The Theory of Interacting Deductions and its Application to Operational Semantics

Andrew Wilson

Doctor of Philosophy
University of Edinburgh
1996

In memory of my father who died after a courageous struggle with a long illness, and to my mother who struggled with him. You taught me faith, hope, love and perseverence, thank you.

Not a single truth has ever been discovered without people first talking utter rot a hundred times or perhaps a hundred thousand times.
— Razumikhin, in Dostoyevsky's "Crime and Punishment".

Abstract

This thesis concerns the problem of complexity in operational semantics definitions. The appeal of modern operational semantics is the simplicity of their metatheories, which can be regarded as theories of deduction about certain shapes of operational judgments. However, when applied to real programming languages they produce bulky definitions that are cumbersome to reason about. The theory of interacting deductions is a richer metatheory which simplifies operational judgments and admits new proof techniques.

An interacting deduction is a pair (F, I), where F is a forest of inference trees and I is a set of interaction links (a symmetric set of pairs of formula occurrences of F), which has been built from interacting inference rules (sequences of standard inference rules, or *rule atoms*). This setting allows one to decompose operational judgments. For instance, for a simple imperative language, one rule atom might concern a program transition, and another a store transition. Program judgments only interact with store judgments when necessary: so stores do not have to be propagated by every inference rule. A deduction in such a semantics would have two inference trees: one for programs and one for stores.

This introduces a natural notion of modularity in proofs about semantics. The *proof fragmentation theorem* shows that one need only consider the rule atoms relevant to the property being proved. To illustrate, I give the semantics for a simple process calculus, compare it with standard semantics and prove three simple properties: nondivergence, store correctness and an equivalence between the two semantics.

Typically evaluation semantics provide simpler definitions and proofs than transition semantics. However, it turns out that evaluation semantics cannot be easily expressed using interacting deductions: they require a notion of sequentiality. The *sequential* deductions contain this extra structure. I compare the utility of evaluation and transition semantics in the interacting case by proving a simple translation correctness example. This proof in turn depends on proof-theoretic concerns which can be abstracted using *dangling interactions*. This gives rise to the techniques of breaking and assembling interaction links. Again I get the proof fragmentation theorem, and also the *proof assembly* theorem, which allow respectively both the isolation and composition of modules in proofs about semantics. For illustration, I prove a simple type-checking result (in evaluation semantics) and another nondivergence result (in transition semantics).

I apply these results to a bigger language, CSP, to show how the results scale up. Introducing a special *scoping* side-condition permits a number of linguistic extensions including nested parallelism, mutual exclusion, dynamic process creation and recursive procedures. Then, as an experiment I apply the theory of interacting deductions to present and prove sound a compositional proof system for the partial correctness of CSP programs.

Finally, I show that a deduction corresponds to CCS-like process evaluation, justifying philosophically my use of the theory to give operational semantics. A simple corollary is the undecidability of interacting-deducibility. Practically, the result also indicates how one can build prototype interpreters for definitions.

Acknowledgments

First and foremost, a big thank you to my supervisor Kevin Mitchell for his painstaking help and advice. Thanks are also due to Gordon Plotkin, Bob Tennant, Robin Milner and Colin Stirling who have variously belonged to my supervision committee, and given helpful suggestions.

Thanks are also due to my officemates Pietro Cenciarelli and Alex Simpson, and to my flatmate Julyan Elbro for their friendship and their comments on early drafts. Thanks also to Saif, Pietro and Alex for making my office days very pleasant, and similarly to Dave Aspinall and all my friends in the LFCS.

Thanks to the Science and Engineering Research Council (now EPSRC) for funding the first three years of my thesis. Thanks also to my mum and the Department of Computer Science which allowed me to scrape a living teaching after the three years had expired.

Finally, I could not have survived a project of this magnitude without the support and care of my family and my friends, of which I have too many to acknowledge by name individually. You know who you are. But especial thanks to my closest friends, Andrew, Douglas and David, and to my flatmates Dave, Julyan, Doug and Dan who put up with the bad times. Thanks also to the members of Mayfield-Salisbury church, and especially the three families who over the years all but adopted me: the Jamiesons, the Simpsons and the Vandersteens. Thanks also to Lt. Heather Chambers and the boys of the 3rd Edinburgh Company, The Boys' Brigade who boosted my flagging spirits during those long depressing months. Finally, thanks to my TEAM co-leader Helen Hopwood who made running TEAM much less stressful than it should have been.

Declaration

This thesis was composed by myself. The work reported herein, unless otherwise stated, is my own.

Andrew Wilson

Table of Contents

Glossary of notations

SETS (see, e.g., [End77])

X, Y: arbitrary sets
$\cap$: intersection
$\cup$: union
$\subseteq$: subset
$\setminus$: set difference
$\in$: membership
$\emptyset$: empty set
$\wp$: powerset

MULTISETS (see figure 2–2 on page 35)

RELATIONS

R: arbitrary binary relations, or labelled binary relations (p. 72)
R^*: transitive closure
$R^\flat$: delabeling ("flattening") a labeled relation
R°: symmetric closure of labeled relation

NUMBERS

m, n: arbitrary naturals
i, j, k: arbitrary indices
$\mathbb{N}$: the set of natural numbers
$\mathbb{Z}$: the integers

SEQUENCES (page 30)

X^ω: set of countable, possibly empty sequences of elements from X
s: arbitrary sequence
ε: empty sequence
$s \cdot s'$: sequence concatenation
$|s|$: the length of s
dom s: the indices of s
$s(i)$: the ith element of s

FUNCTIONS

f, g: arbitrary functions
0: the empty function $0 : \emptyset \to \emptyset$
$f : X \to Y$: total function
$f : X \rightharpoonup Y$: partial function
$f : X \leftrightarrow Y$: bijection
dom f: the domain of f
im f: the image of f
$f \oplus g$: function overwriting: $(f \oplus g)x = g(x)$ if $x \in \mathrm{dom}\ g$, otherwise $f(x)$
$f \upharpoonright X$: restrict domain of f to dom $f \cap X$

LOGIC

ϕ, χ, ψ: arbitrary formulae
$\supset$: implication
$\wedge$: conjunction
$\vee$: disjunction
$\neg$: negation
$\forall x.\phi$: universal quantification
$\exists x.\phi$: existential quantification

LANGUAGE

$\mathcal{L}$: arbitrary language
$A, B, C, \ldots$: arbitrary formulae and formula occurrences
A^i: formula occurrence with position explicit
θA: instantiation of A by substitution θ
$FV(A)$: the free variables of A
$A \equiv B$: occurrences A and B match (i.e., they are occurrences of the same formula).
$+A, -A$: formula perceptions (p. 70)
$X^\pm$: set of perceptions of formulae in X

INFERENCE RULES

a: arbitrary rule atom (p. 15)
θa: instance of atom with respect to θ
r: arbitrary rule
θr: instance of rule with respect to θ
$\mathcal{R}$: arbitrary rule set

DEDUCTION

$\mathcal{T}$: arbitrary deductive system
T: formula tree
F: formula forest
$\lesssim_F$: the preorder of F (p. 18)
$O(F)$: the set of occurrences in forest F
(F, I): I-structure (p. 19)
$(F, I, \sqsubset)$: QI-structure (p. 49)
$(F, I, \sqsubset, D)$: DQI-structure (p. 72)
Σ: arbitrary structures or deductions
Π: arbitrary deductions
$O(\Sigma)$: the set of occurrences in Σ
$D(\Sigma)$: the set of dangling interactions in DQI-structure Σ
$N(\Sigma)$: the set of neighbourhoods of Σ (p. 20)
$N_\Sigma(A)$: the neighbourhood of occurrence A in Σ
$\mathbf{I}(\mathcal{T})$: the set of deductions of I-system $\mathcal{T}$
$\mathbf{QI}(\mathcal{T})$: the set of deductions of QI-system $\mathcal{T}$
$\mathbf{DQI}(\mathcal{T})$: the set of deductions of DQI-system $\mathcal{T}$
$\lesssim_\Sigma$: the preorder of structure Σ
$<_\Sigma = (\lesssim_\Sigma) \setminus (\gtrsim_\Sigma)$
$\sim_\Sigma = (\lesssim_\Sigma) \cap (\gtrsim_\Sigma)$
0: empty deduction
$\perp$: deduction of no information (p. 61)
$\Sigma \vdash A_1, \ldots, A_n$: Σ concludes formula occurrences $A_1, \ldots, A_n$
$\mathcal{T} \vdash A_1, \ldots, A_n$: some $\mathcal{T}$-deduction concludes $A_1, \ldots, A_n$
$\mathcal{T} \Vdash A_1, \ldots, A_n$: some proper $\mathcal{T}$-deduction (DQI) concludes $A_1, \ldots, A_n$ (p. 76)
$\bigotimes_f X$: assembly of a set of structures (p. 74)
$\Sigma \otimes_f \Pi$: binary assembly of Σ and Π (p. 74)
$\Sigma \preceq \Pi$: Π simulates Σ (p. 82)
$\Sigma \simeq \Pi$: $\Sigma \preceq \Pi$ and $\Pi \preceq \Sigma$
$\mathbb{B}(\Sigma)$: result of breaking every interaction link in Σ
$\mathbb{B}(\mathcal{T})$: result of breaking every interaction link in $\mathcal{T}$
$\mathbb{F}_\mathcal{T}(X)$: set of $\mathcal{T}$-fragments of structures in X

PROCESSES (page 22)

p, q, r : arbitrary processes
Nam: set of names
$a, b, c, \ldots$: arbitrary names
$\overline{Nam}$: set of conames of Nam
$\bar{a}, \bar{b}, \bar{c}, \ldots$: arbitrary conames
Lab: labels
l: arbitrary label
τ: silent action
Act: action set
α: arbitrary action
0: the null process
$|$: parallel composition
$\alpha.p$: action prefix
$;$: sequential composition
$p \setminus X$: restriction
$\approx$: weak bisimulation
Ω: set of terminated processes

STORES

σ: arbitrary store
$\sigma(x)$: value of x in σ
$\sigma[v/x]$: updating x to have value v in store σ

Convention

In this thesis, I use the following convention to label lemmas, propositions and theorems. A proposition is either an interesting but not important result about interacting deductions, or an example proof which highlights some point. Propositions are numbered $m.n$ where m is the chapter or appendix number, and n indicates that this is the nth proposition in that chapter or appendix.

A theorem is an interesting result about the theory of interacting deductions, and is numbered using upper case roman numerals.

A lemma is a separate result used in the proof of a proposition or theorem. They are labeled $m(n)$ where m is the label of either the proposition or theorem it helps prove, and n is a lower case roman numeral.

Finally, a corollary is an interesting, but lesser result that is a straightforward consequence of either a proposition or theorem. Corollaries are labeled mn where m is the label of either the theorem or proposition from which the corollary follows, and n is a lower case letter.

Chapter 1

Defining Programming Languages

To learn a foreign language, one has to learn the structure of its sentences, what they mean, and how they can be used to express one's thoughts. Similarly, to learn a programming language, one has to learn the structure of programs, what they mean, and how one can use them to express one's algorithms. These three aspects of language are called *syntax*, *semantics* and *pragmatics*.

Our interest lies in the formal semantics of programming languages. A formal semantics gives a complete and unambiguous account of the defined language. Ideally, this has a number of uses. First, it aids the language developer. Writing a formal semantics forces the designer to fill every hole and resolve every ambiguity that may have existed in the original conception. It may also improve the design, highlighting its poorly thought out, or overly complex areas.

Second, a formal semantics can help the implementer. A formal semantics is an unambiguous compiler specification. The implementer should be able to extract all the information he needs to implement the language before he starts coding. Moreover, it may also suggest implementation strategies. Along similar lines, a formal semantics can also be used as a basis for developing automatic compiler generators or prototype interpreters, static analyzers and program analysis tools such as debuggers and execution profilers. It also can be useful for developing learning tools such as program animators, tutorials and reference manuals.

Third, a formal semantics can help the programmer. Apart from being a complete and concise reference, and the basis for programming tools, it can be used to develop logics that help him reason about his software, in particular showing that it meets a specification.

Fourth, a formal semantics can help the businessman. A formal semantics can serve as a standard, which can ensure portability of software over platforms. It can also be used to write unambiguous implementation contracts that will guarantee such portability.

Ideally. In reality formal semantics (of real languages) are subtle pieces of mathematics which require substantial effort to write, read and use. They are prone to mistakes and ambiguities (e.g., [Kah93]). Even so, the attempt to give a semantics is worthwhile: even a near-perfect semantics can be useful at least informally, and anyway using a semantics will highlight flaws which can then be remedied, hopefully without too much renovation.

This raises the question of correctness: how do we know a semantics defines our language? If it is complete and consistent, it certainly defines something, but is it the language we are concerned about? We cannot resolve this problem conclusively: there is always a gap between ideas and their formal expression. The best we can do is to give a series of different semantics and show that they agree. This is the proposal of consistent and complementary semantics argued in [HL74]: the more different (but mutually consistent) semantics we give a language, the more we can be sure that the language reflects no "irregularity" of a description method. [HK81] gives complementary semantics to the "Real-Time" language TOMAL.

There are many different ways to give semantics (see [deB69,HL74,MLB76, Sve86,Gun91] for a survey. See also [Weg76] for a historical survey of programming languages), but they tend to fall into three categories: *axiomatic*, *denotational* and *operational*, characterized by their different emphases and hence uses. Axiomatic semantics (e.g., Hoare Logics [Apt81] and Predicate Transformers [DS90]) describe programs in terms of the properties (expressible in some assertion language) which they satisfy. [HW73] gives an example axiomatic semantics for most of PASCAL. Their main use is to facilitate reasoning about programs. To justify such reasoning, one has to prove the axiomatic semantics sound with respect to some model of the programming language.

Denotational semantics (e.g., [Sch88]) provide such models: they map programs to *denotations*, which are abstract mathematical representations of program behaviour. [Mos74] gives a denotational semantics for ALGOL 60. Every feature of the language is defined without regard to implementation details. As such they are very useful for linguistic analysis, and for providing correctness criteria for other semantics of languages.

Operational semantics (e.g., [Hen90]) describe how an ideal, abstract machine would simulate the behaviour of a program. [HMT90] gives an operational semantics for STANDARD ML. Operational semantics are useful to the language implementer. Since at least the modern kind tend to be mathematically light, and also programmers tend to understand languages in terms of how a "notional machine" would execute programs [DBOM81,Ber91], these semantics tend to be relatively accessible.

The point is that no one semantic formalism does everything well. We need a variety of different formalisms, first, to be sure that the language we have described is the one we intended to describe. Second, to overcome inherent biases of formalisms to particular kinds of language. For example, traditional denotational semantics uses λ-notation. Both Mosses [Mos91, §18.1], or [Mos90, §6.1] and Abramsky *et al.* [AGN94] argue that this biases denotational semantics towards sequential, functional languages. It is hard to code up and hard to read even conventional imperative or concurrent languages. The third reason is for insurance: we cannot guarantee that every new linguistic feature will be expressible in the current framework of choice.

Therefore there is always a need to develop and evaluate new semantic description methods. This thesis concerns a novel approach to specifying operational semantics. Operational semantics description methods tend to fall into two main

	Description Method	Defined languages
Indirect	Abstract machines	FACILE [GMP89]
	SMoLCS [AR87a,AR87b,SMo]	ADA (Draft) [ABNB+86]
	EVOLVING ALGEBRAS [Gur91,EA]	C, C++, OCCAM, VHDL, MODULA-2, PROLOG, PARLOG, GÖDEL
	ACTION SEMANTICS [Mos91,AS]	STANDARD ML, CML, AMBER, OCCAM, PASCAL, JOYCE
Direct	VDL [Weg72]	PL/I [LW69,AI75]
	W-GRAMMARS [CU73,MLB76]	ALGOL 68 [vWMPK69,vWMP+75]
	SOS [Plo81]	ESTEREL [BG92], FACILE [GMP90], LCS [Ber93b]
	NATURAL SEMANTICS [Kah87,CEN]	STANDARD ML [HMT90,MT90], EIFFEL [ACO93], SISAL [Jab95,ACW95]

Figure 1–1: Some direct and indirect operational semantics of real languages

categories: *indirect* semantics which translate programs into a better understood language, and *direct* semantics which do not. Our approach gives direct semantics. We shall compare indirect and direct semantics to see why this is worthwhile.

Operational semantics

Figure 1–1 is the result of a brief survey of operational descriptions of real programming languages. By "real" I mean languages which have actually been implemented and used in the programming community. It is probably incomplete, and therefore possibly misleading, but it does suggest that both kinds of operational semantics can define real programming languages, more or less equally well.

The reason for the success of indirect semantics is, I think, because we can always define a sufficiently high-level intermediate language, or *abstract machine*, that makes definition easy. The trick is to ensure that the abstract machine is also simple enough to be readily understood. Now there are two philosophies for building such abstract machines. One aims to find a universal abstract machine. The other tailors abstract machines for each language. ACTION SEMANTICS follows the first philosophy, while SMoLCS and EVOLVING ALGEBRAS follow the second.

The universalist philosophy has the advantage that we only have to understand one machine, which facilitates a large amount of foundational analysis and tool-building. It has the disadvantage that we have to work hard to ingest the various notations. For instance, there is an overwhelming amount of action notation. While it has been carefully designed to write near-english descriptions, to understand it properly we have to fathom Mosses's ersatz presentation of structural operational semantics for it [Mos91, App B.6] because near-english descriptions can be misleading. Another disadvantage is that it provides little insurance against unforseen linguistic features.

The individualist philosophy does insure against the future, and does not require prior understanding of an elaborate machine. It does require knowledge of an abstract machine description method, such as Structural Operational Semantics for SMoLCS and a simple rewriting system of first-order algebras for EVOLVING ALGEBRAS. Both are mathematically light and easily understood. Moreover, they both support the construction of tools: for example, both approaches have rapid prototyping (see the relevant webpages for details and more examples [SMo, EA]). Nevertheless, they do require more work per language, and we are less able to compare different languages or build sufficiently high-quality tools. It is more economic to research one machine intensively than many different machines.

Digressing a little, the indirect approach is pandemic in semantics. In denotational semantics, we use *metalanguages*: the traditional λ-notation [Sch88], the Logic of Computable Functions (LCF) [Sco69], FPC [Plo85] and the computational metalanguage [Mog90,Mog91,Cen96]. In this situation, metalanguages are convenient notations for manipulating the mathematical structures required to describe programs. They can also be used to consider denotational definitions abstractly, for instance the computational metalanguage allows us to plug together modules to make a variety of combinations of metalanguages.

At the other end of the spectrum, compiler theory uses intermediate code representations [TS85, ch. 10] because they are free of machine idiosyncrasies, they facilitate optimizations, they speed up compiler development and improve portability.

Yet no matter how useful intermediate languages are, ultimately we need a direct description of the foundations. We have already commented on how simple the semantic description methods are for foundational abstract machines; the great achievement is that these same methods can give manageable definitions of real programming languages too.

Two early examples of direct semantics were VDL (which subsequently became VDM: see [BJ78] for reasons why) and W-GRAMMARS. The first consisted of two parts, a translator and an interpreter. The translator was used both to check grammar and context-sensitive constraints, and also to convert the concrete syntax of real programs into abstract syntax. The interpreter described how abstract syntax representations of programs could evolve over time (in a not dissimilar way to evolving algebras). The second method used two levels of grammar: one for context-free aspects and meta-grammars for context-sensitive aspects of the language. The framework lumped syntax, static semantics and dynamic semantics together.

Syntax is that part of grammar that concerns the due arrangement of words or the construction of sentences. It is usually given inductively using EBNF, but there is a school advocating the use of the simply-typed λ-calculus [PE88] to represent syntax. Static semantics ensures that programs make sense: for instance, that everything is correctly typed, or that every identifier is declared. Dynamic semantics describes the behaviour of sensible programs. Modern semantics tend to separate the three.

There are two modern direct description methods: STRUCTURAL OPERATIONAL SEMANTICS (SOS) and NATURAL SEMANTICS. The first describes an automaton that evaluates programs of the defined language directly. That is, it specifies an abstract machine which directly manipulates programs. In the tradition of automata theory [HU79] a SOS is given by a (labelled) transition system:

> **Definition 1.0 ([Plo81])** *A* labelled transition system *is a quadruple* $\langle \Gamma, \Omega, \Lambda, \longrightarrow \rangle$ *where* Γ *is a set of* configurations*;* $\Omega \subseteq \Gamma$ *is the set of* terminal *configurations;* Λ *is a set of* labels *and* $\longrightarrow \subseteq \Gamma \times \Lambda \times \Gamma$ *is the* transition relation*, such that*
>
> $$\textit{for all } \gamma \in \Omega,\ \lambda \in \Lambda,\ \gamma' \in \Gamma,\ \textit{not}(\gamma \xrightarrow{\lambda} \gamma')$$
>
> *where* $\gamma \xrightarrow{\lambda} \gamma'$ *is an abbreviation for* $\langle \gamma, \lambda, \gamma' \rangle \in \longrightarrow$*. A* stuck *configuration* γ *is any non-terminal configuration such that* $\gamma \xrightarrow{\lambda} \gamma'$ *for no* $\lambda \in \Lambda$ *and* $\gamma' \in \Gamma$*.*
>
> The definition of unlabelled transition systems is even easier: we simply remove every mention of labels from the preceding definition. That is, a transition system is a triple $\langle \Gamma, \Omega, \rightarrow \rangle$.

The main innovation is that the transition relation is defined inductively on the structure of programs, thereby permitting the use of structural induction on syntax to prove properties about transitions. An example of a SOS for the call-by-value λ-calculus can be found in figure 1–2.

Overall, the framework is mathematically simple, and permits one to describe the intuitive computation steps of programs easily. A natural semantics is even simpler: it does not involve a transition system; it simply consists of a set of syntax-directed rules that relate programs to their attributes (i.e., values, types or translations). The method is essentially proof-theoretic (see [Kri68,Kri71] for a survey of proof theory), and a natural semantics can be seen as a *Post System* [Pra71]. The following definition of natural semantics is based on that definition of Post System and the informal description given in [Kah87]:

Abstract syntax

$$M ::= x \mid \lambda x.M \mid MM$$

where x ranges over Var, the set of variable identifiers. Let LC be the set of all λ-calculus terms and $Val = Var \cup \{\lambda x.M \mid x \in Var, M \in LC\}$ be the set of *values*, ranged over by v.

SOS

Transition system $\langle LC, Val, \rightarrow\rangle$ where $\rightarrow$ is defined inductively by:

$$(\lambda x.M)v \rightarrow M[v/x] \qquad \frac{M \rightarrow M'}{MN \rightarrow M'N} \qquad \frac{N \rightarrow N'}{(\lambda x.M)N \rightarrow (\lambda x.M)N'}$$

Natural semantics

$\langle \emptyset, LC, Val, \mathcal{L}, \mathcal{R}\rangle$, where $\mathcal{L} = \{M \Rightarrow v \mid M \in LC, v \in Val\}$ and $\mathcal{R}$ is the set generated by the schemata

$$v \Rightarrow v \qquad \frac{M \Rightarrow \lambda x.M' \quad N \Rightarrow v' \quad M'[v'/x] \Rightarrow v}{MN \Rightarrow v}$$

Figure 1–2: Direct semantics for call-by-value λ-calculus

Definition 1.1 *A* Natural Semantics *is a quintuple* $(\mathcal{E}, \mathcal{C}, \mathcal{A}, \mathcal{L}, \mathcal{R})$ *where* $\mathcal{E}$ *is a set of* environments*,* $\mathcal{C}$ *is a set of* configurations*,* $\mathcal{A}$ *is a set of* attributes*,* $\mathcal{L}$ *is a language of atomic formulae built over the terms in* $\mathcal{E}$*,* $\mathcal{C}$ *and* $\mathcal{A}$*, and* $\mathcal{R}$ *is a set of inference rules determined as the instances of a finite number of schemata of the form*

$$\frac{A_1 \quad \cdots \quad A_n}{B}$$

where $A_1, \ldots, A_n, B$ *are atomic formulae, and* B *contains no parameters not occurring in* $\{A_1, \ldots, A_n\}$*.*

Again, it permits proofs about programs via the technique of rule induction. A natural semantics for the call-by-value λ-calculus can also be found in figure 1–2.

Indirect vs. Direct semantics

Having described indirect and direct semantics it now remains to compare them. Foundationally, indirect semantics are more complex, but they aim to simplify definitions using suitably high-level description languages. The proliferation of indirect definitions of real languages indicates that this goal is successful. The key issue is how to balance the complexity of metatheories against the complexity of definitions.

A first indication of the success of a description method is whether or not it has defined a real language. This is the ultimate purpose of description methods. However, as Mosses says, the description of one language in any framework could merely be testament to the endurance of Man. A better indication is therefore how prolific a method is: the more definitions we have, the better the method must be. Regarding this criterion, figure 1–1 suggests that the modern indirect semantics are better than the direct ones. However, a large number of definitions could in turn merely be testament to the endurance of a small number of research groups trying to promote the method. Further, the small number of real definitions may ultimately be due to the slow uptake of formal methods in industry rather than any demerit of the method. An even better indication therefore is how widespread a method is: the more people that use it, the better the method must be. Regarding this criterion, my experience is that SOS and NATURAL SEMANTICS are by far the most popular.

We need to look more closely. The only hard evidence we have are the definitions themselves. To say anything meaningful, we must consider specific definitions. [GK93] gives the semantics of C using evolving algebras. However, this definition yields a big surprise: the translation from C syntax to evolving algebra rules is completely informal! This is unsatisfactory because it implies incompleteness. For example, their semantics of the `goto` statement is simply "move to the next task", which is given by the NextTask function. This function is never defined: to do so would require a translator from program text to their task-based representation. I suspect that if they had to contain formal translators, evolving algebra definitions would require much more effort and be much less popular.

By contrast, every other approach relates programs to meanings in a completely formal manner. To compare two different semantic description methods, we need to see how each method gives meaning to a particular language (e.g., see [MLB76]). [Wat87] does just that, he compares an action semantics of the core of STANDARD ML with its natural semantics [HMT90]. He gives three arguments for why he favours the action semantics:

1. Action semantics are easier to read, even by the layman.

2. Natural semantics use closures to capture static binding and mutual recursion, which is clumsy.

3. Natural semantics have to propagate stores and environments everywhere, which impares legibility.

The first argument is easily countered. If we want a superficial understanding of a language, we can read an informal semantics. This is easier to write (it doesn't require knowledge of a pseudo-english grammar) and read (action notation scans oddly). To obtain a deep understanding of a language we have to invest a lot of effort anyway, whether to understand action notation or the various techniques used in natural semantics.

This also counters the second argument. Closures are a standard part of the natural semantics arsenal. However, we do not even need to use closures — we can use substitution instead, much like we do in figure 1–2 (see also extended natural semantics [Han93]).

The third argument is more interesting. Certainly, natural semantics rules can be overwhelmed with syntactic notations, e.g., rules (114) and (115) of [HMT90]:

$$\frac{s, E \vdash exp \Rightarrow \texttt{ref}, s' \quad s', E \vdash atexp \Rightarrow v, s'' \quad a \notin \mathrm{Dom}(mem \text{ of } s'')}{s, E \vdash exp\ atexp \Rightarrow a, s'' + \{a \mapsto v\}}$$

$$\frac{s, E \vdash exp \Rightarrow :=, s' \quad s', E \vdash atexp \Rightarrow \{\mathsf{1} \mapsto a, \mathsf{2} \mapsto v\}, s''}{s, E \vdash exp\ atexp \Rightarrow \{\} \text{ in Val}, s'' + \{a \mapsto v\}}$$

It was precisely to reduce the syntactic clutter of rules that they introduced the *state convention*. It permits us to elide states from the presentation of rules by providing a mechanical way to put them in when needed. Watt claims (p.582) that this "does not entirely solve this problem; besides, the [convention] is clearly *ad-hoc*, and not generally applicable". This is wrong. In terms of legibility, it does solve the problem of propagating stores everywhere (except of course where the store is actually used). Furthermore, the convention is not *ad-hoc* because the same mechanism can be used in any semantics whose judgments have the same structure and where stores behave in the same way (most metatheories of operational semantics tend to concern the structure of judgments: e.g., TYPOL [Des88] and GSOS [BIM88]).

I would say that the problem of syntactic overload has less to do with store and environment propagation, and more to do with the amount of auxiliary definitions and notations a language definition requires. Once again, this is the same issue as the previous two points.

Ultimately, the question of which method gives the clearest semantics is subjective. [MLB76] (which was written by three people) remarks when comparing four definitions of a simple language, "Each of us found the technique we knew the best to be the clearest" (p.271). Yet they also remark (p.274) that for formal semantics to become widely accepted outside Theoretical Computer Science, great attention must be paid to human engineering, so that the general reader can gain some benefit from perusing a semantics. ACTION SEMANTICS has been deliberately engineered in this way. However NATURAL SEMANTICS judgments are also readily understandable, for example, "In environment E, program p evaluates to v" is quite clear.

The state convention is an aid to legibility, but it does not formally alter the rules. The rules do propagate stores, and when we are proving results, we have to consider their unabbreviated forms (or at the very least check to see if the rules match the required shape). Moreover, because rules have to carry around the state of the entire system, it makes alteration and extension harder. If I want to introduce a new component to the configurations, I have to rewrite every rule, and reprove every theorem — even if the extension is orthogonal.

Despite this difficulty, NATURAL SEMANTICS (and SOS) have virtues too. For one, their metatheories are very simple, and they permit proofs by induction. Furthermore, they have proved useful as bases for program analysis and theories of tool construction. For example, they have been used in theories of static analysis [Tof87], compiler generation [DJ86], debugging [DS92], program animation [Ber91] and programming environments [CEN].

My thesis is that we can reduce the syntactic clutter of their semantic judgments in a formal way while retaining their virtues: an intuitive metatheory and amenability to simple proofs by induction.

Synopsis

The problem is that semantic rules contain too much information: each judgment must record the entire state of an abstract machine, even those components which are not directly relevant to the rule at hand. For instance, suppose we wished to add stores to the λ-calculus, and wished to modify its SOS. Then the syntax would become:

$$M ::= \ x \mid \lambda x.M \mid MM \mid \mathit{set}\ \alpha\ M \mid \mathit{get}\ \alpha$$

where α ranges over a distinguished set of memory locations, *Loc*. The semantic rules would become

$$\langle (\lambda x.M)v, \sigma\rangle \rightarrow \langle M[v/x], \sigma\rangle \qquad \frac{\langle M, \sigma\rangle \rightarrow \langle M', \sigma'\rangle}{\langle MN, \sigma\rangle \rightarrow \langle M'N, \sigma'\rangle}$$

$$\frac{\langle N, \sigma\rangle \rightarrow \langle N', \sigma'\rangle}{\langle (\lambda x.M)N, \sigma\rangle \rightarrow \langle (\lambda x.M)N', \sigma'\rangle} \qquad \frac{\langle M, \sigma\rangle \rightarrow \langle M', \sigma'\rangle}{\langle \mathit{set}\ \alpha\ M, \sigma\rangle \rightarrow \langle \mathit{set}\ \alpha\ M', \sigma'\rangle}$$

$$\langle \mathit{set}\ \alpha\ v, \sigma\rangle \rightarrow \langle v, \sigma[v/\alpha]\rangle \qquad \langle \mathit{get}\ \alpha, \sigma\rangle \rightarrow \langle M, \sigma\rangle \quad \text{if } M = \sigma(\alpha)$$

and where σ, ranging over the set Store of functions $\mathit{Var} \rightharpoonup \mathit{LC}$, represents the state of the store, such that $\sigma[v/x](y) = \text{if } y = x \text{ then } v \text{ else } \sigma(y)$.

Every rule has to be altered to propagate the store, yet only the last two rules actually interact with it. Every other rule simply passes it around, cluttering up the syntax and obscuring the essential meaning the rule imparts to the connectives. The stores have to be propagated everywhere to capture their serial behaviour.

In this example, the interaction between stores and programs is limited to the *set* and *get* connectives. The other connectives do not interfere with the store in any way. If only we could limit the occurrence of stores to where they were really needed. That is, we should like to interpret the old rules "in an ambient store", and add

$$\frac{}{\mathit{set}\ \alpha\ v \rightarrow v}\ \textit{alter the ambient store} \qquad \frac{}{\mathit{get}\ \alpha \rightarrow v}\ \textit{read the ambient store}$$

The above *state convention* does this. However, as we have said, it is only a convention, and in proofs we still have to consider every rule, either in its full form, or to check that they have the required shape.

We would like to be more formal. When we prove results not concerning the store, we should like not to mention stores. Similarly, when we prove results solely about the store, we should like not to mention programs. We want to have two kinds of transition, one for programs and one for stores. The essential idea is that when the program wishes to write to the store then it induces a store transition. That is, we should like to apply

$$\overline{set\ \alpha\ v \rightarrow v} \qquad \text{and} \qquad \overline{\sigma \rightarrow \sigma[v/\alpha]}$$

simultaneously. We denote simultaneity of application by drawing a line (an "interaction link") between the two rules. Therefore we can reuse the old rules which talk about function application and introduce special rules for the store.

To see how this scheme would work, suppose we want to prove that the first transition of $(\lambda x.M)(set\ \alpha\ v)$ alters the store. The rules show that we first have to prove that $set\ \alpha\ v$ alters the store, and that second, being the argument to $\lambda x.M$ does not alter that fact.

We apply the two interacting rules for *set* to obtain the two interacting proof trees:

$$\overline{set\ \alpha\ v \rightarrow v} \;\text{———}\; \overline{\sigma \rightarrow \sigma[v/\alpha]}$$

We can then apply the third application rule directly to the program transition tree to get

$$\cfrac{\overline{(set\ \alpha\ v) \rightarrow v}}{(\lambda x.M)(set\ \alpha\ v) \rightarrow (\lambda x.M)v} \;\text{———}\; \overline{\sigma \rightarrow \sigma[v/\alpha]}$$

which again consists of two interacting proof trees, one for the program and one for the store. Hence the first transition of the program does alter the store.

Throughout this thesis, we shall be concerned to evaluate the practical qualities of the various proof metatheories we examine. The idea is to study how easy it is to prove properties about the operational semantics we describe. To avoid confusion, I shall use the term "deduction" to apply to the interacting proof trees, and "proof" to refer to the mathematical arguments establishing results about the deductions.

Chapter 2, **"Interacting Deductions"**, describes I-*deduction*, the metatheory of this kind of interacting deduction. It also demonstrates the properties of I-deductions using a simple process calculus called P. We give it a novel transition semantics which does not propagate action information. We compare it to two more standard transition semantics.

We find that the novel semantics is marginally simpler than the standard two. However, when we add stores to P to yield the language P(:=), the gains are increased. Again, we do not have to propagate stores everywhere. Moreover, we also discover that the extension is modular. The new rules for the stores can be added to each of the three semantics without change.

Section 2.5.1 shows how we can extend a simple proof about the nondivergence of P processes to P(:=) processes. It shows how much simpler the extension is in our metatheory than in the standard ones. Section 2.5.2 shows that a simple proof about store transitions remains true regardless of where they are used. In this sense, our metatheory permits modular proofs. Both of these examples make use of the *proof fragmentation theorem*, which allows us to break proofs about

I-deductions into proofs about the individual proof-trees. Section 2.5.3 shows how it can be used to simplify an equivalence proof between our semantics for P and the most standard one.

Chapter 3, **"Evaluation Semantics and Sequential Deductions"**, shows that not only can the I-deductions give transition semantics to P, they can also give natural semantics-like definitions (henceforth called "evaluation semantics"). However, P does not have sequential composition. When we add it (to get P(;)) we find that the evaluation I-semantics become much less perspicuous. Therefore we alter the metatheory to introduce a notion of sequencing, obtaining QI-deduction.

This turns out to be quite subtle. Sequencing already exists in I-deduction (otherwise we could not have given P(;) semantics), but not in a convenient form. The subtlety lies in getting the two forms of sequencing to agree. Once this is resolved, we proceed to compare how the evaluation semantics of P(;) fares in a simple translation correctness result with a standard transition semantics.

The results are mixed. We should expect this — in conventional modern operational semantics, SOS fares better than NATURAL SEMANTICS for concurrent languages. A main problem is that nonterminating behaviour cannot be expressed in an evaluation semantics. Another is that it is hard to define notions of process equivalence in evaluation semantics, making analysis of nondeterminism difficult. These are problems owing to the particular nature of the semantic judgments involved (transition versus complete evaluation), and are not overcome in a QI-system. Thus it is generally not a good idea to use natural semantics-like judgments to capture the meaning of concurrent languages. A notable exception occurs in chapter 5 where we use an evaluation semantics in the soundness proof of a Hoare Logic for the partial correctness of CSP.

Nevertheless, the translation correctness proof is structurally simpler in that we do not have to perform any subinductions on the length of transition sequences. Thus we may still be able to use it profitably for proving results about sequential and deterministic languages.

Both I-deduction and QI-deduction concern systems of inference rules which do not consider the information shared via particular interactions. Chapter 4, **"The Content of Interaction"**, describes a metatheory (DQI-*deduction*) which allows formulae to be assigned to particular interactions, denoting the information which is shared during the interaction.

The definition of DQI-deduction is more naturally inductive than the definitions of I-deduction and QI-deduction. This allows some proofs to proceed directly by induction on the structure of deductions rather than via the proof fragmentation theorem. Moreover it introduces another proof technique called *proof assembly*. For motivation, we use a simple type-checking example. Proof fragmentation is enough to show that no type-checked process fails. This technique is insufficient to show the other direction: that every successful process type-checks. For this we need proof assembly, which allows us to assemble proofs from proofs about fragments.

The family of processes based around P are too simple. We need a larger example to test whether that our semantic method scales up. Chapter 5, **"A**

Semantics and Logic for CSP", gives an evaluation semantics to CSP quite easily. However, just to be sure that CSP is not pathological, we also examine various extensions: nested parallelism, communication via shared variables, dynamic process creation, pre-emption, multicasting and recursive procedures. It turns out that only pre-emptive primitives cannot be given an evaluation semantics.

To show that the semantics is useful, we introduce a novel one-level compositional Hoare Logic for CSP and prove it sound. Usually one-level compositional Hoare Logics need to trace communicative histories carefully. Our method does not. Moreover, the technique is quite general, and should be applicable to other concurrent languages with little alteration. The soundness proof is also very simple, using the evaluation semantics for CSP.

Finally, we finish by confirming that our semantic method really is an operational semantics. The reason that natural semantics are operational is because the deduction of semantic judgments describes how the program evaluated. It is well-known that natural semantics can be seen as PROLOG [CM84] clauses, and that in this light yields a prototype interpreter for the defined language [Des84] (For refinements of the idea, see [And91,DS92].) I believe it is precisely because the inference rules make an interpreter that a natural semantics is an operational semantics. (See [Sun84a] for a discussion on meaning via proof theory.) In chapter 6, **"The Process Calculus Interpretation"**, we show how our interacting rules can be seen as CCS process definitions to yield prototype interpreters. The interpretation is quite straightforward, and also allows us to reinterpret process-calculus results in terms of our proof-theory. For instance, we show that there is no procedure to decide whether a set of formulae is deducible in an arbitrary DQI-system.

Chapter 2

Interacting Deductions

In Natural Deduction, an inference tree traces the proof of a theorem. A leaf is a basic axiom and an internal node an intermediate step. One can imagine a mathematician reasoning from some basic axioms step by step till he deduces his result.

We can think of an interacting deduction as the work of a community of mathematicians, each trying to deduce his or her own theorem. Again, a leaf is a basic axiom and an internal node an intermediate step. However, mathematicians also collaborate, discussing ideas which may be mutually stimulating. Therefore an interacting deduction also records the fact of collaboration: an interaction link records the mutual dependency of two simultaneous inference steps.

This chapter is concerned with the basic theory of interacting deductions, and does not consider the content of interactions (this is considered in chapter 4). In fact we can still do a lot with this basic theory, where interaction is really *synchronization*. We use it to give transition semantics to a simple concurrent language P, which we compare with more standard approaches. We find that both semantic definitions and proofs about them become simpler, easier to extend and more modular.

The key idea is that we do not always have to reason about the entire community of mathematicians to understand the work of a particular individual or group of individuals. Sometimes it is better to think about a fragment of the community in isolation. It is from this independence that our improved extensibility and modularity arise.

The structure of this chapter is as follows. Section 2.1 gives preliminary definitions and notations. Section 2.2 inductively defines ordinary interacting deductions, the I-*deductions*, from sets of I-*rules*. It also characterizes them axiomatically: in future proofs the axiomatic characterization is more convenient. Section 2.3 presents an application: a transition semantics for P using the theory of I-deductions. It shows how one can elide semantic objects from judgments. Thus judgments become syntactically simpler than their Structured Operational Semantics counterparts. Another consequence is that semantic definitions become more extensible, as illustrated by the extension of P to P(:=), a language with stores.

Section 2.4 introduces the technique of *proof fragmentation*: naïve reasoning about I-deductions may lead to clumsy proofs of even simple results. Fragmentation is simply a technique to allow reasoning about parts of deductions. For instance, in the semantics for P(:=), a deduction consists of two interacting trees: one for process transitions and the other for store transitions. Each tree is a fragment of the overall deduction. To reason about the behaviour of stores, it makes sense to prove the property using only the fragments of the rules that mention store transitions. The technique is justified formally by the *proof fragmentation theorem*, which can be (very) loosely phrased as

> To prove a universal property about fragments of I-deductions, it is sound to reason over the rule fragments that build them.

Section 2.5 gives three examples of proof fragmentation. The first example shows how a proof about the termination of P can be extended to a proof about the termination of P(:=). Thus not only are semantic definitions easier to extend, proofs about them are easier to extend too. The second example is a proof about the stores used in the semantics of P(:=). This is simpler than a corresponding proof in a structured operational semantics because only the rules that mention stores are considered. It is also modular for the same reason. The last example uses fragmentation to simplify the proof of equivalence between the I-semantics and the structured operational semantics.

Many of my foundational definitions are modeled on Prawitz's work on Natural Deduction [Pra65,Pra71]. I do not assume that the reader is familiar with it. However, I hope that those who are will see this work as a natural extension.

2.1 Preliminaries

Languages At the moment I do not want to commit myself to any particular notion of language — I believe that any meaningful notion will suffice. I am not here concerned with the properties of any particular language; I wish to concentrate on the structure of deduction. I assume the reader is familiar with the notions of *formula*, *term*, *free variables* and *substitution*. Appendix A gives a more formal account.

I use $\mathcal{L}$ to range over languages, $A, B, C, \ldots$ to range over formulae and t to range over terms. I write $A \in \mathcal{L}$ if A is a formula in language $\mathcal{L}$, and $M \in \mathcal{L}$ if t is a term of the language $\mathcal{L}$. I use x to range over variables, FV for the function that returns the set of free variables of a term or formula and θ for substitutions. I write θA for the instance of A obtained by applying θ to the free variables of A (and similarly for terms).

Intuitively, a deduction will be a forest of interacting inference trees where the interactions are represented as pairs of formula occurrences. The simplest way I have found to maintain such structures (particularly when defining rule application) is to introduce an artificial notion of formula occurrence (i.e., a new datatype) and then build inference trees out of these objects rather than formulae.

To avoid confusion, I shall always use the noun "occurrence" to refer to an artificial occurrence, and the verb "to appear" to mean "to occur within a structure". Of course, no artificial occurrence should actually appear more than once in a deduction. This trick allows me to avoid defining forests as either multisets or sequences of inference trees, both of which are problematic.

I use Girard's device [Gir87, p.29] to represent occurrences. Formally, a *formula occurrence* of $\mathcal{L}$ will be a pair (A, n) of a formula $A \in \mathcal{L}$ and a distinguishing tag $n \in \mathbb{N}$. If $\theta A = B$ I write $\theta(A, n) = (B, n)$. If X is a set of formula occurrences, I shall write $\theta X = \{\theta(A, n) \mid (A, n) \in X\}$. Similarly, I write $FV(A, m) = FV(A)$. I write $(A, n) \equiv (B, m)$ if $A = B$, in which case I say the two occurrences *match* or *have the same shape*. I say two sets of occurrences X and Y match (written $X \equiv Y$) if there exists a bijection $f : X \leftrightarrow Y$ such that for all $(A, n) \in X$, $f(A, n) \equiv (A, n)$.

It is very cumbersome to continually write (A, n) for an occurrence of A. Therefore, following Prawitz [Pra65, p25], I shall use the letters $A, B, C, \ldots$ to range over formula occurrences as well as formulae. I write $A \in \mathcal{L}$ if A is an occurrence of a formula in $\mathcal{L}$. This will afford no confusion most of the time. When I want to distinguish two different occurrences of the same formula A, I shall not write pairs explicitly, but use Prawitz's less intrusive superscript notation: A^1 and A^2, and so on. This is not to be confused with my use of subscripts, where A_1 and A_2 may be occurrences of different formulae. Thus A_1^1 and A_1^2 are different occurrences of A_1, and A_1^1 and A_2^1 are occurrences of different formulae.

Throughout this section I shall assume a fixed $\mathcal{L}$.

Formula Trees A *formula tree* is either a formula occurrence A or the pair (X, A) when X is a finite set of formula trees that share no formula occurrences, and A is a formula occurrence not already appearing in X. While formula trees are usually built from formulae, I use formula occurrences here because it simplifies the treatment of interaction later.

A tree $T = (X, A)$ is written graphically in the following way:

$$\frac{X}{A} \qquad \text{or} \qquad \frac{T_1 \quad \cdots \quad T_n}{A}$$

if $X = \{T_1, \ldots, T_n\}$. I call $T_1, \ldots, T_n$ the *children* of T, and say that A is the *root* of T (written $A = \mathrm{root}(T)$). The *subtree of* relation is easily defined as the reflexive and transitive closure of the "child of" relation.

Formula Forests A *formula forest* F is a set of formula trees such that no occurrence appears more than once. I write $O(F)$ for the set of formula occurrences in F. I write $\mathrm{root}(F) = \{\mathrm{root}(T) \mid T \in F\}$.

Rule atoms *Rule atoms* are used to build interacting rules. They are just the ordinary notion of rule (except that I use occurrences in place of formulae to avoid multisets of premises): a pair $(Prem, C)$ where $Prem \cup \{C\}$ is a finite set of formula occurrences, and each occurrence appears only once. $Prem$ is the set of *premises* of the atom, and C is the *conclusion*. I write $\theta(Prem, C)$ for the *instance* $(\theta Prem, \theta C)$. I write $(Prem_1, C_1) \equiv (Prem_2, C_2)$ when I say they also *match* if $Prem_1 \equiv Prem_2$ and $C_1 \equiv C_2$. I use a to range over atoms.

2.2 Interacting deductions

A rule atom establishes how someone can draw a new inference from a set of previously established facts. An interacting rule establishes how a set of people can draw new inferences in their work after exchanging ideas with each other. For now I am not interested in what ideas are shared, or in who shares them (I shall consider this in chapter 4), so the definition of interacting rule is simply as follows.

I-rules An I-*rule* is a finite, non-empty set of rule atoms, such that no occurrence appears more than once. I drop the I- prefix whenever it is clear from the context. I use r to range over rules.

I write θr for the rule instance obtained by applying θ to its elements. Conventionally, rule instances will also be rules: uninstantiated rules will be *rule schemata* or rule templates that generate their instantiations. Two rules *match* (written $r \equiv r'$) if there exists a bijection $f : r \leftrightarrow r'$ such that for all $a \in r$, $f(a) \equiv a$. I write a rule with three atoms graphically in the following way:

$$\frac{Prem_1}{C_1} \;\rule[0.5ex]{3em}{0.4pt}\; \frac{Prem_2}{C_2} \;\rule[0.5ex]{3em}{0.4pt}\; \frac{Prem_3}{C_3}$$

where I have arbitrarily ordered the rule atoms. (Note that again I could have used sequences of atoms instead of sets, but to do so makes life harder.) When I enumerate the premise set of an atom I shall omit the set brackets. I write other interacting rules similarly.

I-systems An I-*system* $\mathcal{T}$ is a pair $(\mathcal{L}, \mathcal{R})$ where $\mathcal{L}$ is a language and $\mathcal{R}$ is a set of I-rules over formulae in $\mathcal{L}$ (typically, these will be generated from a finite set of rule schemata). In the rest of this chapter, I shall sometimes drop the "I" prefix when convenient.

I-deductions To motivate the definition of I-deduction, consider how our community of forgetful mathematicians deduce facts. At first, they have nothing until they write down some immediately obvious facts (an observation about the world, or an axiom of the formal system they are studying). From then on, at each stage in the community's life, there will exist a body of knowledge represented by a forest F: the current facts are to be thought of as the roots of F. This is advanced whenever a group of (one or more) mathematicians (interested in disjoint sets of facts deduced by $F_1, \ldots, F_n \subseteq F$) inspire each other by exchanging ideas, after which each mathematician is able to draw a new conclusion, $C_1, \ldots, C_n$, respectively. (Thus each mathematician uses a rule atom matching $(\mathrm{root}(F_i), C_i)$). At this point, the premises of the rules have been forgotten: so if a result is required twice, it will have to be deduced twice. Note that it is possible for some current facts given by $F_0 \subseteq F$ not to be used at a particular step, so these remain available for the next step.

