{\triangleright}$ to get $q \times r \ \xrightarrow{\alpha} \ q \times r' \quad \triangleright \quad q' \times r'$, that is, the second of the alternative conclusions. Proposition 3.5.3 is needed to check the side-condition on the PAR_2 rule. □

Note that it is not true that $\triangleright \xrightarrow{\alpha} \ \subseteq \ \xrightarrow{\alpha}$. This is because the rearrangement could be in a part of the process that is distant from the part that performs the transition. This is an unavoidable consequence of separating out the rearrangements. It turns out not to be a problem.

3.5.4. Structural congruence. The following proposition states that rearranging a textual program makes no difference to the graphical program it constructs.

PROPOSITION 3.5.7. *(Rearrangements stay within structural congruence)*

(1) *If* $p \xrightarrow{x\,c(\vec{y})}_1 p'$ *then* $p \times \langle x, c(\vec{y})\rangle \equiv p'$ *(in fact* $p[x \leftrightarrow c(\vec{y})] = p'$ *).*
(2) *If* $p \xrightarrow{\overline{x\,c(\vec{y})}}_1 p'$ *then* $p \equiv \langle x, c(\vec{y})\rangle \times p'$.
(3) *If* $p \ \triangleright \ p'$ *then* $p \equiv p'$.

PROOF. Easy inductions on the derivations of the transitions prove 1 and 2. Induction on the derivation of $\triangleright$ proves 3, using 1 and 2 for the base cases. □

COROLLARY 3.5.8. *(Rearrangements and reactions)*
If $p \ (\triangleright \cup \triangleleft)^* \longrightarrow (\triangleright \cup \triangleleft)^* \ p'$ *then* $p \longrightarrow p'$.

The following simple consequences are a little off track, but will be useful later:

PROPOSITION 3.5.9. *(Input and output form)*

(1) *If* $p \xrightarrow{x\,c(\vec{y})} p'$ *then* $p \times \langle x, c(\vec{y})\rangle \equiv p'$.
(2) *If* $p \xrightarrow{\overline{x}c(\vec{y})} p'$ *then* $p \equiv \langle x, c(\vec{y})\rangle \times p'$.
(3) *If* $p_1 \xleftarrow{\lambda}\xrightarrow{\lambda} p_2$ *or* $p_1 \xleftarrow{\overline{\lambda}}\xrightarrow{\overline{\lambda}} p_2$ *then* $p_1 \equiv p_2$.

PROOF. 3 follows from 1 and 2, which follow from propositions 3.5.7 and 3.5.5. □

3.5.5. One-step asynchrony. Because rearrangements are derived from one-step inputs and outputs, many proofs by induction have as their base cases a statement about one-step transitions. This subsection covers the principal ones.

PROPOSITION 3.5.10. *(One-step asynchrony)*

(1) *If* λ *is an input action (so* $\overline{\lambda}$ *is an output) and* $fn(\alpha) \cap fn(\lambda) = \{\}$ *then*

$$\begin{array}{rcl}
\xleftarrow{\alpha}_1\xrightarrow{\lambda}_1 & \subseteq & \xrightarrow{\lambda}_1\xleftarrow{\alpha}_1 \\
\xrightarrow{\alpha}_1\xrightarrow{\lambda}_1 & \subseteq & \xrightarrow{\lambda}_1\xrightarrow{\alpha}_1 \\
\xleftarrow{\overline{\lambda}}_1\xrightarrow{\alpha}_1 & \subseteq & \xrightarrow{\alpha}_1\xleftarrow{\overline{\lambda}}_1 \\
\xrightarrow{\overline{\lambda}}_1\xrightarrow{\alpha}_1 & \subseteq & \xrightarrow{\alpha}_1\xrightarrow{\overline{\lambda}}_1
\end{array}$$

(2) *If* $p \xrightarrow{\overline{x}b(\vec{y})}_1 p_1$ *and* $p \xrightarrow{\overline{x}c(\vec{z})}_1 p_2$ *then* $b = c$ *and* $|\vec{y}| = |\vec{z}|$ *and* $p_1[\vec{y} \mapsto \vec{z}] = p_2$.
(3) *If* $p \xrightarrow{\lambda}_1 p_1$ *and* $p \xrightarrow{\lambda}_1 p_2$ *then* $p_1 = p_2$.
(4) *If* $x \in fn(p)$ *and* $\vec{y} \notin fn(p)$ *then* $p \xrightarrow{x\,c(\vec{y})}_1 p[x \twoheadrightarrow c(\vec{y})]$.

PROOF. Each of the four parts of 1 can be proved by induction on the derivation of one of the transitions. If the two transitions were derived using the same PAR rule, then use the inductive hypothesis. If they were derived using different PAR rules, then the result follows immediately. This leaves a large number of base cases to check. It is fortunate that in every case one of the transitions must be an input. By proposition 3.5.7, input transitions are just substitutions, so we can use proposition 3.4.4.

2 can be proved by induction on one of the transitions. The crucial point is that both transitions occur on the channel x. By proposition 3.4.2, this greatly constrains the derivations. In fact, they must both be derived using the same rules.

3 is a restatement of part of proposition 3.5.7, and 4 is a restatement of the IN rule. They are included here only for completeness. □

For the purposes of asynchrony, rearrangements behave rather like silent transitions.

PROPOSITION 3.5.11. *(Rearrangement asynchrony)*

If λ is an input action (so $\overline{\lambda}$ is an output), then

$$\begin{array}{lcl} \triangleright \xrightarrow{\lambda}_1 & \subseteq & \xrightarrow{\lambda}_1 \triangleright \\ \xrightarrow{\overline{\lambda}}_1 \triangleright & \subseteq & \triangleright \xrightarrow{\overline{\lambda}}_1 \\ \triangleleft \xrightarrow{\alpha}_1 & \subseteq & \xrightarrow{\alpha}_1 \triangleleft \end{array}$$

PROOF. Each part can be proved by induction on the derivation of the rearrangement. We may assume that the transition was not derived using the IN rule, by proposition 3.3.9. If the rearrangement was derived using a COM rule, we use proposition 3.5.10. If it was derived using the same PAR rule as the transition, we use the inductive hypothesis. If they were derived using opposite PAR rules, the result is immediate. □

3.5.6. Confluence. Rearrangements are partial communications that gradually move a message towards its unique destination. With this intuition, it is a bit of a shock to find that they are not confluent. The problematic case is when two messages approach each other from opposite ends of a chain of wires, as shown in figure [3J]. When they meet, they block each other's progress, and there are several places in which they might meet. This can only occur in badly-typed programs, but I will nevertheless take this case seriously.

The good news is that the non-confluent rearrangements are not needed for deriving labelled transitions. We can identify a *pointless* subset of the rearrangement relation which can be delayed with respect to one-step transitions and other rearrangements, except other pointless ones.

DEFINITION 3.5.12. (Pointless inputs and rearrangments)

Let the *pointless input relation*, with elements written $P \xrightarrow{\lambda}_? P'$ for programs P and P' and input action λ, be the subset of the one-step (input) labelled transition relation that can be derived using the following rules:

$$\frac{\langle t, c(\vec{u})\rangle \xrightarrow{\lambda}_1 p'}{\langle t, c(\vec{u})\rangle \xrightarrow{\lambda}_? p'}\mathrm{IN}_{?1} \qquad \frac{p \xrightarrow{\lambda}_? p' \quad \mathrm{fn}(\lambda) \cap \mathrm{fn}(q) = \{\}}{p \times q \xrightarrow{\lambda}_? p' \times q}\mathrm{PAR}_{?1}$$

$$\frac{\langle c(\vec{u}), t\rangle \xrightarrow{\lambda}_1 p'}{\langle c(\vec{u}), t\rangle \xrightarrow{\lambda}_? p'}\mathrm{IN}_{?2} \qquad \frac{\mathrm{fn}(p) \cap \mathrm{fn}(\lambda) = \{\} \quad q \xrightarrow{\lambda}_? q'}{p \times q \xrightarrow{\lambda}_? p \times q'}\mathrm{PAR}_{?2}$$

Let the *pointless rearrangement relation*, written $\triangleright_?$, be the subset of the rearrangement relation that can be derived from pointless inputs.