Definition 2.0 (I-Deduction) *The set* $\mathbf{I}(\mathcal{T})$ *of* I-deductions *of an* I-*system* $\mathcal{T}$ *is the least set such that*

1. $(\emptyset, \emptyset) \in \mathbf{I}(\mathcal{T})$

2. *if* (a) $(F, I) \in \mathbf{I}(\mathcal{T})$ *where* $F = F_0 \cup F_1 \cup \ldots \cup F_n$ *and* $F_0, \ldots, F_n$ *are all disjoint, and*

   (b)
$$\frac{root(F_1)}{C_1} \;\text{—} \cdots \text{—}\; \frac{root(F_n)}{C_n}$$
   matches a rule of $\mathcal{T}$ *such that* $C_1, \ldots, C_n$ *are distinct occurrences not in* F*, and*

   (c) $I' = I \cup \{(C_i, C_j) \mid 1 \leq i, j \leq n\}$

   then $(F_0 \cup \{\frac{F_1}{C_1}, \ldots, \frac{F_n}{C_n}\}, I') \in \mathbf{I}(\mathcal{T})$

Note that this definition records an interaction by putting interaction links between every pair of new conclusions. Note also that every new conclusion interacts with itself. This is done purely for technical convenience: it ensures that I is a reflexive, symmetric and transitive relation (i.e., an equivalence relation).

For an alternative logical treatment due to Gordon Plotkin, see section 7.3.4.

2.2.1 Examples

Let $\mathcal{L}$ be the language consisting of two formulae, A and B. Let $\mathcal{R}$ be the following set of rules:

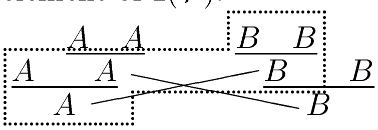
$$\frac{}{A} \qquad \frac{}{B} \qquad \frac{A \quad A}{A} \;\text{———}\; \frac{B \quad B}{B}$$

Then we have an I-system $\mathcal{T} = (\mathcal{L}, \mathcal{R})$, and the following is (the graphical representation of) a simple $\mathcal{T}$-deduction:

$$\dfrac{A \qquad \dfrac{A \quad A}{A} \;\text{————}\; }{A} \;\text{————}\; \dfrac{\dfrac{B \quad B}{B} \qquad B}{B}$$

(Technically, I should overline the leaves of this tree to indicate that they are axiom instances. However, doing this here makes the tree look ugly.) Note that interaction lines in rules are drawn between the rule atoms, but in a forest, they are drawn between the conclusions of rule instances in the forest.

The following is not an element of $\mathbf{I}(\mathcal{T})$:



One cannot apply the rules of $\mathcal{T}$ systematically to create this structure, even though it looks as if it were built from the rules. For instance, the dotted region (called in (non-topological) future terminology a *neighbourhood*) looks like an instance of the third rule.

The problem with this structure is that it contains a dependency loop. The structure contains two instances of the third rule, but each instance depends on conclusions of the other.

The next two subsections prove two important results about I-deductions. The first is that they have no dependency loops. The second result shows that in fact the absence of dependency-loops is a characterizing feature of I-deductions. This fact both gives us a clearer understanding of what deductions are, and allows us to reason about deductions more simply than the inductive definition.

2.2.2 I-deductions have no dependency loops

To show that the definition of I-deduction guarantees the avoidance of dependency loops, I characterize the dependencies between formula occurrences using preorders.

Preorders A preorder over a set X is a reflexive and transitive binary relation over the elements of X. I use the $\lesssim$ symbol to range over preorders. If $a \lesssim b$ then I also write $b \gtrsim a$. If $a, b \in X$, I write $a < b$ for $a \lesssim b$ and $b \not\lesssim a$, and $a \sim b$ for $a \lesssim b$ and $b \lesssim a$.

Preorders of deductions Let F be an occurrence forest. I say A is *below* B in F (written $A \lesssim_F B$) if there exists a tree $T \in F$ such that A is the root of a subtree of T containing B. I say $\lesssim_F$ is the preorder of F. Actually, $\lesssim_F$ is also a partial order, but the preorder of a deduction will not be so in general.

The preorder of a deduction (F, I) is given by

$$\lesssim_{(F,I)} = (\lesssim_F \cup I)^*$$

The rationale is that formula occurrences are deduced from both those above them and those they interact with. I take the transitive closure of $\lesssim_F \cup I$ to obtain a preorder, because (for example) $A \lesssim_F B$ and BIC do not imply $A(\lesssim_F \cup I)C$.

Dependency Loop Freeness I say a pair (F, I) is *dependency-loop free* if the following condition holds:

$$\text{For all } A, B \in O(F), \text{ if } A <_F B \text{ then } A <_{(F,I)} B. \qquad \text{(DLF)}$$

For motivation, again consider the community of mathematicians. If $A <_F B$ then A must have been deduced after B. Given the definition of $\lesssim_{(F,I)}$, this obviously implies $A \lesssim_{(F,I)} B$. However, if A was deduced after B, then it could not also have been deduced simultaneously with B — therefore $A \not\sim_{(F,I)} B$. This means

$A <_{(F,I)} B$. The problem with the non-example on page 17 is that it suggests that two occurrences were deduced both before and after each other.

Proposition 2.1 *Let $\mathcal{T}$ be an* I*-system, and let $(F, I) \in \mathbf{I}(\mathcal{T})$. Then (F, I) satisfies DLF.*

Proof: By induction on the definition of $\mathbf{I}(\mathcal{T})$. The base case is vacuous. For the induction step, assume that the property holds for $(F, I) = (F_0 \cup F_1 \cup \ldots \cup F_n, I)$, and suppose $r = \{a_1, \ldots, a_n\}$ is a rule instance that can be applied to make $(F', I') = (F_0 \cup \{\frac{F_1}{C_1}, \ldots, \frac{F_n}{C_n}\}, I')$, where $C_1, \ldots, C_n$ are the new conclusion occurrences. Now, pick $A, B \in O(F')$ such that $A <_{F'} B$. Then B equals no C_i because it exists above at least one occurrence. Hence $B \in O(F)$. There are two cases: either $A \in O(F)$ or not. If $A \in O(F)$ then $A <_{(F,I)} B$ by induction and thus also $A <_{(F',I')} B$ since $(I' \setminus I) \subseteq \{C_1, \ldots, C_n\}^2$ cannot relate A and B. If $A \notin O(F)$ then $A \lesssim_{(F',I')} B$ (since $A <_{F'} B$ and $\lesssim_{F'} \subseteq \lesssim_{(F',I')}$). The only way that $B \lesssim_{(F',I')} A$ would be if $AI'B$, but as we have seen, $AI'B$ only if B is a new occurrence, which it is not. $\square$

Proposition 2.2 *DLF implies that if $A \sim_{(F,I)} B$ then $A\,I\,B$.*

Proof: Suppose $A \sim_{(F,I)} B$. Then there exists a sequence of occurrences $A_0, \ldots, A_n$ in F such that $A_1 = A$, $A_n = B$ and for all $i = 1, \ldots, n-1$ either $A_i <_F A_{i+1}$ or $A_i I A_{i+1}$. Now, since $A \sim_{(F,I)} B$, we have for $i = 1, \ldots, n-1$, $A_i \sim_{(F,I)} A_{i+1}$ because

$$A \lesssim_{(F,I)} A_i \lesssim_{(F,I)} A_{i+1} \lesssim_{(F,I)} B \lesssim_{(F,I)} A$$

This in turn means that $A_i \not<_F A_{i+1}$ by DLF, which means that $A_i I A_{i+1}$ for all $i = 1, \ldots, n-1$. $\square$

2.2.3 DLF helps characterize the I-deductions

The previous proposition showed that no deduction contains any dependency loops. Dependency loop freeness is an important condition for our intended application (it corresponds to the principle that temporal states are not synchronized with previous states); but it is also important proof-theoretically: DLF helps characterize the set of I-deductions.

Theorem I states this formally. The content of the theorem is that a pair (F, I) is an I-deduction of the I-system $\mathcal{T}$ if it is built from instances of rules in $\mathcal{T}$, and the preorder $\lesssim_{(F,I)}$ satisfies DLF. To show this, I require the notions of I-*structure*, I-*neighbourhood* and how to *match* rules to neighbourhoods.

I-structure An I-*structure* (over a language $\mathcal{L}$) is a pair (F, I) where F is a finite set of finite occurrence trees (with formulae in $\mathcal{L}$) and $I \subseteq O(F) \times O(F)$ is an equivalence relation. Clearly, every I-deduction is an I-structure. Henceforth, I shall assume that every pair written "(F, I)" is an I-structure.

I-neighbourhoods Let C be an occurrence in an I-structure (F, I). The *neighbourhood* of C is the set

$$N_{(F,I)}(C) = \{atom(C') \mid C' \, I \, C\}$$

and $atom(C) = (\mathrm{root}(F_0), C)$ when (F_0, C) is that subtree which concludes C of a tree in F. Thus the neighbourhood of an occurrence returns the putative rule atom that introduces C. Thus the neighbourhood of C in (F, I) is the set of atoms that either conclude C or are connected to the atom that concludes C by a chain of interaction links. (Thus the conclusions of a neighbourhood form an equivalence class of I.) Note that neighbourhoods overlap: the premises of one neighbourhood must be the conclusions of others. I write $N(F, I)$ for the set of neighbourhoods in (F, I).

Neighbourhood Ordering The deduction preorder of (F, I) can be lifted to a preorder over neighbourhoods. The *neighbourhood ordering* of (F, I) is the relation $\leq_{N(F,I)} \subseteq N(F, I) \times N(F, I)$ defined by

$N_1 \leq_{N(F,I)} N_2$ if for all conclusions $A \in O(N_1)$ there exists a conclusion $B \in O(N_2)$ such that $A \lesssim_{(F,I)} B$

Thus N_1 is less than N_2 if some (conclusion occurrence in an) atom in N_1 occurs below an atom of N_2 in (F, I). The definition stresses conclusion occurrences because it is the conclusion occurrences that interact.

Proposition 2.3 *$\leq_{N(F,I)}$ is a partial order when (F, I) satisfies DLF.*

Proof: Reflexivity and transitivity follow from the reflexivity and transitivity of $\lesssim_{(F,I)}$. For antisymmetry, let $N_1 \leq_{N(F,I)} N_2$ and $N_2 \leq_{N(F,I)} N_1$. We show $N_1 = N_2$. Since $N_1 \leq N_2$ there exist conclusions $A_1 \in O(N_1)$ and $A_2 \in O(N_2)$ such that $A_1 \lesssim_{(F,I)} A_2$. Symmetrically, there exist conclusions $B_2 \in O(N_2)$ and $B_1 \in O(N_1)$ such that $B_2 \lesssim_{(F,I)} B_1$. Because they are conclusion occurrences, $A_1 \sim_{(F,I)} B_1$ and $A_2 \sim_{(F,I)} B_2$. But this means

$$A_1 \lesssim_{(F,I)} A_2 \sim_{(F,I)} B_2 \lesssim_{(F,I)} B_1 \sim_{(F,I)} A_1$$

which means $A_1 \sim_{(F,I)} A_2 \sim_{(F,I)} B_2 \sim_{(F,I)} B_1$. Thus, by proposition 2.2, DLF implies that $A_1 I^* A_2$ and $B_1 I^* B_2$. Therefore, $N_1 = N_2$ by the definition of neighbourhood. □

Rule matching An I-rule $r = \{a_1, \ldots, a_n\}$ *matches* a neighbourhood N via $f : O(r) \leftrightarrow O(N)$ (written $r \equiv_f N$) iff

$$N = \{f(a_1), \ldots, f(a_n)\}$$

where $f(\{P_{i1}, \ldots, P_{in_i}\}, C_i) = (\{f(P_{i1}), \ldots, f(P_{in_i})\}, f(C_i))$, f preserves the shape of occurrences and C_i is the conclusion of a_i (for $i = 1, \ldots, n$).

Theorem I *$(F, I) \in \mathbf{I}(\mathcal{T})$ if and only if (F, I) is a* I*-structure satisfying DLF and every neighbourhood matches a rule of $\mathcal{T}$.*

Proof: **$\Leftarrow$:** Let (F, I) be a I-structure satisfying DLF and such that every neighbourhood matches a rule of $\mathcal{T}$. Then (F, I) contains a finite number of neighbourhoods. By proposition 2.3, $\leq_{N(F,I)}$ is a partial order. Let $\leq$ be any total order containing $\leq_{N(F,I)}$. We use induction on the number n of neighbourhoods to show that $(F, I) \in \mathbf{I}(\mathcal{T})$.

When $n = 0$ the structure must equal $(\emptyset, \emptyset)$, which is a member of $\mathbf{I}(\mathcal{T})$ by definition. When $n = k+1$, consider (F', I'), the I-structure consisting of only the k highest (w.r.t. $\leq$) neighbourhoods of (F, I), i.e., not including those parts of the least neighbourhood distinct from every other neighbourhood — its conclusions. Clearly, (F', I') must satisfy DLF, and every neighbourhood must match a rule because (F', I') is contained within (F, I). By induction, $(F', I') \in \mathbf{I}(\mathcal{T})$.

Now, let N_{k+1} be the least (w.r.t. $\leq$) neighbourhood of (F, I). Suppose it matches rule $r = \{a_1, \ldots, a_n\}$ via $f : O(r) \leftrightarrow O(N_{k+1})$. Then $N_{k+1} = \{f(a_1), \ldots, f(a_n)\}$.

Since N_{k+1} is a neighbourhood of (F, I) then its premises must also belong to $\leq$-higher neighbourhoods of (F, I) — i.e., to neighbourhoods in (F', I'). Moreover, these premises must occur as conclusions of (F', I') since they are premises to conclusions of (F, I) which do not exist in (F', I').

Therefore there exist disjoint forests $F_0, F_1, \ldots, F_n$ such that $F' = F_0 \cup F_1 \cup \ldots \cup F_n$ and for $i = 1, \ldots, n$, $\mathrm{root}(F_i)$ equals the set of premises of $f(a_i)$. Thus, since for all $A \in O(r)$, $f(A) \equiv A$, we have $a_i \equiv f(a_i) = (\mathrm{root}(F_i), C_i)$ for $i = 1, \ldots, n$. Moreover, from the definition of neighbourhood, $f(a_i) = atom(C_i)$, which means that $(F_i, C_i) \in F$. Therefore

$$F = F_0 \cup \{\tfrac{F_1}{C_1}, \ldots, \tfrac{F_n}{C_n}\}$$

Also, by the definition of I-structure, the interactions of the neighbourhood must form an equivalence relation. Thus $I = I' \cup \{(C_i, C_j) \mid 1 \leq i, j \leq n\}$.

$\Rightarrow$: By proposition 2.1, we know that every I-deduction satisfies DLF. We show that every neighbourhood matches a rule by induction on the definition of $\mathbf{I}(\mathcal{T})$. **Case $(\emptyset, \emptyset)$:** vacuous. **Case (F, I)** being the result of applying $r = \{a_1, \ldots, a_n\}$ to (F', I'). By induction, every neighbourhood of (F', I') matches a rule of $\mathcal{T}$. From the definition of rule application, we have $F = F_0 \cup F_1 \cup \ldots \cup F_n$ where $a_i \equiv (\mathrm{root}(F_i), C_i)$. The only neighbourhood in (F, I) not in (F', I') is N equals

$$\{atom(C_1), \ldots, atom(C_n)\}$$

From the definition of $atom$, we have $atom(C_i) = (\mathrm{root}(F_i), C_i)$ (which we have already seen matches a_i) for $i = 1, \ldots, n$. Thus if we define $f : O(r) \leftrightarrow O(N)$ such that $f(a_i) = atom(C_i)$ we have $r \equiv_f N$. □

From the above, we can formulate a simple correctness principle for I-deductions based on their graphical representation. If one can trace a loop by following interaction lines and by going down a chain of dependencies in a tree at least once, then the structure is not an I-deduction. Otherwise, if it is also built out of rules, it is a deduction.

Throughout the rest of the thesis, I shall use Π and Σ (variously decorated) to range over deductions. If the set of conclusions of Π is $\{C_1, \ldots, C_n\}$, I write $\Pi \vdash C_1, \ldots C_n$. If further $\Pi \in \mathbf{I}(\mathcal{T})$, I write $\mathcal{T} \vdash C_1, \ldots, C_n$. I also write $\begin{matrix}\Pi\\ A\end{matrix}$ when $\Pi \vdash A$.

2.3 An Example

This section gives an example I-system drawn from Computer Science. It uses I-rules to describe the behaviour of a very simple process calculus based on CCS [Mil89]. The point of this example is both to illustrate the use of I-systems, indicate some of their properties, and also to motivate the technique of fragmentation. I compare this definition with a more traditional structured operational semantics (SOS) [Plo81], and also a semantics given by structured equivalences. I shall refer to the structured operational semantics as "the SOS", and the I-system as "the I-semantics".

The process calculus is built over an *action set Act*. Following Milner [Mil89, p.37], this is built from an infinite set Nam of names. I use $a, b, c, \ldots$ to range over Nam. Names will tend to be english words, but mostly I shall use a, b, c, etc. By $\overline{Nam}$ I denote the set of *co-names* of Nam, which are just the words of Nam overlined. I use $\bar{a}, \bar{b}, \bar{c}, \ldots$ to range over conames. I write $Lab = Nam \cup \overline{Nam}$, ranged over by l. I define *complementation* $\bar{\cdot} : Lab \to Lab$ by: $\bar{l} = \bar{a}$ if $l = a$ and $\bar{l} = a$ if $l = \bar{a}$. The action set $Act = Lab \cup \{\tau\}$ where $\tau \notin Lab$ is called the *silent action*. I use α to range over Act.

The simple calculus is called P. It can be thought of as the sum-, restriction-, recursion- and relabelling-free subset of CCS. Sum and recursion can be added without any problems. We shall consider restriction in section 3.4, and relabelling in section 7.3. The syntax of P is given by the following grammar:

$$p ::= 0 \mid l.p \mid p|p$$

where 0 is the process that does nothing and terminates, $l.p$ is the process that performs an action l from Lab and then behaves as p, and $p|q$ is the parallel composition of processes p and q. I have disallowed processes of form $\tau.p$ for simplicity.

Non-silent actions describe the effect a process has on its environment, which will in turn be a collection of parallel processes. I do not describe what each individual action "is"; for our purposes it is just a name. The crucial point is that each name has a distinguished partner, or co-name. A communication is said to occur between a process and its environment if the process performs an action a, and the environment "absorbs its effect" by performing the opposite action, $\bar{a}$, simultaneously. When this happens, the joint system of processes performs the silent action τ.

2.3.1 Comparative Semantics

The following three semantics formalise these ideas in different ways. In the SOS, when a process performs an action, we decorate the transition relation with the name of the action. In the I-semantics, when we deduce that a process performs an action, we must also simultaneously deduce its communicating partner absorbing its effect. In the structured equivalence setting we cannot deduce the transitions of a communicating pair of processes separately; an atomic transition is a transition of a communicating pair of processes.

Structured Operational Semantics of P

It is given by the labelled transition system $LTS_{SOS} = \langle \mathrm{P}, Act, \rightarrow, \Omega \rangle$ where $\rightarrow \subseteq \mathrm{P} \times Act \times \mathrm{P}$ is the relation defined by $(p, \alpha, q) \in \rightarrow$ if and only if $\mathcal{P}_{SOS} \vdash p \stackrel{\alpha}{\rightarrow} q$ where $\mathcal{P}_{SOS}$ is the deduction system $(\mathcal{L}_{SOS}, \mathcal{R}_{SOS})$ such that $\mathcal{L}_{SOS}$ is the language of judgments of form $p \stackrel{\alpha}{\rightarrow} q$ and the ruleset $\mathcal{R}_{SOS}$ is generated by the following rule schemata

$$\frac{}{l.p \stackrel{l}{\rightarrow} p} \qquad \frac{p \stackrel{\alpha}{\rightarrow} p'}{p|q \stackrel{\alpha}{\rightarrow} p'|q} \qquad \frac{q \stackrel{\alpha}{\rightarrow} q'}{p|q \stackrel{\alpha}{\rightarrow} p|q'} \qquad \frac{p \stackrel{l}{\rightarrow} p' \qquad q \stackrel{\bar{l}}{\rightarrow} q'}{p|q \stackrel{\tau}{\rightarrow} p'|q'}$$

Thus I have regarded the proof system of a structured operational semantics as the special case of I-system whose rules are singletons. The last component of the labelled transition system, Ω, is the set of terminal states of P processes. That is, it is the least set such that

$$0 \in \Omega \qquad \text{and} \qquad p|q \in \Omega \text{ when } p, q \in \Omega$$

It is not hard to see that the quadruple LTS_{SOS} satisfies the conditions on labelled transition systems given in definition 1.0.

I-semantics of P

The I-semantics of P is given by the transition system $TS_I = \langle \mathrm{P}, \rightarrow, \Omega \rangle$ where $\rightarrow \subseteq \mathrm{P} \times \mathrm{P}$ is the relation defined by $(p, q) \in \rightarrow$ if and only if $\mathcal{P}_I \vdash p \rightarrow q$ where $\mathcal{P}_I$ is the deduction system $(\mathcal{L}_I, \mathcal{R}_I)$ such that $\mathcal{L}_I$ is the language of *reduction* judgments of form $p \rightarrow q$ and the ruleset $\mathcal{R}_I$ is generated by the following rule schemata:

$$\overline{a.p \rightarrow p} \;\text{———}\; \overline{\bar{a}.q \rightarrow q}$$

$$\frac{p \rightarrow p'}{p|q \rightarrow p'|q} \qquad \frac{q \rightarrow q'}{p|q \rightarrow p|q'} \qquad \frac{p \rightarrow p' \qquad q \rightarrow q'}{p|q \rightarrow p'|q'}$$

The set of final states, Ω, is as defined above. Note that if I had allowed processes of form $\tau.p$, I would have required an extra rule $\overline{\tau.p \rightarrow p}$.

Structured Equivalences

The reduction (i.e., unlabelled transition) judgment of $\mathcal{P}_I$ is not uniquely a feature of I-semantics. They can also be derived using the technique of *structured equivalences* in a structured operational semantics. Such a semantics for P is given by the I-system $\mathcal{P}_{CONG} = (\mathcal{L}_I, \mathcal{R}_{CONG})$ where $\mathcal{R}_{CONG}$ is the set of rules:

$$\frac{}{a.p \mid \bar{a}.q \;\to\; p|q} \qquad\qquad \frac{p \to p'}{p|q \to p'|q} \qquad\qquad \frac{p \cong p' \qquad p' \to q' \qquad q' \cong q}{p \to q}$$

and $\cong \subset$ P $\times$ P (the structured equivalence relation) is defined to be the least relation satisfying

$$p \cong p \qquad\qquad p|q \cong q|p \qquad\qquad p|(q|r) \cong (p|q)|r$$

$$\frac{p \cong q}{q \cong p} \qquad\qquad \frac{p \cong q}{p|r \cong q|r} \qquad\qquad \frac{p \cong q \quad q \cong r}{p \cong r}$$

Note that there is no rule for action prefixing: we do not need to use such a rule. It is the lack of this rule which prevents the relation from being a structured congruence.

Note also that this approach does not require a rule for two concurrent transitions, whereas one is required in the I-semantics. Without such a rule there, internal communications could not be modeled in an I-deduction. However, this extra rule makes a significant difference: it permits several communication transitions to occur simultaneously, something not allowed in the other two approaches. Section 2.5.3 shows this formally.

We could avoid multiple transitions simply by adding action information to judgments, or by using the structured equivalence approach. In these cases, the interacting rules would be redundant. Shortly I shall indicate how interacting rules can be mixed usefully with the standard techniques.

Comparison

For example, let us see how the three systems deduce a transition of the process $a.p|r|\bar{a}.q$.

$$\mathcal{P}_{SOS}: \;\dfrac{\dfrac{\overline{a.p \xrightarrow{a} p}}{a.p|r \xrightarrow{a} p|r} \qquad \overline{\bar{a}.q \xrightarrow{\bar{a}} q}}{a.p|r|\bar{a}.q \xrightarrow{\tau} p|r|q} \qquad\qquad \mathcal{P}_I: \;\dfrac{\dfrac{\overline{a.p \to p}}{a.p|r \to p|r} \qquad \overline{\bar{a}.q \to q}}{a.p|r|\bar{a}.q \to p|r|q}$$

$$\mathcal{P}_{CONG}: \;\dfrac{a.p|r|\bar{a}.q \;\cong\; a.p|\bar{a}.q|r \qquad \dfrac{\overline{a.p|\bar{a}.q \to p|q}}{a.p|\bar{a}.q|r \to p|q|r} \qquad p|q|r \;\cong\; p|r|q}{a.p|r|\bar{a}.q \to p|r|q}$$

On Actions The first point of difference between the definitions is that $\mathcal{P}_{SOS}$ judgments contain actions, whereas those of the other two do not. In [San93, ch3], Sangiorgi shows that reduction judgments (regardless of how they are deduced) are in general too weak for process algebraic purposes. Reduction bisimulation is "rather weak; in general it is even not preserved by parallel composition" (p.47). Moreover, for CCS, reduction congruence coincides with strong bisimulation only for the divergent-free processes [MS92]. P is too simple to contain divergent processes, but this is accidental. Process calculus transitions should contain actions. $\mathcal{P}_I$ judgments do not for pedagogic reasons.

On Propagation I could easily add actions to the judgments of $\mathcal{P}_I$, however the point is that they are not required to describe the semantics. The judgments of $\mathcal{P}_I$ are syntactically less complex than those of $\mathcal{P}_{SOS}$ because action information does not have to be propagated down inference steps. Nor do we need another mathematical object (an equivalence relation) to achieve this.

The following sections show how one can avoid the propagation of other pieces of information too. Avoiding propagation simplifies the syntactic presentation of judgments, and thus also the statement and proof of theorems about such judgments. It also admits some modularity in definitions and proofs. The rest of this chapter is devoted to illustrating these claims.

2.3.2 Extending the semantics

Sometimes the semantics of an extension to a language can be obtained simply by adding extra rules [FC94]. In some cases, proofs about the language can be extended simply by adding extra cases for the extra rules. However, some extensions (for instance, features that require the addition of an extra component in the transition system configurations) require every rule to be rewritten. In this case, every proof about the base language will have to be rewritten for the extended language.

I shall show how (using I-systems) a simple extension of P that requires an extra component in the configuration translates into a simple extension of a proof about P. The simple extension is to add variable assignment to P, and the extra component of the configuration is a store. It is well known that stores can be implemented in process calculi (see, e.g. [Mil89, ch8]), but I add them explicitly both for pedagogic reasons, and because P is not a real process algebra. The important point will be that the rules of P are included unchanged in those for the new language, despite the extra component.

Another important point is that we can apply this extension almost uniformly to each of $\mathcal{P}_I$, $\mathcal{P}_{SOS}$ and $\mathcal{P}_{CONG}$ without changing any of their rules.

Extending $\mathcal{P}_I$ with stores Let us call the new language P(:=). Let Var be a countably infinite set of variables ranged over by x, Val a set of values ranged over by v, and let *op* range over a set of binary operators *Op*. The grammar of

P(:=) is

$$\begin{aligned} e &::= v \mid x \mid e \ op \ e \\ p &::= 0 \mid l.p \mid p|p \mid (x := e).p \end{aligned}$$

Let *Exp* be the set of all expressions.

The semantics of P(:=) requires a global store component. The precise algebraic definition of stores $\sigma \in \mathit{Store}$ is given later in figure 5–1. The important operations are $\sigma(x)$ which returns the value of x in σ, and $\sigma[v/x]$ which updates the value of x to v.

The I-semantics of P(:=) is given by the transition system $TS_I^{(:=)}$ which equals $\langle \mathrm{P}(:=) \times \mathit{Store}, \rightarrow, \Omega \times \mathit{Store} \rangle$ where $\langle p, \sigma \rangle \rightarrow \langle q, \sigma' \rangle$ if and only if either $\mathcal{P}_I^{(:=)} \vdash p \rightarrow q, \sigma \rightsquigarrow \sigma'$ or $\mathcal{P}_I^{(:=)} \vdash p \rightarrow q$ and $\sigma = \sigma'$. The deduction system $\mathcal{P}_I^{(:=)} = (\mathcal{L}_I^{(:=)}, \mathcal{R}_I^{(:=)})$ is defined such that $\mathcal{L}_I^{(:=)}$ consists of the judgments of the form $p \rightarrow q$, $e \rightarrow v$ and $\sigma \rightsquigarrow \sigma'$, and the rules of $\mathcal{R}_I^{(:=)}$ are those of $\mathcal{R}_I$ plus the rules generated by the following schemata:

$$\frac{}{v \rightarrow v} \qquad \frac{}{x \rightarrow v} \overline{} \frac{\sigma(x) = v}{\sigma \rightsquigarrow \sigma} \qquad \frac{e_1 \rightarrow v_1 \qquad e_2 \rightarrow v_2}{e_1 \ op \ e_2 \rightarrow app(op, v_1, v_2)}$$

$$\frac{e \rightarrow v}{(x := e).p \rightarrow p} \overline{} \quad \frac{}{\sigma \rightsquigarrow \sigma[v/x]} \qquad \frac{\sigma \rightsquigarrow \sigma \qquad \sigma \rightsquigarrow \sigma'}{\sigma \rightsquigarrow \sigma'}$$

where $app : Op \times Val \times Val \rightharpoonup Val$ is a partial function that returns the value of applying *op* to its arguments. Note that the last rule for stores is not a general transitivity rule. As I show later (proposition 2.10), it limits transitions to write to the store at most once.

Thus P(:=) is just P plus an assignment action, which is atomic (avoiding problems of interference). So $(x := 0).(p| \cdots |p)$ where $p = (x := x + 1).0$ returns in x the number of copies of p in the program. For example, figure 2–1 gives the first two transitions of the program $(x := 0).(p|p)$.

Extending $\mathcal{P}_{SOS}$ and $\mathcal{P}_{CONG}$ with stores The I-semantic rules of P(:=) reuse those of P. Here I show how to extend the other two systems, reusing their rulesets. To extend $\mathcal{P}_{SOS}$ to $\mathcal{P}_{SOS}^{(:=)} = (\mathcal{L}_{SOS}^{(:=)}, \mathcal{R}_{SOS}^{(:=)})$, we define the language of judgments $\mathcal{L}_{SOS}^{(:=)}$ to be the set of judgments of form $p \xrightarrow{\alpha} q$, $e \rightarrow v$ and $\sigma \rightsquigarrow \sigma'$ and the ruleset $\mathcal{R}_{SOS}^{(:=)}$ to be $\mathcal{R}_{SOS}$ plus the rules above for stores and expressions with one modification. The rule atom for assignment becomes:

$$\frac{e \rightarrow v}{(x := e).p \xrightarrow{\tau} p}$$

To extend $\mathcal{P}_{CONG}$, we set $\mathcal{L}_{CONG}^{(:=)} = \mathcal{L}_I^{(:=)}$ and $\mathcal{R}_{CONG}^{(:=)}$ to be $\mathcal{R}_{CONG}$ plus the rules above without modification. The only change that has to be made is to the equivalence relation. Currently $\cong \subseteq \mathrm{P} \times \mathrm{P}$. We want $\cong \subseteq \mathrm{P}(:=) \times \mathrm{P}(:=)$. We do not have to change the definition of the equivalence, except insofar as the variables p and q range over P(:=) instead of P.

By contrast, if I had not used interacting rules, the extension of $\mathcal{P}_{SOS}$ and $\mathcal{P}_{CONG}$ would have rewritten every rule in the respective systems to carry around

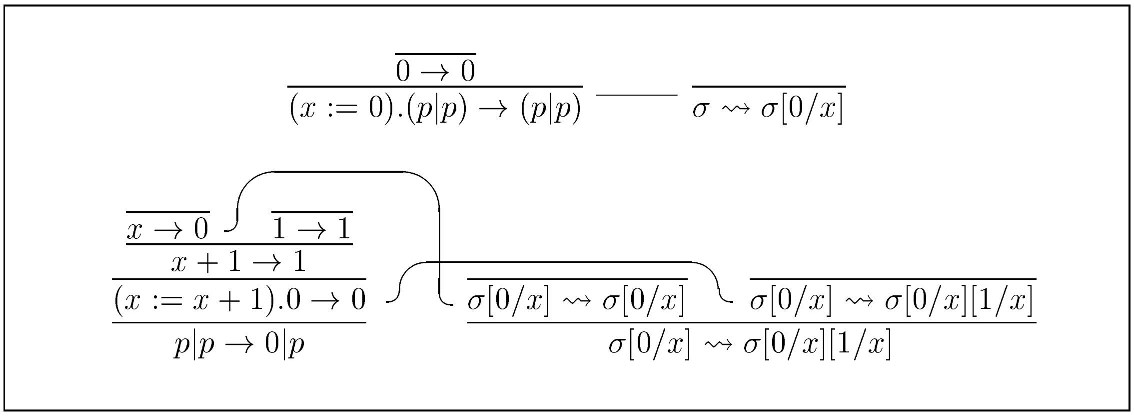


Figure 2–1: P(:=) process transitions of $(x := 0).(p|p)$ where $p = (x := x + 1).0$

a store component. (Section 5.2 discusses a similar treatment for environments.) Thus interacting rules can extend semantics in a more modular fashion than standard approaches.

2.3.3 Reasoning about semantics

We shall see that reasoning about I-semantics can be more modular too. Suppose we have a proof using TS_I that every P process is nondivergent, and that it proceeds by induction on the structure of P processes. Then the proof will concern the rules and judgments of $\mathcal{P}_I$. To prove an analogous result for P(:=) we should only require an extra case for the new production $(x := e).p$. The rule for this construct consists of two atoms, one concluding a process reduction judgment, and another a store transition judgment. Yet the property does not concern store transitions (stores do not store processes). Every $\mathcal{P}_I^{(:=)}$ transition consists of at most two subtransitions: one process transition and one store transition. For a P(:=) process to diverge, there must exist an infinite sequence of process transitions. The proof need only show that no process can be attributed an infinite sequence of transitions. Therefore the proof need only concern itself with the fragment of the rule concerning process reduction.

The key notion is "fragment". We shall see the respective proofs after the definition of fragment, and the proof of the proof fragmentation theorem (theorem II).

2.4 Fragments of deductions

The idea of fragmentation is quite straightforward. Suppose our community of mathematicians is cosmopolitan, and we wish to study the work of the Russians (which is distinguished by the type of facts they deduce). To study them, we have to screen out the work of everyone else. That is, we must isolate the Russian contribution to a deduction (their trees) and consider only how teams of Russians collaborate.

In this section I define rule and deduction fragments. The main result of this section is the proof fragmentation theorem.

Definition 2.4 (Fragment)

- *I-structure* (F_0, I_0) *is a* fragment *of* (F, I) *if* $F_0 \subseteq F$ *and* $I_0 \subseteq I$.
- *Rule* r_0 *is a* (rule) fragment *of rule* r *if* $r_0 \subseteq r$.
- *Ruleset* $\mathcal{R}_0$ *is a* (ruleset) fragment *of ruleset* $\mathcal{R}$ *if* $\mathcal{R}_0$ *contains only fragments of rules in* $\mathcal{R}$.
- I-*system* $(\mathcal{L}_0, \mathcal{R}_0)$ *is a* (system) fragment *of* I-*system* $(\mathcal{L}, \mathcal{R})$ *if* $\mathcal{L}_0 \subseteq \mathcal{L}$, *and* $\mathcal{R}_0$ *is a fragment of* $\mathcal{R}$.

Note that if (F_0, I_0) is a fragment of (F, I) then $I_0 \subseteq I \cap (O(F_0) \times O(F_0))$. This follows both from the definition of fragment and also because $I_0 \subseteq O(F_0) \times O(F_0)$ by the definition of I-structure. Thus a fragment of a structure consists of a subset of its forest and a subset of the interactions between the isolated trees. I define the process of taking fragments of I-structures rather than I-deductions because in general, fragments of deductions in $\mathbf{I}(\mathcal{T})$ will not be deductions in $\mathbf{I}(\mathcal{T})$.

Note also that taking a fragment of a rule does not alter the shape of individual atoms. Further, if $\mathcal{R}_0$ is a fragment of $\mathcal{R}$ then $\mathcal{R}_0$ need not be a subset of $\mathcal{R}$. Indeed, as I show in section 2.5, it is even possible that $\mathcal{R} \subseteq \mathcal{R}_0$ and $\mathcal{R}_0 \neq \mathcal{R}$. I write $(\mathcal{L}, \mathcal{R}) \subseteq (\mathcal{L}', \mathcal{R}')$ if $\mathcal{L} \subseteq \mathcal{L}'$ and $\mathcal{R} \subseteq \mathcal{R}'$.

Proposition 2.5 *The relation "is a fragment of" is a partial order over* I-*structures, rules, rulesets and systems.*

Proof: For I-structures and rules this follows trivially from the fact that $\subseteq$ is a partial order. For rulesets it follows from the fact that fragments of fragments of rules are fragments. For systems, it follows because $\subseteq$ is a partial order and "is a fragment of" is a partial order over rulesets. □

Proposition 2.6 *Let* (F_0, I_0) *be a fragment of* (F, I)*. Then if* (F, I) *satisfies DLF, so does* (F_0, I_0)*.*

Proof: This follows from the fact that $\lesssim_{(F_0,I_0)} \subseteq \lesssim_{(F,I)}$. □

The proof fragmentation theorem

We can consider system fragments to be modules. The proof fragmentation theorem shows that when we prove universal propositions about the deductions of modules they will remain true when the modules are composed. For example, we may show that every Russian deduction satisfies ϕ. Then the theorem entails that every Russian fragment of a cosmopolitan deduction will also satisfy ϕ. Furthermore, as shown in section 2.5.3, it can also be used to simplify proofs within a module.

To state the theorem, I define the set of $\mathcal{T}_0$-fragments of a $\mathcal{T}$-deduction when $\mathcal{T}_0$ is a fragment of $\mathcal{T}$. I say that $\mathbb{F}_{\mathcal{T}_0}(\Pi)$ is the set of fragments of Π such that every neighbourhood matches a rule instance of $\mathcal{T}_0$. I extend this to sets of deductions. Thus I write $\mathbb{F}_{\mathcal{T}_0}(X) = \bigcup_{\Pi \in X} \mathbb{F}_{\mathcal{T}_0}(\Pi)$

Theorem II (Proof fragmentation)

$$\mathbf{I}(\mathcal{T}_0) \supseteq \mathbb{F}_{\mathcal{T}_0}(\mathbf{I}(\mathcal{T})) \textit{ when } \mathcal{T}_0 \textit{ is a fragment of } \mathcal{T}.$$

Proof: Let $\Pi \in \mathbb{F}_{\mathcal{T}_0}(\mathbf{I}(\mathcal{T}))$. Then by proposition 2.6, Π satisfies DLF. By definition, every neighbourhood of Π matches a rule instance of $\mathcal{T}_0$, so by theorem I, $\Pi \in \mathbf{I}(\mathcal{T}_0)$. $\square$

This theorem proves that it is sound to reason universally about fragments of deductions by considering only the rule fragments that build them. Suppose that ϕ is a proposition about all $\mathcal{T}_0$-deductions. For example, proposition 2.10 concerns store deductions. Then the proof fragmentation theorem shows that the result remains true for every $\mathcal{T}_0$-fragment of a $\mathcal{T}$-deduction. Thus the store proposition is true for every store fragment of a $\mathcal{P}_I^{(:=)}$-, $\mathcal{P}_{SOS}^{(:=)}$- and $\mathcal{P}_{CONG}^{(:=)}$-deduction.

The proof technique does not extend to other kinds of sentences, such as existential sentences. Because $\mathbf{I}(\mathcal{T}_0)$ is bigger than $\mathbb{F}_{\mathcal{T}_0}(\mathbf{I}(\mathcal{T}))$ what may be shown to exist in $\mathbf{I}(\mathcal{T}_0)$ may not exist in $\mathbb{F}_{\mathcal{T}_0}(\mathbf{I}(\mathcal{T}))$.

So not every proposition about $\mathcal{T}_0$-fragments of $\mathcal{T}$-deductions will be provable in this way. (That is, it is sound to reason about rule fragments, but in general it is not complete.) It is not the case that $\mathbf{I}(\mathcal{T}_0) \subseteq \mathbb{F}_{\mathcal{T}_0}(\mathbf{I}(\mathcal{T}))$. This is because the other rule atoms in $\mathcal{T}$ may prevent certain $\mathcal{T}_0$ rule fragments from being applied. For example, consider the ruleset consisting of the single rule

$$\frac{}{A} \;\text{———}\; \frac{B}{C}$$

and consider the fragment consisting of $\overline{A}$. The set of deductions over the original ruleset is empty (we cannot infer B), whereas we do have the deduction $\overline{A}$ in the ruleset fragment.

It would be natural to try to find a characterization of the rule sets for which the inclusion does hold. However, I have not yet managed to do this. Similarly, one might wish to classify the kinds of propositions one can prove using this proof technique. Intuitively, we can only prove properties about system fragments that do not depend on the way the rest of the system interacts with the fragment. In chapter 4 we see the *proof assembly theorem* which overcomes this limitation.

2.5 Examples

This section shows how the theory of fragments helps reasoning about P processes. The first result I shall prove is the promised nondivergence proof of P. I shall then extend this to obtain a nondivergence proof of P(:=). In a similar vein, I shall also prove a simple property about stores, which remains true (without modification) irrespective of the different semantics for P. The section finishes with a proof of equivalence between LTS_{SOS} and TS_I.

Broken deductions and systems Let $\mathbb{B}(F, I) = (F, \emptyset)$ be the result of breaking every interaction link in an I-structure. Obviously $\mathbb{B}(\Pi)$ is a fragment of Π. Similarly, let $\mathbb{B}(\mathcal{L}, \mathcal{R}) = (\mathcal{L}, \{\{r_0\} |\ r_0 \in r \in \mathcal{R}\})$ be the result of breaking every interaction link in an I-system. Once again, $\mathbb{B}(\mathcal{T})$ is a fragment of $\mathcal{T}$. For example, $\mathcal{P}_B = \mathbb{B}(\mathcal{P}_I)$ consists of the following rules (note that the two halves of the communication rule are just instances of each other)

$$\frac{}{l.p \to p} \qquad \frac{p \to p'}{p|q \to p'|q} \qquad \frac{q \to q'}{p|q \to p|q'} \qquad \frac{p \to p' \qquad q \to q'}{p|q \to p'|q'}$$

Proposition 2.7 *If* $\Pi \in \mathbf{I}(\mathcal{T})$ *then* $\mathbb{B}(\Pi) \in I(\mathbb{B}(\mathcal{T}))$. □

2.5.1 Extending a nondivergence proof

In this section, we prove that no P process diverges, and then consider how to extend the proof to P(:=). The extension is quite straightforward, and requires less effort when using I-systems than ordinary SOS.

To be precise, we actually prove a non-nontermination proof: that is, that no process performs an infinite sequence of transitions. (Equivalently, that every process either terminates or deadlocks.) This is a stronger result than nondivergence, because divergence is that special case of nontermination which exhibits no visible computation. It is quite possible for a nonterminating process to perform work visibly: for instance it may output an infinite stream of approximations to some result. However, I use the term "nondivergence" because it is less of a mouthful than "non-nontermination".

Sequences I write X^ω for the set of countable, possibly empty sequences of elements from set X. I write ε for the empty sequence. I write sequences using an associative concatenation operator $\cdot$ (e.g., $u \cdot v \cdot \ldots \cdot w$). I use s to range over sequences. I regard elements of X as singleton sequences in X^ω. I write $|s|$ for the length of s, dom $(s) = \{1, \ldots, |s|\}$ and $s(i)$ for the ith element of s when $i \in$ dom (s).

For the following proofs, I say that a *deduction sequence* of $\mathcal{P}_I$ ($\mathcal{P}_B$) is a sequence of deductions $\Pi_1 \cdot \ldots \cdot \Pi_n$ (for $n \geq 1$) such that for $i = 1, \ldots, n$, $\Pi_i \vdash p_i \to p_{i+1}$, for some $p_1, \ldots, p_{n+1} \in \mathrm{P}$. I say $\mathcal{P}_I$ ($\mathcal{P}_B$) *attributes* a deduction sequence $\Pi_1 \cdot \ldots \cdot \Pi_n$ to p if $\Pi_1 \vdash p \to p'$ for some $p' \in \mathrm{P}$.

Proposition 2.8 *No process in* P *diverges.*

Proof: If $\mathcal{P}_I$ attributes a transition sequence to a process p, by definition there must exist a sequence of deductions that conclude the individual transitions. Therefore it suffices to consider deduction sequences.

I show by induction on the structure of p that the rules of $\mathcal{P}_B$ attribute an at most finite deduction sequence to p. **Case $\boldsymbol{p = 0}$** trivial. **Case $\boldsymbol{p = a.q}$** by induction, q has an at most finite deduction sequence s attributed by $\mathcal{P}_B$ (let n be the length of the maximal sequence). Then $a.q$ has a maximal deduction sequence of length $n+1$: $\overline{a.q \to q} \cdot s$ which is finite. **Case $\boldsymbol{p = q|q'}$** by induction, both q and q' are attributed at most finite deduction sequences, s of length n and s' of length m respectively. Then $q|q'$ will have a maximal deduction sequence $s \cdot s'$ of length $n + m$.

Now, let $p \in \mathrm{P}$. Suppose $\mathcal{P}_I$ attributes an infinite deduction sequence $\Sigma_1 \cdot \Sigma_2 \cdot \ldots$ to p. Let $\Sigma_i' = \mathbb{B}(\Sigma_i)$ for $i = 1, 2, \ldots$ be the fragments obtained after breaking every interaction link. By proposition 2.7 this would be an infinite $\mathcal{P}_B$-deduction sequence attributed to p: contradiction. $\square$

Extending the proof to P(:=)

Let $\mathcal{P}^{(:=)}_{PROC}$ be that fragment of $\mathcal{P}^{(:=)}_I$ containing the process transition and expression evaluation rule fragments (i.e., not containing the store atoms). Then the ruleset of $\mathcal{P}^{(:=)}_{PROC}$ contains the ruleset of $\mathcal{P}_I$. Furthermore, $\mathcal{P}^{(:=)}_{BPROC} = \mathbb{B}(\mathcal{P}^{(:=)}_{PROC})$ is just $\mathcal{P}_B$ plus formulae and rules for expression evaluation and the assignment rule atom.

To obtain a proof of nondivergence for the processes of P(:=), it suffices to change every occurrence of $\mathcal{P}_I$ with $\mathcal{P}^{(:=)}_{PROC}$ and $\mathcal{P}_B$ with $\mathcal{P}^{(:=)}_{BPROC}$, in both the definition of deduction sequence and the body of the proof, and then add the following case to the induction proof:

> **Case $\boldsymbol{p = (x := e).q}$** by induction, q has an at most finite sequence s of transitions attributed by $\mathcal{P}^{(:=)}_{BPROC}$ (let n be the length of the maximal sequence). Then $(x := e).q$ has a maximal sequence of transitions of length $n + 1$: $\overline{(x := e).q \to q} \cdot s$ which is finite.

Proposition 2.9 *No process in* P(:=) *diverges.* $\square$

There are two aspects to the work of proving theorems. The first aspect concerns the discovery of proofs, and the second concerns the verification of the discovered proof. When I prove theorems I tend to do both simultaneously: I use my intuitions about what correct proofs are to guide my proof search, and then at each stage I check to see if I have proved what I think I have proved. In a mathematical text the proof has already been discovered — but I still have to check it to believe it (a point of view expounded in [Pol95]). In general, proof discovery is harder and proof checking is easier.

The above extension is not modular in the sense that we have to check that the entire extended proof really is a proof. It is modular in the sense that we do not have to do very much to discover the proof. Because the set of productions in the grammar of P(:=) contains that of P, our first strategy should attempt to reuse the inductive cases of the previous proof. Because the ruleset of $\mathcal{P}^{(:=)}_{BPROC}$ contains that of $\mathcal{P}_B$, we should (at least at the first attempt) expect to reuse the inductive cases altering only the names of the I-systems. And in fact, this is what I did. The candidate extended proof consists of a substitution of names and an extra induction case.

By contrast, if I had attempted to extend a nondivergence proof using SOS then I should have had to rewrite the definition of deduction sequence and the proof itself to reason about judgments with state information. Moreover, since every rule mentions stores, we should have to reconsider every case just to see if they are influenced by the store. So in this sense my approach is more modular than the SOS approach: I simply extended the old proof without having to reconsider anything.

To a small extent, my approach will also require less work when the proof for an extended language requires a different proof strategy than the proof for the original language. For instance, suppose we subsequently introduce to the grammar an iterator production $\mathrm{It}^n C.p$ where n is a natural number and C belongs to a set of process constants, and a production for constants C, with the following semantic rules:

$$\mathrm{It}^0 C.p \;\rightarrow\; 0 \qquad\qquad \mathrm{It}^{n+1} C.p \;\rightarrow\; p[\mathrm{It}^n C.p/C]$$

(For the equivalent SOS rules, add τ superscripts to the arrows.) then this calculus would still be nondivergent, but the induction strategy would have to be altered: $p[\mathrm{It}^n C.p/C]$ is not in general structurally smaller than p (e.g., when $p = C$), so one cannot use structural induction. The modified proof would have to alter the induction hypothesis and reprove every case. (See section 4.3.2 for the proof of a similar result.) The idea of the new proof would be the same in both approaches, but the expression of the new proof would be simpler in my approach than in the SOS approach because of the absence of stores in judgments.

2.5.2 A modular proof about stores

The last section proved a result about P(:=) processes independently of the rules for stores. This section proves a result about stores independently of the rules for processes. The proposition is very simple: store transitions update at most one variable. One consequence of the proposition is that stores do not lose any information.

The Commentary on STANDARD ML [MT90, §2.8] presents some theorems on the evaluation of ML. The first theorem is just "the size of the domain of the store cannot shrink during evaluation". However, the proof of this theorem considers a very large number of rules, many of which do not directly influence the store but have to carry its state around (although the state and exception conventions hide this). In each case we have to show that the "carrying around" preserves the property. Only seven rules actually have to be considered because they directly concern the store.

One way to simplify the proof would be to formalise the state convention as a family of metarules, as [HMT90, §6.7]:

$$\frac{s_0, A_1 \vdash \ phrase_1 \ \Rightarrow \ A_1', s_1 \qquad s_1, A_2 \vdash \ phrase_2 \ \Rightarrow \ A_2', s_2 \qquad \cdots \qquad s_{n-1}, A_n \vdash \ phrase_n \ \Rightarrow \ A_n', s_n}{s_0, A \vdash \ phrase \ \Rightarrow \ A', s_n}$$

and then have one case that proved the "carrying around" result generally, by a subinduction on n, the number of premises. To check this proof correct, I should have to check that every "carrying around" rule does indeed match one of the metarules in the family.

Although ML and P(:=) are different languages, and although the ML semantics is evaluation-style and the one for P(:=) transition-style, the following proof of the theorem for P(:=) illustrates that I-semantics allow one to cut out irrelevant cases. There are only three rules that concern the store, and the proof contains only three cases.

Let $\mathcal{S}_I$ be the fragment of $\mathcal{P}_I^{(:=)}$ consisting of the following rule fragments:

$$\frac{}{\sigma \rightsquigarrow \sigma} \qquad\qquad \frac{}{\sigma \rightsquigarrow \sigma[v/x]} \qquad\qquad \frac{\sigma \rightsquigarrow \sigma \qquad \sigma \rightsquigarrow \sigma'}{\sigma \rightsquigarrow \sigma'}$$

The proof fragmentation theorem can be applied to the following proof to show that every $\mathcal{P}_I^{(:=)}$-, $\mathcal{P}_{SOS}^{(:=)}$-, and $\mathcal{P}_{CONG}^{(:=)}$-transition updates the store at most once.

Proposition 2.10 *If $\mathcal{S}_I \vdash \sigma \rightsquigarrow \sigma'$ then either $\sigma' = \sigma$ or there exists $x \in Var$ and $v \in Val$ such that $\sigma' = \sigma[v/x]$.*

Proof: by induction on the depth of inference of $\Sigma \in \mathbf{I}(\mathcal{S}_I)$ such that $\Sigma \vdash \sigma \rightsquigarrow \sigma'$. **Base cases:** by inspection, the lookup rule gives $\sigma' = \sigma$. The assignment rule gives $\sigma' = \sigma[v/x]$ for some v and x. **Induction step:** the case for the propagation rule follows immediately from the inductive hypotheses: the conclusion has the same shape as the right hand premise. $\square$

2.5.3 An equivalence theorem

This section shows that the I-semantics and the SOS of P are equivalent for the non-deadlocking processes. Once again, the proof uses the proof fragmentation theorem, but this time not for extensibility or modularity reasons. Instead it simplifies the proof: without fragments, even the easy half becomes very cumbersome.

This is actually a flaw of the definition of I-deduction. It is neither particularly elegant nor does it support simple inductive reasoning. Instead, (as we shall see) we always have to recover inductive reasoning using a special kind of fragmentation. One feels that this special notion of fragment should be somehow embodied in the definition of deduction; though at the moment it is not obvious how to do it. This problem will be rectified in chapter 4.

The proofs of the two directions of the equivalence each consist of two parts, only one of which is interesting. The uninteresting part concerns the problem of matching the single communication transitions of $\mathcal{P}_{SOS}$ to the multiple communication transitions of $\mathcal{P}_I$. The interesting part concerns how interactions correspond to communication.

To minimise the uninteresting part, I shall introduce an intermediate structured operational semantics (called $\mathcal{P}_{MSOS}$) which allows multiple communications per transition and simply state the appropriate equivalence results. These proofs are not hard, but are tedious. Then I shall prove $\mathcal{P}_{MSOS}$ equivalent to $\mathcal{P}_I$. Another advantage of having this intermediate semantics is that it shows very clearly the difference between SOS and I-semantics in general.

An intermediate semantics

The idea is that each transition judgment can be decorated with a multiset of labels (collections of silent actions will be represented by the empty multiset.) Figure 2–2 gives a definition of the multiset operations we shall be using.

Let $\mathcal{P}_{MSOS} = (\mathcal{L}_{MSOS}, \mathcal{R}_{MSOS})$ where $\mathcal{L}_{MSOS}$ is the set of judgments of the form $p \xrightarrow{m} q$, and $\mathcal{R}_{MSOS}$ is the set of rules generated by:

$$\frac{}{l.p \xrightarrow{\{\!|l|\!\}} p} \qquad \frac{p \xrightarrow{m} p'}{p|q \xrightarrow{m} p'|q} \qquad \frac{q \xrightarrow{m} q'}{p|q \xrightarrow{m} p|q'}$$

$$\frac{p \xrightarrow{m_1} p' \qquad q \xrightarrow{m_2} q'}{p|q \xrightarrow{m} p'|q'}\ \exists m' \subseteq (m_1 \cap \overline{m_2}). m = (m_1 \setminus m') \cup (m_2 \setminus \overline{m'})$$

The side condition on this last rule is quite complex, but as the following proof shows, it accurately captures the nature of communication in the I-semantics. Suppose m_1 and m_2 are the multiset of visible actions performed by two parallel transitions. Then there may be some communication between the two, in which case there will exist some submultiset m' of actions of m_1 whose complementation will be a submultiset of m_2. That is, $m' \subseteq m_1 \cap \overline{m_2}$. The visible actions of the combined transition will be the union of the visible actions of m_1 and m_2 which have not been communicated, i.e., $(m_1 \setminus m') \cup (m_2 \setminus \overline{m'})$.

Here I shall represent multisets m by their characteristic functions: functions from *Lab* to $\mathbb{N}$, the set of natural numbers. For each action l, $m(l)$ equals the number of occurrences of l in the multiset. Then the following operations can be defined:

Empty multiset	$\emptyset(l) = 0$
Singleton	$\{\!\mid l \mid\!\}(l') = \begin{cases} 1 & \text{if } l = l' \\ 0 & \text{ow} \end{cases}$
Multiset union	$(m_1 \cup m_2)(l) = m_1(l) + m_2(l)$
Multiset intersection	$(m_1 \cap m_2)(l) = \min(m_1(l), m_2(l))$
Multiset difference	$(m_1 \setminus m_2)(l) = \begin{cases} m_1(l) - m_2(l) & \text{if } m_1(l) > m_2(l) \\ 0 & \text{ow} \end{cases}$
Submultiset	$m_1 \subseteq m_2 \quad \text{iff} \quad \forall l. m_1(l) \leq m_2(l)$
Complementation	$\forall l. \overline{m}(l) = m(\bar{l})$

Note that complementation is the extension of label complementation to multisets of labels, and not complementation with respect to some universal multiset. Also, I shall write $\{\!\mid l_1, \ldots, l_n \mid\!\}$ for $\{\!\mid l_1 \mid\!\} \cup \ldots \cup \{\!\mid l_n \mid\!\}$.

Figure 2–2: Multisets

Proposition 2.11 *For all $p, p' \in \mathrm{P}$ and $l \in Lab$,*

$$\text{if } \mathcal{P}_{SOS} \vdash p \xrightarrow{l} p' \text{ then } \mathcal{P}_{MSOS} \vdash p \xrightarrow{\{\!\mid l \mid\!\}} p'$$
$$\text{if } \mathcal{P}_{SOS} \vdash p \xrightarrow{\tau} p' \text{ then } \mathcal{P}_{MSOS} \vdash p \xrightarrow{\emptyset} p'$$

□

A *trace* t is a sequence of labels. t *traces* m if for all l, $m(l)$ equals the number of occurrences of l in t. Note that t may contain no silent actions. A *t-traced deduction sequence* of $\mathcal{P}_{SOS}$ is a sequence of deductions consisting of $|t|+1$ subsequences, the first of which concludes a sequence of transitions $p_0(\xrightarrow{\tau})^* p_1$, and the last $|t|$ concludes a sequence of transitions $p_i(\xrightarrow{\tau})^* \xrightarrow{t(i)} (\xrightarrow{\tau})^* p_{i+1}$, for some $p_0, \ldots, p_{|t|+1} \in \mathrm{P}$. A t-traced deduction sequence is written $\mathcal{P}_{SOS} \vdash p_0 \xrightarrow{t} p_n$.

Proposition 2.12 *For all $p, p' \in \mathrm{P}$ and $m \in \mathbb{N}^{Lab}$, if $\mathcal{P}_{MSOS} \vdash p \xrightarrow{m} p'$ then for all traces t of m, $\mathcal{P}_{SOS} \vdash p \xrightarrow{t} p'$.* □

The interesting equivalence proofs

The following proofs require a different fragment of $\mathcal{P}_I$ than the nondivergence proof. Let $\mathcal{P}_F$ be the fragment of $\mathcal{P}_I$ generated by:

$$\frac{}{a.p \to p} \;\text{———}\; \frac{}{\bar{a}.q \to q} \qquad\qquad \frac{}{l.p \to p}$$

$$\frac{p \to p'}{p|q \to p'|q} \qquad \frac{q \to q'}{p|q \to p|q'} \qquad \frac{p \to p' \qquad q \to q'}{p|q \to p'|q'}$$

The difference between $\mathcal{P}_I$ and $\mathcal{P}_F$ is the inclusion of the rule $\overline{l.p \to p}$. The point is that $\mathcal{P}_F$-deductions can contain both broken and unbroken interaction links. This allows me to either break or put in interaction links as the proof progresses.

For the proofs, I introduce the following notations. Let $\Pi \in \mathbf{I}(\mathcal{P}_F)$. Then it will contain a number of instances of prefix rule instances. I write $\Pi[m]$ if m is the multiset of labels l that occur as prefixes in prefix rule instances $\overline{l.p \to p}$ of Π. (Thus m records the broken interaction links in Π.) In a straightforward extension of notation, I write $\Pi[m] \vdash \Gamma$ if Γ is the set of conclusion occurrences of $\Pi[m]$. I say that an occurrence A of (F, I) is *noninteracting* if for all $(A, B) \in I$, $A = B$.

Lemma 2.13(i) *For all $p, p' \in \mathrm{P}$ and $m \in \mathbb{N}^{lab}$, $\mathcal{P}_{MSOS} \vdash p \xrightarrow{m} p'$ if and only if there exists a $\Pi \in \mathbf{I}(\mathcal{P}_F)$ such that $\Pi[m] \vdash p \to p'$.*

Proof: $\Rightarrow$**:** by induction on the depth of inference of $p \xrightarrow{m} p'$. **Case $\boldsymbol{l.p \xrightarrow{\{\!|\, l \,|\!\}} p}$:** when we have the $\mathcal{P}_F$-deduction $\overline{l.p \to p}[\{\!|\, l \,|\!\}]$.

Case $\boldsymbol{p|q \xrightarrow{m} p'|q}$: this follows from the rule with premise $p \xrightarrow{m} p'$. By induction we have $\Pi[m] \vdash p \to p'$. The result follows by applying the appropriate I-rule to Π to conclude $p|q \to p'|q$. **Case $\boldsymbol{p|q \xrightarrow{m} p|q'}$:** symmetrical to the last case.

Case $\boldsymbol{p|q \xrightarrow{m} p'|q'}$: this follows from the rule with the two premises $p \xrightarrow{m_1} p'$ and $q \xrightarrow{m_2} q'$. By induction, we have $\Pi_1[m_1] \vdash p \to p'$ and $\Pi_2[m_2] \vdash q \to q'$ in $\mathcal{P}_F$. Let $m' \subseteq (m_1 \cap \overline{m_2})$ be such that $m = (m_1 \setminus m') \cup (m_2 \setminus \overline{m'})$. Then we can build the deduction $\Pi_3[m] \vdash p|q \to p'|q'$ in two steps. First, we add interaction links corresponding to the communications specified by m. Let $f_1 : Lab \to \wp(O(\Pi_1))$ be such that $f_1(l)$ isolates $m'(l)$ noninteracting l-communication occurrences in Π_1. Furthermore, let $f_2 : Lab \to \wp(O(\Pi_2))$ be such that $f_2(l)$ isolates $\overline{m'}(l)$ noninteracting $\bar{l}$-communication occurrences in Π_2. Then since for all l, $m'(l) = \overline{m'}(\bar{l})$, there exists a family of bijections $g_l : f_1(l) \leftrightarrow f_2(\bar{l})$. We set I_m to be the reflexive, symmetric closure of $\{(A, B) \mid \exists l. g_l(A) = B\}$, and then add I_m to $\Pi_1 \cup \Pi_2$. The resulting set of interaction links will still form an equivalence relation because in $\mathcal{P}_F$-deductions, each equivalence class consists of at most two conclusions. The second step is simply to apply the parallel composition rule to get the result.

$\Leftarrow$**:** By induction on the depth of inference of $\Pi[m] \vdash p \to p'$. **Case $\boldsymbol{\overline{l.p \to p}}$:** we get $l.p \xrightarrow{\{\!|\, l \,|\!\}} p$. **Case $\boldsymbol{p|q \to p'|q}$:** case follows easily by induction. **Case $\boldsymbol{p|q \to p|q'}$:** similar.

Case $\boldsymbol{p|q \to p'|q'}$: suppose $\Pi[m] \vdash p|q \to p'|q'$. This is inferred from a $\mathcal{P}_F$ deduction of $p \to p', q \to q'$. We break all the links between the deductions of these two conclusions, from which we obtain two $\mathcal{P}_F$ deductions $\Pi_1 \vdash p \to p'$ and $\Pi_2 \vdash q \to q'$. Now suppose the broken links were between pairs of actions $(l_1, \bar{l}_1), \ldots, (l_n, \bar{l}_n)$ (for $n \geq 0$). Let $m' = \{\!|\, l_1, \ldots, l_n \,|\!\}$. Then we get $\Pi_1[m_1 = m_1' \cup m']$ and $\Pi_2[m_2 = m_2' \cup \overline{m'}]$ for some m_1' and m_2' such that $m = m_1' \cup m_2'$. By induction we get $\mathcal{P}_{MSOS} \vdash p \xrightarrow{m_1} p'$ and $\mathcal{P}_{MSOS} \vdash q \xrightarrow{m_2} q'$. We can apply the parallel composition rule to get $\mathcal{P}_{MSOS} \vdash p|q \xrightarrow{m} p'|q'$. $\Box$

Note that even though there is no case for the I-rule $\overline{a.p \to p}$—$\overline{\bar{a}.q \to q}$ in the above induction, I still make essential use of its existence in $\mathcal{P}_F$. It allows me to take $\mathcal{P}_I$-deductions, and then strip away interaction links at my convenience. This allows me to identify the interactions between inference trees that have to be communicated together in $\mathcal{P}_{MSOS}$.

Proposition 2.13 *For all* $p, p' \in \mathrm{P}$, $\mathcal{P}_{MSOS} \vdash p \xrightarrow{\emptyset} p'$ *if and only if* $\mathcal{P}_I \vdash p \rightarrow p'$.

Proof: Follows by proof fragmentation and the fact that $\mathbf{I}(\mathcal{P}_I) \subseteq \mathbb{F}_{\mathcal{P}_F}(\mathbf{I}(\mathcal{P}_I))$. □

From propositions 2.11, 2.12 and 2.13 it follows that

$$\mathcal{P}_I \vdash p \rightarrow^{+} p' \quad \text{iff} \quad \mathcal{P}_{SOS} \vdash p(\xrightarrow{\tau})^{+}p'$$

2.6 Chapter summary

The basic theory of interacting deductions introduces the concept of interaction between groups of inference trees (or mathematicians trying to prove individual results). An interaction link between two occurrences denotes a synchronization of ideas. This allowed us to separate the various parts of a semantic judgment to achieve syntactic simplicity and propagation freeness in semantic rules (section 2.3.1). This made it particularly easy to extend semantic definitions (section 2.3.2). To exploit this simplification and extensibility in proofs about these semantic definitions, we needed the proof fragmentation theorem (section 2.4). This simplified the task of proof extension (section 2.5.1) and improved modularity (section 2.5.2). However, its use in the equivalence result (section 2.5.3) stemmed from the fact that it is not a simple matter to reason inductively from the definition of I-deduction. Though easily remedied using fragmentation, it indicates that the definition is not as good as it should be: somehow it ought to include the notion of fragmentation. We see this in chapter 4.

First though, we have seen how to use a theory of interacting deductions to give transition semantics. The next chapter asks if we can also give evaluation semantics.

Chapter 3

Evaluation Semantics and Sequential Deductions

Evaluation semantics [Hen90] (elsewhere called Natural Semantics [Kah87], Relational semantics [MT92], Martin Löf-style operational semantics [Abr93] and Big-Step semantics [Ast91]) enjoy a number of advantages over transition semantics. They tend to be more concise, less detailed and easier to reason about [Ber91, ch4]. They can also often give more direct semantics to features such as while loops. Practically, they have been used to give semantics to ML and EML [KST94], as the basis of theories of program animation [Ber91], execution profiling [San94], debugging [DS92] and compiler generation [DJ86].

I-systems can describe transition semantics; can they also describe evaluation semantics? We present an evaluation I-semantics for P which answers this question affirmatively. However, P is a very simple language — can we give evaluation semantics to more complex languages, for example containing sequential composition? In fact we can, in at least two different general ways. However, neither approach is easy to understand. Therefore we introduce a notion of sequentiality into the metatheory of deduction, giving QI-deduction. This is harmless, because every QI-system can be coded into an equivalent (albeit less perspicuous) I-system.

We obtain an evaluation semantics for P extended with sequential composition, which is more concise, and more readily comprehensible than a transition semantics. However, the real test is whether or not the evaluation semantics can simplify proofs too. We give a simple translation correctness result which verifies that at least for some applications, it does.

The structure of this chapter is as follows. In section 3.1 we consider how existing techniques for capturing sequential composition in ordinary evaluation semantics can be appropriated in I-systems. In section 3.2 we define the QI-deductions, which are just the I-deductions extended with a notion of sequencing. We show that QI-systems can be coded into equivalent I-systems. Section 3.3 describes a QI-system for P(;) — P plus sequential composition, and uses it to discuss various aspects of QI-deduction. Last, section 3.4 compares the pragmatic value of our evaluation semantics with that of a transition semantics by testing how easily they can prove a simple translation correctness result.

An evaluation semantics of P

Our language P has a very simple evaluation-style I-semantics $\mathcal{P}_I^E$:

$$\frac{}{0\surd} \qquad \frac{p\surd}{l.p\surd} \qquad \frac{q\surd}{\bar{l}.q\surd} \qquad \frac{p\surd \qquad q\surd}{p|q\surd}$$

The judgment $p\surd$ means "p terminates" — i.e., it is equivalent to the sequence of $\mathcal{P}_I$-judgments abbreviated $p \to^* q$ for $q \in \Omega$. This can be shown formally, but it is not important here.

This example illustrates how evaluation semantics can be more concise than transition semantics. There is just one rule for each combinator. I think it is tidier than the transition semantics. It also highlights two important points. Consider the process $a.b.0$. We get $\mathcal{P}_{SOS} \vdash a.b.0 \xrightarrow{a \cdot b} 0$. That is, the evaluation of $a.b.0$ will first perform a and then b and then stop. Now consider the deduction fragment Π of $\mathcal{P}_{QI}$:

$$\cfrac{\cfrac{0\surd}{b.0\surd}}{a.b.0\surd}$$

We have $a.b.0\surd <_\Pi b.0\surd <_\Pi 0\surd$. If we say the *action* of a judgment $l.p\surd$ is l then $<_\Pi$ attributes the following sequence of actions to Π: $a \cdot b$. We can then say that Π attributes this sequence of actions to the conclusion of Π, $a.b.0\surd$. The first point is that the preorder of Π naturally captures the evaluation order of $a.b.0$. The second point is that this order is opposite to the order in which the judgments were deduced.

Adding sequential composition

When we add a sequential composition operator ';' to P, it becomes more difficult to give an evaluation semantics. Let P(;) be P plus ';'. Its grammar is

$$p ::= 0 \mid l.p \mid p|p \mid p;p$$

and its SOS transition rules are:

$$\frac{}{l.p \xrightarrow{l} p} \qquad \frac{p \in \Omega \qquad q \xrightarrow{\alpha} q'}{p;q \xrightarrow{\alpha} q'} \qquad \frac{p \xrightarrow{\alpha} p'}{p;q \xrightarrow{\alpha} p';q}$$

$$\frac{p \xrightarrow{\alpha} p'}{p|q \xrightarrow{\alpha} p'|q} \qquad \frac{q \xrightarrow{\alpha} q'}{p|q \xrightarrow{\alpha} p|q'} \qquad \frac{p \xrightarrow{l} p' \qquad q \xrightarrow{\bar{l}} q'}{p|q \xrightarrow{\tau} p'|q'}$$

Note that the set of terminal states of P(;) is just the set of terminal states of P, Ω. I call this system $\mathcal{P}_{SOS}^{(;)} = (\mathcal{L}_{SOS}^{(;)}, \mathcal{R}_{SOS}^{(;)})$. It is a straightforward extension of $\mathcal{P}_{SOS}$.

Unfortunately I cannot give an evaluation-style I-semantics to P(;) as simply as I could give one to P. In the next section, I survey four methods of capturing sequentiality in standard evaluation semantics. I shall discuss why they are not appropriate, thereby motivating the introduction of extra sequencing technology in the meta-theory.

3.1 Sequential composition in evaluation semantics

This section considers some different ways that sequential composition has been modeled in evaluation semantics. I call them *cutting intermediate states*, *control stacks*, *coding* and *convention*. Each approach either fails or requires complex judgments. Therefore I argue for a meta-theoretic treatment of sequencing, independently of particular semantics.

3.1.1 Cutting intermediate states

The most common way to model sequential composition in an evaluation semantics is to "cut intermediate states". If a program phrase exp_1 is executed before exp_2, then the machine state immediately after executing exp_1 is the state immediately before executing exp_2. The intermediate state is produced and then consumed, and is not mentioned in (i.e., it is "cut out of") the overall evaluation judgment of the composed program $exp_1; exp_2$.

This is how sequentiality is captured in *The Definition of Standard ML* [HMT90]. In ML, sequential composition is a derived form: its semantics is given by the call-by-value interpretation of function application (defined by the rules in [HMT90, §6.7]). However, if it did have a rule for sequential composition, it would be something like:

$$\frac{s, E \vdash exp_1 \Rightarrow v, s' \qquad s', E \vdash exp_2 \Rightarrow v', s''}{s, E \vdash exp_1; exp_2 \Rightarrow v', s''}$$

where E is an environment associating identifiers with their values and s, s' and s'' are store states. A judgment of form $s, E \vdash exp \Rightarrow v, s'$ means that exp is evaluated in environment E and store s to value v, having changed the state to s'. In this rule s is the initial state, s'' the terminal state, and s' is the intermediate state that has been cut out of the conclusion.

This technique does not work directly in an I-semantics. P(;) processes do not interact with stores; but even if they did (e.g., in a similar manner to P(:=)), cutting intermediate states would not guarantee that the interaction links above the 'sequenced' premises would likewise be sequenced. Suppose our semantic judgments were of form $s, p\surd, s'$, and meant that p terminated transforming store s to s'. The proposed rule

$$\frac{s, p\surd, s' \quad s', q\surd, s''}{s, p; q\surd, s''}$$

would not stop the inference trees above the premises interacting. Thus we could have Π such that $\Pi \vdash s, a.0; \bar{a}.0\surd, s$ — which is wrong. If $a.0$ is evaluated before $\bar{a}.0$, they cannot interact.

One way round this difficulty is to introduce an artificial notion of 'state' that captures the global ordering of interaction links. The following four rules capture this:

$$\frac{}{[\kappa]\ 0\surd\ [\kappa]} \qquad \frac{[\kappa_1, n]\ p\surd\ [\kappa_1']}{[\kappa_1]\ l.p\surd\ [\kappa_1']} \quad \rule{2.5em}{0.4pt} \quad \frac{[\kappa_2, n]\ q\surd\ [\kappa_2']}{[\kappa_2]\ \bar{l}.q\surd\ [\kappa_2']}\ \text{if } n > \max(\kappa_1 \cup \kappa_2)$$

$$\frac{[\kappa]\ p\surd\ [\kappa_1] \qquad [\kappa]\ q\surd\ [\kappa_2]}{[\kappa]\ p|q\surd\ [\kappa_1 \cup \kappa_2]} \qquad \frac{[\kappa]\ p\surd\ [\kappa_1] \qquad [\kappa_1]\ q\surd\ [\kappa_2]}{[\kappa]\ p;q\surd\ [\kappa_2]}$$

The sets of natural numbers κ (note I have used κ, n to abbreviate $\kappa \cup \{n\}$) are the artificial states. Each member of an artificial state records the timestamp of an interaction link (a link is timestamped at the communication rule by n). Thus the evaluation judgment $[\kappa]\ p\surd\ [\kappa']$ is intended to mean that "the evaluation of p commences after the interactions recorded in κ and terminates after the interactions recorded in κ'". A straightforward proof shows that if $[\kappa]\ p\surd\ [\kappa']$ is deduced in the above system then $\kappa \subseteq \kappa'$. For example, (where I elide set brackets for legibility)

$$\dfrac{\dfrac{\dfrac{[1,2]\ 0\surd\ [1,2]}{[1]\ b.0\surd\ [1,2]}}{[\emptyset]\ a.b.0\surd\ [1,2]} \qquad \dfrac{\dfrac{[1]\ 0\surd\ [1]}{[\emptyset]\ \bar{a}.0\surd\ [1]} \qquad \dfrac{[1,2]\ 0\surd\ [1,2]}{[1]\ \bar{b}.0\surd\ [1,2]}}{[\emptyset]\ \bar{a}.0;\bar{b}.0\surd\ [1,2]}}{[\emptyset]\ a.b.0|\bar{a}.0;\bar{b}.0\surd\ [1,2]}$$

deduces $[\emptyset]\ a.b.0|\bar{a}.0;\bar{b}.0\surd\ [1,2]$, which says that when the process $a.b.0|\bar{a}.0;\bar{b}.0$ executes, it performs two interactions. (The actual numbers 1 and 2 have no independent meaning.) Thus the above system can sequence interactions correctly. To show that it enforces sequentiality, consider the following attempted deduction of $[\emptyset]\ a.b.0|\bar{b}.0;\bar{a}.0\surd\ [\kappa]$.

$$\dfrac{\dfrac{\dfrac{[1,2]\ 0\surd\ [1,2]}{[1]\ b.0\surd\ [1,2]}}{[\emptyset]\ a.b.0\surd\ [1,2]} \qquad \dfrac{\dfrac{[2]\ 0\surd\ [2]}{[\emptyset]\ \bar{b}.0\surd\ [2]} \qquad \dfrac{[1,2]\ 0\surd\ [1,2]}{\boxed{[2]\ \bar{a}.0\surd\ [1,2]}}}{[\emptyset]\ \bar{b}.0;\bar{a}.0\surd\ [1,2]}}{[\emptyset]\ a.b.0|\bar{b}.0;\bar{a}.0\surd\ [1,2]}$$

The problem here occurs at the boxed occurrence: we cannot apply the communication rule here because $1 \not> 2$. Thus the first example suggests that this system can sequence evaluations correctly, and the second suggests that it cannot sequence evaluations wrongly.

A major point is that this approach does not depend in any way on the language P(;). This is both good and bad. It is good because this means the technique can be used for any language, which in turn means that the basic theory of interacting deductions is powerful enough to give both transition and evaluation semantics to a wide class of languages.

It is bad precisely because the states are artificial: they are not intuitive semantic objects. The stores of P(:=) are intuitive, because stores occur in implementations. But the artificial states would correspond to nothing in an implementation. They are purely declarative. This makes it hard to understand why the technique works: especially since the timestamps correspond to nothing in the language (in the communication rule, where does n come from?). In fact, they relate to the structure of deductions, not to programs.

Moreover, they are bad because they clutter semantic judgments, and have to be propagated everywhere. They cannot be fragmented away like stores because their function is to timestamp the interactions of the trees they occur in. We cannot even appropriate the *state convention* of [HMT90, §6.7] to aid legibility, because our language also has concurrency (see section 3.1.4).

Since my aim is to simplify both the presentation of and proofs about operational semantics, I need a more intuitive semantics for sequential composition that does not clutter judgments and does not require propagation. Moreover, it seems proper to separate information about the structure of deductions (meta-theoretic information) from semantic judgments (object-theoretic information).

3.1.2 Control stacks (or continuations)

An alternative technique is to use a kind of syntactic continuation mechanism, like the *control stacks* used in the SECD MACHINE [Lan64]. This is not a widespread technique, so figure 3–1 gives a simple example: an evaluation semantics for a stack-based expression evaluator.

This technique requires much more effort than a normal semantics for expressions:

$$\frac{}{n \Rightarrow n} \qquad\qquad \frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2}{e_1\ op\ e_2 \Rightarrow app(op, n_1, n_2)}$$

where the judgments are simpler, more perspicuous, and easier to reason about, because the semantics is more abstract. The "continuation-style" semantics is more like an abstract machine. It does not feel like a true evaluation semantics.

This example also underlines the points I made earlier about preorders of deductions. In the deduction Π of figure 3–1, $(1+3)$ was evaluated before 2. Yet the portion of the evaluation concerning the evaluation of 2 is higher up the tree than that evaluating $(1+3)$. That is,

$$[(1+3), 2, -], \varepsilon \Rightarrow 2 \qquad <_\Pi \qquad [2, -], [4] \Rightarrow 2$$

So, if we say the action of $e \cdot C, S \Rightarrow n$ is e (a reasonable thing to do), then $<_\Pi$ sequences the action $(1+3)$ before the action 2. Once again, the preorder (which is opposite to the order in which judgments are deduced) corresponds to the evaluation order.

Let us assume a simple language of expressions:

$$e ::= n \mid e \; op \; e$$

where $n \in \mathbb{Z}$, and for some set Op of operators, $op \in Op$. The semantic rules for evaluation are:

$$\frac{}{\varepsilon, n \cdot S \Rightarrow n} \qquad \frac{C, n \cdot S \Rightarrow n}{n \cdot C, S \Rightarrow n} \qquad \frac{e_1 \cdot e_2 \cdot op \cdot C, S \Rightarrow n}{e_1 \; op \; e_2 \cdot C, S \Rightarrow n}$$

$$\frac{C, app(op, n_1, n_2) \cdot S \Rightarrow n}{op \cdot C, n_2 \cdot n_1 \cdot S \Rightarrow n}$$

where C is the control stack, or syntactic continuation of the values on S, the value stack. $app : Op \times \mathbb{Z} \times \mathbb{Z} \rightharpoonup \mathbb{Z}$ is a partial function meant to capture the "natural interpretation" of an expression. Thus $app(+, 1, 1)$ means 2, etc. A judgment $C, S \Rightarrow n$ means that the continuation C of S results in value n. It is not hard to see that in an expression $e_1 \; op \; e_2$, e_1 is evaluated before e_2. A simple example is

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\varepsilon, [2] \Rightarrow 2}}{[-], [2,4] \Rightarrow 2}}{[2, -], [4] \Rightarrow 2}}{[+, 2, -][3, 1] \Rightarrow 2}}{[3, +, 2, -], [1] \Rightarrow 2}}{[1, 3, +, 2, -], \varepsilon \Rightarrow 2}}{[(1+3), 2, -], \varepsilon \Rightarrow 2}}{[(1+3) - 2], \varepsilon \Rightarrow 2}$$

where I have used $[a, b, c]$ as an abbreviation for $a \cdot b \cdot c \cdot \varepsilon$.

Figure 3–1: A simple "continuation-style" evaluation semantics

Control Stacks and Concurrency

If I tried to use this technique to give semantics to a concurrent language, I would quickly run into complications owing to concurrency. The overheads would be so large I would be better using a transition semantics.

However, the situation is greatly improved when using I-systems: we can elide detail by distributing the control stack across several judgments. The following semantics is based upon an observation of Pietro Cenciarelli. Here judgments concern *process occurrences*, which we denote by pairs (p, n), where p is a process and n a natural number. The judgments come in two forms: the suspended form $(p, n)\surd(q, m)$, intended to mean "when activated, process occurrence (p, n) terminates before process occurrence (q, m) commences," and the active form $p\surd(q.m)$ which means that this occurrence of p will terminate before occurrence (q, m) begings. Thus (q, m) can be seen as a pointer to the next evaluation judgment in the distributed control stack. The semantic rules are:

$$\frac{}{0\surd(0,n)} \qquad \frac{}{0\surd(p,n)} \quad \rule{3em}{0.4pt} \quad \frac{(p,n)\surd(q,m)}{(p,n)\surd(q,m)} \qquad \frac{p\surd(q,m)}{(p,n)\surd(q,m)}$$

$$\frac{p\surd(q,m)}{l.p\surd(q,m)} \quad \rule{3em}{0.4pt} \quad \frac{p'\surd(q',m')}{\bar{l}.p'\surd(q',m')}$$

$$\frac{p\surd(r,m) \qquad q\surd(r,m)}{p|q\surd(r,m)} \qquad\qquad \frac{p\surd(q,n) \qquad (q,n)\surd(r,m)}{p;q\surd(r,m)}\ n \text{ fresh}$$

(Note that the sequential composition rule looks like it is cutting intermediate states, but it does not — q is not cut out of the conclusion.) The key rule is the second one. Its function is to ensure that every parallel process sequenced before p terminates before p commences. For instance, consider the process $(a.0|b.0); c.0$. The I-deduction fragment Π corresponding to its evaluation is:

$$\dfrac{\dfrac{\dfrac{0\surd(c.0,2)}{a.0\surd(c.0,2)} \qquad \dfrac{0\surd(c.0,2)}{b.0\surd(c.0,2)}}{a.0|b.0\surd(c.0,2)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{c.0\surd(0,1)}}{(c.0,2)\surd(0,1)}}{(c.0,2)\surd(0,1)}}{(c.0,2)\surd(0,1)}}{(a.0|b.0);c.0\surd(0,1)}$$

The second rule is used twice to deduce the two $0\surd(c.0, 2)$ judgments. Without it, the evaluation of $a.0$ and $b.0$ could not be deduced. Now, let us say that a judgment is *invisible* when it occurs as a conclusion of the second rule. Invisible occurrences do not correspond to any kind of computation: they just "mark time". Let $(c.0, 2)\surd(0, 1)^3$ be the only visible occurrence of $(c.0, 2)\surd(0, 1)$ in the above fragment (i.e., the highest). Then we see that both $(a.0\surd(c.0, 2)) <_\Pi ((c.0, 2)\surd(0, 1)^3)$ and $(b.0\surd(c.0, 2)) <_\Pi ((c.0, 2)\surd(0, 1)^3)$. Once again the preorder of Π captures the evaluation order.

Of course, these rules do not prohibit us from deducing an arbitrary number of conclusions of form $0\surd p$, but since these deductions do not tell us anything interesting about $\mathrm{P}(;)$ we can ignore them.

Once again, it is straightforward to show that the judgment $a.b.0|(\bar{a}.0;\bar{b}.0)\surd(0,1)$ is deducible, and that $a.b.0|(\bar{b}.0;\bar{a}.0)\surd(0,1)$ is not. This suggests that the above system can sequence correctly, and does not sequence wrongly.

This semantics is much simpler than a traditional control stack semantics owing to the distribution of the stack. If one identifies control stacks with continuations, one can imagine evaluation semantics for continuation-handling operations such as *call/cc* [Cli85,FFHD87]. I have not pursued this topic, but it seems possible. Though it is not clear how continuations mix with concurrency, one might use this technique for sequential languages.

This semantics mentions only naturally occurring semantic objects. So in this sense, it is more natural than the previous semantics. Once again the technique will work for any language. However, it is quite awkward to have to talk about process occurrences. (To avoid them we should have to annotate judgments, cluttering syntax.) Further, once again, sequencing information has to be propagated everywhere. Moreover, it is not perspicuous: one has to think quite hard before one believes this technique works. So even though this style of semantics may have certain applications (e.g., evaluation semantics for continuation-handling languages), it is overkill for sequential composition. Everybody understands sequential composition. It is simple and intuitive. We should like a simple and intuitive characterization of it.

3.1.3 Coding

A third technique is to code sequential composition using other primitives. For example, sequential composition is a derived form in ML: it is coded using function application (appendix A of [HMT90]). However, this only postpones the problem of defining sequentiality. We still have to model call-by-value application: first evaluate the function to its primitive form, then its argument, and then evaluate the body of the function after assigning its parameter to the value of the argument.

It is also coded in CONCURRENT ML [Rep91b], slightly differently than above, but again relying on the call-by-value nature of function application.

Similarly, it can be coded in the CCS. Milner [Mil89, §8.1] defines the *Before* operator using a special communication protocol of "done" signals. In section 3.4, I show that P(;) can be coded into P in a similar way. I could say that the compiler gives the meaning to ";" [Gar63], but there are two difficulties. First, the compilation is complex, and I should like some way to prove that it is correct. To do this, I require direct semantics for both P and P(;). Second, not every language is able to implement sequential composition — for instance, BASIC.

3.1.4 Conventions

A fourth technique is to alter the meaning of inference rules by imposing an order on the premises. An example of this may be found in Berry's work on program animation [Ber91]. However, this is usually justified by a convention about intermediate states: if there are none, then the order does not matter; otherwise the convention is that the premises are ordered according to the order implied by the intermediate states. In the definition of Standard ML, the *state convention* is such an ordering of rule premises. Its function is to simplify the presentation of rules by eliding state information from judgments (except where rules concern state explicitly). The conventional ordering allows one to put state information back in mechanically.

At first sight, this could work: we know how to introduce artificial state information to model sequentiality. Moreover, we can do this almost without knowing anything about the language itself. Our problem is that the premises of a parallel composition rule cannot be ordered — so we cannot appropriate the state convention for P(;) unless we can distinguish premises that are to be ordered from those that are not.

3.1.5 Solution

The previous discussion has led us to conclude that we need an intuitive, but formal and direct way to capture sequentiality without excluding parallelism. The problem is that we must sequence the interactions between inference trees. I introduce a proof-theoretic mechanism to do this.

I introduce some notation first, and formalise its meaning later. The notation indicates that one premise is "sequenced before" another in a rule:

$$\frac{p\surd \quad \blacktriangleright \quad q\surd}{p;q\surd}$$

This rule is intended to say that the evaluation of p must conclude before that of q commences. The view that evaluation order is opposite to deduction order suggests that $q\surd$ ought to be deduced before the deduction of $p\surd$ commences.

(Note that the black triangle $\blacktriangleright$ is not intended to be read as a logical symbol such as consequence. The above rule does not contain one premise that says "$q\surd$ is a consequence of $p\surd$". The triangle occurs between judgments.)

However, $\blacktriangleright$ is not just a side-condition: it adds information. Once an inference tree has been sequenced before another, the trees that interact with the first must also be sequenced before those that interact with the second. Therefore, asserting that one tree must be sequenced before another constrains the way non-local trees can be sequenced too. The following structure violates this condition:

$$\dfrac{\dfrac{\dfrac{}{0\surd}}{a.0\surd} \;\blacktriangleright\; \dfrac{\dfrac{}{0\surd}}{b.0\surd}}{a.0;b.0\surd} \quad\text{---}\quad \dfrac{\dfrac{\dfrac{}{0\surd}}{\bar{b}.0\surd} \;\blacktriangleright\; \dfrac{\dfrac{}{0\surd}}{\bar{a}.0\surd}}{\bar{b}.0;\bar{a}.0\surd}$$

$$\overline{(a.0;b.0)|(\bar{b}.0;\bar{a}.0)\surd}$$

The left-hand sequencing implies that the tree above $\bar{a}.0\surd$ is sequenced before that above $\bar{b}.0\surd$. The right-hand sequencing requires them to be sequenced in the other way.

Therefore, to check that one can apply a sequenced rule to a deduction, one has to check that it sequences inference trees consistently with every other sequencing. Obviously we have to know what these other sequences are, which means that sequencing information ought to be part of the deduction itself.

The next section defines the QI-deductions, or *sequential* deductions. It begins by defining what the black triangle (pronounced "is sequenced before") means in terms of formula occurrences. It defines the QI-deductions inductively from QI-rules, and then characterizes them axiomatically.

Q-atoms a *Q-atom* is a pair $(Prem, C)$ where C is a formula occurrence not appearing in $Prem$, and $Prem$ is a set of finite, non-empty sequences of formula occurrences, such that no occurrence appears more than once. When they occur as premises of Q-atoms, I write sequences of occurrences $A_1 \cdot \ldots \cdot A_n$ graphically as $A_1 \blacktriangleright \ldots \blacktriangleright A_n$. If s is a sequence of occurrences, I write θs for the result of applying θ elementwise. Two sequences of occurrences match if they are of the same length and they match elementwise. I lift substitution and matching to Q-atoms in the obvious way.

I write flat(a) for the operation of flattening out the sequencing structure of atom a:

$$\mathrm{flat}(Prem, C) = (\bigcup\{\{A_1, \ldots, A_n\} \mid A_1 \cdot \ldots \cdot A_n \in Prem\}, C)$$

and I write $\sqsubset_a$ for the relation "is sequenced immediately before" in a:

$$A \sqsubset_{(Prem,C)} B \text{ iff } \exists s, s'. s \cdot A \cdot B \cdot s' \in Prem$$

3.2 Sequential deductions

As before, a rule atom establishes how someone can draw a new inference from a set of previously established facts, and an interacting rule establishes how a set of people can infer new results after exchanging ideas. Now however, our mathematicians are also interested in when the facts they use were deduced. Specifically, they want to know if the history of one fact (i.e., its inference tree) was completed before the history of another was begun. Before I define the QI-deductions, I define the QI-rules and QI-systems. These are straightforward analogues of I-rules and I-systems.

QI-rules A QI-rule r is a finite, non-empty set of Q-atoms such that no occurrence appears more than once. I write QI-rules graphically in a similar way to I-rules. Rule substitution and rule matching is defined as before. I write flat(r) for the result of flattening all the atoms in r, and $\sqsubset_r$ for the union of the sequencing relations of the atoms in r.

QI-systems A QI-*system* $\mathcal{T}$ is a pair $(\mathcal{L}, \mathcal{R})$ where $\mathcal{L}$ is a language and $\mathcal{R}$ is a set of QI-rules over formulae in $\mathcal{L}$. In the rest of this chapter, I shall drop the "QI"-prefix wherever it does not lead to confusion.

QI-structures We have seen that QI-deductions ought to record the sequencing constraints imposed by the rules that build them. The simplest way to do this is to include a sequencing relation $\sqsubset$ between formula occurrences, where $A \sqsubset B$ means "A is sequenced immediately before B". This relation will simply be the union of the sequencing relations of the rules.

Therefore a QI-*structure* is a triple $(F, I, \sqsubset)$ such that (F, I) is an I-structure, and $\sqsubset \subseteq O(F) \times O(F)$. Henceforth, I shall assume that all triples of shape $(F, I, \sqsubset)$ are QI-structures.

Now, suppose $(F, I, \sqsubset)$ is a QI-structure. The relation $\sqsubset$ orders formula occurrences, the intention being to sequence the trees above the formula occurrences. Therefore we define the relation $\hat{\sqsubset} \subseteq O(F) \times O(F)$ by lifting $\sqsubset$ to trees of occurrences in F. It is defined in the following way

$$A \mathrel{\hat{\sqsubset}} B \text{ iff } \exists A', B' \in O(F). A' \sqsubset B' \text{ and } A' \lesssim_F A \text{ and } B' \lesssim_F B$$

The $\hat{\cdot}$ symbolism is meant to suggest the operation of extending the effect of $\sqsubset$ up the trees whose roots are related by $\sqsubset$. Technically, this symbol should be subscripted by the forest that contains the relevant trees; but where the context does not make it clear which forest is intended (e.g., in the proof of theorem III) the result will always be the same whichever forest in the context is chosen.

Histories The last problem is how to verify that a structure satisfies the sequencing constraints imposed by a rule. Specifically, when $(F, I, \sqsubset)$ is a deduction, we wish to know when it is consistent to assert that one tree is completely deduced before another in $(F, I, \sqsubset)$. Ultimately, this question concerns the order in which we can apply our rules to build $(F, I, \sqsubset)$. Therefore, it is natural to use a notion of *deductive history* to verify sequencing constraints. If there exists some history which satisfies all the sequencing constraints then we know that they are consistent.

A *history* of $(F, I, \sqsubset)$ is a function $h : O(F) \to \mathbb{N}$ such that for all $A, B \in O(F)$,

$$\begin{array}{l} \text{if } A <_F B \text{ then } h(A) > h(B) \\ \text{if } A \hat{\sqsubset} B \text{ then } h(A) > h(B) \\ \text{if } A \, I \, B \text{ then } h(B) = h(A) \end{array}$$

The idea is that h timestamps each occurrence according to when it was deduced. If $h(A) > h(B)$ then A was deduced after B. The first condition simply states that if A is lower than B then A must have been deduced after B. Similarly, if A is sequenced before B then (under the view that evaluation order is opposite deduction order) A is deduced after B. Last, if A and B interact, then they must have been deduced simultaneously.

Given a history h, it is easy to check if it deduces the tree above A before that above B. If the tree above A in F is meant to be deduced before that of B, then the earliest occurrence above B must be later than A (which is the latest occurrence in the tree above A). In symbols, I write $\hat{h}(B) > h(A)$, where

$$\hat{h}(B) = \min\{h(C) \mid C \gtrsim_F B\}$$

is the time of the earliest occurrence above B. The $\hat{}$ notation is again meant to symbolise the act of going up the tree above the specified occurrence.

(Using history functions is similar to the idea of using artificial states in the system of section 3.1.1. Here however, I timestamp formula occurrences, not interaction links.)

QI-deductions Now we can define the QI-deductions. The proposal is to add a sense of history to our community of mathematicians. When a mathematician sees a rule with sequenced premises his first task is to check the order in which the trees concluding the premises were deduced. If A is to be deduced before B, he wants to know that the history of B did not begin until A was obtained.

Definition 3.0 *The set* $\mathbf{QI}(\mathcal{T})$ *of* QI-*deductions of* $\mathcal{T}$ *is the least set of triples* $(F, I, \sqsubset)$ *such that*

1. $0 = (\emptyset, \emptyset, \emptyset) \in \mathbf{QI}(\mathcal{T})$
2. *if* (a) $(F, I, \sqsubset)$ *and* r *matches a rule of* $\mathcal{T}$ *such that flat*(r) *can be applied to* (F, I) *to get* (F', I')

   (b) *There exists a history* h *of* $(F, I, \sqsubset)$ *such that for all* $(A, B) \in \sqsubset_r$, $\hat{h}(A) > h(B)$

   then $(F', I', \sqsubset \cup \sqsubset_r) \in \mathbf{QI}(\mathcal{T})$.