This is not the only class of pointless rearrangements that can be identified, but it is all we will need.

PROPOSITION 3.5.13. *(Pointless rearrangements)*

If $p \xrightarrow{\lambda}_? p'$ $\xrightarrow{x\,c(\vec{y})}$ p'' and $x \in fn(\lambda)$ then $p' \xrightarrow{x\,c(\vec{y})}_? p''$.

If $fn(\lambda) \cap fn(\alpha) = \{\}$, *then*

$$\begin{array}{rcl} \xrightarrow{\lambda}_? \xrightarrow{\alpha}_1 & \subseteq & \xrightarrow{\alpha}_1 \xrightarrow{\lambda}_? \\ \xrightarrow{\lambda}_? \triangleright & \subseteq & \triangleright \xrightarrow{\lambda}_? \\ \underset{?}{\triangleright} \xrightarrow{\alpha}_1 & \subseteq & \xrightarrow{\alpha}_1 \underset{?}{\triangleright} \\ \underset{?}{\triangleright\triangleright} & \subseteq & \underset{?}{\triangleright\triangleright} \end{array}$$

PROOF. The fifth part depends on the third and fourth which both depend on the second. The fifth part also depends on the first part, and on propositions 3.5.10 and 3.5.11 The fourth part also uses proposition 3.5.10. All parts can be proved by induction on either of the derivations. There are $4 + 6 + 8 + 8 + 16 = 42$ cases to check, of which most are easy. Symmetry lightens the load to 21 cases. Furthermore, there are patterns. Whenever the two premises were derived using the same PAR rule (5 cases), we use the inductive hypothesis, and whenever they were derived using opposite PAR rules (4 cases) the result is immediate. I will present just the four that depend on other propositions.

For the fourth part, suppose $p \times q \ \triangleright_? \ p' \times q'$ was derived using COM_1 from $p \xrightarrow{\overline{\lambda}}_1 p'$ and $q \xrightarrow{\lambda}_? q'$, and that $p' \times q' \xrightarrow{\alpha}_1 p'' \times q'$ was derived using PAR_1 from $p' \xrightarrow{\alpha}_1 p''$ and $\text{fn}(\alpha) \cap \text{fn}(q') = \{\}$. We may assume by alpha-conversion that $\text{fn}(\lambda) \cap \text{fn}(\alpha) = \{\}$. By proposition 3.5.10, $p \xrightarrow{\alpha}_1 \xrightarrow{\overline{\lambda}}_1 p''$. Therefore by PAR_1 and COM_1 $p \times q \xrightarrow{\alpha}_1 \triangleright_? \ p'' \times q'$ as required.

For the fifth part, suppose $p \times q \ \triangleright_? \ p' \times q'$ was derived using COM_1 from $p \xrightarrow{\overline{\lambda}}_1 p'$ and $q \xrightarrow{\lambda}_? q'$, and that $p' \times q' \ \triangleright \ p'' \times q'$ was derived using PAR_1 from $p' \ \triangleright \ p''$. Then by proposition 3.5.11 $p \ \triangleright \xrightarrow{\overline{\lambda}}_1 \ p''$ and hence $p \times q \ \triangleright\triangleright_? \ p'' \times q'$ as required.

For the fifth part, suppose $p \times q \ \triangleright_? \ p' \times q'$ was derived using COM_1 from $p \xrightarrow{\overline{\lambda}}_1 p'$ and $q \xrightarrow{\lambda}_? q'$, and that $p' \times q' \ \triangleright \ p'' \times q''$ was derived also using COM_1 from $p' \xrightarrow{\overline{x}c(\vec{y})}_1 p''$ and $q' \xrightarrow{xc(\vec{y})}_1 q''$. We have some freedom in choosing the names: we may assume that $\vec{y} \notin \text{fn}(\lambda)$. However, we must consider $x \in \text{fn}(\lambda)$ and $x \notin \text{fn}(\lambda)$ separately. In the former case, we can apply part 1 to get $q' \xrightarrow{xc(\vec{y})}_? q''$ and hence $p' \times q' \ \triangleright_? \ p'' \times q''$ as required. In the latter case $\text{fn}(\lambda) \cap \text{fn}(xc(\vec{y})) = \{\}$, so we can apply proposition 3.5.10 to the outputs and part 2 to the inputs to get $p \xrightarrow{\overline{x}c(\vec{y})}_1 \xrightarrow{\overline{\lambda}} p''$ and $q \xrightarrow{xc(\vec{y})}_1 \xrightarrow{\lambda}_? q''$, and hence $p \times q \ \triangleright\triangleright_? \ p'' \times q''$ as required.

For the fifth part, suppose $p \times q \ \triangleright_? \ p' \times q'$ was derived using COM_1 from $p \xrightarrow{\overline{\lambda}}_1 p'$ and $q \xrightarrow{\lambda}_? q'$, and that $p' \times q' \ \triangleright \ p'' \times q''$ was derived using COM_2 from $p' \xrightarrow{xc(\vec{y})}_1 p''$ and $q' \xrightarrow{\overline{x}c(\vec{y})}_1 q''$. We may assume $\vec{y} \notin \text{fn}(\lambda)$ but an inner induction on the derivation of $\xrightarrow{\lambda}_? \xrightarrow{\overline{x}c(\vec{y})}_1$ is necessary to show that $x \notin \text{fn}(\lambda)$. Now we can use proposition 3.5.10 to obtain $p \xrightarrow{xc(\vec{y})}_1 \xrightarrow{\overline{\lambda}}_1 p''$ and part 2 to obtain $q \xrightarrow{\overline{x}c(\vec{y})}_1 \xrightarrow{\lambda}_? q''$, and hence $p \times q \ \triangleright\triangleright_? \ p'' \times q''$ as required. □

PROPOSITION 3.5.14. *(Confluence)*

If $p_1 \lhd p \rhd p_2$ *then either* $p_1 \lhd_? p \rhd_? p_2$ *or* $p_1 \rhd\lhd p_2$ *or* $p_1 = p_2$.

PROOF. Induction on the derivation of one of the rearrangements. If they were derived using the same PAR rule, use the inductive hypothesis. If they were derived using opposite PAR rules, the second conclusion is immediate. If one was derived using a COM rule and the other using a PAR rule, then part 3 of proposition 3.5.11 gives the second conclusion. This leaves the cases in which they are both derived using COM rules.

If the two rearrangements were both derived using COM_1 (the case in which they are both derived using COM_2 is similar) then $p = q \times r$ and $p_1 = q_1 \times r_1$ and $p_2 = q_2 \times r_2$ and $q_1 \xleftarrow{\overline{x}c(\vec{y})}_1 q \xrightarrow{\overline{w}b(\vec{z})}_1 q_2$ and $r_1 \xleftarrow{xc(\vec{y})}_1 r \xrightarrow{wb(\vec{z})}_1 r_2$. If $w \neq x$ then by alpha-conversion we may assume that all the names are different, and so part 1 of proposition 3.5.10 gives the second conclusion. If $w = x$ then part 2 of proposition 3.5.10 gives $q_1 \times r_1 = q_2 \times r_2$, the third conclusion.

If $q_1 \times r_1 \lhd q \times r$ was derived using COM_1 and $q \times r \rhd q_2 \times r_2$ was derived using COM_2 then $q_1 \xleftarrow{\overline{x}c(\vec{y})}_1 q \xrightarrow{wb(\vec{z})}_1 q_2$ and $r_1 \xleftarrow{xc(\vec{y})}_1 r \xrightarrow{\overline{w}b(\vec{z})}_1 r_2$. Again, if $w \neq x$ then we may assume that all the names are different, and so part 1 of proposition 3.5.10 gives the second conclusion. If $w = x$, the reader may by now have guessed by elimination that we are heading for the first conclusion, that the two input transitions are pointless.