Proposition 3.1 *Every* QI-*deduction has a history.*

Proof: by induction on the definition of $\mathbf{QI}(\mathcal{T})$, for QI-system $\mathcal{T}$. **Case 1:** 0 has the empty history $0 : \emptyset \to \mathbb{N}$. **Case 2:** by induction, $(F, I, \sqsubset)$ has a history. Suppose r can be applied to it to make $(F', I', \sqsubset')$. Then there must exist a history h of $(F, I, \sqsubset)$ such that for all $(A, B) \in \sqsubset_r$, $\hat{h}(A) > h(B)$. We show that $h' : O(F') \to \mathbb{N}$ is a history of $(F', I', \sqsubset')$ where

$$h'(A) = \begin{cases} h(A) & \text{if } A \in O(F) \\ 1 + \max(\text{im } h) & \text{ow} \end{cases}$$

(Where I assume $\max(\emptyset) = 0$.) First, suppose $A <_{F'} B$. Then either $A <_F B$, in which case $h(A) > h(B)$ and so $h'(A) > h'(B)$ or $B \in O(F)$ and $A \notin O(F)$, in which case $h'(A) = 1 + \max(\text{im } h) > h(B) = h'(B)$. Second, suppose $A \widehat{\sqsubset'} B$.

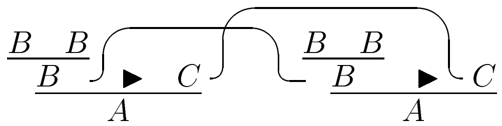
Then either $A\hat{\sqsubset}B$ or not. If so, then $h(A) > h(B)$, whence $h'(A) > h'(B)$. If not, then A and B must be sequenced by $\sqsubset_r$: i.e., there exists A', B' such that $A \gtrsim_F A'$ and $B \gtrsim_F B'$ and $A' \sqsubset_r B'$. But then, $\hat{h}(A') > h(B')$, and so $h'(A) = h(A) \geq \hat{h}(A') > h(B') \geq h(B) = h'(B)$. Last, suppose $A\, I\, B$. Then either $A, B \in O(F)$ or $A, B \notin O(F)$. In the first case, $h'(A) = h(A) = h(B) = h'(B)$, and in the second, $h'(A) = h'(B) = 1 + \max(\mathrm{im}\ h)$. □

3.2.1 Examples

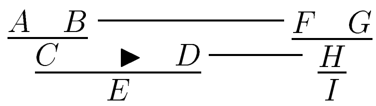
Section 5.1.4 gives an evaluation-style operational semantics for CSP which is a QI-system. I give some simpler examples here. Let $\mathcal{T}$ be the QI-system $(\mathcal{L}, \mathcal{R})$ where $\mathcal{L} = \{A, B, C\}$ and $\mathcal{R}$ consists of the following rules:

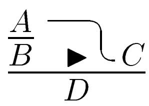
$$\frac{}{B} \qquad \frac{}{C} \text{———} \frac{}{C} \qquad \frac{B \quad B}{B} \text{———} \frac{B \quad B}{B} \qquad \frac{B \quad \blacktriangleright \quad C}{A}$$

Then the following diagram is a graphical representation of a simple $\mathcal{T}$-deduction $(F, I, \sqsubset)$



Two examples of triples $(F, I, \sqsubset)$ which are not QI-deductions are:





In the first diagram, B ought to precede D through sequencing; yet D ought also to precede B through interactions and inferences. It tries to sequence occurrences which are already ordered oppositely. The second diagram tries to strictly order two trees whose leaves interact. I call these errors *sequencing loops* because they attempt to synchronize occurrences which have been strictly sequenced.

3.2.2 QI-deductions have no sequencing loops

Preorders of QI-structures The preorder of a QI-structure $(F, I, \sqsubset)$ is

$$\lesssim_{(F,I,\sqsubset)} = (\lesssim_{(F,I)} \cup \hat{\sqsubset})^*$$

Proposition 3.2 *Let $(F, I, \sqsubset)$ have history h. Then for all $A, B \in O(F)$, $A \lesssim_{(F,I,\sqsubset)} B$ implies $h(B) \leq h(A)$.*

Proof: Suppose $A \lesssim_{(F,I,\sqsubset)} B$. Then there must exist a sequence of occurrences $A_1, \ldots, A_n$ such that $A = A_1$, $B = A_n$ and for $i = 1, \ldots, n-1$, either $A_i <_F A_{i+1}$, $A_i\, I\, A_{i+1}$ or $A_i \hat{\sqsubset} A_{i+1}$. In each case, $h(A_{i+1}) \geq h(A_i)$. The result follows from the transitivity of $\geq$ over the naturals. □

Sequencing Loop Freeness I say $(F, I, \sqsubset)$ is *sequencing loop free* if the following condition holds

$$\text{For all } A, B \in O(F), \text{ if } A <_F B \text{ or } A \hat{\sqsubset} B \text{ then } A <_{(F,I,\sqsubset)} B \tag{SLF}$$

For motivation, again consider the community of mathematicians. If $A <_F B$ or $A \hat{\sqsubset} B$ then A must have been deduced after B. Given the definition of preorder, this obviously implies $A \lesssim_{(F,I,\sqsubset)} B$. However, if A was deduced after B, then it could not also have been deduced simultaneously with B — therefore $A \not\sim_{(F,I,\sqsubset)} B$. That is, $A <_{(F,I,\sqsubset)} B$.

Proposition 3.3 *SLF implies DLF.*

Proof: Let $(F, I, \sqsubset)$ be a QI-structure, and let $A, B \in O(F)$ be such that $A <_F B$. By definition, $A \lesssim_{(F,I)} B$. By SLF, $A <_{(F,I,\sqsubset)} B$. Since $\lesssim_{(F,I)} \subseteq \lesssim_{(F,I,\sqsubset)}$ we get that $B \not\lesssim_{(F,I)} A$ because otherwise $B \lesssim_{(F,I,\sqsubset)} A$ contradicting SLF. □

Proposition 3.4 *Let $(F, I, \sqsubset)$ have history h. Then $(F, I, \sqsubset)$ satisfies SLF.*

Proof: Suppose $A <_F B$. Then $A \lesssim_{(F,I,\sqsubset)} B$ and $h(A) > h(B)$. Now, suppose in fact $B \gtrsim_{(F,I,\sqsubset)} A$ also. Then $A \sim_{(F,I,\sqsubset)} B$, from which $h(A) = h(B)$: contradiction. Therefore $A <_{(F,I,\sqsubset)} B$. Similarly when $A \hat{\sqsubset} B$. □

Proposition 3.5 *Every* QI*-deduction satisfies SLF.*

Proof: From propositions 3.1 and 3.4. □

Proposition 3.6 *SLF implies that $A \sim_{(F,I,\sqsubset)} B$ implies AI^*B.*

Proof: Suppose $A \sim_{(F,I,\sqsubset)} B$. Then there exists a sequence of occurrences $A_0, \ldots, A_n$ in F such that $A_1 = A$, $A_n = B$ and for all $i = 1, \ldots, n-1$ either $A_i <_F A_{i+1}$ or $A_i I A_{i+1}$ or $A_i \hat{\sqsubset} A_{i+1}$. Now, since $A \sim_{(F,I,\sqsubset)} B$, we have for $i = 1, \ldots, n-1$, $A_i \sim_{(F,I,\sqsubset)} A_{i+1}$ because

$$A \lesssim_{(F,I,\sqsubset)} A_i \lesssim_{(F,I,\sqsubset)} A_{i+1} \lesssim_{(F,I,\sqsubset)} B \lesssim_{(F,I,\sqsubset)} A$$

This in turn means that both $A_i \not<_F A_{i+1}$ and $A_i \hat{\not\sqsubset} A_{i+1}$ by SLF, which means that $A_i I A_{i+1}$ for all $i = 1, \ldots, n-1$. □

3.2.3 SLF helps characterize the QI-deductions

Proposition 3.5 showed that every QI-deduction satisfies SLF. To show that SLF is a defining feature of QI-deduction, I need to define the notions of QI-neighbourhoods and rule matching.

QI-neighbourhoods Let A be an occurrence in a QI-structure $(F, I, \sqsubset)$. Then the *(QI-)neighbourhood* of A is the pair $N_{(F,I,\sqsubset)}(A) = (N, \sqsubset_A)$ where

$$N = N_{(F,I)}(A)$$
$$\sqsubset_A = \sqsubset \cap (O(N) \times O(N))$$

Let $N(F, I, \sqsubset)$ be the set of neighbourhoods in $(F, I, \sqsubset)$.

The dependency preorder of a QI-deduction Π can be lifted to a preorder $\leq_{N(\Pi)} \sqsubset N(\Pi) \times N(\Pi)$ over QI-neighbourhoods in exactly the same way that the dependency preorder for I-structures can be lifted to a preorder of I-neighbourhoods. By the same reasoning as proposition 2.3 we get

Proposition 3.7 $\leq_{N(F,I,\sqsubset)}$ *is a partial order when* $(F, I, \sqsubset)$ *satisfies SLF.* □

Proposition 3.8 *Let* $(F, I, \sqsubset)$ *satisfy SLF. Then it has a history.*

Proof: We construct a history. By proposition 3.7, the QI-neighbourhoods of $N(F, I, \sqsubset)$ can be partially ordered. Let $\leq$ be any total order containing this partial order. Then we define $h(A) = i$ if $N(A)$ is the ith neighbourhood of $\geq$ (i.e., the total order reversed). To show that this satisfies the conditions, let $A <_F B$. Then $N(A) < N(B)$, by definition of the partial order. Then $h(B) > h(A)$. The other two cases are similar. □

Rule matching A rule r *matches* a neighbourhood $(N, \sqsubset)$ via $f : O(r) \leftrightarrow O(N)$ (written $r \equiv_f (N, \sqsubset)$) iff

$$(a) \quad \text{flat}(r) \equiv_f N$$
$$(b) \quad \sqsubset = \{ \ (f(A), f(B)) \ \mid \ (A, B) \in \sqsubset_r \}$$

Theorem III $(F, I, \sqsubset) \in \mathbf{QI}(\mathcal{T})$ *if and only if* $(F, I, \sqsubset)$ *is a* QI*-structure satisfying SLF and such that every neighbourhood matches a rule of* $\mathcal{T}$*.*

Proof: $\Leftarrow$: Let $(F, I, \sqsubset)$ be a QI-structure satisfying SLF and such that every neighbourhood matches a rule of $\mathcal{T}$. Then $(F, I, \sqsubset)$ contains a finite number of neighbourhoods. By proposition 3.7, $\leq_{N(F,I,\sqsubset)}$ is a partial order. Let $\leq$ be any total order containing $\leq_{N(F,I,\sqsubset)}$ and $N_n < N_{n-1} < \ldots < N_1$ be the neighbourhoods of $N(F, I, \sqsubset)$ ordered by $\leq$. Thus N_n is the lowest neighbourhood, and N_1 the highest.

I show by induction on the number n of neighbourhoods that $(F, I, \sqsubset)$ belongs to $\mathbf{QI}(\mathcal{T})$. When $n = 0$, the case is trivial. When $n = k+1$, let $(F', I', \sqsubset')$ be the QI-structure consisting only of the $\leq$-highest k neighbourhoods (i.e., $N_k, \ldots, N_1$). Obviously, it will satisfy SLF and every neighbourhood will be a rule of $\mathcal{T}$. Thus by induction, $(F', I', \sqsubset') \in \mathbf{QI}(\mathcal{T})$.

Consider the $k+1$-the neighbourhood $(N, \sqsubset_0)$. It matches rule r. By a similar argument to that of theorem I, we can show that (F, I) is the result of applying flat(r) to (F', I'). It remains to show that the sequencing constraints are satisfied. By proposition 3.8, $(F, I, \sqsubset)$ has history h. So, let $A \sqsubset_0 B$. Then $A \sqsubset B$ by the definition of neighbourhood, and so for all $C \gtrsim_F A$, $C \hat{\sqsubset} B$ and therefore $h(C) > h(B)$. In particular, $\min\{h(C) \mid C \gtrsim_F A\} > h(B)$, i.e., $\hat{h}(A) > h(B)$. Therefore, $(F, I, \sqsubset)$ is a member of $\mathbf{QI}(\mathcal{T})$.

⇒: Let $(F, I, \sqsubset) \in \mathbf{QI}(\mathcal{T})$. By prop 3.5, it satisfies SLF. It remains to show that every neighbourhood matches a rule of $\mathcal{T}$. We show the result by induction on the definition of $(F, I, \sqsubset)$. The base case is vacuous. For the induction step, suppose every neighbourhood of $(F, I, \sqsubset)$ matches a rule of $\mathcal{T}$. Suppose r can be applied to it to yield $(F', I', \sqsubset')$. Now let $(N, \sqsubset_N)$ be the only neighbourhood of $(F', I', \sqsubset')$ not in $(F, I, \sqsubset)$. By a similar argument to that of theorem I we can show that flat(r) matches N via some $f : O(r) \leftrightarrow O(N)$. It remains to show that $\sqsubset_N = \{(f(A), f(B)) \mid (A, B) \in \sqsubset_r\}$. This follows directly from case 2 of definition 3.0. □

I cannot form a simple correctness principle for QI-deductions based on their graphical representation, like I did for I-deductions. One has to mentally "cut out" sequenced trees and stack them in such a way that if $A \sqsubset B$ then the tree above B is stacked above the tree above A. Then one can use the correctness principle for I-deductions.

3.2.4 Coding QI-deduction into I-deduction

In section 3.1 I exhibited two different techniques for capturing sequentiality: cutting intermediate states and using distributed control stacks. Both of these techniques could work independently of the details of the language being defined. This suggests that QI-systems are not more expressive than I-systems — i.e., any language definable in a QI-system is already definable in an I-system.

To prove this result directly, I could either exhibit a coding of the QI-systems which used artificial states, or which used distributed control stacks. In the first case, I would transform a judgment A into judgments of form $[\kappa] \;\; A \;\; [\kappa']$, and in the second, I would use judgments of form A *before* B.

One can do this quite straightforwardly — in fact the translation using artificial states is reminiscent of the state convention of the definition of Standard ML. The proof is not very difficult. However, for reasons of space I shall omit such a proof here. Instead, an even simpler proof goes via the computational semantics of chapter 6. The proof of the following theorem can be found in section 6.3.1.

Theorem IV *For all* QI-*systems* $\mathcal{T} = (\mathcal{L}, \mathcal{R})$, *there exists an* I-*system* $\mathcal{T}^* = (\mathcal{L}^*, \mathcal{R}^*)$ *and a family of injective maps* $f_n : \mathcal{L}^n \to \mathcal{L}^*$ *such that* $\mathcal{T} \vdash A_1, \ldots, A_n$ *if and only if* $\mathcal{T}^* \vdash f_n(A_1, \ldots, A_n)$ □

3.3 An evaluation QI-semantics for P(;)

We use the theory of QI-deductions to give an evaluation-style semantics to P(;). Let $\mathcal{P}^{(;)}_{QI}$ be the QI-system consisting of judgments of form $p\surd$ (where $p \in$P(;)) and the following four QI-rules.

$$\frac{}{0\surd} \qquad \frac{p\surd}{a.p\surd} \;\text{———}\; \frac{q\surd}{\bar{a}.q\surd} \qquad \frac{p\surd \qquad q\surd}{p|q\surd} \qquad \frac{p\surd \;\blacktriangleright\; q\surd}{p;q\surd}$$

Once again, the evaluation semantics is more concise than the transition semantics.

The remainder of this section will be spent confirming that the rule for $p;q$ really does define it to be the sequential composition of p and q. I do this by showing it accords with the sequentiality given by prefixing. The translation of P(;) into P given in section 3.4 will further confirm these rules correct. First I confirm that the semantics does sequence correctly, and second that it does not sequence incorrectly.

The following example suggests that $p;q$ can sequence p before q by giving the deduction of the judgment $(a.0;b.0)|(\bar{a}.\bar{b}.0)\surd$. Since $\bar{a}$ must be communicated before $\bar{b}$, this will show that the rules allow $a.0$ to be evaluated before $b.0$. Let Π be the structure

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0\surd}}{a.0\surd} \;\blacktriangleright\; \dfrac{\dfrac{}{0\surd}}{b.0\surd}}{a.0;b.0\surd} \qquad \dfrac{\dfrac{\dfrac{}{0\surd}}{\bar{b}.0\surd}}{\bar{a}.\bar{b}.0\surd}}{(a.0;b.0)|(\bar{a}.\bar{b}.0)\surd}$$

It is not hard to check that Π is a deduction. For instance, $b.0\surd \not\lesssim_{\Pi} a.0\surd$, and $a.0\surd$ is sequenced before $b.0\surd$. Intuitively, one can cut out the tree above $b.0\surd$ and place it above the tree above $a.0\surd$ without introducing any sequencing loops.

The next example suggests that the rule for $p;q\surd$ only sequences $p\surd$ before $q\surd$. Consider the process $(a.0;b.0)|(\bar{b}.\bar{a}.0)$. The only QI-structure that is composed of $\mathcal{P}^{(;)}_{QI}$ rules and which concludes $(a.0;b.0)|(\bar{b}.\bar{a}.0)\surd$ in isolation is $(F, I, \sqsubset)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0\surd}}{a.0\surd} \;\blacktriangleright\; \dfrac{\dfrac{}{0\surd}}{b.0\surd}}{a.0;b.0\surd} \qquad \dfrac{\dfrac{\dfrac{}{0\surd}}{\bar{a}.0\surd}}{\bar{b}.\bar{a}.0\surd}}{(a.0;b.0)|(\bar{b}.\bar{a}.0)\surd}$$

But this is not a QI-deduction. We have $a.0\surd \widehat{\sqsubset} b.0\surd$ and also $a.0\surd \sim_{(F,I)} \bar{a}.0\surd >_F \bar{b}.\bar{a}.0\surd \sim_{(F,I)} b.0\surd$ (i.e., $b.0\surd <_{(F,I,\sqsubset)} a.0\surd$). This contradicts SLF.

An evaluation semantics for P(:=,;)

Section 2.3 (page 25) showed that transition rules for variable assignment and lookup could be added to those of P in a modular way. They can also be added to the evaluation semantics in a modular way:

$$\frac{}{v \leadsto v} \qquad \frac{}{x \leadsto v} \;\text{———}\; \frac{\sigma(x) = v}{\sigma \leadsto \sigma} \qquad \frac{e_1 \leadsto v_1 \qquad e_2 \leadsto v_2}{e_1 \; op \; e_2 \leadsto app(op, v_1, v_2)}$$

$$\frac{e \leadsto v \quad \blacktriangleright \quad set(x,v) \quad \blacktriangleright \quad p\surd}{(x := e).p\surd} \qquad \frac{}{set(x,v)} \;\text{——}\; \frac{}{\sigma \leadsto \sigma[v/x]} \qquad \frac{\sigma \leadsto \sigma' \quad \blacktriangleright \quad \sigma' \leadsto \sigma''}{\sigma \leadsto \sigma''}$$

Thus in the QI-system for P(:=,;), the rule for sequential composition does not mention stores. We do not need the *state convention* of [HMT90] to aid legibility. They felt the need to remove stores from rules, but could not — so they used a syntactic convention to abbreviate rule elaboration. When my rules do not concern stores, they do not mention them. There is no need to hide irrelevant details.

Note that the cut rule for stores differs from the one given in $\mathcal{S}_I$ (page 33). There I wished transitions to write to the store at most once. However, over the course of its entire evaluation, a program will typically write to the store more than once. Therefore we require a different cut rule for stores.

When interactions occur

The auxiliary judgment $set(x, v)$ is used to ensure that the store is updated only after the appropriate value has been obtained. The "obvious" rule

$$\frac{e \leadsto v \quad \blacktriangleright \quad p\surd}{(x := e).p\surd} \;\text{——}\; \frac{}{\sigma \leadsto \sigma[v/x]}$$

does not capture the behaviour of assignment. To see why, consider the program $(x := x + 1).0$, evaluated in the store σ, where $\sigma(x) = 1$. We would expect to apply the rule to this program and update the store to become σ' where $\sigma'(x) = 2$. Instead, we get

$$\frac{}{\sigma \leadsto \sigma} \qquad\qquad \frac{}{\sigma \leadsto \sigma[2/x]} \;\text{——}\; \frac{\dfrac{\dfrac{}{x \leadsto 1} \qquad \dfrac{}{1 \leadsto 1}}{x + 1 \leadsto 2}}{(x := x + 1).0\surd}$$

To which we cannot apply the cut rule on stores to yield a two-conclusion deduction of $(x := x + 1).0\surd$, $\sigma \leadsto \sigma[2/x]$. The rules of sequencing order the store write before the store read. This is not the behaviour of assignment.

So what is it about the above rule that makes it wrong and my rule right? It is the time during the evaluation of assignment when the store is updated. Intuitively, the store is updated after the expression has been evaluated and before the rest of the program has begun to be evaluated. In my rule, the *set* judgment ensures this. The "obvious" rule attempts to update the store even before the expression has been evaluated!

This raises the following question: given an interacting rule for an evaluation semantics, at which part of the specified evaluation does the interaction occur? That is, in a hypothetical rule

$$\frac{p_1' \Rightarrow v_1'}{p_1 \Rightarrow v_1} \;\text{———}\; \frac{p_2' \Rightarrow v_2'}{p_2 \Rightarrow v_2}$$

at which point does the interaction occur? There are two natural choices. One choice is to say that the interaction occurs before the evaluation of p_1 and p_2. The other is to say that it occurs after them.

In fact, for I-deduction, the interpretation varies between particular semantics (i.e., it is not important proof-theoretically). The semantics $\mathcal{P}_I^E$ (page 40) only makes sense when the interaction occurs prior to the evaluation specified by the premises. It is also the interpretation of interaction required by the distributed control stack semantics of section 3.1.2 for P(;).

We can also find semantics for P(;) in which the interactions are interpreted to occur after the premises. However, for P(;), these are highly artificial, and are harder to understand than any of the ones previously encountered.

The point is that I-deduction does not commit us to interpret interaction as occurring before or after the evaluations described by the premises. By contrast, QI-deduction commits us to interpret interaction as occurring before the evaluations specified by the premises. This can be seen using the rules of $\mathcal{P}_{QI}^{(;)}$. Consider the deduction of $(a.0; b.0)|(\bar{a}.\bar{b}.0)\surd$ on page 55. As far as sequencing is concerned, the interaction of a and $\bar{a}$ occurs before that of b and $\bar{b}$. But this means that the interaction of the rule

$$\frac{0\surd}{a.0\surd} \text{———} \frac{\bar{b}.0\surd}{\bar{a}.\bar{b}.0\surd}$$

must occur before the interactions of $\bar{b}.0\surd$.

We could easily have defined sequencing to fix the other interpretation — it would not have affected the range of languages to which we can give semantics. I chose to fix interaction to occur before premise evaluation both to make the semantics of P(;) simple, and also to make the computational interpretation of deduction in chapter 6 straightforward.

3.4 An example: translation correctness

It is well-known that sequential composition can be coded in CCS. In [Mil89, §8.1], Milner codes sequential composition using restriction and relabelling, of which we have neither. One can work around this lack, but to do so requires more effort: the translation is more complex, and a fancier notion of bisimulation is required which only compares suitably restricted behaviour. Since the aim of this section is to compare the proofs of some result in the evaluation and transition style semantics, and the result we are going to prove is that sequential composition can be coded in P, we might as well save work and add restriction to P(;) and P. We do not include relabelling because first we do not need it (for P(;) simple subsititution sufficies) and because second it is harder to capture in an evaluation semantics (see section 7.3.3).

The structure of this section is as follows. First, we define the translation. Then, in section 3.4.2 we use transition semantics of P(;) plus restriction to define the notion of correctness, and then to prove the translation correct with respect to it. Section 3.4.4 uses evaluation semantics to define the notion of correctness and proves the translation correct with respect to it. It is clear that the evaluation semantics proof involves less work.

The notion of correctness is that the translated process is (almost) weakly trace equivalent to the original one. By almost, I mean that every trace of the translated process will have an extra terminal action. We do not use bisimulation equivalence because it is hard to define in the evaluation semantics. (See [VG90] for a survey of different process calculus equivalences.) We should not expect to define an equivalence finer than a trace equivalence when our semantics considers only complete evaluations. Nevertheless, bisimulation and trace equivalence coincide for deterministic languages. [Ber89] argues that in practice many (if not most) programs are deterministic, so perhaps this is not such a great problem for practical programming languages.

3.4.1 The translation

Let me define the function $\Lambda(p) : \mathrm{P}(;) \rightarrow \mathit{Lab}$ inductively over the terms of p:

$$\Lambda(p) = \begin{cases} \emptyset & \text{if } p = 0 \\ \{l\} \cup \Lambda(q) & \text{if } p = l.q \\ \Lambda(q) \cup \Lambda(q') & \text{if } p = q|q' \text{ or } p = q;q' \end{cases}$$

Let done and $\overline{\text{done}}$ be new actions to $\mathrm{P}(;)$. Then I define the translation using the following inductively defined function

$$\boldsymbol{c}(0) = \overline{\text{done}}.0 \qquad\qquad \boldsymbol{c}(l.p) = l.\boldsymbol{c}(p)$$

$$\frac{p' = \boldsymbol{c}(p)[d_1/\text{done}] \qquad q' = \boldsymbol{c}(q)[d_2/\text{done}] \qquad d_1, d_2, \overline{d_1}, \overline{d_2} \notin \Lambda(\boldsymbol{c}(p)|\boldsymbol{c}(q))}{\boldsymbol{c}(p|q) = (p'|q'|d_1.d_2.\overline{\text{done}}.0) \setminus \{d_1, d_2\}}$$

$$\frac{p' = \boldsymbol{c}(p)[b/\text{done}] \qquad q' = \boldsymbol{c}(q) \qquad b, \bar{b} \notin \Lambda(\boldsymbol{c}(p); \boldsymbol{c}(q))}{\boldsymbol{c}(p;q) = (p'|b.q') \setminus \{b\}}$$

where $p \setminus L$ is the restriction combinator. Informally, it means that p may perform no action in the set of actions L. Therefore, if p can perform such an action, the process can only proceed if this action can be absorbed internally.

The idea here is that every process p is translated into a process $\underline{p'}$ which "behaves the same as" $\underline{p}$, except that it includes a special final action, $\overline{\text{done}}$. Let *Done* be the process $\overline{\text{done}}.0$.

3.4.2 The transition semantics proof of correctness

Restriction The standard transition semantic rule for restriction is

$$\frac{p \xrightarrow{\alpha} p' \qquad \alpha \notin L \cup \bar{L}}{p \setminus L \xrightarrow{\alpha} p' \setminus L}$$

These should be added to the rest of the rules on page 40 to obtain the semantics for $\mathrm{P}(;)$ plus restriction.

The equivalence I say p is *weakly trace-equivalent* to q (written $p \simeq q$) if for all p' and traces t such that $p \xrightarrow{t} p'$ there exists a q' such that $q \xrightarrow{t} q'$ and vice-versa. Extending notation from page 35, $p \xrightarrow{t} \Omega$ means that there exists a sequence of transitions $p_i(\xrightarrow{\tau})^* \xrightarrow{t(i)} (\xrightarrow{\tau})^* p_{i+1}$, for some $p_1, \ldots, p_{|t|+1} \in \mathrm{P}(;)$ and $p_{|t|+1} \in \Omega$. The following are useful properties of trace equivalence:

Proposition 3.9

$$\begin{array}{ll} (i) & \textit{if } p \simeq q \textit{ then } \alpha.p \simeq \alpha.q \\ (ii) & \textit{if } p_1 \simeq p_2 \textit{ and } q_1 \simeq q_2 \textit{ then } p_1|q_1 \simeq p_2|q_2 \\ (iii) & l.(p;q) \simeq (l.p);q \\ (iv) & (p_1;p_2);p_3 \simeq p_1;(p_2;p_3) \end{array}$$

Proof: (i): Let $\alpha.p \xrightarrow{t} p'$. Then $t = \alpha \cdot t'$ and so $q \xrightarrow{t'} q'$ (as $p \simeq q$), which means that $\alpha.q \xrightarrow{t} q'$. The reverse direction is symmetric.
(ii): Let $p_1|p_2 \xrightarrow{t} p_1'|p_2'$. Then there must exist traces t_1 and t_2 such that $p_1 \xrightarrow{t_1} p_1'$ and $q_1 \xrightarrow{t_2} q_1'$ which when combined (according to the above deduction sequence of $p_1|p_2$) form t. Therefore there exist p_2' and q_2' such that $p_2 \xrightarrow{t_1} p_2'$ and $q_2 \xrightarrow{t_2} q_2'$. Then we can zip up t_1 and t_2 in the same way as we had before to get $p_2|q_2 \xrightarrow{t} p_2'|q_2'$. A fully formal proof would require induction on the length of the deduction sequence of $p_1|p_2$. Again the reverse direction is symmetric.
(iii) and (iv): even simpler. □

Well-terminating processes We appropriate from [Mil89] the concept of well-termination. A process p is *well-terminating* if for every p' and t, $p \xrightarrow{t} p'$ implies that $t(i) = \mathrm{done}$ for no $i \in \mathrm{dom}\ t$ and $t(|t|) = \overline{\mathrm{done}}$ implies that $p' \simeq 0$.

Proposition 3.10 *For every $p \in \mathrm{P}(;)$, $\boldsymbol{c}(p)$ is well-terminating.*

Proof: By induction on the structure of p. **Case 0:** $\boldsymbol{c}(0) = \overline{\mathrm{done}}.0$ is obviously well-terminating. **Case $\boldsymbol{l.q}$:** by induction $\boldsymbol{c}(q)$ is well-terminating. Since $l \neq \mathrm{done}$ (because we added done to the set of actions specifically for the translation) the result is obviously true. **Case $\boldsymbol{p|q}$:** by induction, $\boldsymbol{c}(p)$ and $\boldsymbol{c}(q)$ are well-terminating. Suppose $\boldsymbol{c}(p|q)$ is not. Then there must exist a trace t and process r such that $\boldsymbol{c}(p|q) \xrightarrow{t} r$ and either for some $i \in \mathrm{dom}\ t$, $t(i) = \mathrm{done}$ or $t(|t|) = \overline{\mathrm{done}}$ but $p'|q' \not\simeq 0$. The first case is impossible since p and q are well-terminating. To show that the second case is impossible, let $t = t' \cdot \overline{\mathrm{done}}$. We show by induction on the length of t that $r \simeq 0$. **Subcase $\boldsymbol{n = 1}$:** then $\boldsymbol{c}(p)[d_1/\mathrm{done}] \xrightarrow{\overline{d_1}} \Omega$ and $\boldsymbol{c}(q)[d_2/\mathrm{done}] \xrightarrow{\overline{d_2}} \Omega$. But then $r = \Omega|\Omega|0 \simeq 0$. **Subcase $\boldsymbol{n = k+1}$:** then let $\boldsymbol{c}(p|q) \xrightarrow{l} r' \xrightarrow{t} r$. Suppose $(l \cdot t)(|l \cdot t|) = \overline{\mathrm{done}}$. Then $t(|t|) = \overline{\mathrm{done}}$. Now what form has r'? Since $l \neq \overline{\mathit{done}}$, one of p or q must make an l-labelled transition to p' or q'. Therefore one of $\boldsymbol{c}(p)$ or $\boldsymbol{c}(q)$ must make a transition to a process trace equivalent to $\boldsymbol{c}(p')$ or $\boldsymbol{c}(q')$. Therefore r' must be trace equivalent to either $\boldsymbol{c}(p'|q)$ or to $\boldsymbol{c}(p|q')$. Given this fact, and the fact that t has length k, induction tells us that $r \simeq 0$. **Case $\boldsymbol{p;q}$:** by induction $\boldsymbol{c}(p)$ and $\boldsymbol{c}(q)$ are well-terminating. Now let $\boldsymbol{c}(p;q) \xrightarrow{t} r$. Then either $\boldsymbol{c}(p)[b/\mathrm{done}] \xrightarrow{t} p'$ in which case t has the required properties by induction. Alternatively, $\boldsymbol{c}(p)[b/\text{done}] \stackrel{t_1\cdot\bar{b}}{\to} \Omega$ and $b.\boldsymbol{c}(q) \stackrel{b\cdot t_2}{\to} r$, where $t = t_1 \cdot t_2$. Certainly for no $i \in \text{dom}\ t$ is $t(i) = \overline{\text{done}}$ (this follows by induction). Also, if $t(|t|) = \overline{\text{done}}$ then $t_2(|t_2|) = \overline{\text{done}}$, which means that $r \simeq 0$ by induction. □

Proposition 3.11 *For all* $p \in \mathrm{P}(;)$, $p; Done \simeq \boldsymbol{c}(p)$.

Proof: By induction on the structure of p. **Case 0:** trivial. **Case $\boldsymbol{l.p}$:** trivial. **Case $\boldsymbol{p|q}$:** By induction, $p; Done \simeq \boldsymbol{c}(p)$ and $q; Done \simeq \boldsymbol{c}(q)$. By proposition 3.9$(ii)$ we get $p; Done \,|\, q; Done \simeq \boldsymbol{c}(p)|\boldsymbol{c}(q)$. Now Let t be a trace of $(p|q); Done$. Then either it is a trace of $p|q$ or its final action is $\overline{\text{done}}$. In the first case, it is also a trace of $p; Done \,|\, q; Done$ and therefore also of $\boldsymbol{c}(p)|\boldsymbol{c}(q)$ and therefore also of $\boldsymbol{c}(p)[d_1/\text{done}] \,|\, \boldsymbol{c}(q)[d_2/\text{done}]$ since both $\boldsymbol{c}(p)$ and $\boldsymbol{c}(q)$ are well-terminating and t contains no $\overline{\text{done}}$ action. Hence t is also a trace of $\boldsymbol{c}(p|q)$.

Suppose however, $t(|t|) = \overline{\text{done}}$. Let $t = t' \cdot \overline{\text{done}}$. Then t' is a trace of $p|q$. We can project the transition sequence creating trace t' into transition sequences of p and q, with traces t_1 and t_2. Moreover, $p \stackrel{t_1}{\to} \Omega$ and $q \stackrel{t_2}{\to} \Omega$. Therefore $t_1 \cdot \overline{\text{done}}$ is a trace of $\boldsymbol{c}(p)$ and $t_2 \cdot \overline{\text{done}}$ is a trace of $\boldsymbol{c}(q)$. Therefore, $t_1 \cdot \overline{d_1}$ is a trace of $\boldsymbol{c}(p)[d_1/\text{done}]$ and $t_2 \cdot \overline{d_2}$ is a trace of $\boldsymbol{c}(q)[d_2/\text{done}]$. Therefore, by reconnecting the transition sequences, we get $t' \cdot \overline{\text{done}}$ is a trace of $\boldsymbol{c}(p|q)$ by induction on the length of the transition sequence generating t'.

The reverse direction is symmetric.

Case $\boldsymbol{p; q}$: Suppose t is a trace of $(p; q); Done$. Then it is also a trace of $p; (q; Done)$. Either t is a trace of p or $t = t' \cdot t''$ where t' is a trace of p such that $p \stackrel{t}{\to} \Omega$ and t'' is a trace of $q; Done$. In the first case, t is also a trace of $\boldsymbol{c}(p)$ and since it does not contain $\overline{\text{done}}$ and $\boldsymbol{c}(p)$ is well-terminating, it is also a trace of $\boldsymbol{c}(p)[b/\text{done}]$ when $b \notin \Lambda(\boldsymbol{c}(p))$, and therefore also a trace of $\boldsymbol{c}(p; q)$. In the second case, $t' \cdot \bar{b}$ is a trace of $\boldsymbol{c}(p)[b/\text{done}]$ and $b \cdot t''$ is a trace of $b. \cdot (q)$. Since $\boldsymbol{c}(p)$ is well-terminating, then $t' \cdot t''$ is also a trace of $\boldsymbol{c}(p; q)$.

In the reverse direction, let t be a trace of $\boldsymbol{c}(p; q)$. Then we can split it into traces t' of $\boldsymbol{c}(p)[b/\text{done}]$ and t'' of $b.\boldsymbol{c}(q)$. Since $\boldsymbol{c}(p)$ is well-terminating, $\bar{b}$ can only be the last action of t'. If $\bar{b}$ is not the last action of t' then t'' must be empty. Therefore t' is a trace of $\boldsymbol{c}(p)$ not containing $\overline{\text{done}}$, and therefore also of p, and therefore also of $p; (q; Done)$. If $\bar{b}$ is the last action of t', then b must be the first action of t'' (given that b is restricted in $\boldsymbol{c}(p; q)$). Therefore t'' is also a trace of $b.q; Done$. Let t_1 be such that $t' = t_1 \cdot \bar{b}$, and t_2 be such that $t'' = b \cdot t_2$. Then $p \stackrel{t_1}{\to} \Omega$, and so $t_1 \cdot t_2 = t$ must be a trace of $p; q$. □

3.4.3 Tree Pruning and partial deductions

One of the problems in using evaluation semantics to give meaning to processes is that they do not specify aberrant behaviours well. This means that we cannot say anything about the partial behaviour of a program before it deadlocks (for example). This is important. Consider the process $\pitchfork = (l.p) \setminus \{l\}$. It is trivially deadlocking, and so obviously $\pitchfork \surd$ should be undeducible. However, consider the process $a.\pitchfork$. We cannot say anything about this process either, even though it performs action a first.

Because of this, we cannot use evaluation semantics to define a trace equivalence. Of course, trace equivalence belongs to the transition semantics view of processes, but it seems we cannot define anything similar for evaluation semantics. And if we cannot say anything about abberant behaviours, we cannot check that two processes "behave the same".

One solution to this problem is to use a *pruning rule*. This rule is of form $\overline{X}$ where X can be instantiated to any judgment. (Of course, this assumes that our language of judgments permits this kind of instantiation.) I call this the *deduction of no information* (written $\perp$) because it allows us to deduce things from thin air. Using it, we get the following partial deduction of $a.\pitchfork$:

$$\dfrac{\begin{array}{c}\perp\\ \pitchfork\end{array}}{a.\pitchfork}$$

which has one visible occurrence ($a.\pitchfork$), and one instance of the deduction of no information $\perp \vdash \pitchfork$.

Strict pruning As chapter 6 shows, a deduction can be seen as an evaluation of a process. What does $\perp$ mean in this light? It means that the evaluation has been interrupted. In the above example, evaluation would halt when the deadlocking process $\pitchfork$ was encountered. However, if a process halts, no process sequenced after it can proceed. I.e., whenever some occurrence has been pruned, every occurrence sequenced after it must likewise be pruned. I call this *strict pruning*. Let me write $P(\Pi)$ for the set of occurrences of Π which are conclusions of instances of the pruning rule.

Given a QI-structure it is easy to determine the pruned occurrences: simply look at the leaves of the trees, and check if they can be deduced by axioms. If they cannot, then they are pruned. However, this leaves us with a problem. We want to say that everything sequenced after a pruned ocurrence is also pruned; but it could easily be that such an occurrence is actually an axiom instance, and therefore not pruned. It will, however, be a leaf. Formally then, the strict pruning condition can be stated

$$\text{for all } A, B \in O(F, I, \sqsubset),\ \text{if } A \in P(F, I, \sqsubset) \text{ and } A(\hat{\sqsubset})^* B \text{ then } B \text{ is a leaf}$$

Note that these definitions do not concern the semantics of P(;) — they can be equally well be applied to *any* semantics. Therefore the notion of trace equivalence which we shall define is applicable to any (concurrent) language specified in this way, and the method which we use to prove the equivalence can be adopted for these languages.

When $\mathcal{T}$ is a system of interacting rules, I write $\mathbf{QI}_\perp(\mathcal{T})$ for the set of partial, strictly pruned deductions of $\mathcal{T}$.

3.4.4 The evaluation semantics proof of correctness

To ease readability in this section, I shall abbreviate judgments of form $p\surd$ to p. Let $\mathcal{P}^{(;)}_{QF}$ be the system obtained after including the rule to $\mathcal{P}^{(;)}_{QI}$:

$$\frac{p}{l.p}$$

(cf. the I-system $\mathcal{P}_F$ on page 35.) I say a *visible occurrence* of a deduction is an occurence of $l.p$ which does not interact with any other occurrence, and which is not pruned. I write $V(\Pi)$ for the set of visible occurrences of Π.

Restriction To capture restriction in our evaluation semantics, we introduce the following *scoping* side-condition. Let Π be a QI-deduction, let X be a subset of the set of conclusions of Π and let $L \subset Lab$. Then X *scopes* L *in* Π if for all $A, B \in O(\Pi)$ and $l \in L$,

$$\begin{array}{ll} (i) & \text{if } l.p \sim_\Pi \bar{l}.q \text{ and } l.p \gtrsim_\Pi X \text{ then } \bar{l}.q \gtrsim_\Pi X \\ (ii) & \text{for all } l.p \in V(\Pi), \text{ if } l.p \gtrsim X \text{ then } l \notin L \cup \bar{L} \end{array}$$

where $A \gtrsim_\Pi X$ means that $A \gtrsim_\Pi B$ for some $B \in X$. Thus if X scopes L in Π, then either both ends of an L-labelled interaction must occur above X or both ends must not. We write the side-condition graphically

$$\frac{\llcorner P_1 \ \cdots \ P_n \lrcorner_L \ P}{C} \qquad \text{e.g., restriction:} \qquad \frac{\llcorner p \lrcorner_L}{p \setminus L}$$

to mean that $\{P_1, \ldots, P_n\}$ scopes L in whatever deduction the rule atom is applied to. In a deduction, we draw a box around the scoped trees. Throughout the rest of this section, I shall write the name of the system $\mathcal{P}^{(;)}_{QF}$ extended to include restriction $\mathcal{P}$.

The equivalence First I define an equivalence between deductions. I write $\Pi \simeq \Sigma$ if there exists a function $f : V(\Pi) \leftrightarrow V(\Sigma)$ such that for all $l.p \in V(\Pi)$, $f(l.p) = l.q$ for some q, and for all $A, B \in V(\Pi)$

$$A \lesssim_\Pi B \text{ iff } f(A) \lesssim_\Sigma f(B)$$

It says that if $l.p$ is ordered before (or simultaneously with) $l'.q$ in Π, then we get an occurrence $l.p'$ occurring before (or simultaneously with) an $l'.q'$ in Σ. The following two propositions are straightforward (but tedious) to prove:

Proposition 3.12 *Let* $\Pi, \Pi_1, \Pi_2, \Sigma, \Sigma_1, \Sigma_2 \in \mathbf{QI}(\mathcal{P})$ *have single conclusions. Then*

$$\text{if } \Pi \simeq \Sigma \text{ then} \qquad \frac{\begin{array}{c}\Pi\\ p\end{array}}{l.p} \quad \simeq \quad \frac{\begin{array}{c}\Sigma\\ q\end{array}}{l.q}$$

$$\text{if } \Pi_i \simeq \Sigma_i \text{ for } i = 1, 2 \text{ then} \qquad \frac{\begin{array}{cc}\Pi_1 & \Pi_2\\ p & q\end{array}}{p|q} \quad \simeq \quad \frac{\begin{array}{cc}\Sigma_1 & \Sigma_2\\ p' & q'\end{array}}{p'|q'}$$

□

Now we can define the equivalence between processes. We write $p \simeq q$ if for all Π such that $\Pi \vdash p$, there exists a $\Sigma \vdash q$ such that $\Pi \simeq \Sigma$ and vice-versa. This can be seen as a trace equivalence.

Well-terminating processes Once again, we must define the notion of well-terminating process. p is *well-terminating* if for every deduction $\Pi \vdash p$, (1) for no q is $\mathrm{done}.q \in V(\Pi)$, and (2) if $\overline{\mathrm{done}}.q \in V(\Pi)$ then $q \simeq 0$ and for all $A \in V(\Pi)$, $A \lesssim_{\Pi} \overline{\mathrm{done}}.q$.

Proposition 3.13 *For every* $p \in \mathrm{P}(;)$, $\boldsymbol{c}(p)$ *is well-terminating.*

Proof: by induction on the structure of p. **Case 0:** The only complete deduction is:

$$\frac{0}{Done}$$

which has only one visible occurrence, $\overline{\mathrm{done}}.0$. This is obviously well-terminating. **Case $l.p$:** Let $\Pi \vdash \boldsymbol{c}(l.p)$ be such that

$$\Pi = \frac{\begin{array}{c}\Pi' \\ \boldsymbol{c}(p)\end{array}}{l.\boldsymbol{c}(p)}$$

by induction, $\boldsymbol{c}(p)$ is well-terminating, so for no q is $\mathrm{done}.q \in V(\Pi)$ and $l.p \lesssim_{\Pi} \overline{\mathrm{done}}.q$ if $\overline{\mathrm{done}}.q \in V(\Pi)$ because then it must be a member of $V(\Pi')$, from which we also get $q \simeq 0$. **Case $p|q$:** Let $\Pi \vdash \boldsymbol{c}(p|q)$ consist of the three subdeductions $\Pi_1 \vdash \boldsymbol{c}(p)[d_1/\mathrm{done}]$, $\Pi_2 \vdash \boldsymbol{c}(q)[d_2/\mathrm{done}]$ and $\Pi_3 \vdash d_1.d_2.\overline{\mathrm{done}}.0$. Now by induction, $\boldsymbol{c}(p)$ and $\boldsymbol{c}(q)$ are well-terminating, so the two parts of well-terminatedness are satisifed by $\Pi_1[\mathrm{done}/d_1] \vdash \boldsymbol{c}(p)$ and $\Pi_2[\mathrm{done}/d_2] \vdash \boldsymbol{c}(q)$. They are obviously true of Π_3. Therefore, since in Π, d_1 and d_2 are restricted, they must be connected to their partners in Π_3 by an interaction link. But this means that every visible occurrence in Π_1 and Π_2 precedes the only occurrence of $\overline{\mathrm{done}}.0$, in Π_3. **Case $p;q$:** Let $\Pi \vdash \boldsymbol{c}(p;q)$ consist of the two deductions $\Pi_1 \vdash \boldsymbol{c}(p)[b/\mathrm{done}]$ and $\Pi_2 \vdash b.\boldsymbol{c}(q)$. By induction, $\boldsymbol{c}(p)$ and $\boldsymbol{c}(q)$ are well-terminating, so therefore $\Pi_1[\mathrm{done}/b]$ must satisfy the two properties of well-terminatedness, and so must Π_2. Therefore $\bar{b}.q$ is preceded by every visible occurrence in Π_1 and $q \simeq 0$. Since b is restricted, the occurrence of $\bar{b} \cdot 0$ must be linked to $b.\boldsymbol{c}(q)$ in Π, so therefore, the occurrence of $\overline{\mathrm{done}}.0$ in Π_2 must be preceded by every other visible occurrence in Π. $\square$

In the following proof, when Π_1, Π_2 are single-conclusioned, let me write $\Pi_1 \blacktriangleright \Pi_2$ for

$$\frac{\begin{array}{ccc}\Pi_1 & & \Pi_2 \\ p & \blacktriangleright & q\end{array}}{p;q}$$

This proof also depends on a result proved in section 4.2.1.

Corollary VIIa *Let Σ_1 and Σ_2 be* QI*-structures whose formulae are those of $\mathcal{P}$ such that $\Sigma_1 \simeq \Sigma_2$. If Σ_1 is the result of breaking interaction links in Σ_1' then there exists a Σ_2' such that $\Sigma_1' \simeq \Sigma_2'$ and Σ_2 is the result of breaking interaction links of Σ_2'.*

Proposition 3.14 *For all $p \in$*P(;)*, $p; Done \simeq \boldsymbol{c}(p)$.*

Proof: By induction on the structure of p. **Case 0:** trivial. **Case $\boldsymbol{l.p}$:** trivial.

Case $\boldsymbol{p_1|p_2}$: let $\Pi \vdash (p_1|p_2); Done$. Then breaking the interaction links between them, we obtain three smaller deductions: $\Pi_1 \vdash p_1$, $\Pi_2 \vdash p_2$ and $\Pi_3 \vdash Done$. Now by induction, $p_1; Done \simeq \boldsymbol{c}(p_1)$ and $p_2; Done \simeq \boldsymbol{c}(p_2)$. Therefore, we obtain partial deductions $\Sigma_1 \vdash \boldsymbol{c}(p_1)$ and $\Sigma_2 \vdash \boldsymbol{c}(p_2)$ such that $\Pi_1 \blacktriangleright \Pi_3 \simeq \Sigma_1$ and $\Pi_2 \blacktriangleright \Pi_3 \simeq \Sigma_2$. Now, there are two cases, depending on whether or not Π_3 deduces *Done* by the pruning rule or not. If so, then neither Σ_1 nor Σ_2 will have a $\overline{\text{done}}$-labelled visible occurrence. Therefore, when $\Sigma_3 = \perp$, we obtain Σ' equal to

$$\dfrac{\boxed{\dfrac{\dfrac{\begin{array}{c}\Sigma_1[d_1/\mathrm{done}]\\ \boldsymbol{c}(p)'\end{array} \qquad \begin{array}{c}\Sigma_2[d_2/\mathrm{done}]\\ \boldsymbol{c}(q)'\end{array}}{\boldsymbol{c}(p)' \mid \boldsymbol{c}(q)'} \qquad \begin{array}{c}\Sigma_3\\ d_1.d_2.Done\end{array}}{\boldsymbol{c}(p)' \mid \boldsymbol{c}(q)' \mid d_1.d_2.Done}}_{\{d_1,d_2\}}}{\boldsymbol{c}(p|q)}$$

where $\boldsymbol{c}(p)'$ is an abbreviation of $\boldsymbol{c}(p)[d_1/done]$ and $\boldsymbol{c}(q)'$ is an abbreviation of $\boldsymbol{c}(q)[d_2/done]$. By proposition 3.12, $\Sigma' \simeq \Pi'$ where Π' is the parallel composition of Π_1 and Π_2. By corollary VIIa we can reflect the pattern of interaction links between Π_1 and Π_2 in Π to between their corresponding occurrences in Σ_1 and Σ_2 in Σ', to yield Σ such that $\Sigma \simeq \Pi$.

The second case, when $\Pi_3 \neq \perp$ implies that $P(\Pi_1) = P(\Pi_2) = \emptyset$ by strict pruning. Therefore, Σ_1 and Σ_2 must be deductions of $\boldsymbol{c}(p)$ and $\boldsymbol{c}(q)$. And therefore, when Σ_3 is the deduction

$$\dfrac{\dfrac{\dfrac{\overline{\;0\;}}{\overline{\mathrm{done}}.0}}{d_2.\overline{\mathrm{done}}.0}}{d_1.d_2.\overline{\mathrm{done}}.0}$$

We construct Σ as above, except that before we reflect the pattern of interactions between Π_1 and Π_2 between Σ_1 and Σ_2, we also add interaction links between $\overline{d_1}.0$ of Σ_1 and $d_1.d_2.Done$ of Σ_3 and also between $\overline{d_2}.0$ of Σ_2 and $d_2.Done$ of Σ_3. Once again, $\Sigma \simeq \Pi$.

The reverse direction is symmetrical.

Case $\boldsymbol{p_1; p_2}$: Let $\Pi \vdash (p_1; p_2); Done$. Then Π sequences $\Pi_3 \vdash Done$ after $\Pi_2 \vdash p_2$ which in turn is sequenced after $\Pi_1 \vdash p_1$. There are two cases. In the first case, $\Pi_2 = \Pi_3 = \perp$. By induction, $p_1; Done \simeq \boldsymbol{c}(p_1)$. Therefore we obtain a $\Sigma_1 \vdash \boldsymbol{c}(p_1)$ such that $\Pi_1 \blacktriangleright \perp \simeq \Sigma_1$. This means that $\Pi_1 \simeq \Sigma_1$. This means that

$\overline{\text{done}}.0$ is not a visible occurrence of Σ_1. Therefore, we can build Σ equal to

$$\dfrac{\boxed{\dfrac{\begin{matrix}\Sigma_1 \\ \boldsymbol{c}(p)[b/\text{done}]\end{matrix} \qquad \begin{matrix}\bot \\ b.\boldsymbol{c}(q)\end{matrix}}{\boldsymbol{c}(p)[b/\text{done}] \mid b.\boldsymbol{c}(q)}}_{\{b\}}}{\boldsymbol{c}(p;q)}$$

and it is easy to see that $\Pi \simeq \Sigma$, because the only visible occurrences of Π occur in Π_1, and the only visible occurrences of Σ occur in Σ_1. The second case is when $\Pi_2 \neq \bot$. This must mean that Π_1 contains no pruned occurrences, because otherwise $\Pi_2 = \bot$. Then by induction, we get $\Sigma_1 \simeq (\Pi_1 \blacktriangleright \Delta)$ (where Δ is a complete deduction of *Done*) and $\Sigma_2 \simeq (\Pi_2 \blacktriangleright \Pi_3)$. Then we construct Σ

$$\dfrac{\boxed{\dfrac{\begin{matrix}\Sigma_1 \\ \boldsymbol{c}(p)[b/\text{done}]\end{matrix} \qquad \begin{matrix}\Sigma_2 \\ \dfrac{\boldsymbol{c}(q)}{b.\boldsymbol{c}(q)}\end{matrix}}{\boldsymbol{c}(p)[b/\text{done}] \mid b.\boldsymbol{c}(q)}}_{\{b\}}}{\boldsymbol{c}(p;q)}$$

where we have placed an interaction link between the occurrence of $\overline{b}.0$ in $\Sigma_1[b/\text{done}]$ and $b.\boldsymbol{c}(q)$ in Σ_2. It is not hard to see that $\Sigma \simeq \Pi$. □

3.4.5 Appraisal (equivalence, nondeterminism, nontermination)

Perhaps the most obvious lesson is that evaluation semantics are restrictive in the kinds of equivalences they can define. We have seen a trace equivalence which is fairly natural. However, it is well known that trace equivalence can equate possibly deadlocking processes with non-deadlocking processes. *Failures equivalence* [Hoa85], or equivalently *testing equivalence* [dNH84], appears to be the weakest equivalence which does not confuse deadlocking and deadlock-free processes. Can we give an evaluation semantics characterization of it?

A *failure* is a pair (t, X) where t is a trace and X is a set of actions none of which a process having this failure will be able to respond to after t. The problem is that this requires the notion of intermediate state, something which evaluation semantics do not express naturally[1]. So whereas we probably could characterize failures equivalence, it would not be very natural (nevertheless we shall shortly see another, more natural way to characterize deadlock in evaluation semantics).

Another difficulty is that in order to capture partial traces, we had to introduce extra machinery to prune deductions strictly. Neither of these ideas are hard, but

[1] *Note added in Proof:* for an alternative approach, see [PR96]

their use seems to subtract from the conceptual simplicity of evaluation semantics. But then again, we do not use transition semantics 'neat' either — we have to define what weak traces are using the notions of deduction sequences and silent action, neither of which are required in the evaluation semantics.

Nonetheless, in terms of the actual proof, once the problem of partiality has been overcome, and the idea of "deduction as trace" has been accepted, we do notice a reduction in complexity. The proofs of propositions 3.10 and 3.11 require nested inductions on the length of transition sequences which are missing in the corresponding evaluation semantics proofs (of propositions 3.13 and 3.14 respectively).

The problem of Angelic nondeterminism

Actually, strict pruning is more than a technique to capture a notion of trace in evaluation semantics. By providing a uniform treatment of "interrupted" evaluations, the pruning rule has potential to describe deviant behaviour naturally, such as escape features. In section 5.1.4, we use it to describe a propagation-free characterization of self-abortion.

It also solves the problem of *angelic nondeterminism*. Consider the instantly deadlocking process $\pitchfork$. We saw that without pruning, we could not deduce $a.\pitchfork\surd$ (quite rightly, since $a.\pitchfork$ does not terminate successfully), but more, we could not say anything about its behaviour. Thus when we have a nondeterministic choice operator $+$, an evaluation semantics can only give meaning to and reason about the successful choices. An example of this problem occurs in the completeness proofs for total correctness Hoare Logics (e.g., see [NN92, p.197]).

Nontermination

In fact, the pruning rule is based on the pruning judgments of [Mit94] (of form $p \Rightarrow \bot$ for some program p) where he uses them to handle nontermination. However, the judgment form $p \Rightarrow \bot$ does not distinguish nonterminating, divergent and deadlocked behaviours. It simply means "p may (or may not) exhibit unsuccessful behaviour". The same is true in my system: the judgment $p\surd$ says nothing about whether or not p terminates successfully.

However, we do much better when we look at the Heyting semantics of a judgment. Let me write $\mathcal{T}(A)$ for the set of $\mathcal{T}$-deductions of A. Then we obtain a very natural partial order $\sqsubseteq$ over $\mathcal{P}_\bot(p\surd)$:

$$\Pi_1 \sqsubseteq \Pi_2 \quad \text{iff} \quad \lesssim_{\Pi_1} \subseteq \lesssim_{\Pi_2}$$

For example, we obtain the following chain of deductions in $\mathcal{P}_\bot(a.p|\bar{a}.q\surd)$:

$$\begin{array}{c}\bot\\ a.p|\bar{a}.q\surd\end{array} \;\sqsubseteq\; \dfrac{\overset{\bot}{a.p\surd}\quad \overset{\bot}{\bar{a}.q\surd}}{a.p|\bar{a}.q\surd} \;\sqsubseteq\; \dfrac{\dfrac{\overset{\bot}{p\surd}}{a.p\surd}\quad\dfrac{\overset{\bot}{q\surd}}{\bar{a}.q\surd}}{a.p|\bar{a}.q\surd} \;\sqsubseteq\; \dfrac{\dfrac{\overset{\bot}{p\surd}}{a.p\surd}\;\text{—}\;\dfrac{\overset{\bot}{q\surd}}{\bar{a}.q\surd}}{a.p|\bar{a}.q\surd} \;\sqsubseteq \cdots$$

This looks similar to the bottom-up method of constructing deductions, related to the tableaux method [Sun84b,Und95,MP93]. In fact, given the strict pruning condition, it also corresponds to an evaluation order for processes.

Now, if X is a chain of elements of $\mathcal{P}_{\perp}(p\surd)$ then we say it has a *limit* if there exists a deduction $\Pi \in \mathcal{P}_{\perp}(p\surd)$ such that $\lesssim_{\Pi} = \bigcup_{\Sigma \in X} \lesssim_{\Sigma}$. Let me write $\lim X$ for the set of limits of chains in X. Then we can distinguish different behaviours of p:

p may terminate if $\mathcal{P}(p\surd) \neq \emptyset$
p may deadlock if $\lim \mathcal{P}_{\perp}(p\surd) \neq \mathcal{P}(p\surd)$
p might not terminate if there exists a limitless chain in $\mathcal{P}_{\perp}(p\surd)$

Note that the presence of limitless chains means that $\sqsubseteq$ is not a complete partial order. To obtain a complete partial order, one needs to introduce a notion of infinite (or non well-founded) deduction.

Coinductive definitions

The interpretion of $p\surd$ as "p terminates successfully" is due to the inductive definition of QI-deduction. If it were *coinductively* defined [MT92,Sch95,Tof87] (i.e., we take the greatest fixed point interpretation of the inference rules [Acz77]) then nonterminating evaluations could be represented by non-well founded QI-deductions, or *codeductions*. Non-well founded codeductions are not finite and therefore cannot be QI-structures, but it is not hard to define a notion of structure which may contain infinitely deep trees. The codeductive interpretation of $p\surd$ is "p does not deadlock".

However, when I tried this (for an extension of P(;) which introduced nontermination), I discovered problems with sequencing, related to the problems [Sch95] notes when using infinitely deep trees to give an evaluation semantics for a simple applicative language. There, a judgment has form $\rho \vdash e \Downarrow v$ intended to mean "in environment ρ, e evaluates to v". But if e does not terminate, one can codeduce $\rho \vdash e \Downarrow v$ for *any* v. Therefore what the judgment actually means is "in environment ρ, it is not the case that e doesn't evaluate to v". This is not useful. So in both his and my situations, there are too many codeductions.

[Sch95] solves the problem by constraining the set of acceptable deductions. The basic idea is that if one can codeduce $\rho \vdash e \Downarrow v$ for *any* v, then the only *acceptable* codeduction for e concludes $\rho \vdash e \Downarrow \perp$ where $\perp$ is the least-defined value familiar from denotational semantics. I managed to transfer this idea to my setting, but it was not particularly simple. I tried to use this theory of codeductions to prove the correctness of a coding for sequential composition in the nonterminating case. I gave up because the proof techniques became sophisticated, and idea of the proof became obscured by detail. I concluded that a transition semantics (whether ordinary or interacting) would be much better. Perhaps codeduction could be used usefully in [Sch95]'s setting of abstract interpretation.

Representativity

The translation correctness example has indicated that the statement and proof of the correctness of the compilation of P(;) into P is structurally simpler using an evaluation semantics than a transition semantics. The question is how representative the example is: the translation is very simple.

However, my example is representative in that no matter how complex the compiler is, each direction of the correctness proof will always entail a simulation proof. Also, the example used every feature of QI-rules. More complex languages like CSP can also be expressed using QI-systems. Therefore, I should expect that even though the individual rules may be more complex, the kinds of reasoning involved in a correctness proof will remain the same. Last, I believe that the example is representative in that evaluation semantics are less detailed than transition semantics, and the correctness result does not require the extra detail.

3.5 Chapter Summary

We saw that the basic theory of interacting deductions was powerful enough to capture sequential composition, and in at least two language independent ways. Hence the theory is powerful enough to give evaluation semantics to a wide variety of programming languages. Moreover, it managed to give a simple treatment of control-stack semantics.

However, though both methods were general, they were not perspicuous. Therefore we added a notion of sequencing to the metatheory, purely to make representation easier. It does not affect the range of languages we can give semantics to. The presentation of P(;) was much simpler in the evaluation semantics than in the SOS. However, the issue was not so clear regarding the translation correctness proof. The structure of the evaluation semantics proof was simpler than that of the transition semantics proof, but it required extra machinery to deliver the partial deductions. Moreover, all that gave us was a trace equivalence: it was hard to see how to define other equivalences.

On the other hand, the pruning technique is not *ad-hoc*: it provides us with a general way of talking about aberrant behaviours of programs: deadlock, nontermination and escaping.

Chapter 4

The content of interaction

So far deductions have recorded interactions using pairs of occurrences. This approach merely records that an interaction occurred. It does not record what the interaction was about. In this chapter we shall consider a notion of deduction that includes this extra information.

The result, DQI-deduction, is more elegant than both I- and QI-deduction. Moreover, it supports not only proof fragmentation, but also a dual technique called *proof assembly*, illustrated with a type-checking example and another non-divergence proof (section 4.3). In short it improves the modular treatment of systems (in particular semantic definitions) and proofs about them.

However, the primary purpose of DQI-deduction is to provide a finer (and more accessible) account of QI-deduction. The need is clear. First, the definition of QI-deduction is not particularly elegant. Second, we have been forced to define concepts in the semantics which are more properly concepts of the structure of deductions. For example, the visibility of occurrences (defined in section 3.4.4) depends solely on whether or not the occurrence would interact in a deduction. For another example, the proof of proposition 3.14 (just like the proof of lemma 2.13(i)) uses the technique of breaking and then reassembling interaction links between the children of an inference tree: something which could be done generally. The problem with definition 3.0 is that it does not support this kind of reasoning easily.

To achieve this finer account in a general way implies that we should enrich the structure of deduction again to include the notion of what it is for an interaction link to be broken.

Observations about the environment

What does it mean for an interaction to be broken? In our leading illustration, the community of mathematicians, "breaking" would suggest that communication is cut off (e.g., between the Russians and the Americans). This is a good analysis of what happens during fragmentation. However, the idea is too strong for the proof technique of breaking and reassembling interaction links. There we break links in order to restrict our field of view (say to the Russians). Reassembly means that once we have studied the Russians we are then interested in how they interact with the Americans. Thus breaking an interaction link only means that we are not currently interested in both halves of the interaction.

For example, the KGB might be interested in the activities of Russian mathematicians, but not their American colleagues. They would tape all of the international conversations, but would not care about American work. Similarly, the CIA might be interested in American mathematicians, but not Russian ones. Only the UN might be interested in both: having taped every conversation between the two countries from both ends, it is in the unique position of being able to reconstruct the entire history of interaction between the two countries, simply by matching up the taped conversations.

So a broken, or better, *dangling* interaction link is just the perception of an interaction from one side. The content of the interaction, the conversation or flow of ideas, is shared by both parties, although they will perceive it in opposite ways: Russians hear what Americans say, and vice-versa.

Therefore, a dangling interaction is formally a pair (A, α) where A is a formula occurrence and α is a *perception of a formula*: a pair (A, s) where A is a formula denoting the content of an interaction (for technical simplicity later on, I shall assume that it takes the form $P(t_1, \dots, t_n)$ for some n-ary predicate letter P) and $s \in \{+, -\}$ is the polarity of the perception. I write $+A$ for $(A, +)$ and $-A$ for $(A, -)$. It does not matter what $+$ and $-$ "mean"; it is enough that they denote opposite perceptions. I shall write $\overline{+A} = -A$ and $\overline{-A} = +A$. Last, if X is a set of formulae, I write $X^{\pm}$ for the set of perceptions of formulae.

Hypotheses about the environment

Thus we can view α as an observation about the environment in which the relevant mathematician works. However, we can also view it as a *hypothesis*: the perception α is a hypothesis of A. The crucial point here is that a Russian will perceive α only if an American simultaneously percieves $\bar{\alpha}$ — i.e., when an interaction occurs between them. An interaction link is formed when two dangling interactions (A, α) and $(B, \bar{\alpha})$ are assembled. (The slogan is *assembly is discharge*: section 4.3). Just as in Natural Deduction, a formula is properly deduced only when there exist no undischarged hypotheses — i.e., no dangling interactions. Therefore, the process of deduction must consist of rule application (inference) and assembly (hypothesis discharge).

Rules and the principle of meaningful communication

Since we are incorporating assembly into the definition of deduction, the simplest definition of L-labelled DQI-rule is the pair (a, D), where a is a Q-atom, and D is a finite set of perceptions of L-formulae. Finiteness ensures that deductions are finite objects. Conceptually, it corresponds to the fact that mathematicians can only share a finite number of ideas at any point in time: if they can only share what they have deduced, and at each point in time they can have deduced only a finite amount of information, they cannot talk about an infinite number of facts.

Since an interaction is an exchange of ideas, the objects of discourse must be familiar to both parties. In a rule schema, the objects of discourse are represented by the free variables of the perceived interaction (or interaction hypothesis). Therefore, since objects of discourse must be familiar to both parties, every free variable in a perception must either appear in the rule atom (i.e., be used in the inference) or in another perception (in which case the rule is relaying this information). I call this the *principle of meaningful communication*. Formally, it is expressed:

$$\text{for all } \alpha \in D, FV(\alpha) \subseteq FV(a) \cup \bigcup\{FV(\beta) \mid \beta \in D,\ \beta \neq \alpha\}$$

Semantic rules will tend to observe this principle naturally.

I write DQI-rules graphically as

$$\frac{P_1 \quad \ldots \quad P_n}{C}\ D$$

It will often be convenient to write sets of rules as if they were connected via some pattern of interaction links. Thus I should write

$$\frac{Prem_1}{C_1} \ \overset{A}{\text{———}}\ \frac{Prem_2}{C_2} \qquad \text{for} \qquad \frac{Prem_1}{C_1}\ {+A} \quad \text{and} \quad \frac{Prem_2}{C_2}\ {-A}$$

for example. As a further convention, given that $A = P(x_1, \ldots, x_n)$ in a rule schema, I shall often abbreviate A above a pseudo-interaction link to P. The idea in this case is that the variables are just those used or relayed by both parties on either side of the link.

DQI-systems A DQI-system is a pair $(\mathcal{L}, \mathcal{R})$, where $\mathcal{L}$ is a language and $\mathcal{R}$ is a set of $\mathcal{L}$-labelled DQI-rules which mention only formulae in $\mathcal{L}$.

To illustrate the following definitions, we give a DQI-system $\mathcal{P}^{(;)}_{DQI} = (\mathcal{L}^{(;)}_{DQI}, \mathcal{R}^{(;)}_{DQI})$ where $\mathcal{L}^{(;)}_{DQI} = \mathcal{L}^{(;)}_{QI} \cup \{comm(a) \mid a \in Nam\}$, and $\mathcal{R}^{(;)}_{DQI}$ is the set of rules generated by:

$$\frac{}{0\surd} \qquad \frac{p\surd}{a.p\surd} \ \overset{comm(a)}{\text{———}}\ \frac{p\surd}{\bar{a}.p\surd} \qquad \frac{p\surd \qquad q\surd}{p|q\surd} \qquad \frac{p\surd \quad \blacktriangleright \quad q\surd}{p;q\surd}$$

where given our convention, the communication rule is really the two rules

$$\frac{p\surd}{a.p\surd}\ {+comm(a)} \qquad \frac{p\surd}{\bar{a}.p\surd}\ {-comm(a)}$$

4.1 DQI-deduction

4.1.1 Structures, Histories and Binary Assemblies

Labelled binary relations Let X be a set and L a set of labels. Then a *labelled binary relation* R is a subset of $X \times L \times X$. I say R is symmetric if $(A, \alpha, B) \in Y$ implies $(B, \bar{\alpha}, A) \in Y$. The symmetric closure of R, written R° is the set $R \cup \{(B, \bar{\alpha}, A) \mid (A, \alpha, B) \in R\}$. I write $R^\flat = \{(A, B) \mid \exists \alpha.(A, \alpha, B) \in R\}$ for the flattening of R to a binary relation.

For example, when we come to assemble two opposite dangling interactions in a DQI-structure, the interaction link will be represented by a pair of triples $(a.p\surd, +comm(a), \bar{a}.q\surd)$ and $(\bar{a}.q\surd, -comm(a), a.p\surd)$.

DQI-structures

An L-*labelled* DQI-*structure* is a quadruple $(F, I, \sqsubset, D)$ where F is a formula forest, $I \subseteq O(F) \times L \times O(F)$ is symmetric, $\sqsubset \subseteq O(F) \times O(F)$ and $D \subseteq O(F) \times L$ is finite. I shall use Σ to range over structures. If $\Sigma = (F, I, \sqsubset, D)$, I write $D(\Sigma)$ for D, the set of its dangling interactions. Two structures are disjoint if they share no common occurrences. The *union* of two disjoint structures is just the pointwise union of the data in the quadruples.

For example, the following is a graphical representation of a DQI-deduction built out of the previous ruleset:

$$\dfrac{\dfrac{0\surd^1}{a.0\surd} \qquad \dfrac{\dfrac{0\surd^2}{\bar{a}.0\surd} \;\blacktriangleright\; \dfrac{0\surd^3}{b.0\surd}\; +comm(b)}{\bar{a}.0; b.0\surd}}{a.0|(\bar{a}.0; b.0)\surd}$$

Where F is the singleton set containing the tree of occurrences, $\sqsubset = \{(\bar{a}.0\surd, b.0\surd)\}$, $I = \{(a.0\surd, +comm(a), \bar{a}.0\surd), (\bar{a}.0\surd, -comm(a), a.0\surd)\}$ and D is the singleton set containing only $(b.0\surd, +comm(a))$.

Note that being $(F, I, \sqsubset, D)$ a DQI-structure does not imply that $(F, I, \sqsubset)$ is a QI-structure: apart from being labelled, I is only symmetric and not an equivalence relation. This is a consequence of the decision to take a finer view of interaction than "everybody interacts with everybody else". The QI-structure corresponding to $(F, I, \sqsubset, D)$ is $(F, (I^\flat)^*, \sqsubset)$.

Preorders Dangling interactions do not add extra dependency information, so the preorder of a DQI-structure $(F, I, \sqsubset, D)$ is just the analogue of the preorder for QI-structures:

$$\precsim_{(F,I,\sqsubset,D)} = \precsim_{(F,(I^\flat)^*,\sqsubset)} = (\precsim_F \cup I^\flat \cup \widehat{\sqsubset})^*$$

Histories

Let $\Sigma = (F, I, \sqsubset, D)$ be a DQI-structure. Then $h : O(\Sigma) \to \mathbb{N}$ is a *history* of Σ if for all $A, B \in O(\Sigma)$,

$$\begin{array}{l} \text{if } A <_F B \text{ then } h(B) > h(A) \\ \text{if } A \hat{\sqsubset} B \text{ then } h(B) > h(A) \\ \text{if } A \, I \, B \text{ then } h(B) = h(A) \end{array}$$

It is easy to show that if h is a history of Σ and $A \lesssim_\Sigma B$ then $h(B) \geq h(A)$ (simply consider the path of occurrences from A to B in Σ).

For example, the following is a history of the above deduction:

A	$a.0 \vert (\bar{a}.0; b.0)\surd$	$\bar{a}.0; b.0\surd$	$a.0\surd$	$\bar{a}.0\surd$	$0\surd^2$	$b.0\surd$	$0\surd^3$	$0\surd^1$
$h(A)$	6	5	4	4	3	2	1	0

And note that this history does indeed correspond to a possible sequence of rules applications to build the deduction.

Proposition 4.0 *Σ has a history iff Σ satisfies SLF.*

Proof: Similar to that of propositions 3.4 and 3.8. □

Assemblies

We want to take a set of structures and connect their dangling interactions together. To do this we must specify which dangling interactions are to be assembled. A *connector* is a function that maps dangling interactions to their intended partners. Of course, this means that a connector should be involutive. However, we also want to ensure that we do not introduce a sequencing loop. The easiest way is to use a history: if two dangling interactions can be synchronized in a history then they can be connected.

Connectors A *connector* of a DQI-structure Σ with respect to a history $h : O(\Sigma) \to \mathbb{N}$ is an involution $f : H \leftrightarrow H$ (where $H \subseteq D(\Sigma)$) such that for all $(A, \alpha) \in H$, $f(A, \alpha) = (A', \bar{\alpha})$ for some $A' \neq A$, and $h(A) = h(A')$. I write

$$\bigotimes{}_f H = \{(A, \alpha, B) \mid (A, \alpha) \in H, f(A, \alpha) = (B, \bar{\alpha})\}^\circ$$

Whenever f is a connector for Σ. Extending notation, I say that a connector of a set of disjoint structures is a connector for the union of the structures.

The above example has only one connector: the empty one. However, suppose its interaction link was broken, i.e., that $I = \emptyset$ and D contained three elements $(a.0\surd, +comm(a))$, $(\bar{a}.0\surd, -comm(a))$ and $(b.0\surd, +comm(b))$. Then we should have another connector $f : H \leftrightarrow H$ where H contains the two elements $(a.0\surd, +comm(a))$ and $(\bar{a}.0\surd, -comm(a))$, and

$$\begin{array}{l} f(a.0\surd, +comm(a)) = (\bar{a}.0\surd, -comm(a)) \\ f(\bar{a}.0\surd, -comm(a)) = (a.0\surd, +comm(a)) \end{array}$$

This is a connector with respect to the history given on the previous page. In this case, $\bigotimes_f = \{(a.0\surd, +comm(a), \bar{a}.0\surd), (\bar{a}.0\surd, -comm(a), a.0\surd)\}$ which is just the set I of the original deduction. If $b = a$, then we would have another possibility, of connecting $\bar{a}.0\surd$ to $b.0\surd$, but this would require a different history because $h(\bar{a}.0\surd) \neq h(b.0\surd)$.

Assembly The *assembly* of a DQI-structure $\Sigma = (F, I, \sqsubset, D)$ with respect to a connector f (written $\bigotimes_f \Sigma$) is the quadruple $(F, I', \sqsubset, D')$ where

$$I' = I \cup \bigotimes_f(\text{dom } f)$$
$$D = (D \setminus \text{dom } f)$$

In what follows, whenever I write $\bigotimes_f \Sigma$, I shall assume that f is a connector of Σ with respect to some history h. Extending notation again, I say that the assembly of a set of disjoint structure with respect to a connector is just the assembly for the union of the structures (i.e., $\bigotimes_f\{\Sigma_1, \ldots, \Sigma_n\} = \bigotimes_f(\Sigma_1 \cup \ldots \cup \Sigma_n)$).

Proposition 4.1 $\bigotimes_f(Y \cup \{\bigotimes_g X\}) = \bigotimes_{f\oplus g}(X \cup Y)$

Proof: Since dangling interactions can be assembled only once, we get dom $g \cap$ dom $f = \emptyset$ and im $g \cap$ im $f = \emptyset$. Since there is no interference between the two connectors we can easily combine them. The result, $f \oplus g$ is involutive because f and g are, maps perspectives into their opposite perspectives because f and g do and satisfies the history requirement because f and g do. □

Proposition 4.2 *Let Σ satisfy SLF. Then $\bigotimes_f \Sigma$ satisfies SLF.*

Proof: Since Σ satisfies SLF, it has a history (proposition 4.0). Now, by the definition of connector, we know that for some history of Σ, for all $(A, \alpha), (A', \bar{\alpha}) \in$ dom f such that $f(A, \alpha) = (A', \bar{\alpha})$, that $h(A) = h(A')$. But this means that h is also a history of $\bigotimes_f(\Sigma)$: if (B, β, B') is an interaction in $\bigotimes_f(\Sigma)$ then either it is an interaction of Σ, in which case $h(B) = h(B')$, or it belongs to $\bigotimes_f(\text{dom } f)$. But then, $f(B, \beta) = (B', \bar{\beta})$, and so $h(B) = h(B')$ again. The other two properties of histories follow because we do not alter the trees or sequencing relations of Σ. Therefore by proposition 4.0, $\bigotimes_f \Sigma$ satisfies SLF. □

Binary Assembly A special case is the assembly of two disjoint structures together. This is called *binary assembly*. A binary assembly of two disjoint structures Σ_1 and Σ_2 only adds links between the two structures, as specified by a *binary connector*. Formally, a binary connector of Σ_1 and Σ_2 is a connector of $\{\Sigma_1, \Sigma_2\}$ such that

$$\text{for all } (A, \alpha) \in (\text{dom } f \cap D(\Sigma_i)), f(A, \alpha) \in D(\Sigma_{3-i})$$

for $i = 1, 2$. (Note that when $i = 1$, $3 - i = 2$ and vice-versa.) I write $\Sigma_1 \otimes_f \Sigma_2$ for $\bigotimes_f\{\Sigma_1, \Sigma_2\}$ when f is a binary connector. Binary assembly is used in the definition of deduction because it does not satisfy the absorption propery of full assembly (proposition 4.1). If we need to assemble n structures together, we have to use $n - 1$ binary connectors. This allows us to determine a unique size for deductions, which in turn facilitates proofs by induction on the size of deductions.

For example, consider the two deductions

$$\frac{0\surd}{a.0\surd}\ {+comm(a)} \qquad\qquad \frac{\dfrac{0\surd}{b.0\surd}}{a.b.0\surd}\ {+comm(a)} \quad \frac{\dfrac{0\surd}{\bar{b}.0\surd}}{\bar{a}.\bar{b}.0\surd}\ {-comm(a)}$$

We cannot apply a binary connector to the second deduction once it has been connected. Once a deduction has been assembled, there is no way for us (using binary assembly) to go back and rewire its internals. So again, there are only two binary assemblies: the empty one, and the one that connects the dangling interaction $(a.0\surd, +comm(a))$ to $(\bar{a}.\bar{b}.0\surd, -comm(a))$.

Proposition 4.3 $\Sigma_1 \otimes_f \Sigma_2 = \Sigma_2 \otimes_f \Sigma_1$ □

For the following, if g is an injective function with disjoint domain and image, I say the *involutive closure* of g is the function $g \oplus g^{-1}$, which I write $\mathrm{Inv}(g)$. Binary connectors will be the involutive closure of an isomorphism between subsets of the dangling interactions of two structures.

Proposition 4.4 $(\Sigma_1 \otimes_f \Sigma_2) \otimes_g \Sigma_3 = \Sigma_1 \otimes_{f \oplus g_1} (\Sigma_2 \otimes_{g_2} \Sigma_3)$
where $g_i = Inv(g \restriction D(\Sigma_i))$ *for* $i = 1, 2$.

Proof: First, g_i is an isomorphism because g is and because dom $(g \restriction D(\Sigma_i)) \cap$ im $(g \restriction D(\Sigma_i)) = \emptyset$. The definition of g_i shows immediately that it is involutive. Now g_i is contained in g (since $g = g^{-1}$), so it follows that g_i satisfies the properties of being a binary connector: it connects Σ_i and Σ_{3-i}. Therefore, as dom f and dom g_i are disjoint, $f \oplus g_1$ must be a binary connector too. Now $(\Sigma_1 \otimes_f \Sigma_2) \otimes_g \Sigma_3$ equals $\bigcup_{f \oplus g}\{\Sigma_1, \Sigma_2, \Sigma_3\}$, which equals $\Sigma_1 \otimes_{f \oplus g_1} (\Sigma_2 \otimes_{g_2} \Sigma_3)$ because $(f \oplus g_1) \oplus g_2 = f \oplus g$. □

4.1.2 Deduction

This section uses the previous notions to define DQI-deduction. All deductions are DQI-structures. We use histories to capture sequencing, and assembly to model hypothesis discharge. For an informal account of the process, we return to our community of mathematicians. Previously, inference steps (rule applications) were made simultaneously by a group of mathematicians. Now we allow mathematicians to make individual steps on their own as long as they record what hypotheses they are using (or what they have perceived about a colleague's work). What makes this work is a second kind of inference step: the binary assembly of two deductions.

Definition 4.5 *DQI-deduction Let $\mathcal{T}$ be a* DQI*-system. Then the set* $\mathbf{DQI}(\mathcal{T})$ *of* DQI*-deductions of $\mathcal{T}$ is the least set such that:*

1. $0 = (\emptyset, \emptyset, \emptyset, \emptyset) \in \mathbf{DQI}(\mathcal{T})$.
2. *If* $\Sigma_1, \Sigma_2 \in \mathbf{DQI}(\mathcal{T})$ *then* $\Sigma_1 \otimes_f \Sigma_2 \in \mathbf{DQI}(\mathcal{T})$.
3. *If* (a) $(F, I, \sqsubset, D) \in \mathbf{DQI}(\mathcal{T})$
   (b) *There exists a history h of $(F, I, \sqsubset)$ such that for all $(A, B) \in \sqsubset_r$,* $\hat{h}(A) > h(B)$
   (c) *r matches a rule of $\mathcal{T}$ such that*

$$\mathit{flat}(r) = \frac{\mathrm{roots}(F)}{C}\ D_r$$

then $(\{\frac{F}{C}\}, I, \sqsubset \cup \sqsubset_r, D \cup \{(C, \alpha) \mid \alpha \in D_r\}) \in \mathbf{DQI}(\mathcal{T})$.

I write $\Pi \vdash A_1, \ldots, A_n$ if Π is a deduction with conclusions $A_1, \ldots, A_n$. I also write $\mathcal{T} \vdash A_1, \ldots, A_n$ if $\Pi \in \mathbf{DQI}(\mathcal{T})$. A deduction Π is *proper* if $D(\Pi) = \emptyset$, in which case I write $\Pi \Vdash A_1, \ldots, A_n$ and $\mathcal{T} \Vdash A_1, \ldots, A_n$. It is *acyclic* if the only cycles of interaction links are of the form $(A, \alpha, B), (B, \bar{\alpha}, A)$. A DQI-system is acyclic if all its deductions are.

The size of deductions This definition is much more elegant than the previous definitions, because we have not had to apply the several rule atoms at once. Thus we shall be able to prove results such as proposition 3.14 and lemma 2.13(i) more directly, without requiring special "fragmented" systems which allowed us to break interaction links in inductive proofs. To aid inductive proofs, let me define the *size* of a DQI-deduction as follows:

$$\mathrm{size}(\Sigma) = \begin{cases} 0 & \text{if } \Sigma = 0 \\ 1 + \max(\mathrm{size}(\Sigma_1), \mathrm{size}(\Sigma_2)) & \text{if } \Sigma = \Sigma_1 \otimes_f \Sigma_2 \\ 1 + \mathrm{size}(\Sigma_1) & \text{if } \Sigma = \dfrac{\Sigma_1}{C}\ D \end{cases}$$

Note that this is well-defined. As mentioned previously, because we only use binary assemblies in the definition of deduction, and binary assemblies only add interactions *between* deductions (i.e., they assemble no extra interactions *within* an argument deduction), it follows that there must exist a fixed number of binary assemblies within each deduction.

Proposition 4.6 *Every* DQI*-deduction has a history* □

Proposition 4.7 *Every* DQI*-deduction satisfies SLF.*

Proof: Straightforward from propositions 4.6 and 4.0. □

Characterizing the DQI-deductions

Once again the DQI-deductions are the DQI-structures that satisfy SLF and which are built only from the appropriate DQI-rules. The following treatment is straightforward.

DQI-neighbourhoods We need the notion of DQI-neighbourhood. As usual, I shall drop the prefix, and simply refer to neighbourhoods. Let A be an occurrence of a DQI-structure $(F, I, \sqsubset, D)$. Then its DQI-*neighbourhood* (written $N(A)$) is the triple $(N, \sqsubset_N, D_N)$ where

$$\begin{array}{l} N = \{atom(A)\} \\ \sqsubset_N = \sqsubset \cap O(N) \times O(N) \\ D_N = \{\alpha \mid (A, \alpha) \in D\} \cup \{\alpha \mid \exists B.(A, \alpha, B) \in I\} \end{array}$$

Thus the DQI-neighbourhood of A is simply the atom that introduces A in F together with information about how its premises are sequenced and its interactive ability (i.e., the interactions and dangling interactions incident to A in the structure).

Rule matching I say a rule (a, D_a) *matches* a neighbourhood $(N, \sqsubset, D_N)$ via $f : O(a) \leftrightarrow O(N)$ (written $(a, D_a) \equiv_f (N, \sqsubset, D_N)$) if

$$\begin{array}{ll} (a) & \{a\} \equiv_f (N, \sqsubset_N) \\ (b) & D_a = D_N \end{array}$$

Lemma V(i) *Let $\Sigma = (F, I, \sqsubset, D)$ be an L-labelled* DQI-*structure, and let $\Sigma_i = (F_i, I_i, \sqsubset_i, D_i)$ (for $i = 1, 2$) be disjoint structures such that $F_1 \cup F_2 = F$, $\sqsubset = \sqsubset_1 \cup \sqsubset_2$ and $I_i \subseteq I$ and for all $A \in D(\Sigma_i)$, $N_{\Sigma_i}(A) = N_\Sigma(A)$. Then there exists a connector f such that $\Sigma = \bigotimes_f \{\Sigma_1, \Sigma_2\}$*

Proof: Let us define $\Sigma' = \Sigma_1 \cup \Sigma_2 = (F, I', \sqsubset', D')$. We know that $I' = I_1 \cup I_2 \subseteq I$. Therefore, by the definition of DQI-neighbourhood, $D' = D_1 \cup D_2 \supseteq D$, otherwise there would exist a neighbourhood in either Σ_1 or Σ_2 which had a different communicative ability from its corresponding neighbourhood in Σ, which is again impossible.

Let us define $X = \{(A, \alpha) \mid \exists B.(A, \alpha, B) \in I \setminus I'\}$. We show $X = D' \setminus D$. **First** we show $X \subseteq D' \setminus D$. Let $(A, \alpha) \in X$. Then there exists B such that $(A, \alpha, B) \in I \setminus I'$. Therefore $(A, \alpha) \in D_A$ where $N_\Sigma(A) = (atom(A), \sqsubset_A, D_A) = N_{\Sigma'}(A)$. Therefore, since $(A, \alpha, B) \notin I'$, $(A, \alpha) \in D'$. Since $(A, \alpha, B) \in I$, $(A, \alpha) \notin D$. Therefore $(A, \alpha) \in D' \setminus D$. **Second** we show that $D' \setminus D \subseteq X$. Let $(A, \alpha) \in D' \setminus D$. Then $(A, \alpha) \in D_A$, which means (because $(A, \alpha) \notin D$, that for some B, $(A, \alpha, B) \in I \setminus I'$, i.e., that $(A, \alpha) \in X$.

Therefore, let $p : (I \setminus I') \leftrightarrow (D' \setminus D)$ be the isomorphism defined by $p(A, \alpha, B) = (A, \alpha)$. To assemble Σ let $f : (D' \setminus D) \leftrightarrow (D' \setminus D)$ be defined by $f = p \circ \mathrm{swap} \circ p^{-1}$, where $\mathrm{swap}(A, \alpha, B) = (B, \bar{\alpha}, A)$. Let h be any history of Σ. Then, since $\lesssim_{\Sigma'} \subseteq \lesssim_{\Sigma}$, h is also a history of Σ'. We show f is a connector.

First, $f(A, \alpha) = p \circ \mathrm{swap}(A, \alpha, B) = p(B, \bar{\alpha}, A) = (B, \bar{\alpha})$. Second, it is involutive because swap is involutive. Third, to show that it satisfies the history property, Let $(A, \alpha), (A', \bar{\alpha}) \in D' \setminus D$ be such that $f(A, \alpha) = (A', \bar{\alpha})$. Since (A, α, A') is part of an interaction link in h, it follows that $h(A) = h(A')$.

The last thing to show is that $\Sigma = \bigotimes_f \{\Sigma_1, \Sigma_2\}$. The only nontrivial things to check are that

$$\begin{aligned} I &= I_1 \cup I_2 \cup \bigotimes_f (D' \setminus D) \\ D &= D \setminus (D' \setminus D) \end{aligned}$$

But these follow from the definitions and elementary set-theoretic manipulation.

□

We can strengthen this result: when $I_i = I \cap (O(F_i) \times L \times O(F_i))$ we can show that the f constructed is in fact binary. This follows because in this case the only interaction links in $I \setminus I'$ are those that occur between the structures Σ_1 and Σ_2.

Theorem V *Let $\mathcal{T} = (\mathcal{L}, \mathcal{R})$ be a* DQI*-system. Then $(F, I, \sqsubset, D) \in \mathbf{DQI}(\mathcal{T})$ if and only if $(F, I, \sqsubset, D)$ is an $\mathcal{L}^{\pm}$-labelled* DQI*-structure satisfying SLF and such that every neighbourhood matches a rule of $\mathcal{T}$.*

Proof: $\Rightarrow$: Let $\Sigma \in \mathbf{DQI}(\mathcal{T})$. Then by proposition 4.7, it satisfies SLF. We show by strong induction on the size n of Σ that every neighbourhood matches a rule of $\mathcal{T}$. **Case $n = 0$:** vacuously true. **Case $n = k + 1$:** two subcases. First, suppose $\Sigma = \otimes_f \Sigma'$ where $\Sigma' = (F', I', \sqsubset', D') = \Sigma_1 \cup \ldots \cup \Sigma_n$, $f : H \leftrightarrow H$, and $\Sigma_i = (F_i, I_i, \sqsubset_i, D_i)$. Then by induction, every neighbourhood of Σ_i matches a rule in $\mathcal{T}$. Suppose there exists an occurrence A (say from the Σ_j part) whose neighbourhood $(N, \sqsubset_N, D_N)$ does not match a rule of $\mathcal{T}$. Let $(N', \sqsubset'_N, D'_N)$ be its neighbourhood in Σ_j. By induction, it matches (a, D_a). By the definition of neighbourhood, $(N, \sqsubset_N) = (N', \sqsubset'_N)$, so it must be that $D_N \neq D_a$ since $(N, \sqsubset_N, D_N)$ does not match (a, D_a). Thus:

$$\begin{aligned} D'_N &= \{\alpha \mid (A, \alpha) \in D_j\} \cup \{\alpha \mid \exists B.(A, \alpha, B) \in I_j\} \\ D_N &= \{\alpha \mid (A, \alpha) \in D' \setminus H\} \\ &\qquad \cup \{\alpha \mid \exists B.(A, \alpha, B) \in I' \cup \textstyle\bigotimes_f H\} \\ &= \{\alpha \mid (A, \alpha) \in (D_j \setminus H)\} \cup \{\alpha \mid \exists B.(A, \alpha, B) \in I_j \cup \textstyle\bigotimes_f H\} \end{aligned}$$

since $A \in O(\Sigma_j)$. But this means that $D_N = D'_N$ since every dangling interaction in H becomes half an interaction in $\bigotimes_f H$. Contradiction.

Second, $\Sigma = \dfrac{\Sigma_1}{C} D$ follows easily from induction, and the fact that the rule introducing C must match that rule applied to Σ_1 that made Σ.

$\Leftarrow$: Let $\Sigma = (F, I, \sqsubset, D)$ be a DQI-structure satisfying SLF and such that every neighbourhood matches a rule. We show by strong induction on the number of occurrences in F that $(F, I, \sqsubset, D)$ is in $\mathbf{DQI}(\mathcal{T})$. **Case $n = 0$:** trivial. **Case $n = k + 1$:** There are two subcases. First, suppose $F = \{\frac{F_1}{C}\}$. Let

$(atom(C), \sqsubset_C, D_C)$ be the neighbourhood of C, and let it match r via f. By SLF, there are no interaction links incident to C, so by induction, the structure $\Sigma_1 = (F_1, I, \sqsubset \setminus \sqsubset_C, D \setminus D_C)$ is in $\mathbf{DQI}(\mathcal{T})$. Now, since Σ satisfies SLF, then it satisfies the sequencing constraints of r (i.e., for every history h of Σ, and every $(A, B) \in \sqsubset_C = \{(f(A), f(B)) \mid (A, B) \in \sqsubset_r\}$, $\hat{h}(A) > h(B)$) then we can apply $f(r)$ to Σ_1 to obtain Σ. Hence $\Sigma \in \mathbf{DQI}(\mathcal{T})$.

The second subcase occurs when $|F| > 1$. Then we arbitrarily split it into two disjoint, nonempty subsets F_1 and F_2. We define (for $i = 1, 2$) $\Sigma_i = (F_i, I_i, \sqsubset_i, D_i)$ where $F = F_1 \cup F_2$, $I_i = I \cap O(F_i) \times \mathcal{L}^{\pm} \times O(F_i)$ and for all $A \in O(\Sigma_i)$, $N_{\Sigma_i}(A) = N_\Sigma(A)$. Then by induction, $\Sigma_1, \Sigma_2 \in \mathbf{DQI}(\mathcal{T})$. By lemma V(i), we find a connector f such that $\Sigma = \bigotimes_f \{\Sigma_1, \Sigma_2\}$. Moreover, since $I_i = I \cap O(F_i) \times \mathcal{L}^{\pm} \times O(F_i)$, we can show that f is binary, hence $\Sigma = \Sigma_1 \otimes_f \Sigma_2$, whence $\Sigma \in \mathbf{DQI}(\mathcal{T})$. □

4.1.3 Coding QI-deduction into DQI-deduction

The point of DQI-deduction is to provide a finer and more accessible account of QI-deduction. This section makes the claim more precise. The following theorem shows how QI-deduction can be simulated by proper DQI-deduction. To do this we have to break up QI-rules into DQI-rules such that these new rules only interact when they are pieces of the same QI-rule.

Encoding rules and rulesets We say a graph G *encodes* r with respect to L if $G = (r, E)$ (i.e., its vertices are the atoms of r) and it is both connected and acyclic, and every edge is labelled uniquely from L. I write $aE_\alpha a'$ if there is an edge from a to a' labelled by α. Let us pick one such encoding arbitrarily for each rule $r \in \mathcal{R}$ such that no two rules share a label in their encodings. Let us write G_r for this distinguished encoding of r. Then we say the graph $G_\mathcal{R}$ *encodes* $\mathcal{R}$ with respect to L if $G_\mathcal{R}$ is the union of the distinguished encodings of its rules.

Now, let $G_\mathcal{R} = (V, E)$, and let $\leq$ be an arbitrary total order over V. Let me write $a\vec{E}_\alpha a'$ if there exists an α-labelled edge from a to a' and moreover $a < a'$ in the total order. Then we define the interactive ability of each atom a of $G_\mathcal{R}$ as follows:

$$D_a = \{+\alpha \mid a\vec{E}_\alpha a'\} \ \cup \ \{-\alpha \mid a'\vec{E}_\alpha a\}$$

Then we write $\mathcal{R}(G_\mathcal{R}, \leq)$ for the set of rules $\{(a, D_a) \mid a \in V\}$, the DQI-encoding of $\mathcal{R}$ with respect to the edges of $G_\mathcal{R}$ and the order $\leq$.

Encoding QI-systems Let $\mathcal{T} = (\mathcal{L}, \mathcal{R})$ be a QI-system. Then we say that a DQI-system $\mathcal{T}' = (\mathcal{L}', \mathcal{R}')$ *encodes* $\mathcal{T}$ if there exists a set L of formulae disjoint from $\mathcal{L}$ such that $\mathcal{L}' = \mathcal{L} \cup L$ and there exists an encoding $G_\mathcal{R}$ of $\mathcal{R}$ with respect to L and a total order $\leq$ over the vertices of $G_\mathcal{R}$ such that $\mathcal{R}' = \mathcal{R}(G_\mathcal{R}, \leq)$.