What is required is the following result: If $q_1 \xleftarrow{\overline{x}c(\vec{y})}_1 q \xrightarrow{xb(\vec{z})}_1 q_2$ then $q \xrightarrow{xb(\vec{z})}_? q_2$. This can be proved by induction on the derivation of one of the transitions. If either is derived using a PAR rule, then the other must be derived using the same PAR rule (we may neglect the IN rule by proposition 3.3.9), so use the inductive hypothesis. This leaves the case in which the output is derived using an OUT rule, which constrains the form of q enough to show that the input is pointless. □

DEFINITION 3.5.15. (Rearrangements equivalence)

Let the *rearrangement equivalence relation*, written $\bowtie$, be the smallest equivalence containing the rearrangement relation.

PROPOSITION 3.5.16. *(Rearrangement equivalence is a strong bisimulation)*

$$\bowtie \xrightarrow{\alpha} \subseteq \xrightarrow{\alpha} \bowtie$$

PROOF. The result follows by induction on the derivation of $\bowtie$ from $\rhd \xrightarrow{\alpha} \subseteq \xrightarrow{\alpha} \bowtie$ (proposition 3.5.6) and $\lhd \xrightarrow{\alpha} \subseteq \xrightarrow{\alpha} \bowtie$. The latter follows (by proposition 3.5.5) from $\lhd \rhd^* \xrightarrow{\alpha}_1 \subseteq \xrightarrow{\alpha} \bowtie$, which follows (by proposition 3.5.6) from $\lhd \rhd^* \xrightarrow{\alpha}_1 \subseteq \rhd^* \xrightarrow{\alpha}_1 \bowtie$. This we will prove by induction on the number of $\rhd$ steps.

The base case, $\lhd \xrightarrow{\alpha}_1 \subseteq \xrightarrow{\alpha}_1 \lhd$, is part 3 of proposition 3.5.11. The inductive hypothesis is that if $p \lhd \rhd^{(n+1)} \xrightarrow{\alpha}_1 p'$ then either $p \lhd_? \rhd^n \xrightarrow{\alpha}_1 \rhd_? p'$ or $p \rhd \lhd \rhd^n \xrightarrow{\alpha}_1 p'$ or $p \rhd^n \xrightarrow{\alpha}_1 p'$. This follows from proposition 3.5.14 and in the first case $n+1$ applications of proposition 3.5.13. □

3.5.7. Asynchrony. Selinger's asynchrony conditions characterise a class of asynchronous communication. More precisely, a labelled transition system belongs to the class if it is weakly bisimilar to one satisfying the conditions. Selinger worked out the conditions for CCS-like calculi, and they need to be generalised in order to include channel passing. I presented this generalisation in chapter 2.

We have already seen, in propositions 3.5.10 and 3.5.11, that the one-step labelled transition relation satisfies the asynchrony conditions if we think of rearrangements as silent transitions. The fit is very tight: not only is it weakly bisimilar to one satisfying the conditions, it actually satisfies them itself. However, both the one-step labelled transition system and the rearrangement relation are fictions that don't make much sense in the graphical notation.

In this section, I will show that the full labelled transition system satisfies the asynchrony conditions up to $\bowtie$. Proposition 3.5.16 assures us that this is sufficient to call MIN asynchronous. Furthermore, we know by part 3 of proposition 3.5.7 that $\bowtie$ only relates textual programs that construct the same graphical program, so in the graphical notation the labelled transition system is once again satisfied up to $=$.

THEOREM 3.5.17. *(Asynchrony)*

(1) *If λ is an input action (so $\overline{\lambda}$ is an output) and $fn(\alpha) \cap fn(\lambda) = \{\}$ then*

$$\begin{aligned}
\overleftarrow{\alpha}\,\overrightarrow{\lambda} &\subseteq \overrightarrow{\lambda}\bowtie\overleftarrow{\alpha} \\
\overrightarrow{\alpha}\,\overrightarrow{\lambda} &\subseteq \overrightarrow{\lambda}\,\overrightarrow{\alpha}\bowtie \\
\overleftarrow{\overline{\lambda}}\,\overrightarrow{\alpha} &\subseteq \overrightarrow{\alpha}\bowtie\overleftarrow{\overline{\lambda}} \\
\overrightarrow{\overline{\lambda}}\,\overrightarrow{\alpha} &\subseteq \overrightarrow{\alpha}\,\overrightarrow{\overline{\lambda}}\bowtie
\end{aligned}$$

(2) *If $p \xrightarrow{\overline{x}b(\vec{y})} p_1$ and $p \xrightarrow{\overline{x}c(\vec{z})} p_2$ then $b = c$ and $|\vec{y}| = |\vec{z}|$ and $p_1[\vec{y} \mapsto \vec{z}] \bowtie p_2$.*

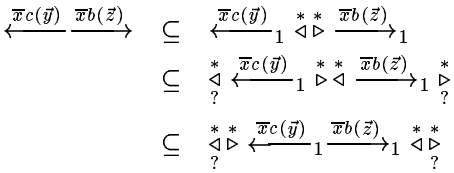
(3) *If $p \xrightarrow{\lambda} p_1$ and $p \xrightarrow{\lambda} p_2$ then $p_1 \bowtie p_2$.*

(4) *If $x \in fn(p)$ and $\vec{y} \notin fn(p)$ then $p \xrightarrow{x\,c(\vec{y})} p[x \mapsto c(\vec{y})]$.*

PROOF. For 1, reason as follows using propositions 3.5.5, 3.5.6, 3.5.10, 3.5.11 and 3.5.14 (the third and fourth parts are similar to the first and second respectively):

$$\begin{aligned}
\overleftarrow{\alpha}\,\overrightarrow{\lambda} &\subseteq \overleftarrow{\alpha}_1\,{}^{*}_{\triangleleft}\,{}^{*}_{\triangleright}\,\overrightarrow{\lambda}_1 \\
&\subseteq {}^{*}_{?\triangleleft}\,\overleftarrow{\alpha}_1\,{}^{*}_{\triangleright}\,{}^{*}_{\triangleleft}\,\overrightarrow{\lambda}_1\,{}^{*}_{\triangleright ?} \\
&\subseteq {}^{*}_{?\triangleleft}\,{}^{*}_{\triangleright}\,\overleftarrow{\alpha}_1\overrightarrow{\lambda}_1\,{}^{*}_{\triangleleft}\,{}^{*}_{\triangleright ?} \\
&\subseteq \bowtie\overrightarrow{\lambda}_1\overleftarrow{\alpha}_1\bowtie \\
&\subseteq \overrightarrow{\lambda}\bowtie\overleftarrow{\alpha} \\
\overrightarrow{\alpha}\,\overrightarrow{\lambda} &\subseteq {}^{*}_{\triangleright}\,\overrightarrow{\alpha}_1\,{}^{*}_{\triangleright}\,\overrightarrow{\lambda}_1 \\
&\subseteq {}^{*}_{\triangleright}\,\overrightarrow{\alpha}_1\overrightarrow{\lambda}_1\,{}^{*}_{\triangleright} \\
&\subseteq {}^{*}_{\triangleright}\,\overrightarrow{\lambda}_1\overrightarrow{\alpha}_1\,{}^{*}_{\triangleright} \\
&\subseteq \overrightarrow{\lambda}\,\overrightarrow{\alpha}\,{}^{*}_{\triangleright}
\end{aligned}$$

For 2, start in the same way as for part 1 of 1:



then use part 2 of proposition 3.5.10. The same approach works for 3. Since any one-step labelled transition is also a labelled transition, part 4 follows immediately from part 4 of proposition 3.5.10. □

3.6. Congruence results

In most of the process calculus literature, a congruence result is pretty much the last theorem proved. However, in at least one case (Milner's book on CCS [**1**]) the congruence result doesn't actually depend on the previous result, that the structural congruence is a strong bisimulation. They could have appeared in either order. In my attempts to prove the latter result for MIN, I have not been able to avoid a dependence on the former, so unusually the congruence result comes first, here.

In English, the result states that if two programs are equivalent, then they remain so after linking each with a third. This is one of the principal pillars supporting the main theorem of this chapter. For this purpose, linking is the only operation for which a congruence result is important. Congruence with respect to other operations is interesting from a pragmatic point of view, as a way of proving programs equivalent, and from a theoretical point of view, as a way of comparing MIN with other process calculi, but for the moment such considerations are off-topic.