Theorem VI *Let $\mathcal{T}$ be a* QI*-system, and let $\mathcal{T}'$ be a* DQI*-encoding of it. Then $\mathcal{T} \vdash A_1, \ldots, A_n$ if and only if $\mathcal{T}' \Vdash A_1, \ldots, A_n$.*

Proof: $\Rightarrow$: Let $\mathcal{T}'$ be constructed from $\mathcal{T}$ via L, $G_{\mathcal{R}} = (V, E)$ and $\leq$. Suppose there exists a QI-deduction $\Sigma = (F, I, \sqsubset)$. Then we construct a proper DQI-deduction $\Sigma' = (F, I', \sqsubset, \emptyset)$ such that $I = ((I')^\flat)^*$ by induction on the n, number of QI-neighbourhoods of Σ. Let $\leqslant$ be any total order containing the neighbourhood ordering of Σ. **Case $n = 0$:** trivial. **Case $n = k + 1$:**. Consider the least (with respect to $\leqslant$) neighbourhood. Let $\Sigma_1 = (F_1, I_1, \sqsubset_1)$ consist of the highest (w.r.t. $\leqslant$) k-neighbourhoods of Σ. Then by induction, we have a proper deduction $\Sigma_1' = (F_1, I_1', \sqsubset_1, \emptyset)$. Let the lowest neighbourhood match r, and let $\{(a_1, D_1), \ldots, (a_n, D_n)\}$ be the set of DQI-rules that encode it, ordered by $\leq$ (the total order over the vertices V of $G_{\mathcal{R}}$). We know that the sequencing constraints of r are satisfied by Σ_1, so they will be satisfied by Σ_1' too (because $I_1 = ((I_1')^\flat)^*$, hence $\lesssim_{\Sigma_1} = \lesssim_{\Sigma_1'}$). We fragment Σ_1' into $\Pi_0, \ldots, \Pi_n$ such that

$$\Sigma_1' = \bigotimes{}_f \{\Pi_0, \Pi_1, \Pi_2, \ldots \Pi_n\}$$

(using the absorption property of assembly) and for $i = 1, \ldots, n$, Π_i concludes the premises of a_i. Then we can apply each rule (a_i, D_i) to Π_i to yield Π_i'. To glue them back together to get Σ, we must construct another connector, f'. Suppose the old connector $f : H \leftrightarrow H$ is defined with respect to history $h : O(\Sigma_1') \to \mathbb{N}$. We construct H' by

$$\begin{aligned} H' = H \ & \cup \{(C_i, +\alpha) \mid atom(C_i)\ \vec{E}_\alpha\ atom(C_j)\} \\ & \cup \{(C_i, -\alpha) \mid atom(C_j)\ \vec{E}_\alpha\ atom(C_i)\} \end{aligned}$$

(where i, j range over 1 to n). Then we construct $f' : H' \leftrightarrow H'$ and $h' : O(\Sigma') \to \mathbb{N}$ by

$$f'(A, \alpha) = \begin{cases} f(A, \alpha) & \text{if } (A, \alpha) \in H \\ (B, \bar{\alpha}) & \text{if } (A, \alpha) \notin H \text{ and } atom(A)\ E_\alpha\ atom(B) \end{cases}$$

$$h'(A) = \begin{cases} h(A) & \text{if } A \in O(\Sigma_1') \\ 1 + \max(\text{im } h) & \text{ow} \end{cases}$$

It is straightforward to show that $f' : H' \leftrightarrow H'$ and that it is a connector with respect to h'. Then we glue them together to get Σ':

$$\bigotimes{}_{f'} \{\Pi_0, \Pi_1', \Pi_2', \ldots, \Pi_n'\}$$

which satisfies SLF. It remains to show that Σ' is proper. Suppose not. Then there exists a dangling interaction $(C_i, +\alpha)$ (or $(C_i, -\alpha)$, which is a symmetric case) in Σ'. But then $atom(C_i)\ E_\alpha\ atom(C_j)$ which implies that $(C_j, -\alpha)$ exists in Σ'. This is impossible by the definition of H'.

$\Leftarrow$: Let $\Sigma = (F, I, \sqsubset, \emptyset)$ be a proper $\mathcal{T}'$ deduction, and let $\Sigma' = (F, (I^\flat)^*, \sqsubset)$ be a QI-structure. Obviously it will satisfy SLF, since the preorder of the DQI-deduction also takes the reflexive and transitive closure of $I^\flat$. Thus it remains to show that evey QI-neighbourhood of Σ matches a rule. Let $(N, \sqsubset)$ be the

QI-neighbourhood of A in Σ'. It will consist of $n \geq 1$ atoms, $a_1, \ldots, a_n$. Each will be connected by I in Σ', and no other atom will be connected by I. Let $N_1, \ldots, N_n$ be the distinct DQI-neighbourhoods of Σ such that $N_i = (a_i, \sqsubset_i, D_i)$ for some $\sqsubset_i$ and D_i. Since the atoms in each neighbourhoods are connected, it must be that for every dangling interaction $\alpha \in D_i$ there exists $\bar{\alpha} \in D_j$ (for $i \neq j$). But regarding the encoding, this must mean that $a_1, \ldots, a_n$ match the vertices of some connected component of $G_{\mathcal{R}}$ (where $\mathcal{R}$ is the ruleset of $\mathcal{T}$). This means that $(\{a_1, \ldots, a_n\}, \bigcup_i \sqsubset_i)$ must match a rule of $\mathcal{R}$. The result follows by theorem V. □

4.2 Proof Fragmentation

This section is devoted to the theory of fragments of DQI-systems and structures. In a sense it is simply continuing the work of the last theorem: reinterpreting the old theory in terms of the new situation. We obtain the *interaction replacement theorem*, which justifies the technique of breaking links, proving results and then reassembling links afterwards. This was required in the proof of proposition 3.14, and should also be useful in any similar situation. Once again, we have the proof fragmentation theorem, and we prove one half of a simple type-checking example (section 4.2.3).

Definition 4.8 (Fragment)

System fragments *Let $\mathcal{T} = (\mathcal{L}, \mathcal{R})$ and $\mathcal{T}' = (\mathcal{L}', \mathcal{R}')$ be* DQI-*systems. Then $\mathcal{T}$ is a* fragment *of $\mathcal{T}'$ if $\mathcal{L} \subseteq \mathcal{L}'$ and $\mathcal{R} \subseteq \mathcal{R}'$.*

Structure fragments *Let $\Sigma = (F, I, \sqsubset, D)$ and $\Sigma' = (F', I', \sqsubset', D')$ be* DQI-*structures. Then Σ is a fragment of Σ' if $F \subseteq F'$, $I \subseteq I'$ and for all $A \in O(\Sigma)$, $N_\Sigma(A) = N_{\Sigma'}(A)$.*

Thus if Σ is a fragment of Σ', it contains a subset of the trees of Σ', exactly the sequencing constraints of Σ' restricted to Σ, and while it contains no extra interaction links, the interactive ability of each neighbourhood is preserved. So in the following picture Σ' is not a fragment of Σ because D is able to interact in Σ, while it cannot in Σ'.

$$\Sigma = \cfrac{\cfrac{A \quad B}{C} \quad \alpha \qquad D}{E} \text{———} \cfrac{\bar{\alpha}\cfrac{F \quad A}{H}}{I} \qquad\qquad \Sigma' = \cfrac{\cfrac{A \quad B}{C} \quad \alpha \qquad D}{E}$$

Proposition 4.9 *Let Σ be a* DQI-*structure such that Σ_1 and Σ_2 are disjoint fragments of Σ such that $O(\Sigma) = O(\Sigma_1) \cup O(\Sigma_2)$. Then there exists an f such that $\Sigma = \bigotimes_f \{\Sigma_1, \Sigma_2\}$.*

Proof: Since Σ_1 and Σ_2 are fragments, and $O(\Sigma) = O(\Sigma_1) \cup O(\Sigma_2)$, then the forest of Σ is the union of the disjoint forests of Σ_1 and Σ_2. Therefore the results follow from lemma V(i). $\square$

Proposition 4.10 *Let Σ satisfy SLF. Then every fragment of Σ satisfies SLF too.* $\square$

Proposition 4.11 *Every fragment of a* DQI*-deduction is a* DQI*-deduction.*

Proof: Let $\mathcal{T}$ be a DQI-system, let $\Pi \in \mathbf{DQI}(\mathcal{T})$ and let Σ be a fragment of Π. Then by theorem V, Π satisfies SLF and every neighbourhood matches a rule of $\mathcal{T}$. By proposition 4.10, Σ satisfies SLF. Furthermore, every neighbourhood of Σ is a neighbourhood of Π and so it must match a rule of $\mathcal{T}$ also. Therefore, by theorem V, $\Sigma \in \mathbf{DQI}(\mathcal{T})$. $\square$

4.2.1 The interaction reflection theorem

Proposition 3.14 showed that the coding of a P(;) process as a P process did not affect the behaviour of the process. To do this, it showed that whenever a process could be evaluated, its coding could be too with the same behaviour and vice-versa. In current terminology, having the "same behaviour" meant that the two "equivalent" deductions could be assembled in the same way to the same context. Now whereas we could have proved this result specifically for the system $\mathcal{P}_{QF}^{(;)}$, it is more obviously a property which can be proved independently of particular systems: assembly is a concept belonging to the metatheory of deduction.

In this section, we prove the *Interaction reflection theorem*. This says that if we have two structures where the dangling interactive dependencies of one are "contained" within that of the other, then whenever we assemble the larger one, we can reflect the pattern of connections of the larger structure in the smaller one to obtain a parallel assembly.

Simulation Let Σ_1 and Σ_2 be DQI-structures. Then Σ_2 *simulates* Σ_1 (written $\Sigma_1 \preceq \Sigma_2$) if there exists an isomorphism $f : D(\Sigma_1) \leftrightarrow D(\Sigma_2)$ such that for all $(A, \alpha) \in D(\Sigma_1)$, $f(A, \alpha) = (A', \alpha)$ for some A', and for all $(A, \alpha), (B, \beta) \in D(\Sigma_1)$,

$$\text{if } A \lesssim_{\Sigma_1} B \quad \text{then} \quad A' \lesssim_{\Sigma_2} B'$$

where $f(A, \alpha) = (A', \alpha)$ and $f(B, \beta) = (B', \beta)$. We write $\Sigma_1 \simeq \Sigma_2$ if $\Sigma_1 \preceq \Sigma_2$ and $\Sigma_2 \preceq \Sigma_1$.

Theorem VII (Interaction Reflection) *Let Σ_1' be an assembly of Σ_1, and let Σ_2 be such that $\Sigma_2 \preceq \Sigma_1$. Then there exists an assembly Σ_2' of Σ_2 such that $\Sigma_2' \preceq \Sigma_1'$.*

Proof: Suppose $\Sigma_1' = \bigotimes_{f_1} \Sigma_1$, and that f is a connector with respect to history h_1. let $g : D(\Sigma_2) \leftrightarrow D(\Sigma_1)$ be the function that witnesses $\Sigma_2 \preceq \Sigma_1$. We show that $f_2 = g^{-1} \circ f_1 \circ g$ is a connector of Σ_2. First, let $(A, \alpha) \in \text{dom } f'$. Then $f_2(A, \alpha) = (g^{-1} \circ f_1)(A', \alpha) = g^{-1}(B', \bar{\alpha}) = (B, \bar{\alpha})$ for some $A', B' \in O(\Sigma_1)$ and $B' \in O(\Sigma_2)$. Second, f_2 must be involutive because f_1 is. Third, let h_2 be any history of Σ_2. Then we construct histories h_1' and h_2' of Σ_1 and Σ_2 such that $h_1'(A) \leq h_1'(B)$ if and only if $h_1(A) \leq h_1(B)$ and such that for all $(A, \alpha) \in D(\Sigma_1)$, if $g(A, \alpha) = (A', \alpha)$, $h_1'(A) = h_2'(A')$. The method of construction is iterative. First, we totally order the dangling interactions with respect to the preorder of Σ_1. Then for the ith dangling interaction $g(A_i, \alpha_i) = (A_i', \alpha_i)$, if $h_{1i}(A_i) < h_{2i}(A_i')$ then add the difference to every timestamp in h_{1i} greater than or equal to that of A_i. Then h_{1i+1} equals the altered h_{1i}, and $h_{2i+1} = h_{2i}$. If $h_{1i}(A_i) > h_{2i}(A_i')$ then we alter h_{2i} in the same way. h_{1i+1} and h_{2i} are calculated in the symmetric way to the previous case. h_1' and h_2' are h_{1n} and h_{2n} for the final n.

Now it is easy to show that for all $(A, \alpha), (A', \bar{\alpha}) \in \text{dom } f_2$ $h_2'(A) = h_2'(A')$: $h_2'(A) = h_1'(A) = h_1'(A') = h_2'(A')$.

Therefore we get $\Sigma_2' = \bigotimes_{f_2} \Sigma_2$. It is straightforward to show that $\Sigma_2' \preceq \Sigma_1'$ because we have not altered the relative dependencies of the remaining dangling interactions: we can show simulation using $g \restriction D(\Sigma_2')$. □

Application to the translation correctness proof

We complete the proof of proposition 3.14. That proof was an indication of the way one would like to reason about deductions, and so left out the messy detail of preserving interactions, which (as we have seen) could be done generally. The following corollary simply interprets the interaction reflection theorem in terms of the terminology of section 3.4.4.

Corollary VIIa *Let Σ_1 and Σ_2 be* QI*-structures whose formulae are those of $\mathcal{P}_{QI}^{(;)}$ such that $\Sigma_1 \simeq \Sigma_2$. If Σ_1 is the result of breaking interaction links of Σ_1' then there exists a Σ_2' such that $\Sigma_1' \simeq \Sigma_2'$ and Σ_2 is the result of breaking interaction links of Σ_2'.*

Proof: We define an isomporhism between QI-structures and DQI-structures if they have the same preorder. We define an isomorphism between the visible occurrences of a QI-structure and the dangling interactions of the isomorphic DQI-structure, and then the corollary follows naturally. □

4.2.2 The Proof Fragmentation Theorem

Once again, we define $\mathbb{F}_{\mathcal{T}}(X)$ for the set of all fragments of DQI-fragments of DQI-deductions in X that are built from rules only in $\mathcal{T}$.

Theorem VIII (Proof Fragmentation) *Let $\mathcal{T}'$ be a system fragment of $\mathcal{T}$. Then* $\mathbf{DQI}(\mathcal{T}') \supseteq \mathbb{F}_{\mathcal{T}'}(\mathbf{QI}(\mathcal{T}))$

Proof: Let $\Pi \in \mathbb{F}_{\mathcal{T}'}(\mathbf{QI}(\mathcal{T}))$. Then by theorem V, Π satisfies SLF and by definition every neighbourhood matches a rule of $\mathcal{T}'$. Hence by theorem V again, $\Pi \in \mathbf{DQI}(\mathcal{T}')$. $\square$

4.2.3 An example: Type checked processes do not fail

To illustrate proof fragmentation and motivate the proof assembly technique, we consider a simple type-checking example. We introduce the notions of *value* and *type* to P (calling the language P(V)). Processes in this language may fail to terminate successfully in one of two ways: they may deadlock, or attempt to evaluate an expression with wrongly typed arguments. This latter event I call a *run-time error*. If a process exhibits a run-time error, I say it *fails*.

In the following subsections I introduce the language, its dynamic semantics and its static semantics. The static semantics simply checks types — the intention being that a process type-checks if and only if it exhibits no run-time errors. Since no P process can execute indefinitely, we could also test for deadlock-freeness — perhaps using a type system like that of [MG95] where *types are behaviours* (see also [GS86]). However, testing for deadlock is in general undecidable (seen via a reduction to the halting problem: consider a program which sequences a trivially deadlocking process after a possibly nonterminating process) though perhaps one may be able to decide deadlock-freeness for restricted, but nonetheless interesting, classes of processes.

We shall be able to prove that no type-checked process fails using proof fragmentation. However, we shall not be able to prove that every process which does not fail type checks in this way. In this case we really have to take account of what interactions are about. This motivates the technique of proof assembly described in section 4.3.

The Language P(V)

Consider the langage P(V), a typed value-passing variant of P. It consists of the following syntax:

$$\begin{aligned} p &::= 0 \mid a!e.p \mid a?x.p \mid p|p \\ e &::= v \mid x \mid e \; op \; e \end{aligned}$$

where x ranges over the countable set Var, v over the set Val of integers and booleans, op over a set of binary integer and boolean operations OP, and a ranges over the action set $\mathcal{A}$. The grammar of expressions is a straightforward extension of the expressions of P(:=) (page 25) to include boolean expressions. (We need at least two types to make type-checking non-trivial.) The grammar defines the set Exp, ranged over by e.

Dynamic semantics

The dynamic semantics of P(V) is given by the DQI-system $\mathcal{P}^{(v)}_{QI} = (\mathcal{L}^{(v)}_{QI}, \mathcal{R}^{(v)}_{QI})$ where $\mathcal{L}^{(v)}_{QI}$ is the set of judgments of form

$$E \Rrightarrow p\surd \quad \text{and} \quad E \Rrightarrow e \rightsquigarrow v \quad \text{and} \quad comm(a, v)$$

Where E ranges over the set of environments $E : Var \rightharpoonup Val$, $p \in$P(V), $e \in Exp$, and $v \in Val$. It is customary to use turnstile notation, $E \vdash p\surd$, for environment judgments: E is seen as a collection of hypotheses about variables. However, I already use $\vdash$ in the metatheory, so I use $\Rrightarrow$ instead, because of its connotations of implication. The ruleset $\mathcal{R}^{(v)}_{QI}$ is given by the following rules:

$$\frac{}{E \Rrightarrow 0\surd} \qquad \frac{E \Rrightarrow p_1\surd \qquad E \Rrightarrow p_2\surd}{E \Rrightarrow p_1|p_2\surd} \qquad \frac{E \Rrightarrow e \rightsquigarrow v \quad \blacktriangleright \quad E \Rrightarrow a!v.p\surd}{E \Rrightarrow a!e.p\surd}$$

$$\frac{E \Rrightarrow p_1\surd \qquad typof(a) = typof(v)}{E \Rrightarrow a!v.p_1\surd} \quad \xrightarrow{comm(a,v)} \quad \frac{E'[v/x] \Rrightarrow p_2\surd}{E' \Rrightarrow a?x.p_2\surd}$$

$$\frac{}{E \Rrightarrow v \rightsquigarrow v} \qquad \frac{}{E \Rrightarrow x \rightsquigarrow E(x)} \qquad \frac{E \Rrightarrow e_1 \rightsquigarrow v_1 \qquad E \Rrightarrow e_2 \rightsquigarrow v_2}{E \Rrightarrow e_1 \; op \; e_2 \rightsquigarrow app(op, v_1, v_2)}$$

The partial function $app : OP \times Val \times Val \rightharpoonup Val$ applies operations to their arguments. It is not defined when its arguments are wrongly typed.

Static semantics

Let $BT = \{\text{int}, \text{bool}\}$ be the set of base types, and $Types$ (ranged over by τ) be the least set containing BT and $\tau_1 \times \tau_2$ when $\tau_1, \tau_2 \in Types$. For base types and actions, let $typof : Val \cup OP \cup \mathcal{A} \rightarrow Types$ assign arbitrary types to actions and for the base types be such that $typof(b) = \text{bool}$ for all booleans b; $typof(n) = \text{int}$ for all integers n; and for operations op, $typof(op) = \tau_1 \times \tau_2 \times \tau_3$ for some τ_1, τ_2 and τ_3 if and only if for every v_1, v_2 and v_3, if $app(op, v_1, v_2) = v_3$ then $typof(v_i) = \tau_i$ for $i = 1, 2, 3$. This fact is used in the proof of lemma 4.12(i). A *typing environment* is a function $T : Var \rightarrow Types$. Then I say a P(V) process p (expression e) is *type-checked* by type environment T if $T \Rrightarrow p$ (for expressions, $T \Rrightarrow e : \tau$ for some $\tau \in Types$) is deducible in the following system $\mathcal{TP}$:

$$\frac{}{T \Rrightarrow 0} \qquad \frac{T \Rrightarrow p_1 \qquad T \Rrightarrow p_2}{T \Rrightarrow p_1|p_2} \qquad \frac{T \Rrightarrow e : typof(a) \qquad T \Rrightarrow p}{T \Rrightarrow a!e.p}$$

$$\frac{T[typof(a)/x] \Rrightarrow p}{T \Rrightarrow a?x.p} \qquad \frac{}{T \Rrightarrow v : typof(v)} \qquad \frac{}{T \Rrightarrow x : T(x)}$$

$$\frac{T \Rrightarrow e_1 : \tau_1 \qquad T \Rrightarrow e_2 : \tau_2 \qquad typof(op) = \tau_1 \times \tau_2 \times \tau_3}{T \Rrightarrow e_1 \; op \; e_2 : \tau_3}$$

We extend the typing judgment to environments, writing $E : T$ when for all $x \in \text{dom } E$, $T(x) = typof(E(x))$.

Type-checked processes do not fail

This condition can be stated formally: if $\mathcal{TP} \vdash T \Rrightarrow p$ then $\mathcal{P}^{(v)}_{\mathrm{QI}} \vdash E \Rrightarrow p\surd$ when $E : T$. Note that I do not ask that the evaluation judgment be properly deduced: that would amount to asserting deadlock-freeness of type-checked processes.

This is a candidate for the proof fragmentation theorem: we can simply break the communication interaction link, and prove the property by induction on the size of $\mathcal{P}^{(v)}_{QI}$-deductions. First, however, we need a lemma attesting that type-checked expressions do not fail. The expression evaluation fragment does not involve interaction, so we can prove both directions of this lemma in the usual manner.

Lemma 4.12(i) *For all $e \in Exp$, $v \in Val$, and environments E and T such that $E : T$, we have $\mathcal{P}^{(v)}_{QI} \vdash E \Rrightarrow e \rightsquigarrow v$ if and only if $\mathcal{TP} \vdash T \Rrightarrow e : \mathrm{typof}(v)$.* □

This lemma is the heart of the type checking proof: run-time errors occur when an operation is applied to wrongly-typed arguments. Nevertheless I do not give its proof; first because it is so simple the result should not be in doubt, but mainly because we are more concerned with the way one proves results about interacting deductions.

Proposition 4.12 *Let $E : T$. Then if $\mathcal{TP} \vdash T \Rrightarrow p$ then $\mathcal{P}^{(v)}_{\mathrm{QI}} \vdash E \Rrightarrow p\surd$*

Proof: by induction on the depth of inference of $T \Rrightarrow p$. **Case $\boldsymbol{T \Rrightarrow 0}$:** trivial. **Case $\boldsymbol{T \Rrightarrow p|q}$:** straightforward induction. **Case $\boldsymbol{T \Rrightarrow a!e.p}$:** this follows when $T \Rrightarrow e : \mathit{typof}(a)$ and $T \Rrightarrow p$. By lemma 4.12(i), we know that if $E \Rrightarrow e \rightsquigarrow v$ then $\mathit{typof}(v) = \mathit{typof}(a)$. By induction, $E \Rrightarrow p\surd$. Therefore, $E \Rrightarrow a!v.p\surd$ Since $E \Rrightarrow e \rightsquigarrow v$ we can also infer $E \Rrightarrow a!e.p\surd$.

Case $\boldsymbol{T \Rrightarrow a?x.p}$: this follows when $T[\mathit{typof}(a)/x] \Rrightarrow p$. By induction, we know for all E' such that $E' : T[\mathit{typof}(a)/x]$ that $E' \Rrightarrow p\surd$. In particular, this will be true in those cases when $E' = E[v/x]$ and $\mathit{typof}(v) = \mathit{typof}(a)$. Since this binding of v to x does not appear in E, we can conclude that $E \Rrightarrow a?x.p\surd$ is deducible. □

4.2.4 Proof fragmentation is not always enough

So we have shown that type-checked processes do not fail. However, the proof of the result that terminating processes type-check fails at the case for input:

> **case input:** Suppose $E \Rrightarrow a?x.p$ and $E : T$. (We are required to prove that $T \Rrightarrow a?x.p$ is deducible by $\mathcal{TP}$.) Then $E[v/x] \Rrightarrow p\surd$ is deduced by a smaller tree. By induction we know that for all T' such that $E[v/x] : T'$, we can deduce $T' \Rrightarrow p$. In particular, this is true when $T' = T[\mathit{typof}(v)/x]$ ...

The problem here is that we cannot guarantee that $\mathit{typof}(v) = \mathit{typof}(a)$, which would allow us to infer that $T' = T[\mathit{typof}(a)/x]$, from which we could deduce $T \Rrightarrow a?x.p$. Yet the other half of the communication rule guarantees $\mathit{typof}(a) =$

typof(v): therefore the case for the input rule must follow *when the rule fragment is known to be part of a* $\mathcal{P}^{(v)}_{QI}$*-deduction.*

4.3 Proof Assembly

We could not prove that terminating processes type-check because the case for input required more information than was contained in the premises to the input rule atom:

$$\frac{E'[v/x] \Rrightarrow p_2\surd}{E' \Rrightarrow a?x.p_2\surd}\ -comm(a, v)$$

Specifically, we wanted to know that *typof*(a) = *typof*(v). The object of study of this chapter is the content of interaction. In the type-checking proof, we know that *typof*(a) = *typof*(v) holds in proper deductions. Therefore, whatever the content of the interaction labelled $comm(a, v)$ is, it must imply that *typof*(a) = *typof*(v).

At this point the idea that dangling interactions are hypotheses becomes useful. We simply view $-comm(a, v)$ as a hypothesis about the environment that the deduction occurs within. Then, in the proof, whenever we see a $-comm(a, v)$ dangling interaction, we simply hypothesize that $comm(a, v)$ implies *typof*(a) = *typof*(v). We have already noted that the case for the output rule will guarantee this. So whenever we see $+comm(a, v)$, we have to view this as an obligation to prove that $comm(a, v)$ implies *typof*(a) = *typof*(v).

Assembly is Discharge

Thus labels are interpreted as propositions and dangling interactions mark interaction hypotheses and interaction guarantees. The slogan is *Assembly is Discharge*: assembling a hypothesis with its guarantee will discharge the hypothesis.

The idea is borrowed from the world of parallel program verification [Jon83]. There the goal is to prove results about processes compositionally. This is achieved by using sets of *rely* and *guarantee* conditions (elsewhere called *assumption* and *commitment* conditions [ZdBdR83]) which encapsulate the relevant information about the interactive behaviour of a process and its environment.

The proof fragmentation theorem enables one to reason about deduction fragments using system fragments. The *proof assembly theorem* enables one to compose proofs of fragments "in parallel" to obtain a result about deductions.

The proof assembly theorem

In this section, we assume that formulae of a language have tree structure, and we write them $P(M_1, \ldots, M_n)$ where P is the principal syntactic constructor of the formula and $M_1, \ldots, M_n$ are the subformulas or terms that occur as children of the formula. So, for example, the evaluation judgment of P(V) would be written $(\cdot \Rrightarrow \cdot\surd)(E, p)$. We say a judgment is *atomic* if its arguments are all terms.

Operational semantics judgments and interaction labels by convention, are atomic. Therefore, we shall restrict our discussion to atomic judgments: this makes the treatment very easy.

Of course, this restricts the following notion of proof assembly to those semantics which use a standard tree notion of syntax for judgments and programs. However, since the aim of the restriction is to allow a straightforward notion of formula interpretation, I believe that the technique could be extended to other notions of syntax (e.g., higher-order abstract syntax [PE88,Han93]) straightforwardly, given a suitable notion of interpretation.

Interpretation Let $\mathcal{T} = (\mathcal{L}, \mathcal{R})$ be a DQI-system, and let J_n be the set of all judgment constructors P that have n arguments in $\mathcal{L}$. Then an *interpretation* of $\mathcal{T}$ is a family of functions $\mathcal{I}_n : J_n \to \wp(\mathcal{L}^n)$. Thus an interpretation maps each judgment constructor J to a set of n-tuples of terms and formulae. That is, the interpretation returns the extension of the property the formula represents. In the example above, the interpretation of *comm* would be the set of pairs (a, v) such that $\mathit{typof}(a) = \mathit{typof}(v)$.

If $\mathcal{I}$ is an interpretation, and Σ is a DQI-structure, then Σ is $\mathcal{I}$*-consistent* with respect to a judgement P if for all $(A, \pm P(t_1, \ldots, t_n)) \in D(\Sigma)$, $(t_1, \ldots, t_n) \in \mathcal{I}_n(P)$.

On Hypotheses and Guarantees Let X be a set of DQI-structures, and $\mathcal{I}$ an interpretation. Then a judgment constructor P is *guaranteed* in X if for all $\Sigma \in X$, Σ is $\mathcal{I}$-consistent with respect to P. Dually, P is *hypothesized* in X if there exists a $\Sigma \in X$ such that Σ is $\mathcal{I}$-consistent with respect to P. For a non-empty set X, if P is guaranteed in X it is also hypothesized. Thus in a given proof (which will determine the interpretation) about $\mathcal{T}$ we can determine which labels are hypotheses and which are guarantees. If the inverse of every hypothesis is a guarantee then we shall be able to use the technique of proof assembly.

Relative guarantees We want to be able to say that something is guaranteed when certain hypotheses hold. In notation reminiscent of [Sti88,MC81], I write $X \models_{\mathcal{I}} [H]\ \phi\ [G]$ where ϕ is some formula, and H and G are sets of formula perceptions if for all $\Sigma \in X$, if Σ is $\mathcal{I}$-consistent with respect to every constructor in H, then it is also $\mathcal{I}$-consistent with respect to the constructors in G. Thus H can be seen as a set of hypotheses, and G a set of guarantees. Obviously, $X \models_{\mathcal{I}} [H]\ \phi\ [G]$ implies $X \models_{\mathcal{I}} [H']\ \phi\ [G']$ when $H \subseteq H'$ and $G' \subseteq G$. $X \models_{\mathcal{I}} \phi$ abbreviates $X \models_{\mathcal{I}} [\emptyset]\ \phi\ [\emptyset]$.

Parallel proofs Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be disjoint fragments of $\mathcal{T}$, and suppose ϕ_1 is a property about $\mathcal{T}_1$-deductions, and ϕ_2 a property about $\mathcal{T}_2$-deductions. Then I write $X \models_{\mathcal{I}} [H]\ \phi_1 \wedge \phi_2\ [G]$ if under the hypotheses H, G is guaranteed and ϕ_1 is true about every $\mathcal{T}_1$-fragment of structures in X, and ϕ_2 is true about every $\mathcal{T}_2$ fragment.

Theorem IX (Proof Assembly) *Let $\mathcal{T}$ be an acyclic* DQI*-system and $\mathcal{I}$ be an interpretation of $\mathcal{T}$. Let $\mathcal{T}_1, \ldots, \mathcal{T}_n$ (for $n \geq 1$) be disjoint fragments of $\mathcal{T}$, such that $\mathcal{T}_1 \cup \ldots \cup \mathcal{T}_n = \mathcal{T}$ and $\mathbf{DQI}(\mathcal{T}_i) \models_{\mathcal{I}} [H_i]\ \phi_i\ [G_i]$. Then if $\bigcup_{i=1}^{n} H_i \subseteq \bigcup_{i=1}^{n} \overline{G_i}$ then $\mathbf{QI}(\mathcal{T}) \models_{\mathcal{I}} \phi_1 \wedge \ldots \wedge \phi_n$*

Proof: Let $\Sigma \in \mathbf{QI}(\mathcal{T})$ and let $\Sigma_1, \ldots, \Sigma_n$ be fragments of Σ such that Σ_i is the $\mathcal{T}_i$ fragment of Σ, and such that Σ is the assembly of them all. We know that if Σ is $\mathcal{I}$-consistent with respect to H_i then both Σ_i satisfies ϕ_i and Σ_i is $\mathcal{I}$-consistent with respect to G_i. Thus we have to show that Σ_i is $\mathcal{I}$-consistent with respect to the labels in H_i.

Suppose not. Then there exists $P \in H_i$ and $(A, \pm P(M_1, \ldots, M_n)) \in D(\Sigma_i)$ such that $(M_1, \ldots, M_n) \notin \mathcal{I}_n(P)$. Since Σ is proper there must exist a Σ_j $(i \neq j)$ with a dangling interaction $(B, \mp P(M_1, \ldots, M_n))$ that is assembled with $(A, P(M_1, \ldots, M_n))$. Now, since $\bigcup_{i=1}^{n} H_i \subseteq \bigcup_{i=1}^{n} \overline{G_i}$, $\mp P \in G_j$. Since $(M_1, \ldots, M_n) \notin \mathcal{I}_n(P)$, then there exists in turn a $P' \in H_j$ such that Σ_j is not $\mathcal{I}$-consistent with respect to P'. By similar reasoning, we find an infinite chain of dangling interactions in $\Sigma_1 \cup \ldots \cup \Sigma_n$ such that the dangling interaction is not consistent with its label's interpretation. Since $\mathcal{T}$ is acyclic, this must mean that there exists an infinite number of dangling interactions in $\Sigma_1 \cup \ldots \cup \Sigma_n$, and hence an infinite number of interactions in Σ. But this is impossible because deductions can contain only a finite number of interactions. $\square$

The next section shows how proof assembly can be used to prove that terminating processes type-check. However, this is a result stated and proved using evaluation semantics. We cannot use the proof assembly theorem so easily for a result about a transition semantics, which will concern sequences of deductions (section 4.3.2). Proof fragmentation can be used either for transition or evaluation semantics, so it is only natural to expect a form of proof assembly for sequences of deductions. The following corollary is such a form.

Proof assembly for sequences of deductions

Given an interpretation $\mathcal{I}$ of $\mathcal{T}$, we say that a sequence of $\mathcal{T}$-deductions is $\mathcal{I}$-*consistent* if all of its elements are. Then, if X is a set of sequences of deductions, we write $X \models_{\mathcal{I}} [H]\ \phi\ [G]$ if for all $s \in X$, if s is $\mathcal{I}$-consistent with respect to H then it is also $\mathcal{I}$-consistent with respect to G, and ϕ is true about s.

Let $\mathcal{T}'$ be a fragment of $\mathcal{T}$, and let s be a $\mathcal{T}$-deduction sequence. Then I say that s' is a $\mathcal{T}'$-fragment of s if $|s'| = |s|$ and for all $i \in \text{dom } s$, $s'(i)$ is the $\mathcal{T}'$-fragment of $s(i)$.

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be disjoint fragments of $\mathcal{T}$, and ϕ_1 is a property about $\mathcal{T}_1$-deduction sequences, and ϕ_2 a property about $\mathcal{T}_2$-deduction sequences. Then I write $X \models_{\mathcal{I}} [H]\ \phi_1 \wedge \phi_2\ [G]$ if under the hypotheses H, G is guaranteed and ϕ_1 is true about every $\mathcal{T}_1$-fragment of sequences in X, and ϕ_2 is true about every $\mathcal{T}_2$ fragment.

Corollary IXa (Proof assembly for sequences) *Let $\mathcal{T}$ be an acyclic* DQI-*system and $\mathcal{I}$ be an interpretation of $\mathcal{T}$. Let $\mathcal{T}_1, \ldots, \mathcal{T}_n$ (for $n \geq 1$) be disjoint fragments of $\mathcal{T}$, such that $\mathcal{T}_1 \cup \ldots \cup \mathcal{T}_n = \mathcal{T}$ and $\mathbf{DQI}(\mathcal{T}_i)^\omega \models_{\mathcal{I}} [H_i]\ \phi_i\ [G_i]$. Then if $\bigcup_{i=1}^n H_i \subseteq \bigcup_{i=1}^n \overline{G_i}$ then $\mathbf{QI}^\omega(\mathcal{T}) \models_{\mathcal{I}} \phi_1 \wedge \ldots \wedge \phi_n$*

Proof: Let $s \in \mathbf{QI}^\omega(\mathcal{T})$. We show by transfinite induction on n that every $\mathcal{T}_i$-fragment of a prefix of s of length n is $\mathcal{I}$-consistent with respect to H_i. **Case $\boldsymbol{n = 0}$:** vacuous. **Case $\boldsymbol{n = k + 1}$:** By induction, the result holds for the deductions $s(1)$ to $s(k)$. It is trivial to show that under the hypothesis H_i that $s_i(k+1)$ is $\mathcal{I}$-consistent with respect to H_i, guaranteeing G_i. By proof assembly, every $\mathcal{T}_i$-fragment of $s(k+1)$ is $\mathcal{I}$-consistent with respect to H_i. **Case $\boldsymbol{n = \omega}$:** if not, there must exist a finite k for which the k-prefix fails the property. This is contradicted by induction.

Therefore, if we have an $s \in \mathbf{QI}^\omega(\mathcal{T})$ failing to satisfy $\phi_1 \wedge \ldots \wedge \phi_n$ then there must be some $\mathcal{T}_i$ such that the $\mathcal{T}_i$-fragment of s fails to satisfy ϕ_i. But this can only happen when s is not $\mathcal{I}$-consistent with respect to H_i, which we have just seen is impossible. □

4.3.1 Example: Terminating processes type-check

To illustrate this theorem, we prove the other half of the type-checking theorem, that type-checking is sufficient. Here the input rule hypothesizes that the content of the interaction $comm(a, v)$ implies that $typof(a) = typof(v)$, and that the output rule guarantees this consequence.

Proposition 4.13 *For all $p \in \mathrm{P}(\mathrm{V})$, and environments E, T such that $E : T$, we have that $\mathcal{P}_{QI}^{(v)} \vdash E \Rrightarrow p\surd$ if and only if $\mathcal{TP} \vdash T \Rrightarrow p$.*

Proof: We prove that if $E : T$ then $\mathcal{P}_{QI0}^{(v)} \vdash E \Rrightarrow p\surd$ iff $\mathcal{TP} \vdash T \Rrightarrow p$, under the interaction hypothesis that $-comm(a, v)$ implies $typof(a) = typof(v)$, and guaranteeing that $+comm(a, v)$ implies $typof(a) = typof(v)$. Then the result will follow by the proof assembly theorem.

by induction on the depth of inference of $E \Rrightarrow p\surd$. **Case $\boldsymbol{E \Rrightarrow 0\surd}$:** trivial. **Case $\boldsymbol{E \Rrightarrow p|q\surd}$:** straightforward induction. **Case $\boldsymbol{E \Rrightarrow a!v.p\surd}$:** this follows when $E \Rrightarrow p\surd$ and $typof(a) = typof(v)$ (which guarantees the interaction hypothesis). We know that $T \Rrightarrow v : typof(v)$, so therefore $T \Rrightarrow v : typof(a)$. By induction we know $T \Rrightarrow p$. Therefore, $T \Rrightarrow a!v.p$.

Case $\boldsymbol{E \Rrightarrow a!e.p\surd}$: this follows from $E \Rrightarrow e \rightsquigarrow v$ and $E \Rrightarrow a!v.p\surd$. By lemma 4.12(i) we know that $T \Rrightarrow e : typof(v)$. By induction we know that $T \Rrightarrow a!v.p$, which means $typof(a) = typof(v)$. Therefore $T \Rrightarrow e : typof(a)$, and so $T \Rrightarrow a!e.p$.

Case $\boldsymbol{E \Rrightarrow a?x.p\surd}$: this follows from $E[v/x] \Rrightarrow p\surd$. By induction, we know for all T' such that $E[v/x] : T'$ that $T' \Rrightarrow p$. In particular, this is true when $T' = T[typof(v)/x]$. By the interaction hypothesis, $T[typof(v)/x] = T[typof(a)/x]$. Therefore $T \Rrightarrow a?x.p$. □

A similar result would be to extend P(:=) with types, and show that only correctly typed processes could evaluate. There the problem is to ensure that the store maintains correct typing. Thus we get two parallel subproofs: one proof for the P(:=) programs, and the other for stores. They depend on each other: the program proof relies on the store proof to maintain correct typing, and guarantees that everything it sends the store is correctly typed. The store proof relies on the program proof to send it correct information, and guarantees that it outputs correct information.

Thus we obtain two kinds of proof dependency: the usual "functional" dependency of the proof for processes on the proof for expressions, and the "interactive" dependency between the proof for stores and the proofs for expressions and processes. The next section illustrates an interactive dependency between subproofs of a nondivergence result.

4.3.2 Example: Another nondivergence proof

In this section, we apply the technique of proof assembly to show that our process calculus P extended with the iterators of section 2.5.1 preserves the nondivergence property of P. In order to use the proof assembly theorem we need some kind of interaction between two different fragments. Therefore we shall treat the iterator constants as process variables which have to be looked up in the store. Thus to prove the nondivergence result, we shall assemble a proof concerning process transitions and a proof concerning the contents of the store.

Let P(IT) be the language with the grammar

$$p ::= 0 \mid l.p \mid p|p \mid It^n C.p \mid C$$

where $n \in \mathbb{N}$ and C is a member of $Const$, a countable set of process constants.

The dynamic semantics is given by the transition system $TS_I = \langle \mathrm{P(IT)}, \rightarrow, \Omega \rangle$ where Ω is the same set of final states as for P and $\rightarrow \subseteq \mathrm{P(IT)} \times \mathrm{P(IT)}$ is the relation defined by $(p,q) \in \rightarrow$ if and only if $\mathcal{P}_I^{(It)} \vdash p \rightarrow q$ where $\mathcal{P}_I^{(It)}$ is the DQI-system $(\mathcal{L}_I^{(It)}, \mathcal{R}_I^{(It)})$ such that $\mathcal{L}_I^{(It)}$ consists of the following judgments

$$p \rightarrow q \qquad \sigma \rightsquigarrow \sigma' \qquad comm(a) \qquad write(C,n,p) \qquad read(C,p)$$

where $\sigma, \sigma' \in ItStore$ which is the set of stores that bind processes to process constants, and the ruleset $\mathcal{R}_I^{(It)}$ consists of the following rules:

$$\frac{}{a.p \to p} \;\frac{comm(a)}{}\; \frac{}{\bar{a}.q \to q}$$

$$\frac{p \to p'}{p|q \to p'|q} \qquad \frac{q \to q'}{p|q \to p|q'} \qquad \frac{p \to p' \qquad q \to q'}{p|q \to p'|q'}$$

$$\frac{}{It^0 C.p \to 0} \qquad \frac{}{It^{n+1}C. \to p[C_n/C]} \;\frac{write(C,n,p)}{}\; \frac{}{\sigma \rightsquigarrow \sigma[It^n C.p/C_n]}$$

$$\frac{}{C \to p} \;\frac{read(C,p)}{}\; \frac{\sigma(C) = p}{\sigma \rightsquigarrow \sigma}$$

Note that this semantics is quite reckless: it takes no account of scoping. There may be two iterators of C, and if they are executed in parallel then one iterator will overwrite the other. We could simply state a static condition to exclude such processes. Then a *nicely bound* process would be such that no constant was bound more than once, and every constant occurs bound. However, while nice binding makes sense, it does not affect the nondivergence proof.

Thus this system is just the system $\mathcal{P}_I$ extended with labels and with rules for iterators and constants. To aid induction, we define the size of p inductively as follows:

$$|p| = \begin{cases} 1 & \text{if } p = 0 \text{ or } p = C_n \\ 1 + |q| & \text{if } p = l.q \\ 1 + \max(|q_1|, |q_2|) & \text{if } p = q_1|q_2 \\ 1 + n|q| & \text{if } p = It^n C.q \end{cases}$$

Let me say that a process is *finite* if it can be attributed no infinite deduction sequence. Then the nondivergence proof for processes becomes:

Lemma 4.14(i) *For all processes p, under the hypothesis that $read(C,q)$ implies that q is finite, then p is finite, and $write(C,n,p)$ is guaranteed to imply that $It^n C.p$ is finite.*

Proof: By strong induction on $n = |p|$. **Case $n = 1$:** then there are three subcases, when $p = 0$, when $p = C$ and when $p = It^0 C.q$. The first follows because 0 is attributed no transition sequences; the second from the interaction hypothesis; the third is trivial. **Case $n = k + 1$:** There are three sub cases. **First, $p = l.q$:** trivial. **Second, $p = q_1|q_2$:** follows from similar reasoning to that in proposition 2.8. **Third, $p = It^{m+1}C.q$:** This is the interesting case. Since $|It^m C.q| < |It^{m+1}C.q|$, we know by strong induction that $It^m C.q$ is finite. Thus we have guaranteed the condition on $write(C, m, q)$. Since also $|It^{m+1}C.q| > |q[C_m/C]|$, we know by induction that q is finite, and so therefore neither is p. □

I say a deduction sequence attributed to the empty store `empty` *binds finitely* if for every prefix of the deduction sequence ending in σ, every constant bound in σ is bound to a finite process.

Lemma 4.14(ii) *Under the interaction hypothesis that write(C, n, p) implies that $It^n C.p$ is finite, then every deduction sequence attributed to the empty store binds finitely, and read(C, p) is guaranteed to imply that p is finite.*

Proof: by transfinite induction on the length n of prefixes of deduction sequences attributed to the empty store. **Case $\boldsymbol{n = 0}$:** vacuous. **Case $\boldsymbol{n = k+1}$:** suppose not. Let s be the sequence that does not bind finitely. By induction, we know that the k-length prefix of s binds finitely, so it must be the last store which violates the property. Consider then the last transition: $\sigma \rightsquigarrow \sigma'$. There are two cases. Either it is the result of a read transition, in which case $\sigma' = \sigma$ (and therefore the read condition is guaranteed), or it is the result of a write transition, in which case $\sigma' = \sigma[It^n C.p/C_n]$. In the first case, σ' cannot violate the property because σ does not. In the second, we know by the interaction hypothesis that C_n is bound to a finite process, and we know σ satisfies the invariant. Contradiction.

Case $\boldsymbol{n = \omega}$: suppose not. Then there must exist a finite first point when the invariant was violated. By induction this is impossible. □

Proposition 4.14 *No process in* P(IT) *diverges.*

Proof: This follows by the assembly of lemmas 4.14(i) and 4.14(ii) (and of course also by the corollary to the proof assembly theorem). □

4.4 Chapter summary

We extended the treatment of interacting deduction to include an analysis of the content of interaction. Intuitively, this is that which is discussed when two ideas are shared. The result was a more elegant and usable definition, constructed from the two notions of rule application (inference) and interaction assembly (assumption discharge). We obtained the proof fragmentation theorem again, which allows us to ignore contexts of trees, and also the dual proof assembly theorem which allows us to build proofs from proofs concerning individual trees. We saw a simple evaluation-semantics type-checking example, and also a transition-semantics nondivergence proof.

Chapter 5

A Semantics and Logic for CSP

To date my examples have all been extensions of a trivial process calculus P. However, this is not a convincing programming language. COMMUNICATING SEQUENTIAL PROCESSES (CSP) [Hoa78] is the canonical imperative parallel programming language. CSP is widely used in the analysis of concurrent algorithms, and is also the core of the language OCCAM [Wex89].

In this chapter I give an evaluation semantics for CSP and consider various extensions. There are many different dialects of CSP: I base this account on Plotkin's [Plo83], explaining differences as I proceed. The major difference is the syntactic simplicity of the judgments.

I also give a compositional Hoare Logic for the partial correctness of CSP and prove it sound with respect to my semantics. This Hoare Logic uses the method of invariants to establish communicative relationships between processes, but unlike other compositional systems, it does not make use of traces.

Section 5.1 describes the syntax and semantics of our version of CSP, which allows top-level parallel composition only. Section 5.2 describes six different alternative features: nested parallelism, communication via shared variables, dynamic process creation, pre-emption, multicasting and recursive procedures. Section 5.3 presents a Hoare Logic for the partial correctness of CSP, and proves it sound.

5.1 A Definition of CSP

5.1.1 Syntax and informal semantics

The abstract syntax of CSP is given by the following EBNF grammar.

programs:

$$p ::= \; R :: c \mid p \parallel p$$

commands:

$$c ::= \; x := e \mid \texttt{skip} \mid \texttt{abort} \mid c; c \mid \texttt{if } g \texttt{ fi} \mid \texttt{do } g \texttt{ od} \mid R?x \mid Q!e$$

guarded commands:

$$g ::= \; b \Rightarrow c \mid g \mathbin{[\!]} g$$

expressions:

$$e ::= \; v \mid x \mid e \; op \; e$$

The grammar defines the sets *Prog* of programs, *Com* of commands, *GCom* of guarded commands and *Exp* of expressions where x ranges over a set *Var* of variables, b over a set $BExp \subseteq Exp$ of boolean expressions, *op* over the set of operators $\{+, -, \times, \div, <, \leq\}$ and P, Q and R over a set *Pid* of process identifiers.

Plotkin ignores expressions. We do not because here the treatment of reading and writing variables is important.

Informal semantics

A CSP program is the parallel composition of a collection of sequential processes. Processes do not communicate via shared variables (as in, e.g., Concurrent Pascal [BH83] and Modula [Wir83]). Instead, they communicate via *handshaking* (or *synchronized message-passing*), where two processes synchronize, one to send data, and the other to receive it.

Handshaking is not via named channels (as in CCS [Mil89], Concurrent ML [Rep91a], Occam [Wex89] and later versions of CSP [Hoa85]), but by explict addressing. Thus if a variable is shared by two processes, it must remain constant. Each process is identified by a name and communication occurs when a process Q directs output to process R, and R simultaneously receives it from Q. For this scheme to work, each name must refer to a unique process. This is ensured by the static semantics of CSP (which also requires that no process may write to a shared variable) found in section 5.1.2.

The relevant commands are: $R!e$ to evaluate e and send its value to process R; $Q?x$ to input a message from Q and store it in variable x; $R :: c$ to declare the process R to have sequential body c and $p_0 \parallel p_1$ to execute p_0 and p_1 in parallel.

The sequential processes are based on the programs of Dijkstra's Guarded Command Language [Dij76] augmented with the commands for input and output. The syntax is specified in the command, guarded command and expression syntactic categories.

Expression evaluation is straightforward. Constants evaluate to themselves, variables are looked up in the store, and operations are applied to their arguments in the standard way. Unlike [Plo83], expressions (and hence guarded commands) never raise an error.

A guarded command is either primitive ($b \Rightarrow c$ where b is the *guard* of c) or the alternative composition of two guarded commands ($g_0 \talloblong g_1$). The primitive guarded command is Plotkin's concurrent variant of Dijkstra's guarded command, $b \rightarrow c$. Dijkstra's command only permits c to be executed if b is true; otherwise it fails. In an alternative composition, only one of the true-guarded commands is selected. If every guard is false then the composition fails as a whole.

Plotkin's strongly guarded command $b \Rightarrow c$ only permits c to be selected if both b evaluates to true and c can proceed (i.e., it is not blocked waiting to communicate). The point of this is to allow external processes to determine which command is selected. If the first action of each command $c_1, \ldots, c_n$ is a communication, and the guards $b_1, \ldots, b_n$ all evaluate to true then the alternative composition of $b_1 \Rightarrow c_1$ up to $b_n \Rightarrow c_n$ will select one of the commands that can successfully communicate at selection time. Note that if no true-guarded command can proceed, the guarded command blocks until such time as one can. It does not fail. Thus a guarded command will either fail if no guard is true, block, or select a command.

The command `skip` does nothing. The sequential composition $c_0; c_1$ of c_0 and c_1 executes c_0 first and then c_1 after c_0 has finished. `abort` stops the execution of the entire program signalling an error. The conditional `if` g `fi` selects and executes a true-guarded command that can proceed from g. If g fails, an error has occurred and the command aborts the program. The repetition command `do` g `od` repeatedly executes the guarded command g until it fails.

5.1.2 Static Semantics

We mentioned above that the static semantics for CSP consisted of two conditions: no two processes should share a name, and processes may only share read-only variables. Another important property is the well-typedness of expressions. I could present a separate deduction system for the static semantics, in the manner of [Plo83]; however to do so would distract attention — it would not be used in what follows.

Instead, we shall add a suitable side condition to the parallel composition rule of the dynamic semantics. This side condition depends on the functions $FV(c)$, $WV(c)$ and $A(p)$ that return the variables occurring in c (or p), the variables written to by c (or p), and the agent names given bodies in p respectively. They are defined by the following tables.

	v	x	$e_1\ op\ e_2$
FV	$\emptyset$	$\{x\}$	$FV(e_1) \cup FV(e_2)$
WV	$\emptyset$	$\emptyset$	$\emptyset$

Expressions

	$x := e$	`skip`	`abort`	$c_0; c_1$	`if` g `fi`	`do` g `od`
FV	$\{x\} \cup FV(e)$	$\emptyset$	$\emptyset$	$FV(c_0) \cup FV(c_1)$	$FV(g)$	$FV(g)$
WV	$\{x\}$	$\emptyset$	$\emptyset$	$WV(c_0) \cup WV(c_1)$	$WV(g)$	$WV(g)$

The sequential component

	$P?x$	$Q!e$	$p_0 \parallel p_1$	$R :: c$
FV	$\{x\}$	$FV(e)$	$FV(p_0) \cup FV(p_1)$	$FV(c)$
WV	$\{x\}$	$\emptyset$	$WV(p_0) \cup WV(p_1)$	$WV(c)$
A	–	–	$A(p_0) \cup A(p_1)$	$\{R\}$

The concurrent component

	$b \Rightarrow c$	$g_0 \parallel g_1$
FV	$FV(b) \cup FV(c)$	$FV(g_0) \cup FV(g_1)$
WV	$WV(c)$	$WV(g_0) \cup WV(g_1)$

The guarded command component

$$\begin{array}{rl}
\text{sorts:} & Store, IVar, BVar, Var, Val \\
\text{functions:} & \texttt{empty} : Store \\
 & \cdot[\cdot/\cdot] : Store \times Val \times Var \to Store \\
 & \cdot(\cdot) : Store \times Var \to Val \\
\text{metavars:} & \sigma, \sigma', \ldots, \sigma_0, \sigma_1, \ldots \\
\text{equations:} & \texttt{empty}(x) = 0 \;\; \text{if } x \in IVar \\
 & \texttt{empty}(x) = \mathit{ff} \;\; \text{if } x \in BVar \\
 & \sigma[v/x](x) = v \\
 & \sigma[v/x](y) = \sigma(y) \;\; \text{if } y \neq x
\end{array}$$

Figure 5–1: Algebraic definition of stores

5.1.3 Auxiliary Definitions: Stores

The semantics of CSP requires a store. Let $IVar$ be a countable set of *integer variables* and $BVar$ be a disjoint countable set of *boolean variables*. Then a store maps elements of $Var = IVar \cup BVar$ to elements of the set Val of *values* (ranged over by v), consisting of both the integers n and the booleans b. The set $Store$ (ranged over by σ) is defined algebraically in figure 5–1 such that $\sigma(x)$ returns the value of x in σ, and $\sigma[v/x]$ updates the value of x to v. A good introduction to algebraic definitions is [Wec92].

5.1.4 Dynamic semantics

This section gives an evaluation DQI-semantics of CSP. The main purpose of this section is to demonstrate that the techniques established in previous chapters scale up to a larger semantics. Note that in appendix B we outline the proof of an equivalence result (proposition B.12) between the following evaluation semantics and a more traditional structural operational semantics. This theorem helps both to explain the nonstandard evaluation judgments and also to improve our confidence that the following semantic definition really does define CSP.

The abortion-free fragment

The dynamic semantics for the abortion-free fragment of CSP can be found in figure 5–2. Let us call the abortion-free fragment $\mathcal{CSP}$. The rules are very concise because the judgments have minimal structure. The following table describes the intended meaning of the abortion-free judgments.

Judgment	Intended meaning
$R : e \to v$	agent R evaluates e to v
$R : c$	agent R performs c successfully
$R : g$	agent R evaluates guard g successfully
$R * g$	agent R fails to evaluate guard g
p	the agents of p co-operate successfully
$\sigma \rightsquigarrow \sigma'$	the agents transform store σ to σ'

Once again, we require two extra judgments: $R : \mathit{set}(x, v)$ and $R : \mathit{out}(Q, v)$. The need for these was discussed in section 3.3 in the context of the language P(;,:=). Figure 5–3 illustrates the evaluation of a simple CSP program.

Aborting programs

The most straightforward way to capture abortion is to use propagation rules, in much the same way as exceptions are propagated in the definition of Standard ML. This increases the number of rules — for each ordinary rule with n premises, n extra propagation rules are required. To aid legibility of the definition, the authors introduce an *exception convention* (see [HMT90, §6.7]) to minimise the number of rules they have to write.

We can use this technique to capture abortion as follows (we shall give another technique which avoids propagation shortly). We use the following judgments:

Judgment	Intended meaning
$p\dagger$	the agents of p aborted
$R \dagger c$	agent R aborted while running c
$R \dagger g$	agent R aborted while evaluating guard g

The rules can be found in figure 5–4. We use *abortion contexts* to cut down the number of propagation rules required. These are based on the *evaluation contexts* of [FFHD87] in transition semantics. Their use is cosmetic: when proving results about languages defined with evaluation contexts, we still have to unwind the definition of "context", which may require an extra lemma.

Our abortion contexts are defined by the following grammar:

$$\begin{aligned} \boldsymbol{G} &::= [\cdot] \mid \boldsymbol{G} \mathbin{[\!]} g \mid g \mathbin{[\!]} \boldsymbol{G} \\ \boldsymbol{A} &::= [\cdot] \mid \boldsymbol{A}; c \mid \texttt{if}\ \boldsymbol{G}\ \texttt{fi} \mid \texttt{do}\ \boldsymbol{G}\ \texttt{od} \end{aligned}$$

Aborting parallel programs is an interesting problem. Hoare [Hoa78, p.668] simply says that "the parallel command terminates successfully only if and when [the processes] have all successfully terminated". This is a loose specification: the semantics of abort may be either to kill every process simultaneously or to wait until the other processes have terminated (which may not happen).

The second choice is more natural from an implementer's point of view, and is the semantics given in figure 5–4. It is also easier to express in an evaluation semantics (unlike the transition semantics of [Plo83], where it is marginally easier to express the former).

Rules for expressions

$$\frac{}{R : v \to v} \qquad \frac{}{R : x \to v} \; \overset{lookup}{\rule{3em}{0.4pt}} \; \frac{}{\sigma \rightsquigarrow \sigma} \; \sigma(x) = v$$

$$\frac{R : e_1 \to v_1 \qquad R : e_2 \to v_2}{R : e_1 \; op \; e_2 \to app(op, v_1, v_2)}$$

Rules for the sequential component

$$\frac{R : e \to v \;\; \blacktriangleright \;\; R : set(x, v)}{R : x := e} \qquad \frac{}{R : set(x, v)} \; \overset{assign}{\rule{3em}{0.4pt}} \; \frac{}{\sigma \rightsquigarrow \sigma[v/x]}$$

$$\frac{R : c_0 \quad \blacktriangleright \quad R : c_1}{R : c_0; c_1} \qquad \frac{R : g}{R : \texttt{if}\ g\ \texttt{fi}} \qquad \frac{\sigma \rightsquigarrow \sigma' \quad \blacktriangleright \quad \sigma' \rightsquigarrow \sigma''}{\sigma \rightsquigarrow \sigma''}$$

$$\frac{R : g \quad \blacktriangleright \quad R : \texttt{do}\ g\ \texttt{od}}{R : \texttt{do}\ g\ \texttt{od}} \qquad \frac{R * g}{R : \texttt{do}\ g\ \texttt{od}} \qquad \frac{}{R : \texttt{skip}}$$

Rules for guards

$$\frac{R : b \to tt \quad \blacktriangleright \quad R : c}{R : b \Rightarrow c} \qquad \frac{R : b \to ff}{R * b \Rightarrow c}$$

$$\frac{R : g_0}{R : g_0 \talloblong g_1} \qquad \frac{R : g_1}{R : g_0 \talloblong g_1} \qquad \frac{R * g_0 \quad R * g_1}{R * g_0 \talloblong g_1}$$

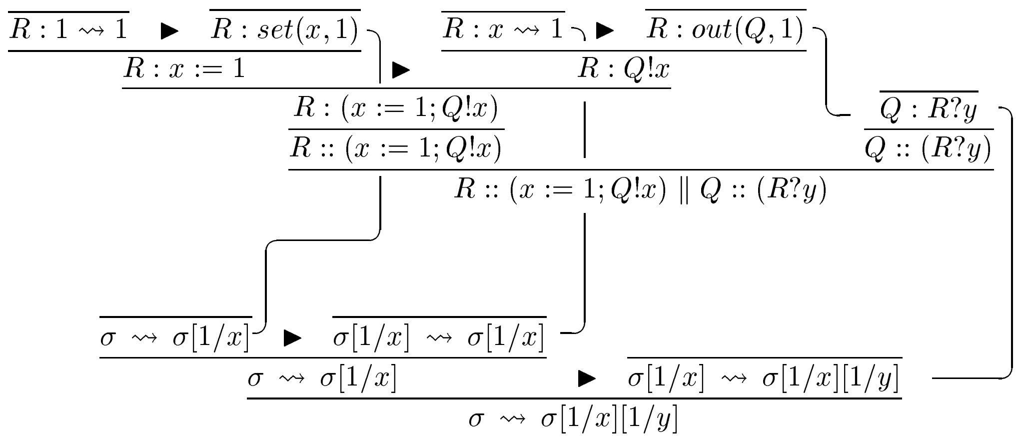
Rules for the concurrent component

$$\frac{}{R : out(Q, v)} \; \overset{comm}{\rule{3em}{0.4pt}} \; \frac{}{Q : R?x} \; \overset{assign}{\rule{3em}{0.4pt}} \; \frac{}{\sigma \rightsquigarrow \sigma'[v/x]}$$

$$\frac{R : e \to v \;\; \blacktriangleright \;\; R : out(Q, v)}{R : Q!e} \qquad \frac{R : c}{R :: c}$$

$$\frac{p_0 \quad p_1}{p_0 \parallel p_1} \; FV(p_0) \cap WV(p_1) = FV(p_1) \cap WV(p_0) = A(p_0) \cap A(p_1) = \emptyset$$

Figure 5–2: Evaluation semantics rules for the abortion-free fragment of CSP



Note that the interaction links are flowing behind the upper inference tree.

Figure 5–3: Evaluation of a simple program

Rules that generate abortion

$$\frac{}{R \dagger \texttt{abort}} \qquad \frac{R * g}{R \dagger \texttt{if}\ g\ \texttt{fi}}$$

Rules that propagate abortion

$$\frac{R : c_0 \quad \blacktriangleright \quad R \dagger c_1}{R \dagger c_0; c_1} \qquad \frac{R : g \quad \blacktriangleright \quad R \dagger \texttt{do}\ g\ \texttt{od}}{R \dagger \texttt{do}\ g\ \texttt{od}} \qquad \frac{R : b \to tt \quad \blacktriangleright \quad R \dagger c}{R \dagger b \Rightarrow c}$$

$$\frac{R \dagger c}{R \dagger \boldsymbol{A}[c]} \qquad \frac{R \dagger c}{R :: c\dagger} \qquad \frac{p_0\dagger \quad p_1}{p_0 \parallel p_1\dagger} \qquad \frac{p_0 \quad p_1\dagger}{p_0 \parallel p_1\dagger} \qquad \frac{p_0\dagger \quad p_1\dagger}{p_0 \parallel p_1\dagger}$$

Figure 5–4: Abortion rules for CSP

Propagation-free abortion rules

We can model self-abortion (and I think also exceptions) without the use of propagation rules. Instead we use the strictly pruned deductions of section 3.4.3. The main observation is that in a sequential process, the leaves of an evaluation deduction after an aborting command are sequenced. Therefore, if we ensure that aborting commands can only be deduced using the pruning rule, everything sequenced after an abort will be pruned too (in a strictly pruned deduction). That is, the program sequenced after the aborting command will not be evaluated.

Figure 5–5 contains the rules. The special formula "abort" cannot be deduced by any rule except pruning. We use the formula "aborted" to indicate that abortion has occurred. The scoping of the *abort* signal ensures that we cannot ignore this indicator.

Note that it makes use of the scoping condition borrowed from section 3.4.4. It has to be redefined for the current context:

Let Π be a DQI-deduction, let X be a set of copremises of Π and let L be a set of formula perspectives. Then X *scopes* L *in* Π if for all $A, B \in O(\Pi)$ and $\alpha \in L$,

$$\begin{array}{ll} (i) & \text{if } AI_\alpha B \text{ and } A \gtrsim_\Pi X \text{ then } B \gtrsim_\Pi X \\ (ii) & D(\Pi) \cap (O(\Pi) \times L) = \emptyset \end{array}$$

where I_α is the set of α-labelled interactions of Π, and $A \gtrsim_\Pi X$ means that $A \gtrsim_\Pi B$ for some $B \in X$. Thus if X scopes L in Π, then either both ends of an L-labelled interaction must occur above X or both ends must not. Again we write the side-condition graphically

$$\frac{\llcorner P_1 \ \cdots \ P_n \lrcorner_L \ P}{C}$$

to mean that $\{P_1, \ldots, P_n\}$ scopes L in whatever deduction the rule atom is applied to. In a deduction, we draw a box around the scoped trees.

For these rules to make sense, we have to stress that the pruning rule is used only to terminate aborted programs. That is, for each program p, we take the deductions to be the limits of the chains in $\mathcal{CSP}_\bot^{(\dagger)}(p)$. These deductions will also include deadlocking evaluations, and other stuck states. The following deduction is an example, where $g_1 = tt \Rightarrow \texttt{skip}$ and $g_2 = tt \Rightarrow \texttt{abort}$:

$$\frac{\boxed{\dfrac{\dfrac{\dfrac{\overline{tt \to tt} \ \blacktriangleright \ \overline{R : \texttt{skip}}}{R : tt \Rightarrow \texttt{skip}}}{R : g_1 \mathbin{[\!]} g_2} \ \blacktriangleright \ \dfrac{\dfrac{\dfrac{\overline{tt \to tt} \ \blacktriangleright \ \dfrac{\dfrac{\bot}{\text{abort}}}{R : \texttt{abort}}}{R : tt \Rightarrow \texttt{abort}}}{R : g_1 \mathbin{[\!]} g_2} \ \blacktriangleright \ \dfrac{\bot}{R : \texttt{do}\ g_1 \mathbin{[\!]} g_2\ \texttt{od}}}{R : \texttt{do}\ g_1 \mathbin{[\!]} g_2\ \texttt{od}}}{R : \texttt{do}\ g_1 \mathbin{[\!]} g_2\ \texttt{od}} \qquad \overline{\text{aborted}}}_{\,K}}{R :: \texttt{do}\ g_1 \mathbin{[\!]} g_2\ \texttt{od}\dagger}$$

$$\frac{\text{abort}}{R : \texttt{abort}} \xrightarrow{abort} \frac{}{\text{aborted}} \qquad \frac{R * g \quad \blacktriangleright \quad \text{abort}}{R : \texttt{if}\ g\ \texttt{fi}} \xrightarrow{abort} \frac{}{\text{aborted}}$$

$$\frac{\llcorner R : c \qquad \text{aborted} \lrcorner_K}{R :: c\dagger} \qquad \frac{\llcorner R : c \lrcorner_K}{R :: c}$$

$$\frac{p_0\dagger \quad p_1}{p_0 \parallel p_1\dagger} \qquad \frac{p_0 \quad p_1\dagger}{p_0 \parallel p_1\dagger} \qquad \frac{p_0\dagger \quad p_1\dagger}{p_0 \parallel p_1\dagger}$$

Where $K = \{abort\}$. These rules are added to the rule of $\mathcal{CSP}_\perp$ except for the old naming rule, which is replaced by the two naming rules above. Call the resulting system $\mathcal{CSP}_\perp^{(\dagger)}$.

Figure 5–5: Propagation-free abort rules for $\mathcal{CSP}_\perp$

5.2 Some alternative features

We consider ways of capturing the semantics of six different alternative features, intended to capture different aspects of concurrent languages. It turns out that four extensions require the scoping side-condition.

The alternative features are:

- Nested parallelism
- Shared variables
- Dynamic Process Creation
- Pre-emption
- Multicasting
- Recursive Procedures

5.2.1 Nested Parallelism

If a program has nested parallelism it means that the parallel composition operator can occur arbitrarily deeply in a program, not just at the top level. The simplest way to introduce this is to add an extra production to the syntax of commands:

$$c ::= \ldots \mid [p]$$

Where $p \in Prog$. The major difficulty is one of scope. If we can name processes arbitrarily then we can introduce holes in the scope of namings, where a parallel composition nested in R may include another process named R. Within that nested composition, all communications naming R will refer to the inner process. Outside it, all communications naming R will refer to the outer process. For instance, consider the two programs

$$R :: [Q :: R!1 \parallel R :: Q?x] \parallel Q :: \texttt{skip}$$
$$R :: [Q :: \texttt{skip} \parallel R :: Q?x] \parallel Q :: R!1$$

the first will not deadlock, whereas the second will. This means that the obvious rule for $[p]$, namely $\dfrac{p}{R : [p]}$, is not sufficient — it would not force the second example to deadlock. There is no notion of scoping. Therefore we use the scoping side-condition.

$$\frac{\llcorner p \lrcorner_{AGENTS(p)}}{R : [p]}$$

where $AGENTS(p) = \{comm(R, Q, v) \mid v \in Val \text{ and } R \in A(p) \text{ or } Q \in A(p)\}$ is the set of all labels that record a communication with an agent defined in the top level of p.

5.2.2 Shared Variables

CSP communicates by message-passing. The alternative way to communicate is via shared variables. This requires mutual exclusion primitives such as *semaphores* [Dij65] or *monitors* [Hoa74]. CONCURRENT PASCAL [BH83], MODULA [Wir83], and the communication paradigm LINDA [BCGL88] use shared variables to communicate.

It is easy to introduce shared variables to our semantics: we drop the "no shared variable" condition from the static semantics. To add mutual exclusion primitives is harder. Mutual exclusion is the function of the `await` command [OG76]:

$$\texttt{await } b \texttt{ then } c$$

where c does not contain a nested `await` , communication or parallel composition. When a process attempts to execute an `await` command, it is delayed until b is true. Then c is executed as an indivisible action. If a number of processes are awaiting b, only one may proceed when b is true. The evaluation of b is treated as if it were part of the indivisible action c. This implies some kind of scheduling mechanism, but we do not need to build one into the semantics.

One can use this primitive to code semaphores. If s is a binary semaphore (a variable with value 0 or 1) then the operations $P(s)$ (wait until s is free, i.e., $s = 0$, and then grab it) and $V(s)$ (signal that s is free) can be coded:

$$\begin{array}{ll} P(s) & \texttt{await } s = 0 \texttt{ then } s := 1 \\ V(s) & s := 0 \end{array}$$

The await command is not intended as a practical programming language feature: it is too powerful to be implemented efficiently. But it does embody all the relevant semantic features of mutual exclusion and synchronization. The semantic rule for `await` is given by:

$$\frac{\llcorner R : b \rightarrow tt \;\; \blacktriangleright \;\; R : c \qquad \sigma \rightsquigarrow \sigma' \lrcorner_{STORE}}{R : \texttt{await } b \texttt{ then } c} \quad \frac{atomic}{\qquad} \quad \frac{}{\sigma \rightsquigarrow \sigma'}$$

Where $STORE$ is the set of all *assign* and *lookup* labels. That is,

$$\begin{aligned} STORE = \; & \{assign(x, v) \mid x \in Var, v \in Val\} \cup \\ & \{lookup(x, v) \mid x \in Var, v \in Val\} \end{aligned}$$

The scoping condition forces every store interaction of the evaluation of b and the execution of c to be with the inference tree above the $\sigma \rightsquigarrow \sigma'$ premise of the `await` rule atom. No outside process can interfere with this internal store tree. When c has finished, the outer, world copy of the store is altered (in one step — i.e., atomically) according to how the atomic action altered it.

5.2.3 Dynamic process creation

The next feature we wish to consider is dynamic process creation, such as the `spawn` function in Concurrent ML [Rep91b]. In the version of CSP with nested parallelism, if the parallel composition of a number of processes is composed sequentially before c (say), then c will only be executed after all of the processes have terminated.

The behaviour of `spawn` is different. Once a process has been spawned the spawning process can proceed immediately, executing concurrently with the new process. One does not usually find parallel composition and spawn together in a language, so this "extension" is not really an extension. Instead, we remove parallel composition from our language, and add a spawn command, to get:

$$\begin{aligned} p &::= R :: c \\ c &::= \cdots \mid \texttt{spawn}\ (Q, c) \end{aligned}$$

Where $Q \in \mathit{Pid}$. For the dynamic semantics, we introduce a special thread judgment to provide a place where spawned processes can be evaluated:

$$\frac{}{\text{thread}(X)} \qquad\qquad \frac{}{R : \texttt{spawn}\ (Q, c)} \;\text{——}\; \frac{Q : c \qquad \text{thread}(X \setminus \{Q\})}{\text{thread}(X \cup \{Q\})}\; R \notin X$$

The judgment thread(X) is simply a space-filler, where X denotes the set of process identifiers not in use. The condition $X \cup \{Q\}$ in the conclusion of the second atom of the second rule ensures that Q is not already being used, and $X \setminus \{Q\}$ ensures that no other process can use it.

5.2.4 Pre-emption

A control feature is *pre-emptive* if it can influence the behaviour of another process. Common pre-emptive features are *interruption*, *restart*, *checkpointing*, *alternation* (all in [Hoa85, §5.4]), *abortion* and *suspension* (in [Ber93a]).

[Ber93a] distinguishes two forms of pre-emption: *must pre-emption* where the parallel processes are pre-empted instantaneously and *may pre-emption* where the pre-emption may be delayed for an arbitrarily large period of time. This latter form of pre-emption is the only one which can be defined in CSP or CCS, because they have no intrinsic notion of time. In chapter 6 I show that the computational semantics of DQI-deduction is (a subset of) the CCS. This implies that we cannot use our system of evaluation semantics to model must pre-emption, and therefore languages like ESTEREL [BG92].

5.2.5 Multicasting

Multicasting is sending a message to a collection of named processes. It is a generalisation of the point-to-point communication paradigm we have been considering so far. It is a restriction of broadcasting which sends a message to every process. We extend the syntax of CSP commands to become

$$\begin{aligned} c &::= \cdots \mid \langle \mathit{Pidlist} \rangle ! e \\ \mathit{Pidlist} &::= \mathit{Pid} \mid \mathit{Pid}, \mathit{Pidlist} \end{aligned}$$

Intuitively, $R : \langle Q_1, \ldots, Q_n \rangle ! e$ will evaluate e and send it to each process $Q_1, \ldots, Q_n$ simultaneously. (Of course there is a static condition: no process ought to multicast to itself.) We use the structure of DQI-systems in an essential way to give us a collection of building bricks, out of which we can build chains of communications.

$$\frac{R : e \rightarrow v \quad \blacktriangleright \quad R : ms(Q_1, \ldots, Q_n, v)}{R : \langle Q_1, \ldots, Q_n \rangle ! e}$$

$$\frac{}{R : ms(Q_1, \ldots, Q_n, v)} \; ms(R, Q_1, \ldots, Q_n, v)$$

$$\overline{ms}(R, x_1, \ldots, x_{i-1}, Q_i, \ldots, Q_n, v) \; \frac{1 \le i \le n}{Q_i : R?x_i} \; ms(R, x_1, \ldots, x_i, Q_{i+1}, \ldots, Q_n, v)$$

$$\overline{ms}(R, x_1, \ldots, x_n, v) \; \frac{}{\sigma \rightsquigarrow \sigma[v/x_1] \ldots [v/x_n]}$$

Broadcasting (e.g. [Pra91,Pra93]) could be implemented in a similar manner, but it requires that the semantic rules maintain a central record of nonterminated processes. This is a problem in its own right.

5.2.6 Procedures

We introduce procedures and block structure to CSP. Let *Proc* be a set of procedure names (ranged over by π). Then we extend the grammar of commands to include three new constructs:

$$c ::= \ldots \mid \texttt{proc}\ \pi(x)\ \texttt{is}\ c \mid \pi(e) \mid \texttt{var}\ x = v\ \texttt{in}\ c$$

The first defines a procedure π with parameter x (it is a straightforward extension to handle many parameters) to have body c. The second calls procedure π with argument the value of e. The third declares variable x to have initial value v with scope c.

Procedures introduce non-local flow-of-control into programs: the definition of a procedure will not be part of a program's local structure. Consequently, there may be a difference in variable bindings when a procedure is defined and when it is invoked. The question is which binding to look up when running a procedure. The *static* binding discipline is to use the define-time variable bindings. The *dynamic* binding discipline is to use the call-time variable bindings.

A simple store-based approach is no longer sufficient. The reason is one of scope: a variable may refer to different locations at different times. For instance, a procedure call may temporarily rebind a variable.

Therefore, modeling the binding disciplines requires both environments and stores. An environment maps variables to store *locations* Loc and procedure names to procedure *closures*. A closure is either *static* or *dynamic*, depending on the binding discipline. Intuitively, a static closure is a triple $\langle x, c, E\rangle$ which parcels up the formal parameter x, the body c and the define-time environment E (thus environments and static closures are mutually dependent notions).

Formally, we define the *Static Environments*, $SEnv$ to be the set $\bigcup_{i\in\mathbb{N}} SEnv_i$ where

$$\begin{aligned} SEnv_0 &= Var \rightharpoonup Loc \\ SEnv_{i+1} &= SEnv_i \cup (Proc \rightharpoonup Var \times Com \times SEnv_i) \end{aligned}$$

This scheme is sufficient to allow us to define recursive procedures (and also, with more work, families of mutually recursive procedures [Mit].) An alternative approach, using coinduction, is to allow infinite closures that model recursively defined procedures directly [MT92,Sch95].

A dynamic closure is a pair $\langle x, c\rangle$. Formally, we define the *Dynamic Environments*, $DEnv$ to be the set

$$DEnv = (Var \rightharpoonup Loc) \cup (Proc \rightharpoonup Var \times Com)$$

We use E to range over both $SEnv$ and $DEnv$, and σ to range over stores in $Stores$. The definition is the same as in section 5.1.3 except that it maps locations in Loc to values in Val. We introduce a special judgment to record environments: $R : E$ means that the environment of process R is E. (In a shared variable context, the environment would be global like the store.) The rules for the extension of CSP can be found in figure 5–6, where we have abbreviated labels to their names for clarity.

The rules depend on the auxiliary definition $ENV(R)$. This is the set of all environment accessing labels (i.e., the *def*, *call*, *read* and *write* labels) that mention R. Its precise definition is similar to that of $STORE$:

$$\begin{aligned} ENV(R) = \; & \{def(R,\pi,x,c) \mid \pi \in Procs, x \in Var, c \in Com\} \cup \\ & \{call(R,\pi,x,c,E) \mid \pi \in Procs, x \in Var, c \in Com, E \in Env\} \cup \\ & \{block(R,E,n,v) \mid E \in Env, n \in \mathbb{Z}, v \in Val\} \cup \\ & \{read(R,x,v) \mid x \in Var, v \in Val\} \cup \\ & \{write(R,x,v) \mid x \in Var, v \in Val\} \end{aligned}$$

The main rules are the first two, which deal with procedure definition and invocation. We have only given one half of these rules: they can be completed to support static or dynamic binding. The rule fragments for static binding are:

$$\frac{\overline{def}(R,\pi,x,c)}{} \quad \frac{R : E[\langle x, c, E\rangle/\pi]}{R : E}$$

$$\frac{\overline{call}(R,\pi,x,c,E')}{} \quad \frac{R : E \qquad E(\pi) = \langle x, c, E'\rangle}{R : E}$$

$$\frac{}{R : \texttt{proc}\ \pi(x)\ \texttt{is}\ c} \quad \frac{\scriptstyle def(R,\pi,x,c)}{}$$

$$\frac{\llcorner R : c \qquad R : E[n/x] \lrcorner_{ENV(R)} \qquad n \notin \mathrm{dom}\ E}{R : call(\pi, v)} \quad \frac{}{} \quad \frac{}{\sigma \rightsquigarrow \sigma[v/n]} \quad \frac{\scriptstyle call(R,\pi,x,c,E)}{}$$

$$\frac{R : e \rightsquigarrow v \quad \blacktriangleright \quad R : call(\pi, v)}{R : \pi(e)}$$

$$\frac{\llcorner R : c \qquad R : E[n/x] \lrcorner_{ENV(R)} \qquad n \notin \mathrm{dom}\ E}{R : \texttt{var}\ x = v\ \texttt{in}\ c} \quad \frac{\scriptstyle block}{} \quad \frac{R : E}{R : E} \quad \frac{\scriptstyle assign}{} \quad \frac{}{\sigma \rightsquigarrow \sigma[v/n]}$$

$$\frac{}{R : x \rightsquigarrow v} \quad \frac{\scriptstyle read}{} \quad \frac{R : E}{R : E} \quad \frac{\scriptstyle lookup}{} \quad \frac{\sigma(E(x)) = v}{\sigma \rightsquigarrow \sigma}$$

$$\frac{}{R : x \rightsquigarrow v} \quad \frac{\scriptstyle write}{} \quad \frac{R : E}{R : E} \quad \frac{\scriptstyle assign}{} \quad \frac{}{\sigma \rightsquigarrow \sigma[v/E(x)]}$$

$$\frac{}{R : out(Q, v)} \quad \frac{}{} \quad \frac{}{Q : R?x} \quad \frac{\scriptstyle write}{} \quad \frac{Q : E}{Q : E} \quad \frac{\scriptstyle assign}{} \quad \frac{}{\sigma \rightsquigarrow \sigma[v/E(x)]}$$

$$\frac{}{R : E}$$

Figure 5–6: DQI-Rules for recursive procedures and blocks

While the rule fragments for dynamic binding are:

$$\frac{\overline{def}(R,\pi,x,c)}{} \quad \frac{R : E[\langle x, c\rangle/\pi]}{R : E}$$

$$\frac{\overline{call}(R,\pi,x,c,E)}{} \quad \frac{R : E \qquad E(\pi) = \langle x, c\rangle}{R : E}$$

This example is quite untidy: it shows that while we can add environments and procedures "in a modular way", if we do so, we end up with a mess. It would probably be better to attach environments to evaluation judgments explicitly, especially since each process has its own environment.

5.3 An Application: Program Verification

Program verification is an important area of computer science: in the real world, erroneous programs cause disasters. It follows that "correctness" and "safety" proofs (i.e., proofs that programs meet some specification) are important.

But programs are complex, which means that reasoning about them will be complex too. We need to minimise the cost of verification. One way is to reason in a syntax-directed manner: one inserts assertions about the state of the world between program statements and shows that the program satisfies them [Flo67]. For instance, if we have a program $x := 0; y := 1$, we might add assertions

$$\{\text{true}\}\ x := 0\ \{x = 0\}\ y := 1\ \{x = 0 \wedge y = 1\}$$

which tell us what we know about the store after each atomic step of the program.

The question is how to prove that a program satisfies these assertions. There is a wide variety of techniques: for instance, weakest preconditions [Dij76], "Hoare Logics" [Hoa69], specification logic [Rey81], evaluation logic [Pit91], dynamic logic [Har84], and other modal and temporal logics [Sti92].

We consider a "Hoare Logic" for the partial correctness of CSP programs. Since our version of CSP does not have full procedures, this approach is "quite satisfactory" [OPTT95]. (The problem with procedures is that they can introduce *aliasing* which means that differently named variables can interfere with each other. See *op. cit.* for a full discussion.)

A Hoare Logic is really a deductive system whose judgments relate pre- and post-conditions to program statements. The judgments are of form $\{\phi\}\ c\ \{\psi\}$, and intuitively mean "if ϕ is true immediately before executing c then ψ is true immediately after". [Apt81] is a comprehensive survey of Hoare Logics for sequential while-languages. [Hoo86] is a survey of deductive systems for partial correctness of CSP.

There are three main points motivating this application. First, it is an example application: we can use (a modified form of) $\mathcal{CSP}$ to prove the rules of the Hoare Logic sound. Second, it is distinct from previous Hoare Logics because it is an

DQI-system in its own right. Third, because it is a DQI-system, it demonstrates an application of our work outside the sphere of operational semantics.

To illustrate the method, we consider the ubiquitous greatest common divisor algorithm, as implemented by the program GCD:

$$
\begin{array}{l}
i := m; \\
j := n; \\
\texttt{do}\ \ i > j \ \Rightarrow\ i := i - j \\
\quad\ \ [\!]\, j > i \ \Rightarrow\ j := j - i \\
\texttt{od}
\end{array}
$$

The program is intended to run in the situation where both m and n are constants greater than zero. We have to prove that when started in this situation, the program will terminate such that $i = j = gcd(m, n)$. That is, we want to prove

$$\{m > 0 \wedge n > 0\} \ \ GCD \ \ \{i = j = gcd(m, n)\}$$

Figure 5–7 gives our deductive system for CSP. For the moment, we ignore the concurrency rules, and use the sequential system to prove the above program correct. The deduction is too big to write on the page, so instead we give the following *proof outline* [AO91] which can be thought of as a flattened deduction:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\{m > 0 \wedge n > 0\}} \\
\quad & \multicolumn{3}{l}{i := m;} \\
\multicolumn{4}{l}{\{m > 0 \wedge n > 0 \wedge i = m\}} \\
\quad & \multicolumn{3}{l}{j := n;} \\
\multicolumn{4}{l}{\{m > 0 \wedge n > 0 \wedge i = m \wedge j = n\}} \\
\multicolumn{4}{l}{\{gcd(i, j) = gcd(m, n)\}} \\
\quad & \texttt{do}\ \ i > j & \Rightarrow & \{gcd(i - j, j) = gcd(m, n)\} \\
 & & & \quad i := i - j \\
 & & & \{gcd(i, j) = gcd(m, n)\} \\
 & \quad [\!]\, j > i & \Rightarrow & \{gcd(i, j - i) = gcd(m, n)\} \\
 & & & \quad j := j - i \\
 & & & \{gcd(i, j) = gcd(m, n)\} \\
\quad & \multicolumn{3}{l}{\texttt{od}} \\
\multicolumn{4}{l}{\{gcd(i, j) = gcd(m, n) \wedge i = j\}}
\end{array}
$$

The proof depends on the well-known fact about the greatest common divisor:

$$i > j \wedge gcd(i, j) = gcd(m, n) \ \supset\ gcd(i - j, j) = gcd(m, n)$$

In the following we shall abbreviate proof outlines to record only the essential information.

Verifying concurrent programs

Verifying concurrent programs is much harder. The reason is that one often wants to verify relationships that hold between processes. In a strictly compositional system (such as [MC81] or [ZdBdR83]) one has to break global relationships of processes into invariant properties of communication. It is not clear how to break global relationships of distributed algorithms such as the distributed GCD algorithm [AO91].

Levin and Gries's proof system for CSP [LG81] splits the verification process into two halves. First, one proves properties about the sequential processes in isolation. Second, one proves an *interference freedom* result (in the spirit of Owicki-Gries's method [OG76]) to show that no process inadvertently falsifies the assertions of another process. CSP processes do not share variables, but in this system they can share the *auxiliary* variables (so one can prove global relationships using global auxiliary variables). The main problem here is that the interference freedom test "if approached mechanically, is an awesome task" [LG81, p291]. To reduce the work, one has to structure one's program and choose assertions carefully.

Apt *et al.*'s proof system [AFdR80] also splits the verification process into two halves. However, it forbids even auxiliary variables to be shared between processes. Again first, proofs about the sequential processes are done, and then global relationships are established second (the "co-operation proof"). This approach handles global relationships naturally. However, the co-operation proof also involves a large amount of work, having to consider every syntactically matching pair of communication commands.

These approaches are not compositional, and so are not suited to aid program development. The advantage of strictly compositional proof systems are that one knows if one's program is correct as soon as one has a proof outline for the program. Therefore one can develop one's programs in a modular manner.

The deductive system in figure 5–7 is compositional, belonging to the school of [ZdBdR83], yet is inspired by the rules of [AFdR80]. We shall use it to deduce that a simple set partitioning program is correct.

A Set Partitioning program

This is taken from [AFdR80], and assumes that CSP is extended to include sets of integers and operations for manipulating them. The modifications required to the semantics are trivial.

Given two disjoint sets of integers S and T, the problem is to partition $S \cup T$ into S' and T' such that $|S| = |S'|$ and $|T| = |T'|$ and $\max(S') < \min(T')$. The solution is given in the following program $SP = p_1 \parallel p_2$, where p_1 and p_2 are

$$
\begin{array}{ll}
Q_1 :: & mx := \max(S); \\
& Q_2!mx;\;\; S := S \setminus \{mx\}; \\
& Q_2?x;\;\; S := S \cup \{x\}; \\
& mx := \max(S); \\
& \texttt{do}\;\; mx > x \Rightarrow \\
& \qquad Q_2!mx;\;\; S := S \setminus \{mx\}; \\
& \qquad Q_2?x;\;\; S := S \cup \{x\}; \\
& \qquad mx := \max(S); \\
& \texttt{od}
\end{array}
\qquad \text{and} \qquad
\begin{array}{ll}
Q_2 :: & Q_1?y;\;\; T := T \cup \{y\}; \\
& mn := \min(T); \\
& Q_1!mn;\;\; T := T \setminus \{mn\}; \\
& \texttt{do}\;\; \text{true} \Rightarrow \\
& \qquad Q_1?y;\;\; T := T \cup \{y\}; \\
& \qquad mn := \min(T); \\
& \qquad Q_1!mn;\;\; T := T \setminus \{mn\}; \\
& \texttt{od}
\end{array}
$$

respectively. Intuitively, processes p_1 and p_2 own S and T respectively. The program repeatedly swaps the greatest element of S and the least element of T until the greatest element of S is less than the least element of T.

5.3.1 A Hoare Logic for the partial correctness of CSP

Our deductive system is similar to Misra and Chandy's system in that it is compositional and uses rely and guarantee conditions. It does not require an interference-freedom or co-operation proof. It is dissimilar in that it does not use trace variables. It is similar to Levin and Gries's system in that a special class of auxiliary variables called *views* can be shared between communicating processes. Last, it is similar to Apt *et al.*'s system in that no auxiliary variable can be used globally.

In the following, I assume a first-order language L of assertions with equality and sets over Var ranged over by ϕ, χ, ψ. I use the symbols $\wedge$ for conjunction and $\supset$ for implication. Let Tr_L be the set of true sentences of L (perhaps described by some other deductive system). Let $FV : L \to \wp(Var)$ be a function which returns the set of program variables contained in an assertion. Similarly, I use the function FV defined in section 5.1.2 that returns the set of variables mentioned in a program fragment.

The rules in figure 5–7 are mostly similar to those of [LG81,AFdR80]. (I leave their dependence on the set Tr_L implicit.) The rule for do-loops uses the following auxiliary function

$$Gu(g) = \begin{cases} b & \text{if } g = b \Rightarrow c \\ Gu(g_1) \vee Gu(g_2) & \text{if } g = g_1 \mathbin{[\!]} g_2 \end{cases}$$

The main difference lies with the concurrent rules. One feature is that the rule for parallel composition is quite simple. Another is that the rules for communication have dangling interactions. Each dangling interaction is labelled with a special formula that expresses an invariant property of the communicated information.

Views When process R transmits the value of expression e to Q, we say that Q has a *view* of the value of e in R. Now R knows what Q's view of e will be since it sent it. Thus there exists a natural class of information which is shared between communicating processes. We model this information using a special class of auxiliary variables called *views*, which may be altered only at communication, and which are shared between communicating processes.

However, views will not be related to particular expressions (i.e., data) but to particular occurrences of output commands in the program (i.e., locations). The reason is that expressions may play different roles at different parts of a program, whereas each part of a program plays one well-defined role, established by its pre- and post- conditions in a proof outline. The "role" of a location in a particular deduction is formalised by a special property called the *communication invariant*. This is the information that is shared with a communicating process about the actual message. In fact, it can be thought of as an abstract message.

Actually, it is more convenient to identify sets of locations that play the same role, and associate communication invariants with sets of locations. For instance, in our example program, process p_1 sends the maximum element of S from two locations. It makes sense and simplifies reasoning to have one view of the maximum element of S.

In what follows, I use u, w to range over sets of occurrences of output commands in programs, and I shall annotate programs to indicate how output occurrences are referred to in proofs. I write $\underline{u}$ for the view of u.

Counters The proof technique works by associating invariant properties to each view. The proof outline at the output command occurrence u ought to guarantee that the invariant property holds of $\underline{u}$. Then the proof outline at an input command occurrence can rely on that invariant property. But this depends on the fact that we know which input commands communicate with which output commands. To do this, we introduce special auxiliary views called *counters* which return the last view shared between two communicating processes. Initially, we set the counters to a special null value, written $\perp$. There will be two counters for every pair of processes: $\underline{QR}$ records R's last view of Q (i.e., the last communication from Q to R) and $\underline{RQ}$ records Q's last view of R. These counters have to be maintained by the communication invariants. As a convention, for every program we use $\vec{RQ}$ for the set of all occurrences in R that output to Q.

Communication Invariants Typical communication invariants relate old and new values of views. At the communication rules, we write $\underline{u}'$ for the new value of view u and $\underline{RQ}'$ for the new value of the counter RQ. It is a temporary name used when working out the postcondition of the input rule. It does not itself occur in any pre- or postcondition. I write $\psi^\natural$ for the result of de-priming every primed view or counter in ψ. So, for example, $(\underline{u}' = 5)^\natural$ is $\underline{u} = 5$. This operation occurs in the conclusion of the input rule.

Let $Views$ be the set of all views and counters. I also assume a function $vw : L \to \wp(Views)$ that returns the set of views and counters mentioned in an assertion. Note that views and counters may not appear in a program. Last, I write $VV : L \to \wp(Views \cup Var)$ for the function $VV(\phi) = FV(\phi) \cup vw(\phi)$.

Rules for Input and Output

The only complex rules are the ones for input and output. The output rule is simpler, so we explain it first. All it has to do is show that the precondition implies the postcondition and guarantees the appropriate communication invariant. The main difficulty is that we have to update the appropriate view and counter. Since this is like an assignment, we borrow the "backward assignment" idea and write

$$\phi = \psi[e/\underline{u}][u/\underline{RQ}]$$

where ϕ is the precondition, ψ the postcondition, u the relevant location and the communication sends e from R to Q. This is straightforward. To guarantee the

$\boxed{\text{Rules for the sequential component}}$

$$\frac{}{\{\phi\}\ \texttt{skip}\ \{\phi\}_R} \qquad\qquad \frac{}{\{\phi[e/x]\}\ x := e\ \{\phi\}_R}$$

$$\frac{\{\phi\}\ c_1\ \{\psi\}_R \qquad \{\psi\}\ c_2\ \{\chi\}_R}{\{\phi\}\ c_1; c_2\ \{\chi\}_R} \qquad\qquad \frac{\{\phi\}\ g\ \{\psi\}_R}{\{\phi\}\ \texttt{if}\ g\ \texttt{fi}\ \{\psi\}_R}$$

$$\frac{\{\phi\}\ g\ \{\phi\}_R}{\{\phi\}\ \texttt{do}\ g\ \texttt{od}\ \{\phi \wedge \neg Gu(g)\}_R}$$

$\boxed{\text{Rules for guards}}$

$$\frac{\{\phi \wedge b\}\ c\ \{\psi\}_R}{\{\phi\}\ b \Rightarrow c\ \{\psi\}_R} \qquad\qquad \frac{\{\phi\}\ g_1\ \{\psi\}_R \qquad \{\phi\}\ g_2\ \{\psi\}_R}{\{\phi\}\ g_1 \parallel g_2\ \{\psi\}_R}$$

$\boxed{\text{Rules for the concurrent component}}$

$$\frac{+CI}{} \quad \frac{\phi = \psi[e/\underline{u}][u/\underline{RQ}] \qquad \phi \supset CI(u)[e/\underline{u}'][u/\underline{RQ}']}{\{\phi\}\ u : Q!e\ \{\psi\}_R}$$

$$\frac{-CI}{} \quad \frac{\forall u \in \overrightarrow{QR}.\ \phi \wedge CI(u)\ \supset\ \psi[\underline{u}'/x] \wedge \underline{QR}, \underline{u} \notin VV(\psi)}{\{\phi\}\ Q?x\ \{\psi^{\natural}\}_R}$$

$$\frac{\{\phi\}\ c\ \{\psi\}_R}{\{\phi\}\ R :: c\ \{\psi\}}$$

$$\frac{\{\phi_1\}\ p_1\ \{\psi_1\} \qquad \{\phi_2\}\ p_2\ \{\psi_2\}}{\{\phi_1 \wedge \phi_2\}\ p_1 \parallel p_2\ \{\psi_1 \wedge \psi_2\}}$$

$\boxed{\text{Logical rules}}$

$$\frac{\phi \supset \phi' \qquad \{\phi'\}\ c\ \{\psi'\} \qquad \psi' \supset \psi}{\{\phi\}\ c\ \{\psi\}} \qquad\qquad \frac{\phi \supset \phi' \qquad \{\phi'\}\ p\ \{\psi'\} \qquad \psi' \supset \psi}{\{\phi\}\ p\ \{\psi\}}$$

Figure 5–7: A Hoare Logic for the partial correctness of CSP communication invariant, we need values for the temporary new view $\underline{u}'$ and new counter $\underline{RQ}'$. But since we are sending the message, we know both these values: e and u respectively. Thus we show

$$\phi \supset CI(u)[e/\underline{u}'][u/\underline{RQ}']$$

The input rule is more complex because we do not know the values of the new view and new counter. These have to be supplied by the communication invariant. However, we do not know which communication invariant gives the right answers. Therefore for soundness, we have to check that the postcondition follows from the conjunction of the precondition and each communication invariant. This would be restrictive were it not for the counters. Suppose the communication is from R to Q. The precondition to the input rule should know what the current value of the counter $\underline{RQ}$ is. Then every communication invariant $CI(u)$ which asserts a different value for this counter will contradict the precondition, from which anything (in particular the desired postcondition) follows. The correct communication invariant will entail the desired postcondition naturally. The idea is related to Lamport's *at* predicates [LS84].

Once again, we have to fiddle with the primed views and counters. For each viewable location u we show two things. First that

$$\phi \wedge CI(u) \ \supset \ \psi[\underline{u}'/x]$$

which allows the postcondition to contain references to the variable x which receives the information (in particular it allows us to prove the important fact $x = \underline{u}'$ trivially). Second that

$$\phi \wedge CI(u) \ \supset \ \underline{RQ}, \underline{u} \notin VV(\psi)$$

which is really a condition on the kinds of postcondition we allow. Essentially, this ensures that when we deprime ψ to adopt the new values of the view and counter we do not conflict with the old values.