We are interested in several different equivalences and pre-orders. These include strong and weak bisimilarity, weak similarity and mutual weak similarity, and coupled similarity and mutual coupled similarity (all labelled not barbed). Even in CCS we famously find that one plausibly defined equivalence (weak bisimilarity) is not a congruence with respect to all the operations (unguarded sum), and in more complicated calculi such as the π-calculus, the array of equivalences that are and are not congruences is intimidating. It is rather pleasant, therefore, to discover that all of the important equivalences and pre-orders on MIN programs are context-closed.

We do not have to check every relation separately. Instead, I will prove the soundness of a technique for constructing context-closed simulations. By applying this technique to the largest simulation (of a given kind), we will immediately obtain a context-closed version of it. Since the context-closed version can be no larger than the largest, this implies that the largest is context-closed. All the hard work goes into the proof technique, and is thereby shared. Moreover, the proof technique is independently useful.

In fact, we need to consider two cases: strong and weak simulations must be considered separately. The spirit of the two cases is the same, though.

3.6.1. Strong simulations. Recall from chapter 2 the following definitions:

DEFINITION 3.6.1. (Strong simulations)

Say a well-formed relation $\underline{S}$ is a *strong simulation* iff whenever $P \ \underline{S} \ Q$ and $Q \xrightarrow{\alpha} Q'$ we find some P' such that $P \xrightarrow{\alpha} P'$ and $P' \ \underline{S} \ Q'$.

Say $\underline{S}$ is a *strong bisimulation* iff it is symmetric and a strong simulation.[3]

Let *strong bisimilarity*, written $\sim$, be the largest strong bisimulation.

The output of the proof technique is a strong bisimulation. The input must obey the following definition:

DEFINITION 3.6.2. (Strong simulation up to linking)

Given a well-formed relation $\underline{S}$, define $\hat{\underline{S}}$ to be the smallest relation containing S and satisfying the following axiom:

$$\frac{P_1 \ \hat{\underline{S}} \ Q_1 \quad P_2 \ \hat{\underline{S}} \ Q_2}{P_1 \times P_2 \ \hat{\underline{S}} \ Q_1 \times Q_2}$$

Say $\underline{S}$ is a *strong simulation up to linking* iff whenever $P \ \underline{S} \ Q$ and $Q \xrightarrow{\alpha} Q'$ we find some P' such that $P \xrightarrow{\alpha} P'$ and $P' \ \hat{\underline{S}} \ Q'$.

Strong simulations up to linking are easier to find than ordinary strong simulations on account of the hat on the final $\hat{\underline{S}}$. In fact, any strong simulation is trivially a strong simulation up to linking. To make it even easier, we can with impunity add some more axioms to the definition of $\hat{\underline{S}}$:

$$\frac{P \ \hat{\underline{S}}^* \ Q}{P \ \hat{\underline{S}} \ Q} \qquad \frac{P \ \hat{\underline{S}} \ Q \quad \sigma : \langle \text{name} \rangle \to \langle \text{name} \rangle}{P\sigma \ \hat{\underline{S}} \ Q\sigma} \qquad \frac{P \ \sim \hat{\underline{S}} \sim \ Q}{P \ \hat{\underline{S}} \ Q}$$

In the third, $\sim$ can be replaced by any smaller strong simulation, such as $\triangleright^*$. I will leave the reader to verify that none of these additions makes any difference to the soundness of the technique, proved next.

PROPOSITION 3.6.3. *(Strong simulation up to linking)*

If $\underline{S}$ is a strong simulation up to linking, then $\hat{\underline{S}}$ is a strong simulation.

PROOF. We need to show that if $P \ \hat{\underline{S}} \ Q$ and $Q \xrightarrow{\alpha} Q'$ then $P \xrightarrow{\alpha} P'$ and $P' \ \hat{\underline{S}} \ Q'$. Proceed by induction on the derivation of the transition. If $P \ \underline{S} \ Q$ the result is immediate. Otherwise, $P = P_1 \times P_2$ and $Q = Q_1 \times Q_2$ and $P_1 \ \hat{\underline{S}} \ Q_1$ and $P_2 \ \hat{\underline{S}} \ Q_2$. Distinguish cases according to the derivation of $Q_1 \times Q_2 \xrightarrow{\alpha} Q'$. We may neglect the IN rule by proposition 3.3.9.

If $Q_1 \times Q_2 \xrightarrow{\alpha} Q'$ was derived using the PAR'_1 rule (the PAR'_2 rule is similar) then $Q_1 \xrightarrow{\alpha} Q'_1$ and $\text{fn}(\alpha) \cap \text{fn}(Q_2) = \{\}$ and $Q' = Q'_1 \times Q_2$. By the inductive hypothesis, $P_1 \xrightarrow{\alpha} P'_1$ and $P'_1 \ \hat{\underline{S}} \ Q'_1$. By the PAR'_1 rule, $P_1 \times P_2 \xrightarrow{\alpha} P'_1 \times P_2$. Finally, $P_1 \times P_2 \ \hat{\underline{S}} \ Q'_1 \times Q_2$ as required.

[3] The conventional definition is that both $\underline{S}$ and its inverse must be strong simulations. Then their union is a bisimulation by my definition.

If $Q_1 \times Q_2 \xrightarrow{\alpha} Q'$ was derived using the COM'_1 rule (the COM'_2 rule is similar) then $Q_1 \xrightarrow{\overline{\lambda}} Q'_1$ and $Q_2 \xrightarrow{\lambda} Q'_2$ and $Q'_1 \times Q'_2 \xrightarrow{\alpha} Q'$. Applying the inductive hypothesis to the first two premises, we deduce that $P_1 \xrightarrow{\overline{\lambda}} P'_1$ and $Q_1 \xrightarrow{\lambda} Q'_1$ and $P'_1 \times P'_2 \ \hat{\underline{S}} \ Q'_1 \times Q'_2$. Applying the inductive hypothesis again to the third premise, we deduce $P'_1 \times P'_2 \xrightarrow{\alpha} P'$ and $P' \ \hat{\underline{S}} \ Q'$ as required. □

COROLLARY 3.6.4. *(Strong bisimilarity is a congruence)*

$\sim$ is a strong simulation, hence a strong simulation up to linking, so $\hat{\sim}$ is a strong simulation. It is also symmetric, hence a strong bisimulation, so $\hat{\sim} \subseteq \sim$. Since $\sim \subseteq \hat{\sim}$ by definition, we have $\sim = \hat{\sim}$. In other words, $\sim$ is a congruence.

3.6.2. Weak simulations. Recall the following definitions from chapter 2:

DEFINITION 3.6.5. (Weak simulations)

Say a well-formed relation $\underline{S}$ is a *weak simulation* iff:

- Whenever $P \ \underline{S} \ Q$ and $Q \xrightarrow{\tau} Q'$, we find some P' such that $P \xrightarrow{\tau}{}^* P'$ and $P' \ \underline{S} \ Q'$.
- Whenever $P \ \underline{S} \ Q$ and $Q \xrightarrow{\alpha} Q'$ with $\alpha \neq \tau$, we find some P' such that $P \xrightarrow{\tau}{}^* \xrightarrow{\alpha} \xrightarrow{\tau}{}^* P'$ and $P' \ \underline{S} \ Q'$.

Let *weak similarity*, written $\gtrsim$, be the largest weak simulation, and let *mutual weak similarity*, written $\approx$, be its largest symmetric subrelation.

Say $\underline{S}$ is a *weak bisimulation* iff it is symmetric and a weak simulation.[4]

Let *weak bisimilarity*, written $\approx$, be the largest weak bisimulation.

Say $\underline{S}$ is *coupled* if $\underline{S} \subseteq \xrightarrow{\tau}{}^* \underline{S}^{-1}$. Say it is a *coupled simulation* if it is also a weak simulation.[5]

Let *coupled similarity*, written $\gtrsim_C$, be the largest coupled simulation, and let *mutual coupled similarity*, written $\approx_C$, be its largest symmetric subrelation.

There are many other interesting equivalences, but no more are used in this thesis. According to the definition of MIN, $P \gtrsim_C Q$ means that Q is a valid implementation of P. Weak bisimilarity is interesting mainly because it is easier to construct weak bisimulations (when they exist) than coupled simulations. Similarity is not particularly interesting, but its properties fall out of the maths for free.

DEFINITION 3.6.6. (Weak simulation up to linking)

Given a relation $\underline{S}$, define $\hat{\underline{S}}$ to be the smallest relation containing S and satisfying the following axioms:

$$\frac{P_1 \ \hat{\underline{S}} \ Q_1 \quad P_2 \ \hat{\underline{S}} \ Q_2}{P_1 \times P_2 \ \hat{\underline{S}} \ Q_1 \times Q_2} \qquad \frac{P \sim \hat{\underline{S}} \sim Q}{P \ \hat{\underline{S}} \ Q}$$

Say $\underline{S}$ is a *weak simulation up to linking* iff:

[4] Again, the conventional definition is that both $\underline{S}$ and its inverse must be weak simulations.

[5] The conventional definition of a coupled simulation is a pair of weak simulations $\underline{S_1}$ and ${\underline{S_2}}^{-1}$ such that $\underline{S_1} \subseteq \xrightarrow{\tau}{}^* \underline{S_2}$ and $\underline{S_2} \subseteq \underline{S_1} \xleftarrow{\tau}{}^*$. Then $\underline{S_1} \cup {\underline{S_2}}^{-1}$ is a coupled simulation by my definition.

- Whenever $P \ \underline{S} \ Q$ and $Q \xrightarrow{\tau} Q'$, we find a P' such that $P \xrightarrow{\tau}{}^* P'$ and $P' \ \hat{\underline{S}} \ Q'$.
- Whenever $P \ \underline{S} \ Q$ and $Q \xrightarrow{\alpha} Q'$ with $\alpha \neq \tau$, we find a P' such that $P \xrightarrow{\tau}{}^* \xrightarrow{\alpha} \xrightarrow{\tau}{}^* P'$ and $P' \ \hat{\underline{S}} \ Q'$.

We can with impunity add the following extra axioms:

$$\frac{}{P \ \hat{\underline{S}} \ P} \qquad \frac{P \ \hat{\underline{S}} \ Q \quad \sigma : \langle \text{name} \rangle \to \langle \text{name} \rangle}{P\sigma \ \hat{\underline{S}} \ Q\sigma}$$

but the transitivity axiom is unsound.

Because the COM rules deal only with strong transitions, we need a lemma to replace them.

LEMMA 3.6.7. *(Weak COM rules)*

Suppose $P \xrightarrow{\tau}{}^* \xrightarrow{\overline{\lambda}} \xrightarrow{\tau}{}^* P'$ *and* $Q \xrightarrow{\tau}{}^* \xrightarrow{\lambda} \xrightarrow{\tau}{}^* Q'$, *or the same with* λ *and* $\overline{\lambda}$ *exchanged.*

- *If* $P' \times Q' \xrightarrow{\tau}{}^* \xrightarrow{\alpha} \xrightarrow{\tau}{}^* R'$ *then* $P \times Q \xrightarrow{\tau}{}^* \xrightarrow{\alpha} \xrightarrow{\tau}{}^* \triangleright^* R'$.
- *If* $P' \times Q' \xrightarrow{\tau}{}^* R'$ *then* $P \times Q \xrightarrow{\tau}{}^* \triangleright^* R'$.

In fact, in the first case $P \times Q \xrightarrow{\tau}{}^* \xrightarrow{\alpha} \xrightarrow{\tau}{}^* R'$, and in the second case there will be at most one $\triangleright$ step, but we don't need this stronger result.

PROOF. First let me introduce some more variables for some of the intermediate states: $P \xrightarrow{\tau}{}^* P_1 \xrightarrow{\overline{\lambda}} P_2 \xrightarrow{\tau}{}^* P'$ and $Q \xrightarrow{\tau}{}^* Q_1 \xrightarrow{\lambda} Q_2 \xrightarrow{\tau}{}^* Q'$. Now by the PAR rules, $P \times Q \xrightarrow{\tau}{}^* \xrightarrow{\tau}{}^* P_1 \times Q_1$. By the $\text{COM}_1^{\triangleright}$ rule, $P_1 \times Q_1 \triangleright P_2 \times Q_2$. By the PAR rules, $P_2 \times Q_2 \xrightarrow{\tau}{}^* \xrightarrow{\tau}{}^* P' \times Q'$. In summary, $P \times Q \xrightarrow{\tau}{}^* \triangleright \xrightarrow{\tau}{}^* \xrightarrow{\alpha} R'$. The result then follows from proposition 3.5.6. □

PROPOSITION 3.6.8. *(Weak simulation up to linking)*

If $\underline{S}$ *is a weak simulation up to linking, then* $\hat{\underline{S}}$ *is a weak simulation.*

PROOF. Similar to that of proposition 3.6.3, but using lemma 3.6.7 in place of the COM rules, and using the second axiom of definition 3.6.6 to absorb the extra $\triangleright$ steps. □

THEOREM 3.6.9. *(Congruence results)*

Each of $\sim$, $\gtrsim$, $\eqsim$, $\approx$, $\gtrsim_C$ *and* $\eqsim_C$ *is context-closed.*

PROOF. $\sim$ is a congruence by corollary 3.6.4. Similar reasoning, but using the weak version of simulation up to linking, nets the others. □

3.7. Structural congruence

We now have everything necessary to fill in the second last hole in the table at the start of section 3.4, namely that $\equiv$ is a strong bisimulation. The last, namely that $\longrightarrow$ and $\xrightarrow{\tau}\equiv$ coincide, will follow quickly after that.

Since we know that $=$ is a strong bisimulation, that strong bisimulations are symmetric, that the composition of two strong bisimulations is a strong bisimulation, it suffices to find, for each of the axioms of definition 3.3.4, a strong bisimulation up to linking containing the relation generated by it alone. Each of the first two axioms in fact defines a strong bisimulation, just like in CCS. The STAR3 rule defines a strong bisimulation up to $\bowtie$, and the WIRE rule defines a strong bisimulation up to linking. The BUF rule, however, is remarkably awkward. We not only need to work up to $\bowtie$ and up to linking, but also up to the other four strong bisimulations.

We can easily handle all the input transitions. This reduces the number of cases we have to consider later, so it's worth doing straight away.

PROPOSITION 3.7.1. *(Input transitions and the structural congruence)*

Suppose $p \equiv q$, and that λ is an input transition. Then $p \xrightarrow{\lambda}\equiv r'$ iff $q \xrightarrow{\lambda}\equiv r'$.

PROOF. The existence of a matching transition follows from the last part of theorem 3.5.17. The form of the resulting program is then constrained by proposition 3.5.9. □

3.7.1. The easy rules.

PROPOSITION 3.7.2. *(STAR1 and STAR2 define strong bisimulations)*

(1) *If $p \xrightarrow{\alpha} p'$ then $p \times 1 \xrightarrow{\alpha} p'$ and* vice versa.
(2) *If $p \times q \xrightarrow{\alpha} r'$ then $r' = p' \times q'$ and $q \times p \xrightarrow{\alpha} q' \times p'$.*

PROOF. For the forwards direction of 1, use the PAR_1 rule. For the backwards direction, observe that only the PAR_1 rule is possible. For 2, perform a case analysis on the derivation of the transition (no induction is required). □

PROPOSITION 3.7.3. *(STAR3 defines a strong bisimulation up to $\bowtie$)*

If $p \times (q \times r) \xrightarrow{\alpha} s'$ then $s' = p' \times (q' \times r')$ and $(p \times q) \times r \xrightarrow{\alpha} \triangleright^* (p' \times q') \times r'$.

A symmetric result holds of $(p \times q) \times r$.

PROOF. In fact two stronger propositions hold:

(1) If $p \times (q \times r) \triangleright s'$ then $s' = p' \times (q' \times r')$ and $(p \times q) \times r \triangleright (p' \times q') \times r'$.
(2) If $p \times (q \times r) \xrightarrow{\alpha}_1 s'$ then $s' = p' \times (q' \times r')$ and $(p \times q) \times r \xrightarrow{\alpha}_1 (p' \times q') \times r'$.

Each of these can be proved by case analysis on the derivation of the transition (no induction is required). The result then follows from propositions 3.5.5 and 3.5.6. A symmetric argument yields the symmetric result. □

PROPOSITION 3.7.4. *(WIRE defines a strong bisimulation up to linking)*

If $\langle t, u\rangle \xrightarrow{\alpha} p'$ then $p' = \langle \vec{t}, \vec{u}'\rangle$ and $\langle u, t\rangle \xrightarrow{\alpha} \langle \vec{u}', \vec{t}\rangle$.

PROOF. Case analysis on the derivation of the transition. □

DEFINITION 3.7.5. ($\equiv_\times$)

Let $\equiv_\times$ be the smallest congruence satisfying the STAR1, STAR2, STAR3 and WIRE rules of definition 3.3.4 and containing $\bowtie$.

PROPOSITION 3.7.6. *($\equiv_\times$ is a strong bisimulation)*

$$\equiv_\times \xrightarrow{\alpha} \quad \subseteq \quad \xrightarrow{\alpha} \equiv_\times$$

PROOF. Immediate from propositions 3.7.2, 3.7.3 and 3.7.4. □

3.7.2. The BUF rule. One of the things that makes the BUF rule awkward is that it is quite different forwards from backwards. We must consider the two directions separately. One of the directions is rather easier than the other, and I'll do that first.

LEMMA 3.7.7. *(base case of proposition 3.7.8)*

If $fn(p) \cap fn(\langle x, y\rangle) = \{x\}$ and y is a $\langle$name$\rangle$ and $p[x \twoheadrightarrow y] \xrightarrow{\alpha} q'$ then $p \times \langle x, y\rangle \xrightarrow{\alpha} \equiv q'$.

PROOF. We may assume by alpha-conversion that $x \notin \mathrm{fn}(\alpha)$, but we must distinguish two cases according to whether $y \in \mathrm{fn}(\alpha)$. If $y \notin \mathrm{fn}(\alpha)$, then by proposition 3.4.4 we have $p \xrightarrow{\alpha} p'$ and $q' = p'[x \mapsto y]$, so by the PAR_1 rule $p \times \langle x, y\rangle \xrightarrow{\alpha} p' \times \langle x, y\rangle \equiv p'[x \mapsto y]$ as required. If $y \in \mathrm{fn}(\alpha)$ then either α is an input or (using proposition 3.4.2) $\alpha = \overline{y}c(\vec{z})$ for some $\vec{z} \notin \mathrm{fn}(p[x \twoheadrightarrow y])$ (so $\vec{z} \notin \mathrm{fn}(p)$ too). Therefore, choosing fresh $\vec{w}$, $p \xrightarrow{\overline{x}c(\vec{w})} p'$ and $\langle x, y\rangle \xrightarrow{xc(\vec{w})} \xrightarrow{\overline{y}c(\vec{z})} \langle \vec{w}, \vec{z}\rangle$. Apply the PAR_2 and COM_1 rules. □

PROPOSITION 3.7.8. *(BUF rule backwards)*

If $fn(p) \cap fn(\langle x, t\rangle) = \{x\}$ and $p[x \twoheadrightarrow t] \xrightarrow{\alpha} q'$ then $p \times \langle x, t\rangle \xrightarrow{\alpha} \equiv q'$.

PROOF. Induction on the structure of t. If t is a $\langle$name$\rangle$, use lemma 3.7.7. Otherwise $t = c(\vec{u})$. Choosing fresh $\vec{y}$ and putting $n = |\vec{y}| = |\vec{u}|$, we have

$$\begin{array}{rcl} p[x \twoheadrightarrow c(\vec{u})] & = & p[x \twoheadrightarrow c(\vec{y})][y_1 \mapsto u_1] \ldots [y_n \mapsto u_n] \\ p \times \langle x, c(\vec{u})\rangle & \triangleright & p[x \twoheadrightarrow c(\vec{y})] \times \langle \vec{y}, \vec{u}\rangle \\ & \equiv_\times & (\ldots (p[x \twoheadrightarrow c(\vec{y})] \times \langle y_n, u_n\rangle) \ldots) \times \langle y_1, u_1\rangle \end{array}$$

Applying the inductive hypothesis to

$$p[x \twoheadrightarrow c(\vec{y})][y_1 \mapsto u_1] \ldots [y_{\overline{n}} \twoheadrightarrow u_n] \quad \xrightarrow{\alpha} \quad q'$$

we obtain

$$\begin{array}{rcl} & & (p[x \twoheadrightarrow c(\vec{y})] \times \langle y_n, u_n\rangle)[y_1 \mapsto u_1] \ldots [y_{n-1} \mapsto u_{n-1}] \\ & = & p[x \twoheadrightarrow c(\vec{y})][y_1 \mapsto u_1] \ldots [y_{n-1} \mapsto u_{n-1}] \times \langle y_n, u_n\rangle \\ & \xrightarrow{\alpha} \equiv & q' \end{array}$$

Repeating this step a total of n times gives the desired result. □

Now for the forwards direction.

LEMMA 3.7.9. *(Base case of BUF rule forwards)*

If $fn(p) \cap fn(\langle c(\vec{u}), t\rangle) = fn(c(\vec{u}))$ *and* $p \times \langle c(\vec{u}), t\rangle \xrightarrow{\alpha} q'$ *and* α *is not an input action then one of the following holds:*

- $p \xrightarrow{\alpha} p'$ *and* $fn(\alpha) \cap fn(\langle c(\vec{u}), t\rangle) = \{\}$ *and* $p' \times \langle c(\vec{u}), t\rangle \bowtie q'$.
- $t = w$, *a* $\langle name\rangle$, *and* $\alpha = \overline{w}c(\vec{z})$ *and* $p \times \langle \vec{u}, \vec{z}\rangle \bowtie q'$.

PROOF. Induction on the derivation of the transition. If it was derived using PAR_1 then the first conclusion holds. If it was derived using PAR_2 then the second holds. COM_2 is not possible. If it was derived using COM_1 then $p \xrightarrow{\overline{\lambda}} p_1$ and $\langle c(\vec{u}), t\rangle \xrightarrow{\lambda} \langle c(\vec{u}'), t\rangle$ and $p_1 \times \langle c(\vec{u}'), t\rangle \xrightarrow{\alpha} q'$. Apply the inductive hypothesis to the latter. If it returns the second conclusion, then return the second conclusion. If it returns the first, then $p_1 \xrightarrow{\alpha} p_1'$ and $p_1' \times \langle c(\vec{u}'), t\rangle \bowtie q'$. Use theorem 3.5.17 to deduce $p \xrightarrow{\alpha} p' \xrightarrow{\overline{\lambda}} \bowtie p_1'$, then $\text{COM}_1^{\triangleright}$ to deduce $p' \times \langle c(\vec{u}), t\rangle \triangleright \bowtie p_1' \times \langle c(\vec{u}'), t\rangle$ as required. □

COROLLARY. *If* $fn(p) \cap fn(\langle x, t\rangle) = \{x\}$ *and* $p \xrightarrow{\overline{x}c(\vec{y})} p'$ *and* $p' \times \langle c(\vec{y}), t\rangle \xrightarrow{\alpha} q'$ *and* α *is not an input action, then one of the following holds:*

- $p \xrightarrow{\alpha} p''$ *and* $fn(\alpha) \cap fn(\langle x, t\rangle) = \{\}$ *and* $p'' \times \langle x, t\rangle \bowtie q'$ *(again, this requires theorem 3.5.17).*
- $t = w$, *a* $\langle name\rangle$, *and* $\alpha = \overline{w}c(\vec{z})$ *and* $p' \times \langle \vec{y}, \vec{z}\rangle \bowtie q'$.

PROPOSITION 3.7.10. *(BUF rule forwards)*

If $fn(p) \cap fn(\langle x, t\rangle) = \{\}$ *and* $p \times \langle x, t\rangle \xrightarrow{\alpha} q'$ *then* $p[x \mapsto t] \xrightarrow{\alpha} \equiv q'$.

PROOF. Proposition 3.7.1 leaves the cases in which α is not an input action. Reason by induction on the structure of t. Distinguish cases according to the derivation of $p \times \langle x, t\rangle \xrightarrow{\alpha} q'$. If it was derived using PAR_1, then the result is immediate. PAR_2 is not possible. If it was derived using COM_1 then use lemma 3.7.9. This leaves the COM_2 case, in which $t = c(\vec{u})$ and $p \xrightarrow{xc(\vec{y})} p'$ (from which $p[x \mapsto c(\vec{y})] \bowtie p'$) and $p' \times \langle \vec{y}, \vec{u}\rangle \xrightarrow{\alpha} q'$. Also, $p[x \mapsto t] = p[x \mapsto c(\vec{y})][\vec{y} \mapsto \vec{u}]$. This puts us on territory familiar from proposition 3.7.8, and we can finish the proof in the same way. □

3.7.3. Structural congruence is a strong bisimulation.

THEOREM 3.7.11. *(Structural congruence is a strong bisimulation)*

$$\equiv \xrightarrow{\alpha} \quad \subseteq \quad \xrightarrow{\alpha} \equiv$$

PROOF. Since $\equiv$ is the smallest congruence containing $\equiv_\times$ and satisfying the BUF rule of definition 3.3.4, it suffices to show that the smallest equivalence containing $\equiv_\times$ and satisfying the BUF rule is a strong bisimulation up to linking. This follows from propositions 3.7.6, 3.7.8 and 3.7.10. □

3.7.4. Reactions and silent transitions. We can now prove the assertion made way back in section 3.1.2, namely that the reaction relation agrees with the silent transition relation up to structural congruence. This will fill in the final hole in the table at the start of section 3.4. It is also an important pillar supporting the main theorem of this chapter.

THEOREM 3.7.12. *(Reactions and silent transitions)*

(1) *If* $P \longrightarrow P'$ *then* $P \xrightarrow{\tau} \equiv P'$.
(2) *If* $P \xrightarrow{\tau} P'$ *then* $P \longrightarrow P'$.

PROOF. Prove 1 by induction on the derivation of $P \longrightarrow P'$. The REACT and PAR rules of definition 3.3.5 map respectively onto the REACT and PAR_1 rules of definition 3.3.7, and (at last!) we can handle the STRUCT rule using theorem 3.7.11.

For 2, use propositions 3.5.5 and 3.5.7 to deduce $P \equiv \xrightarrow{\tau}_1 P'$, and then show by induction on its derivation that $\xrightarrow{\tau}_1 \subseteq \longrightarrow$. Finally, use the STRUCT rule. □

We should also check that the definition of the labelled transition relation given in chapter 1, and repeated in section 3.1.2, agrees with definition 3.3.7. After all, they do not look the same at all. The former implicitly uses the structural congruence, while the latter is inductive on a construction of the graph. All the hard work is contained in theorem 3.7.11; we just have to finish it off.

PROPOSITION 3.7.13. *(Alternative definition of the labelled transition relation)*

(1) $P \xrightarrow{\tau} \equiv P'$ iff $P \longrightarrow P'$.
(2) $p \xrightarrow{x\,c(\vec{y})} \equiv p'$ iff $x \in \text{fn}(p)$ and $\vec{y} \in \text{fn}(p')$ and $p \times \langle x, c(\vec{y})\rangle \equiv p'$.
(3) $p \xrightarrow{\overline{x}c(\vec{y})} \equiv p'$ iff $x \in \text{fn}(p)$ and $\vec{y} \in \text{fn}(p')$ and $p \equiv \langle x, c(\vec{y})\rangle \times p'$.

PROOF. 1 is simply a restatement of theorem 3.7.12. The 'only if' parts of 2 and 3 are simply a restatement of propositions 3.4.2 and 3.5.9. For the 'if' part of 2, derive $p \xrightarrow{x\,c(\vec{y})} \equiv p \times \langle x, c(\vec{y})\rangle$ using the IN rule. For the 'if' part of 3, derive $\langle x, c(\vec{y})\rangle \times p' \xrightarrow{\overline{x}c(\vec{y})} \equiv p'$ using OUT_1 and PAR_1, and then apply theorem 3.7.11. □

We will never need this proposition; we can always derive any transitions we want using definition 3.3.7. It is nice to know that it is true, though.

3.8. The main theorem

We now know a great deal about the labelled equivalences. We have two equivalent definitions of the labelled transition relation, and we can switch between them at will. One of those definitions provides an easy way of enumerating the transitions of a program exhaustively. The other provides a simple characterisation of the effect of a transition on a program. We know that all the interesting labelled equivalences are congruences. We have good ways of proving that programs are equivalent (construct a simulation) or are not equivalent (find a context in which they do not have the same transitions). The former task is undecidable in general, but we are beginning to see some powerful tools for achieving it in practice (simulation 'up to' various things).

Apart from studying my provisional type scheme in chapter 4, this is as far as I am going to push the side of the theory that concerns the labelled transition system. We are close to the frontier of what is known for any calculus, and we have not taken very long to get there. What lies beyond is to some extent calculus-independent. This includes proof techniques such as simulation up to expansion, and the fascinating study of 'up to' techniques in general [**2**, section 2.4]. There are also techniques that will require some foundational results before they can be applied to MIN, but which will then work just as they do for other calculi. These include the expansion law[6] and the unique solution theorem (same reference). I feel that there is a pressing need to automate the whole lot in the form of a program that can be used by non-specialists [ref: Concurrency Workbench].

The remaining job for this chapter is to establish that knowing about labelled equivalences is of any use. We have to show that labelled coupled similarity coincides with barbed coupled similarity, the relation used in the definition of MIN.

3.8.1. Barbed coupled similarity. Recall the definition of barbed coupled similarity (this version is untyped):

DEFINITION 3.8.1. (Barbed coupled similarity)

Let $\langle\text{constructor}\rangle'$ be larger than $\langle\text{constructor}\rangle$ by one element: 'test'.[7] Keeping $\langle\text{destructor}\rangle$ and $\langle\text{name}\rangle$ unchanged, derive $\langle\text{tree}\rangle'$, $\langle\text{atom}\rangle'$, $\langle\text{process}\rangle'$, $\langle\text{rewrite}\rangle'$, $\langle\text{environment}\rangle'$ and $\langle\text{program}\rangle'$ by substituting $\langle\text{constructor}\rangle'$ for $\langle\text{constructor}\rangle$ everywhere in definition 3.3.1. Extend fn( ), $\equiv$, $\longrightarrow$ and the labelled transition relation accordingly. Do not extend the labelled equivalences, such as $\sim$ and $\gtrsim_C$.

Say a well-formed relation $\underline{S}$ on $\langle\text{program}\rangle'$ is *sound* unless it relates two programs let E in p and let F in q such that p contains 'test' but q does not, or *vice versa*.

Say $\underline{S}$ is *reduction-closed* if $\underline{S} \longrightarrow^* \ \subseteq \ \longrightarrow^* \underline{S}$.

Say $\underline{S}$ is *context-closed* if $P \times R \ \underline{S} \ Q \times R$ and $R \times P \ \underline{S} \ R \times Q$ whenever $P \ \underline{S} \ Q$.

Say $\underline{S}$ is *coupled* iff $\underline{S} \ \subseteq \ \longrightarrow^* \underline{S}^{-1}$.

Say $\underline{S}$ is a *barbed coupled simulation* iff it is sound, reduction-closed, context-closed and coupled. Let *barbed coupled similarity*, written $\geq_C$, be the restriction to $\langle\text{program}\rangle$ of the largest barbed coupled simulation, and let *mutual barbed coupled similarity*, written $=_C$, be its largest symmetric subrelation.

It is easy to see that each of the four properties is true of $\equiv$, and is closed under unions and under the operation that maps $\underline{S}$ and $\underline{T}$ to $\underline{S}\,\underline{T} \cup \underline{T}\,\underline{S}$. The largest barbed coupled simulation therefore exists, contains $\equiv$, and is reflexive and transitive.

To show that $\geq_C$ and $\gtrsim_C$ coincide, we must find a (labelled) coupled simulation containing $\geq_C$ and a barbed coupled simulation containing $\gtrsim_C$. The proofs are not hard, but they use nearly every trick we have met, jumping between reactions and

[6] There are two different uses of the word 'expansion' in this paragraph, and more widely in the process calculus literature.

[7] I considered calling the new constructor 'twdfldsapstiiti' for 'thing with dazzling flashing lights, deafening siren and putrid stench, that is impossible to ignore', as this more accurately describes the way in which it is treated by the maths.

silent transitions, treating the structural congruence as equality, and reasoning up to linking.

3.8.2. Forwards. In the forwards direction, we get lucky: $\geq_C$ itself is a labelled coupled simulation.

PROPOSITION 3.8.2. *($\geq_C$ is a labelled coupled simulation)*

If $\alpha \neq \tau$ then

$$\begin{array}{rcl} \geq_C \xrightarrow{\tau} & \subseteq & \xrightarrow{\tau}{}^* \geq_C \\ \geq_C \xrightarrow{\alpha} & \subseteq & \xrightarrow{\tau}{}^* \xrightarrow{\alpha} \xrightarrow{\tau}{}^* \geq_C \\ \geq_C & \subseteq & \xrightarrow{\tau}{}^* \leq_C \end{array}$$

PROOF. The first part simply states that $\geq_C$ is reduction-closed, and the last that it is coupled. It is the second part that is interesting. Suppose $P \geq_C Q$ and $Q \xrightarrow{\alpha} Q'$. If α is an input action, the result follows from the last part of theorem 3.5.17 (and $\mathrm{fn}(P) = \mathrm{fn}(Q)$), proposition 3.5.9 and context-closure. We may therefore suppose it is an output action $\overline{x}c(\vec{y})$, with $c \neq \mathrm{test}$. By proposition 3.4.2 $x \in \mathrm{fn}(Q)$ and $\vec{y} \neq \mathrm{fn}(Q)$.

Construct a test harness, a $\langle \mathrm{program} \rangle' R = \mathrm{let}\ E\ \mathrm{in}\ d(\vec{y}, x)$ where the $\langle \mathrm{environment} \rangle$ E is defined as follows:

$$\begin{array}{rcl} E & = & \{d(\vec{y}, c(\vec{z})) \longrightarrow d(\vec{y}, \mathrm{test}(\vec{z})), \\ & & \quad d(\vec{y}, \mathrm{test}(\vec{z})) \longrightarrow \langle \vec{y}, \vec{z} \rangle\} \end{array}$$

Let $\underline{S}$ be the largest barbed coupled simulation, so $P \ \underline{S}\ Q$, and by context-closure $P \times R \ \underline{S}\ Q \times R$ (note that $P \times R \not\geq_C Q \times R$ because $\geq_C$ is only a relation on $\langle \mathrm{program} \rangle$).

Since $Q \times R \longrightarrow Q_1 = Q'[\vec{y} \mapsto \vec{z}] \times \mathrm{let}\ E\ \mathrm{in}\ d(\vec{y}, \mathrm{test}(\vec{z}))$, reduction-closure dictates that $P \times R \longrightarrow^* P_1$ and $P_1 \ \underline{S}\ Q_1$ for some P_1. By theorem 3.7.12, $P \times R \xrightarrow{\tau}{}^* \equiv P_1$. By soundness, P_1 contains 'test'. By induction on the derivation of $P \times R \xrightarrow{\tau}{}^* \equiv P_1$, we deduce that $P \xrightarrow{\tau}{}^* \xrightarrow{\overline{x}c(\vec{y})} \xrightarrow{\tau}{}^* P'$ and $P_1 \equiv P'[\vec{y} \mapsto \vec{z}] \times \mathrm{let}\ E\ \mathrm{in}\ d(\vec{y}, \mathrm{test}(\vec{z}))$.

Similarly, since $Q_1 \longrightarrow Q_2 = Q'[\vec{y} \mapsto \vec{z}] \times \mathrm{let}\ E\ \mathrm{in}\ \langle \vec{y}, \vec{z} \rangle$, we deduce that $P' \xrightarrow{\tau}{}^* P''$ and $P''[\vec{y} \mapsto \vec{z}] \times \mathrm{let}\ E\ \mathrm{in}\ \langle \vec{y}, \vec{z} \rangle \equiv P_2 \ \underline{S}\ Q_2$ for some P'' and P_2.

Note that $Q_2 \equiv Q' \times \mathrm{let}\ E\ \mathrm{in}\ 1$ and $P_2 \equiv P'' \times \mathrm{let}\ E\ \mathrm{in}\ 1$. Now clearly the reaction relation is invariant under injective renaming of destructors, and also under addition to or removal from the environment of rewrites that can never be used. We can easily construct barbed coupled simulations containing these transformations. Since such transformations (and $\equiv$) are all that distinguish Q' from $Q' \times \mathrm{let}\ E\ \mathrm{in}\ 1$ (and similarly for P''), we conclude that $P'' \geq_C Q'$ as required. □

3.8.3. Backwards. $\gtrsim_C$ is not a barbed coupled simulation, because it is a relation only on $\langle\text{program}\rangle$, and is therefore not context-closed with respect to $\langle\text{program}\rangle'$.

The obvious thing to do is to define labelled similarities on $\langle\text{program}\rangle'$. However, the resulting relations are not sound. For example, $\langle \text{test}(), \text{test}() \rangle$ has no labelled transitions at all, and is therefore labelled strong bisimilar to 1. Indeed, if this approach worked, I would not have bothered to define $\langle\text{program}\rangle'$ at all.

The correct thing to do is to close up $\gtrsim_C$ under linking. This does not give the largest barbed coupled simulation (consider, for example, the barbed bisimulation that relates any two programs that contain 'test' permanently), but all that is required is that it gives some barbed coupled simulation.

PROPOSITION 3.8.3. *($\gtrsim_C$ generates a barbed coupled simulation)*

Let $\underline{S}$ be the smallest relation containing $\gtrsim_C$ and $\equiv$ and satisfying $P_1 \times P_2 \;\; \underline{S} \;\; Q_1 \times Q_2$ whenever $P_1 \;\; \underline{S} \;\; Q_1$ and $P_2 \;\; \underline{S} \;\; Q_2$. Then $\underline{S}$ is context-closed by definition. It is also sound, reduction-closed and coupled.

PROOF. Show $\underline{S}$ is sound by induction on its derivation. For the base cases, observe that $\gtrsim_C$ only relates $\langle\text{program}\rangle$s (which do not contain 'test'), and that $\equiv$ is sound. The inductive case is no less obvious.

To show that $\underline{S}$ is reduction-closed, show that it is a weak simulation, and then apply theorem 3.7.12. This is easy to do: $\equiv$ is a strong bisimulation by theorem 3.7.11, and $\gtrsim_C$ is a weak simulation by definition, so their union is a weak simulation, and hence a weak simulation up to linking.

Show that $\underline{S}$ is coupled by induction on its derivation. For the base cases, $\equiv$ is not only coupled but symmetric, and $\gtrsim_C$ is coupled by definition. For the inductive step, suppose $P_1 \;\; \longrightarrow^* \underline{S}^{-1} \;\; Q_1$ and $P_2 \;\; \longrightarrow^* \underline{S}^{-1} \;\; Q_2$. Then $P_1 \times P_2 \;\; \longrightarrow^* \underline{S}^{-1} \;\; Q_1 \times Q_2$ as required. □

3.8.4. The main theorem.

THEOREM 3.8.4. *(Barbed and labelled coupled similarity coincide)*

$$\gtrsim_C \;\; = \;\; \geq_C$$

PROOF. Propositions 3.8.2 and 3.8.3. □

Bibliography

[1] Milner, R., A Calculus of Communicating Systems.
[2] Sangiorgi, D. and Walker D., The π-calculus.