5.3.2 An example partial correctness proof

In this section, we show SP to be partially correct using our Hoare Logic. We simply prove the result: it will be appraised in section 5.3.4. The bulk of the proof concerns the deduction of p_1, and we need an auxiliary variable T' in p_1 that represents p_1's view of T in p_2. To keep T' up to date we need auxiliary communications of T. The bodies of the two processes become:

$$
\begin{array}{ll}
Q_1 :: & Q_2?T'; \\
& mx := \max(S); \\
& Q_2!mx; \quad S := S \setminus \{mx\}; \\
& Q_2?x; \quad S := S \cup \{x\}; \\
& Q_2?T'; \\
& mx := \max(S); \\
& \texttt{do} \ \ mx > x \Rightarrow \\
& \qquad Q_2!mx; \quad S := S \setminus \{mx\}; \\
& \qquad Q_2?x; \quad S := S \cup \{x\}; \\
& \qquad Q_2?T'; \\
& \qquad mx := \max(S); \\
& \texttt{od}
\end{array}
\qquad
\begin{array}{ll}
\text{and } Q_2 :: & Q_1!T; \\
& Q_1?y; \quad T := T \cup \{y\}; \\
& mn := \min(T); \\
& Q_1!mn; \quad T := T \setminus \{mn\}; \\
& Q_1!T; \\
& \texttt{do} \ \ \text{true} \Rightarrow \\
& \qquad Q_1?y; \quad T := T \cup \{y\}; \\
& \qquad mn := \min(T); \\
& \qquad Q_1!mn; \quad T := T \setminus \{mn\}; \\
& \qquad Q_1!T; \\
& \texttt{od}
\end{array}
$$

We label the output occurrences $Q_2!mx$ in p_1, u; the two occurrences of $Q_1!mn$ in p_2, w_1; and the three occurrences of $Q_1!T$ in p_2, w_2. The communication invariants are

$$\begin{array}{ll}
CI(u) = & \underline{Q_1Q_2} \in \{\perp, u\} \ \wedge\ \underline{Q_1Q_2}' = u \\
CI(w_1) = & \underline{Q_2Q_1} = w_2 \ \wedge\ \underline{Q_2Q_1}' = w_1 \ \wedge \quad \underline{w_1}' \le \underline{u} \\
CI(w_2) = & \underline{Q_2Q_1} \in \{\perp, w_1\} \ \wedge \underline{Q_2Q_1}' = w_2 \ \wedge (\underline{Q_2Q_1} = \perp \supset \underline{w_2}' = T_0) \\
& \qquad\qquad\qquad\qquad\qquad\ \wedge (\underline{Q_2Q_1} = w_1 \supset \underline{w_2}' = \underline{w_2} \cup \{\underline{u}\} \setminus \{\underline{w_1}\})
\end{array}$$

The abbreviated proof outline for the modified forms of p_1 and p_2 can be found in figures 5–8 and 5–9 respectively. Applying the parallel composition and consequence rules, we get

$$\begin{array}{c}
\{|S| = n_1 > 0,\ |T| = n_2 > 0,\ S = S_0,\ T = T_0,\ S \cap T = \emptyset\} \\
SP \\
\{S \cap T = \emptyset,\ S \cup T = S_0 \cup T_0,\ \max(S) < \min(T)\}
\end{array}$$

Which is the desired result.

5.3.3 Soundness of the Hoare Logic

The above proof only works if the proof rules are valid — i.e, if they make sound inferences from sound premises. [Sti88,Apt83] prove validity for Hoare Logics of concurrent languages using a transition-style operational semantics. [ZdRvEB85] proves validity for their Hoare Logic of DNP (a CSP variant) with respect to a denotational semantics of process traces. In this section, we sketch a proof of the validity of our Hoare Logic with respect to a modification of our evaluation semantics $\mathcal{CSP}$.

The meaning of programs and all that

The soundness proof is going to be split into two: one half for the sequential component and one half for the concurrent component. In this subsection we modify $\mathcal{CSP}$ to incorporate a semantic account of views and counters. We also define some auxiliary notations and definitions including an important *interference-freedom* property of deductions. Last we define the meaning of a program to be the set of all interference-free deductions of the program.

Views and Counters in $\mathcal{CSP}$ The required modification to $\mathcal{CSP}$ is slight. It consists of two changes. In the following I shall use u, w to range over occurrences of output commands, and I shall write $u : Q!e$ if $Q!e$ is occurrence u in the overall program. Let us extend the function vw to range over commands and programs: let $vw(c)$ (or $vw(p)$) return the set of views of output occurrences in c (or p). Let $VV = FV \oplus vw$ as before.

First, we replace the rules for communication with the following:

$$\frac{R : e \to v \quad \blacktriangleright \quad R : out(Q, v, u, \sigma)}{R : u : Q!e} \qquad\qquad \frac{R : in(Q, x, v, u, \sigma)}{R : Q?x}$$

$$\frac{}{R : out(Q, v, u, \sigma)} \ \stackrel{out(R,Q,v,u,\sigma)}{\overline{\qquad\quad}} \ \frac{}{\sigma \rightsquigarrow \sigma[v/x][v/\underline{u}][u/\underline{RQ}]} \ \stackrel{in(R,Q,x,v,u,\sigma)}{\overline{\qquad\quad}} \ \frac{}{Q : in(R, x, v, u, \sigma)}$$

$$\{|S| = n_1 > 0,\ S = S_0,\ S \cap T' = \emptyset,\ T' = T_0,\ \underline{Q_2Q_1} = \underline{Q_1Q_2} = \perp\}$$

$$\boxed{Q_1 ::} \quad \boxed{Q_2?T';} \qquad \{\ldots \underline{w_2} = T_0,\ \underline{Q_2Q_1} = w_2\}$$

$$\boxed{mx := \max(S);} \qquad \{\ldots mx = max(S)\}$$

$$u : \boxed{Q_2!mx;} \qquad \{\ldots mx = \underline{u},\ \underline{Q_1Q_2} = u\}$$

$$\boxed{S := S \setminus \{mx\};}$$

$$\{\ldots S \neq S_0,\ S \cap (T_0 \cup \{\underline{u}\}) = \emptyset,\ S \cup (T_0 \cup \{\underline{u}\}) = S_0 \cup T_0\}$$

$$\boxed{Q_2?x;} \qquad \{\ldots x = \underline{w_1} \leq \underline{u},\ \underline{Q_2Q_1} = w_1\}$$

$$\boxed{S := S \cup \{x\};}$$

$$\{\ldots S \cap (T_0 \cup \{\underline{u}\} \setminus \{\underline{w_1}\}) = \emptyset,\ S \cup (T_0 \cup \{\underline{u}\} \setminus \{\underline{w_1}\}) = S_0 \cup T_0\}$$

$$\boxed{Q_2?T';}$$

$$\{\ldots\ T' = \underline{w_2},\ S \cap T' = \emptyset,\ S \cup T' = S_0 \cup T_0,\ \underline{Q_2Q_1} = w_2\}$$

$$\boxed{mx := \max(S);}$$

$$\boxed{\texttt{do}\ \ mx > x \Rightarrow} \qquad \{LI_1\}$$

$$u : \boxed{Q_2!mx;} \ \boxed{S := S \setminus \{mx\};}$$

$$\{\ldots, mx = \underline{u},\ \underline{Q_1Q_2} = u,$$
$$S \cap (T' \cup \{\underline{u}\}) = \emptyset,\ S \cup (T' \cup \{\underline{u}\}) = S_0 \cup T_0\}$$

$$\boxed{Q_2?x;} \ \boxed{S := S \cup \{x\};}$$

$$\{\ldots x = \underline{w_1} \leq \underline{u},\ \underline{Q_2Q_1} = w_1,$$
$$S \cap (T' \cup \{\underline{u}\} \setminus \{\underline{w_1}\}) = \emptyset,$$
$$S \cup (T' \cup \{\underline{u}\} \setminus \{\underline{w_1}\}) = S_0 \cup T_0\}$$

$$\boxed{Q_2?T';}$$

$$\{\ldots\ T' = \underline{w_2},\ S \cap T' = \emptyset,\ S \cup T' = S_0 \cup T_0,$$
$$\underline{Q_2Q_1} = w_2\}$$

$$\boxed{mx := \max(S);} \qquad \{LI_1\}$$

$$\boxed{\texttt{od}}$$

$$\{|S| = n_1 > 0,\ S \cap \underline{w_2} = \emptyset,\ S \cup \underline{w_2} = S_0 \cup T_0,\ \max(S) = \underline{w_1},\ \ldots\}$$

Where LI_1 is the loop invariant:

$$LI_1 : \quad |S| = n_1 > 0,\ \ T' = \underline{w_2},\ \ x = \underline{w_1},\ \ \underline{Q_2Q_1} = w_2,$$
$$S \cap T' = \emptyset,\ \ S \cup T' = S_0 \cup T_0,\ \ \max(S) = mx \geq x$$

Figure 5–8: An abbreviated proofoutline for p_1

$$\{|T| = n_2 > 0,\ T = T_0,\ \underline{Q_1Q_2} = \underline{Q_2Q_1} = \bot\}$$

$$
\begin{array}{llll}
\boxed{Q_2 ::} & w_2 : & \boxed{Q_1!T;} & \{\ldots \underline{Q_2Q_1} = w_2, \underline{w_2} = T_0\} \\
 & & \boxed{Q_1?y;} & \{\ldots \underline{Q_1Q_2} = u, y = \underline{u}\} \\
 & & \boxed{T := T \cup \{y\};} & \{\ldots T = \underline{w_2} \cup \{\underline{u}\},\ |T| = n_2 + 1\} \\
 & & \boxed{mn := \min(T);} & \{\ldots mn = \min(\underline{w_2} \cup \underline{u})\} \\
 & w_1 : & \boxed{Q_1!mn;} & \{\ldots \underline{w_1} \leq \underline{u}, \underline{Q_2Q_1} = w_1\} \\
 & & \boxed{T := T \setminus \{mn\};} & \{\ldots T = (\underline{w_2} \cup \{\underline{u}) \setminus \underline{w_1}\}, \underline{w_1} < \min(T)\} \\
 & w_2 : & \boxed{Q_1!T;} & \{\ldots T = \underline{w_2},\ \underline{Q_2Q_1} = w_2\} \\
 & & \boxed{\texttt{do}\ \ \text{true} \Rightarrow} & \{LI_2\}
\end{array}
$$

$$
\begin{array}{lll}
 & \boxed{Q_1?y;} & \{\ldots \underline{Q_1Q_2} = u, y = \underline{u}\} \\
 & \boxed{T := T \cup \{y\};} & \{\ldots T = \underline{w_2} \cup \{\underline{u}\},\ |T| = n_2 + 1\} \\
 & \boxed{mn := \min(T);} & \{\ldots mn = \min(\underline{w_2} \cup \underline{u})\} \\
w_1 : & \boxed{Q_1!mn;} & \{\ldots \underline{w_1} \leq \underline{u}, \underline{Q_2Q_1} = w_1\} \\
 & \boxed{T := T \setminus \{mn\};} & \{\ldots T = (\underline{w_2} \cup \{\underline{u}) \setminus \underline{w_1}\}, \underline{w_1} < \min(T)\} \\
w_2 : & \boxed{Q_1!T;} & \{LI_2\}
\end{array}
$$

$$\boxed{\texttt{od}}$$

$$\{|T| = n_2,\ T = \underline{w_2},\ \min(T) > \underline{w_1}\}$$

Where LI_2 is the loop invariant

$$|T| = n_2,\quad T = \underline{w_2},\ \min(T) > \underline{w_1},\ \underline{Q_1Q_2} = u,\ \underline{Q_2Q_1} = w_2$$

Figure 5–9: An abbreviated proofoutline for p_2

There are three differences. First, the judgment $R : out(\cdots)$ has been extended to contain more variables. Second, we have a new judgment $R : in(\cdots)$ which also includes variables not in the input command. These extra variables are used when hypothesizing and guaranteeing communication invariants. The third change is that the store judgment has been extended to update views and counters and also altered to interact with both the input and output commands. (Thus in this system every interaction involves the store.) Formerly, it did not interact with the output atom. This is to facilitate reasoning about the shared views and counters.

The second change is to add a null store transition:

$$\overline{\sigma \rightsquigarrow \sigma}$$

This alteration is done purely to simplify the following definitions and proofs. For convenience, let us also call this modified system $\mathcal{CSP}$ (this brooks no confusion: we never refer to the old system in this section).

Assertions and stores Let $\models \subseteq Store \times L$ be a relation such that $\sigma \models \phi$ if the values of variables (or views) in σ satisfy the assertion ϕ (i.e., substituting the variables in ϕ with their values in σ returns a sentence in Tr_L). Technically, the relation should be subscripted by Tr_L, but we omit this to ease clutter.

Interference Freedom: programs and commands Intuitively, a deduction Π is interference free over a set of views (output occurrences) and variables X if no element of X is altered in the store by a process outside Π. That is, every update of an element of X must be sanctioned by an interaction link in Π. That is, the behaviour of the program fragment of Π with respect to that part of the store pertinent to X is fully specified by the deduction. For assignment and lookup, this means that there can be no *assign* or *lookup* dangling interactions in Π that mention variables in X. However, since both sides of a communication interact with the store, we can allow one of the interactions to dangle; though if Π contains the output command, then it cannot be the *out* link, and if it contains the input, it cannot be the *in* link.

Therefore we say a $\mathcal{CSP}$-deduction Π is *interference-free* with respect to a set of variables and output occurrences X (written $\Pi\checkmark X$) if for all $(A, \alpha) \in D(\Pi)$,

$$\begin{array}{ll} (i) & FV(\alpha) \cap X = \emptyset \\ (ii) & vw(\alpha) \cap X = \emptyset \text{ or } \alpha = \pm in(\cdots) \end{array}$$

Condition (i) asserts interference freedom on variables; condition (ii) states that either no relevant view occurs, or if it does, it occurs in an input link. (Note that if Π contains both input and output commands of an interaction then condition (i) will ensure that the input link is intact, and (ii) that the output link is intact.)

Proposition 5.0 *Let $\Sigma \vdash \sigma \rightsquigarrow \sigma'$ be interference-free with respect to X. Then for all ϕ such that $VV(\phi) \subseteq X$, if $\sigma \models \phi$ then $\sigma' \models \phi$.*

Proof: Straightforward induction on size(Σ). □

Accessible views and variables The assertions of a sequential process may refer to any variable in that process. We want to ensure that if a command does not mention certain variables or views, then their values will remain untouched. So, whenever we reason about the meaning of a command, we shall want to talk about those deductions that are interference-free with respect to the entire set of views and variables in the sequential process. We define

$$Acc(R,p) = \begin{cases} VV(c) & \text{if } p = R :: c \\ \emptyset & \text{if } p = Q :: c \text{ and } R \neq Q \\ Acc(R,p_1) \cup Acc(R,p_2) & \text{if } p = p_1 \parallel p_2 \end{cases}$$

For the set of views and variables accessible by process R in p.

The meaning of commands and programs The *meaning* of a command (program) will be the set of all its deductions which are interference-free over the variables accessible by the command (program). Thus the meaning of a command *c belonging to process R in program p* is defined to be the set

$$\mathcal{M}_{(R,p)}[c] = \{\Pi \in \mathbf{DQI}(\mathcal{CSP}) \mid \Pi \vdash R : c, \sigma \rightsquigarrow \sigma' \text{ and } \Pi \checkmark Acc(R,p)\}$$

We really need to restrict attention to the deductions which are interference-free on $Acc(R,p)$. It is not enough to assert interference-freedom for the views and variables of the particular program fragment c, because the pre- and post-assertions of c may refer to any view or variable accessible to R.

The meaning of a program p

$$\mathcal{M}[p] = \{\Pi \in \mathbf{DQI}(\mathcal{CSP}) \mid \Pi \vdash p, \sigma \rightsquigarrow \sigma' \text{and } \Pi \checkmark VV(p)\}$$

The meaning of Hoare Triples is easily defined by

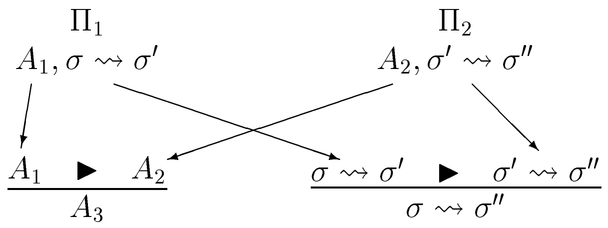
$$\begin{array}{l} \models_{(R,p)} \{\phi\}\ c\ \{\psi\} \text{ iff} \\ \qquad\qquad \text{for all } \Pi \in \mathcal{M}_{(R,p)}[c], \text{ if } \Pi \vdash R : c, \sigma \rightsquigarrow \sigma' \text{ and } \sigma \models \phi \text{ then } \sigma' \models \psi \\ \models \{\phi\}\ p\ \{\psi\} \text{ iff} \\ \qquad\qquad \text{for all } \Pi \in \mathcal{M}[p], \text{ if } \Pi \vdash p, \sigma \rightsquigarrow \sigma' \text{ and } \sigma \models \phi \text{ then } \sigma' \models \psi \end{array}$$

The meaning of a Hoare triple can be paraphrased "for every deduction fragment of c (or p) interference-free on every variable or view in c (or p), such that ϕ holds before executing it, ψ holds afterwards".

Notes on Communication Invariants In the following, let us write $\vdash^{CI}_{(R,p)}$ $\{\phi\}\ c\ \{\psi\}$ if the Hoare triple is provable in the above system, when c is a command occurrence of R in p, and the communication invariants are given by $CI : Views \rightarrow L$. Moreover, we say a set of deductions X is *CI-input correct* if for all $\Pi \in X$, for all occurrences of form $R : in(Q,x,v,u,\sigma) \in O(\Pi)$, $\sigma \models CI(u)[v/\underline{u}'][u/\underline{QR}']$. Similarly, we say a set X of deductions is *CI-output correct* if for all $\Pi \in X$, for all occurrences of form $R : out(Q,x,v,u,\sigma) \in O(\Pi)$, $\sigma \models CI(u)[v/\underline{u}'][u/\underline{RQ}']$.

A convenient notation and related results

To aid the following proof, we use the following notation. Let $\Pi_1 \vdash A_1, \sigma \rightsquigarrow \sigma'$ and $\Pi_2 \vdash A_2, \sigma' \rightsquigarrow \sigma''$. Then we write $\Pi = \Pi_1 \blacktriangleright \Pi_2$ if Π is the deduction



Note that in $\mathcal{CSP}$, A_3 is uniquely determined by A_1 and A_2. We extend the notation: if $\Sigma_1 \vdash \sigma \rightsquigarrow \sigma'$ and $\Sigma_2 \vdash \sigma' \rightsquigarrow \sigma''$ we write $\Sigma_1 \blacktriangleright \Sigma_2$ for the deduction of $\sigma \rightsquigarrow \sigma''$ with children Σ_1 and Σ_2 in sequence.

Proposition 5.1 (Store Reshuffling) *Let $\Sigma \in \mathbf{DQI}(\mathcal{CSP})$ be such that $\Sigma \vdash \sigma \rightsquigarrow \sigma'$. Let $X \subseteq D(\Sigma)$ be upwards closed with respect to $\lesssim_\Sigma$. Then there exists a store σ'' and $\mathcal{CSP}$-deductions $\Pi_1 \vdash \sigma \rightsquigarrow \sigma''$ and $\Pi_2 \vdash \sigma'' \rightsquigarrow \sigma'$ such that $X = D(\Pi_2)$ and $\Sigma \simeq (\Pi_1 \blacktriangleright \Pi_2)$.*

Proof: By induction on $n = |X| \times \mathrm{size}(\Sigma)$. **Case $n = 0$:** Two cases: either $|X| = \emptyset$ or $\mathrm{size}(\Sigma) = 0$. We set

$$\Pi_1 = \Sigma \qquad\qquad \Pi_1 = \overline{\sigma \rightsquigarrow \sigma}$$
$$\Pi_2 = \overline{\sigma' \rightsquigarrow \sigma'} \quad \text{and} \quad \Pi_2 = \Sigma$$

respectively. In both cases, the other conditions are satisfied trivially. **Case $n > 0$:** We have $\Sigma = (\Sigma_1 \blacktriangleright \Sigma_2)$ where $\Sigma_1 \vdash \sigma \rightsquigarrow \sigma''$ and $\Sigma_2 \vdash \sigma'' \rightsquigarrow \sigma'$. There are two cases. **First**, either $X \subseteq D(\Sigma_2)$, in which case by induction there exists Π_{21} and Π_{22} such that $\Sigma_2 \simeq (\Pi_{21} \blacktriangleright \Pi_{22})$ and $D(\Sigma_2) = D(\Pi_{22})$. Then we set $\Pi_1 = (\Sigma_1 \blacktriangleright \Pi_{21})$ and $\Pi_2 = \Pi_{22}$, and the conditions follow trivially.

Second, $X \cap D(\Sigma_1) \neq \emptyset$ (in which case $D(\Sigma_2) \subseteq X$ as X is upwards closed). Let $Y = X \cap D(\Sigma_1)$. Clearly $X = Y \cup D(\Sigma_2)$. By induction, there exists Π_{11} and Π_{12} such that $\Sigma_1 \simeq (\Pi_{11} \blacktriangleright \Pi_{12})$ and $Y = D(\Pi_{12})$. Then we set $\Pi_1 = \Pi_{11}$ and $\Pi_2 = (\Pi_{12} \blacktriangleright \Sigma_2)$, and the conditions follow trivially. □

Proposition 5.2 *Let $\Pi \vdash A_3, \sigma \rightsquigarrow \sigma'$ be a $\mathcal{CSP}$ deduction such that A_3 is introduced by a rule atom instance with premises $A_1 \blacktriangleright A_2$. Then there exist σ'' and $\mathcal{CSP}$-deductions $\Pi_1 \vdash A_1, \sigma \rightsquigarrow \sigma''$ and $\Pi_2 \vdash A_2, \sigma'' \rightsquigarrow \sigma'$ such that $\Pi = \Pi_1 \blacktriangleright \Pi_2$.*

Proof: (Sketch) essentially, the command judgment occurrence A_3 is deduced from the deduction concluding A_1 and A_2. We want to split the tree concluding $\sigma \rightsquigarrow \sigma'$ into two subtrees concluding $\sigma \rightsquigarrow \sigma''$ and $\sigma'' \rightsquigarrow \sigma'$, such that the first subtree interacts only with the tree concluding A_1 and the second interacts only with that concluding A_2. The precise details of how this is done are tedious.

Briefly: first we break every interaction link of Π. Second, we isolate the set of dangling interactions that used to interact with the tree concluding A_2. Then we take its upper closure (with respect to $<_\Pi$) and apply proposition 5.1 to build the two desired subtrees. Reassembly is easy because proposition 5.1 alters neither the set of dangling interactions above the store tree, nor the relative dependencies between the dangling interactions. Therefore, one can apply the interaction reflection theorem using the identity function to get the result. $\square$

Soundness for commands

Lemma 5.3(i) *If* $\vdash^{CI}_{(R,p)} \{\phi\}\ c\ \{\psi\}_R$ *and* $\mathcal{M}_{(R,p)}[c]$ *is CI-input correct then* $\models_{(R,p)} \{\phi\}\ c\ \{\psi\}_R$ *and* $\mathcal{M}_{(R,p)}[c]$ *is CI-output correct.*

Proof: By induction on the depth of inference of $\vdash \{\phi\}\ c\ \{\psi\}$. We shall only give the cases for input, output and sequential composition: all the rest follow by simpler arguments.

Case $\{\phi\}\ Q?x\ \{\psi\}_R$. Let $\Pi \in \mathcal{M}_{(R,p)}[Q?x]$ be such that $\Pi \vdash R : Q?x, \sigma \rightsquigarrow \sigma'$ and $\sigma \models \phi$.

The bottom-most command rule in Π will be the input rule. This depends on the premise $R : in(Q, x, v, u, \sigma_0)$ for some store σ_0. By CI-input correctness, $\sigma_0 \models CI(u)[v/\underline{u}'][u/\underline{QR}']$ ($\spadesuit$).

Now, since Π is interference-free on all variables and views accessible to R, this means that any writes to x in the store must be marked by an interaction link between the appropriate store transition formula occurrence and command evaluation formula occurrence. But in Π, the only formula occurrence the store can interact with is $R : in(Q, x, v, u, \sigma_0)$. Let

$$\overset{out(Q,R,v,u,\sigma_0)}{\overline{\qquad\quad}}\ \ \frac{}{\sigma_0 \rightsquigarrow \sigma_0[v/x][v/\underline{u}][u/\underline{QR}]} \qquad \overset{in(Q,R,x,v,u,\sigma_0)}{\overline{\qquad\quad}}\ \ \frac{}{R : in(Q, x, v, u, \sigma_0)}$$

for some σ_0 be this single instance of the input rule in Π. By interference freedom, we know that this rule instance is the only one that can change an accessible variable or view. Therefore, since $\sigma \models \phi$ so also $\sigma_0 \models \phi$ (proposition 5.0). By $\spadesuit$, we get $\sigma_0 \models \phi \wedge CI(u)[v/\underline{u}']$. Together with the premise to the input rule, we get $\sigma_0 \models (\psi[v/x])[v/\underline{u}'][u/\underline{QR}']$. Since $u, QR \notin VV(\psi)$, we get $\sigma_0 \models (\psi[v/x])[v/\underline{u}'][u/\underline{QR}']^\natural$, i.e., $\sigma_0 \models (\psi^\natural)[v/x][v/\underline{u}][u/\underline{QR}]$. Therefore $\sigma_0[v/x][v/\underline{u}][u/\underline{QR}] \models \psi^\natural$. Last, by interference-freedom (and proposition 5.0) we know that no other variable or view accessible to R in p is altered in Π. Therefore $\sigma' \models \psi^\natural$.

Case $\{\phi\}\ u : Q!e\ \{\psi\}$. Let $\Pi \in \mathcal{M}_{(R,p)}[Q!e]$. Then $\Pi \vdash_{(R,p)} R : Q!e, \sigma \rightsquigarrow \sigma''$ and $\sigma \models \phi$. From the premises of the Hoare rule, $\sigma \models \psi[e/\underline{u}][u/\underline{RQ}]$ and also (by modus ponens), $\sigma \models CI(u)[e/\underline{u}'][u/\underline{RQ}']$. Now $R : Q!e$ is inferred from $R : e \to v$ and $R : out(Q, v, u, \sigma_0)$ for some σ_0 (sequenced in that order). Therefore, $\sigma \models \psi[v/\underline{u}][u/\underline{RQ}]$ and $\sigma \models CI(u)[v/\underline{u}'][u/\underline{RQ}']$..

By interference freedom (and proposition 5.0), we know that Π must contain an instance of the rule fragment

$$\frac{}{R : out(Q, v, u, \sigma_0)} \quad \overline{}^{\,out(R,Q,v,u,\sigma_0)} \quad \frac{}{\sigma_0 \rightsquigarrow \sigma_0[v/x][v/\underline{u}][u/\underline{RQ}]} \quad \overline{}^{\,in(R,Q,x,v,u,\sigma_0)}$$

Moreover, it also follows from interference freedom that since $R : Q!e$ does not write to any variables of R, the accessible variables will remain untouched. The only accessible view altered is $\underline{u}$, and we know how that is altered. Therefore, for every σ_0 mentioned in Π, since $\sigma_0 \models \phi$, we know $\sigma_0 \models CI(u)[v/\underline{u}'][u/\underline{RQ}']$. Thus Π is CI-output correct.

We also know $\sigma_0 \models \phi$, and so also trivially $\sigma_0[v/\underline{u}][u/\underline{QR}] \models \psi$, and so also by interference freedom (and proposition 5.0), $\sigma' \models \psi$.

Case $\{\phi\}\ c_1; c_2\ \{\psi\}$. If $\vdash_{(R,p)} \{\phi\}\ c_1; c_2\ \{\psi\}$ then it was inferred from $\vdash_{(R,p)} \{\phi\}\ c_1\ \{\chi\}$ and $\vdash_{(R,p)} \{\chi\}\ c_2\ \{\psi\}$ for some cut-formula χ.

Let $\Pi \in \mathcal{M}_{(R,p)}[c_1; c_2]$ be such that $\Pi \vdash_{(R,p)} R : c_1; c_2, \sigma \rightsquigarrow \sigma'$ and $\sigma \models \phi$. By proposition 5.2, we can find σ'' and two deductions $\Pi_i \in \mathcal{M}_{(R,p)}[c_i]$ for $i = 1, 2$, such that $\Pi_1 \vdash R : c_1, \sigma \rightsquigarrow \sigma''$ and $\Pi_2 \vdash R : c_2, \sigma'' \rightsquigarrow \sigma'$ such that $\Pi = \Pi_1 \blacktriangleright \Pi_2$. Now, if $\mathcal{M}_{(R,p)}[c_1; c_2]$ is CI-input correct, so must $\mathcal{M}_{(R,p)}[c_i]$ be, since the input occurrences of $c_1; c_2$ is the union of the input occurrences of c_1 and c_2.

Therefore, by induction, $\models_{(R,p)} \{\phi\}\ c_1\ \{\chi\}$ and $\models_{(R,p)} \{\chi\}\ c_2\ \{\psi\}$. Therefore, whenever $\sigma \models \phi$, we get $\sigma'' \models \chi$ and so $\sigma' \models \psi$. Similarly, $\mathcal{M}_{(R,p)}[c_1; c_2]$ will be CI-output correct because $\mathcal{M}_{(R,p)}[c_1]$ and $\mathcal{M}_{(R,p)}[c_2]$ are. □

Soundness of Programs

Lemma 5.3(ii) *If $\vdash \{\phi\}\ p\ \{\psi\}$ and $\mathcal{M}[p]$ is CI-input correct then $\models \{\phi\}\ p\ \{\psi\}$ and $\mathcal{M}[p]$ is CI-output correct.*

Proof: by induction on p. **Case $p = R :: c$:** the result follows from lemma 5.3(i). **Case $p = p_1 \parallel p_2$:** let $\Pi \in \mathcal{M}[p_1 \parallel p_2]$ be such that $\Pi \vdash p_1 \parallel p_2, \sigma \rightsquigarrow \sigma'$ and $\sigma \models \phi_1 \wedge \phi_2$. We remove the bottom parallel composition rule to get $\Pi_0 \vdash p_1, p_2, \sigma \rightsquigarrow \sigma'$. Now, let Π_1, Π_2 be the fragments of Π_0 such that $\Pi_i \vdash p_1, \sigma \rightsquigarrow \sigma'$ and $\Pi_i \checkmark VV(p_i)$, for $i = 1, 2$. Thus, the only interaction links broken to get Π_i are the links between the store trees and the tree concluding p_{3-i} (for $i = 1, 2$).

Now we get $\Pi_i \in \mathcal{M}_{p'}[p_i]$ $(i = 1, 2)$. Therefore, by induction, we get $\models \{\phi_i\}\ p_i\ \{\psi_i\}$ (for $i = 1, 2$). That is, if $\sigma \models \phi_i$ then $\sigma' \models \psi_i$. Since $\sigma \models \phi_1 \wedge \phi_2$, we get $\sigma' \models \psi_1 \wedge \psi_2$. □

The previous lemma shows that the deductive system is sound for programs whenever the meaning of a command is CI-input correct. This condition can only be guaranteed in the context of the entire program. To complete the soundness proof, we simply have to show that CI-input correctness is guaranteed by CI-output correctness.

We say $p \in Prog$ is *whole* if every process identifier mentioned in p is bound to a process in p. Then interference-freedom implies that for whole p, $\mathcal{M}[p] \subseteq \mathbf{QI}(\mathcal{CSP})$ (essentially because every output occurrence viewed in p will occur in p.)

Proposition 5.3 *If* $\vdash \{\phi\}\ p\ \{\psi\}$ *then* $\models \{\phi\}\ p\ \{\psi\}$ *when* p *is whole.*

Proof: Suppose $\vdash \{\phi\}\ p\ \{\psi\}$ where p is whole. By lemma 5.3(ii), if $\mathcal{M}[p]$ is CI-input correct, then $\models \{\phi\}\ p\ \{\psi\}$ and $\mathcal{M}[p]$ is CI-output correct.

Now, since $\mathcal{M}[p] \subseteq \mathbf{QI}(\mathcal{CSP})$, if we break every *in*- and *out*-labelled interaction of every deduction in $\mathcal{M}[p]$ to get X, we have $X \subseteq \mathbf{DQI}(\mathcal{CSP})$. Then input- and output-correctness correspond to the following interaction hypotheses and guarantees:

(1)	$+out(R, Q, v, u, \sigma) : \sigma \models CI(u)[v/\underline{u}'][u/\overline{RQ}']$	guaranteed
(2)	$-out(R, Q, v, u, \sigma) : \sigma \models CI(u)[v/\underline{u}'][u/\overline{RQ}']$	hypothesized
(3)	$+in(R, Q, v, u, \sigma) : \sigma \models CI(u)[v/\underline{u}'][u/\underline{RQ}']$	guaranteed
(4)	$-in(R, Q, v, u, \sigma) : \sigma \models CI(u)[v/\underline{u}'][u/\underline{RQ}']$	hypothesized

(1) is guaranteed by output correctness. (3) is guaranteed under the hypothesis of (2). Hypothesizing (4) allows us to prove the first half of input-correctess. Thus we have the following result: for all $\Pi \in \mathbf{DQI}(\mathcal{CSP})$, under the above interaction hypotheses, if $\Pi \in X$ and $\vdash^{CI} \{\phi\}\ p\ \{\psi\}$ then if $\Pi \vdash p, \sigma \rightsquigarrow \sigma'$ and $\sigma \models \phi$ then $\sigma' \models \psi$ and the above guarantees are guaranteed. By proof assembly, we can discharge the interaction hypotheses and guarantees to get soundness. □

5.3.4 Appraisal

The first thing to notice is that the actual proof system for the Hoare Logic did not make much use of the dangling interactions. They only seemed to remove the need to propagate the communication invariant everywhere. This is reminiscent of the convention employed in [ZdBdR83] to avoid propagating their assumption and commitment conditions.

The burden of proof

The most obvious thing about the partial correctness proof of section 5.3.2 was that although there were two processes which performed roughly equal amounts of work, the burden of the proof fell on one process (see figure 5–8). The role of the other process was to maintain communication invariants (and to show that $\underline{w_1} < \max(T)$).

Thus the proof technique seems to enforce a master-slave paradigm on the processes: the bulk of the proof is performed in the master, and the slaves simply fill in the appropriate gaps. This was acceptable for our example SP, but it is more difficult for programs such as the *distributed greatest common divisor* program DGCD given in [AO91]. Here we wish to find the common divisor of n numbers.

We build a ring of identical processes, each of which owns one of the variables. At each step of the algorithm each process either sends its value to the next processor or accepts a value of the previous one. It then adjusts its value accordingly.

To prove this correct, I had to postulate a special *observer* process which at each step read the values from each processor and showed that the set of all the values satisfied a global invariant condition. (That is, at each step, the gcd of all the values equalled the gcd of the original values.)

The most obvious problem is that if we modify the communicative behaviour of the processes to continually pass around important values, how do we know if the original program is correct? In our example, it should be clear that the auxiliary communications do not upset the original communications, but this might not always be so evident.

I believe that with more work this problem could be circumvented in a straightforward manner. One solution might be to "piggy-back" the auxiliary communications on top of existing communications. One would send tuples of views, each of which would have to be broken into their consituents by the communication invariant at the input commands.

More speculatively, we might rectify the anomalous distribution of work in a proof by introducing global communication invariants. The idea here would be to introduce a special "broadcasting" primitive (purely for auxiliary communications) to access the global invariant. The idea would be that each view of a broadcasting location would be globally accessible. Associated to each broadcast view would be a global invariant. Each broadcast would have to show that it maintained the invariant. It is unlike the input and output rules in that the global views would be accessible everywhere in the proof outline. That is, the global invariant would have to be propagated everywhere. However, at the moment, this is an idea I have not pursued very far.

Of this and other Hoare Logics

Another point of interest is that our Hoare Logic for CSP is very simple. It is simpler than the systems of Apt *et al.* [AFdR80] and Levin and Gries [LG81] in that it is a compositional, one-level system. It is more compositional than that of Lamport *et al.* [LS84], in that our invariants are decomposed at the parallel rule. Last, it is simpler than those of Misra and Chandy [MC81] and Zwiers *et al.* [ZdBdR83,ZdRvEB85] because it does not use trace variables. However, I think we could easily maintain trace variables manually for those programs whose correctness proof requires them (e.g., *rebound sorting* [Oss83, §4]).

The soundness proof

The soundness proof was very simple. It required a minimal number of concepts. Structurally, it was just an assembly of a simple inductive proof. However, somewhat ironically, the proof could have been even simpler if we had not fragmented the store and program judgments. Since CSP processes do not interfere via shared variables, we could have used traditional natural semantics judgments of form $R : c, \sigma \Rightarrow \sigma'$ and $p, \sigma \Rightarrow \sigma'$. Most of the rules would have been straightforward — the only two interesting cases would have been the communication and parallel composition rules:

$$\frac{}{R : Q!v, \sigma_1 \Rightarrow \sigma_1[v/\underline{u}][u/\underline{RQ}]} \quad \frac{}{}\textit{comm} \quad \frac{}{Q : R?x, \sigma_2 \Rightarrow \sigma_2[v/x][v/\underline{u}][u/\underline{RQ}]}$$

$$\frac{p_1, (\sigma \restriction VV(p_1)) \Rightarrow \sigma_1 \qquad p_2, (\sigma \restriction VV(p_2)) \Rightarrow \sigma_2}{p_1 \parallel p_2, \sigma \Rightarrow \sigma_1 \oplus \sigma_2}$$

Where $\sigma \restriction X$ is an operation to restrict the domain of the store, and $\sigma_1 \oplus \sigma_2$ is an operation to overwrite σ_1 with σ_2. These would have to be defined algebraically.

In this setting we would not have needed to prove propositions 5.1 and 5.2. The fact that at least the first proof is modular is irrelevant. Thus we have seen a case where fragmentation is not always useful. However, the technique we have seen would be applicable for soundness proofs of Hoare Logics for concurrent languages with shared variables.

Proving Deadlock-freeness

One important property we should like to prove about CSP programs is that processes are deadlock-free. I do not currently know how we could prove this directly within our Hoare Logic, since it is a property of the system as a whole. [AFdR80] prove deadlock-freeness by extending their two-level proof system to reason about the set of all blocked states of a program. Deadlock-freeness follows when every blocked states fails a global invariant test. It is not clear how to do this compositionally: [MC81] say it is impossible.

Total Correctness

However, given such a technique we can alter the Hoare Logic in a standard way to prove the total correctness of CSP programs. Following [Apt83], we say a program is *totally correct* if it is partially correct, terminates, does not abort, and is deadlock-free. We know how to obtain partial correctness proofs. Termination (or more properly, non-nontermination) can be achieved with a modification of the repetitive rule:

$$\frac{\phi(0) \supset \neg Gu(g) \qquad \phi(n+1) \supset Gu(g) \qquad \{\phi(n+1)\}\ g\ \{\phi(n)\}_R}{\{\exists n.\phi(n)\}\ \mathtt{do}\ g\ \mathtt{od}\ \{\phi(0)\}_R}$$

Non-abortion can be proved by altering the alternative composition rule:

$$\frac{\phi \supset Gu(g) \qquad \{\phi\}\ g\ \{\psi\}_R}{\{\phi\}\ \mathtt{if}\ g\ \mathtt{fi}\ \{\psi\}_R}$$

Relative completeness (in the sense of Cook)

I do not know if our Hoare Logic is complete relative to the set of true sentences of the assertion language L. If it is not, I would be surprised if we could not rectify the situation by adding some extra logical rules and rules for manipulating auxiliary variables — every other Hoare Logic seems to have them.

I think such a completeness proof would follow using the standard technique of *strongest postconditions*. The strongest postcondition of a program and precondition $SP(p, \phi)$ is such that $\sigma \models SP(p, \phi)$ if and only if

$$\exists \sigma'.\sigma' \models \phi \ \wedge \ \exists \Pi \in \mathcal{M}[p].\Pi \vdash p, \sigma' \rightsquigarrow \sigma$$

This would require an assertion language sufficiently expressive to code the semantics. (For instance the language of Peano arithmetic.) Then the completeness proof would proceed in two steps. First, to show that if $\models \{\phi\}\ p\ \{\psi\}$ then $SP(p, \phi) \supset \psi$, and second to show that for all p and ϕ, $\vdash \{\phi\}\ p\ \{SP(p, \phi)\}$. This should follow by induction on the structure of programs.

5.4 Chapter Summary

This chapter uses interacting deductions to give a semantics to CSP. The point was to show that we could extend our techniques to less trivial languages, and to more usable proofs. We considered how to model various different concurrent features in the evaluation semantics setting, mostly with success. One interesting feature was that we could give propagation-free rules for abortion.

We saw in section 5.2.4 that evaluation semantics is not well suited to specifying pre-emptive primitives. Also, we saw that it is not well suited to features that operate on a snapshot of the entire system of processes (e.g., broadcasting and global abort).

An important lesson was that modularity may sometimes be achieved at the expense of other desirable properties of semantics. We saw this twice. First, in section 5.2.6, we gave a modular semantics for recursive procedures. (That is, we added the rules for procedures to CSP without changing any of the original rules.) The result was a mess. One feels that environments ought to be bound closely to judgments. Second, in the soundness proof of the Hoare Logic, because we separated the store judgments from the program judgments, we had to do more work, both in defining the notion of interference-freedom, and also in having to prove proposition 5.1. Neither would have had to have been done if we had not fragmented rules. (Of course this example is mitigated by the fact that the fragmented stores could be useful in other contexts.)

Another lesson, gleaned from section 5.2 was that the scoping condition was ubiquitous.

Finally, in section 5.3, we saw a very simple Hoare Logic for the partial correctness of CSP. This in itself is an achievement. (There were some difficulties, but future work should solve them.) One nice feature was that despite the above reservations, its soundness proof was very easy using the evaluation semantics and proof assembly. In fact the above reservations suggest that the proof could be made even easier.

Chapter 6

The process calculus interpretation

So far I have only given a syntactic account of interacting deductions. Now I intend to give a semantic account. This account is intended to explain why we can call DQI-systems like $\mathcal{CSP}$ *operational semantics*. This chapter shows that the computational content of an interacting deduction is a set of traces of some CCS-like process. Using Heyting's idea that the meaning of a formula is the set of its deductions, the meaning of an "evaluation" judgment like "$R : c$" is a set of evaluations of a process corresponding to $R : c$. In this sense $\mathcal{CSP}$ is an operational semantics.

On a more practical level, the result indicates that in principle we can use process calculus simulators (e.g., languages like Pict [PT95], or tools such as the Concurrency Workbench [CPS89,CWB] or similar [IP91], perhaps with some modification) to build prototype interpreters for languages from semantic definitions. This is similar to the way that Typol systems use Prolog to build prototypers for Natural Semantics definitions [Des84].

In fact process evaluation is a semidecision procedure which is a natural analogue to the tableaux method. Another corollary of this result is that we can obtain suitable models for deductive systems.

We can also go in the reverse direction, and show that every evaluation of a CCS process corresponds to the deduction of a certain formula. A corollary is that there is no procedure that decides if a set of occurrences is deducible in an arbitrary DQI-system.

The heart of this chapter is section 6.2 which contains both the interpretation and its soundness and completeness results. Section 6.1 describes the relevant preliminaries, and section 6.3 explores the aforementioned consequences.

6.1 Preliminaries

6.1.1 On formula occurrences

The heart of our semantic theorem depends upon the close association of a formula in a deductive system with a process constant defined in some corresponding system of equations. A technicality is that deductions concern formula occurrences. There is no formal notion of constant occurrences in process calculus, because one can easily distinguish different occurrences of a constant in a term according to their positions in the (abstract syntax tree of the) term.

There is no notion of position in a set — that is, sets are not rigid objects. Therefore, in order to map sets of formula occurrences into process terms which are rigid, I introduce a total order over the set of all formula occurrences of a language that will distinguish one enumeration of any set of occurrences.

Let $\trianglelefteq_{\mathcal{L}}$ be a total order over the formulae in $\mathcal{L}$. (For brevity we omit the subscript $\mathcal{L}$: it will always be clear which language is intended.) When our language is defined algebraically from a finite number of constructors — as one would expect a language of operational judgments to be — some kind of gödelnumbering [Men79] of the terms and formulae will provide this total order. A crucial point is that the gödelnumbering function is primitive recursive, which means that the total order over formulae is decidable.

I lift $\trianglelefteq$ to be an order on formula occurrences and formula perspectives. In chapter 2 I said that a formula occurrence was a pair (A, n) where A is a formula and n is a natural number. Thus the total order $\trianglelefteq$ over occurrences is just the lexicographic order over pairs (A, n):

$$(A, n) \trianglelefteq (A', n') \text{ iff } A \trianglelefteq A' \text{ or } (A = A' \text{ and } n \leq n')$$

I write $A \triangleleft B$ if $A \trianglelefteq B$ and $A \neq B$.

The total order $\trianglelefteq$ over formula perspectives is obtained by ignoring the sign information:

$$(A, s) \trianglelefteq (A', s') \text{ iff } A \trianglelefteq A'$$

Thus both $+A \trianglelefteq -A$ and vice-versa. This property is required on page 136 when I define an order over $D(\Pi)$ from a history of Π. It is important that connectable dangling interactions are assigned equal position in that order: this is used in the proof of lemma X(i) (specifically, near the middle of page 138).

As a **convention**, unless otherwise stated, whenever I write $\Pi \vdash A_1, \ldots, A_n$ I also mean $A_1 \trianglelefteq \ldots \trianglelefteq A_n$. Also, when I write sets of premise occurrences $\{P_1, \ldots, P_n\}$ or perspectives $\{\alpha_1, \ldots, \alpha_n\}$ I shall also assume that they are ordered. This is unproblematic except when both $+A$ and $-A$ are perspectives dangling from the same occurrence. In that case, I order $+A$ before $-A$. (Such a case seems unnatural in practice, and it cannot occur in acyclic systems.)

6.1.2 The process calculus

For the interpretation we use CCS extended with sequential composition. In fact, we shall use the $Before$ operator defined by Milner [Mil89, ch 8]. For this to work, our interpretation must use his *well-terminating* processes. Throughout, I assume that the reader is familiar with the theory of CCS.

Syntax Like P, CCS is built over an action set Act. This is built from an infinite set Nam of names, ranged over by $a, b, c, \ldots$. $\overline{Nam}$ denotes the set of *co-names* of Nam (distinct from Nam), ranged over by $\bar{a}, \bar{b}, \bar{c}, \ldots$. The set of *labels* is $Lab = Nam \cup \overline{Nam}$, ranged over by l. I define *complementation* $\bar{\cdot} : Lab \to Lab$ by: $\bar{l} = \bar{a}$ if $l = a$ and $\bar{l} = a$ if $l = \bar{a}$. The action set $Act = Lab \cup \{\tau\}$ (and is ranged over by α) where $\tau \notin Lab$ is called the *silent action*.

Last, let C range over a countable set $Const$ of process constants. Then the syntax of CCS is given by

$$p ::= 0 \mid \alpha.p \mid p|p \mid p \setminus M \mid p[f] \mid \sum_{i \in I} p_i \mid C$$

For some index set I and function $f : Act \to Act$ such that for all $l \in Lab$, $f(\bar{l}) = \overline{f(l)}$ and $f(\tau) = \tau$. I use p and q to range over CCS processes.

Semantics The constructors have their usual meanings: inaction, prefixing, parallel composition, restriction, relabelling, summation and constants. Technically, the semantics is given by a labelled transition system $LI_\Delta = \langle \mathrm{CCS}, Act, \to, \Omega \rangle$ where $\Delta : Const \rightharpoonup \mathrm{CCS}$ gives the constant bindings and Ω is the least set such that

$$\begin{array}{cc} 0 \in \Omega & p \setminus M \in \Omega \text{ when } p \in \Omega \\ p|q \in \Omega \text{ when } p, q \in \Omega & p[f] \in \Omega \text{ when } p \in \Omega \\ \sum_{i \in I} p_i \in \Omega \text{ when every } p_i \in \Omega & \end{array}$$

And the labelled transition relation is defined by $(p, \alpha, q) \in \to$ iff $\mathcal{C}_\Delta \vdash p \xrightarrow{\alpha} q$, where the system $\mathcal{C}_\Delta = (\mathcal{L}, \mathcal{R})$ and $\mathcal{L}$ is just the language of labelled transition judgments and $\mathcal{R}$ consists of the following rules:

$$\alpha.p \xrightarrow{\alpha} p \qquad \frac{p \xrightarrow{\alpha} p'}{p|q \xrightarrow{\alpha} p'|q} \qquad \frac{q \xrightarrow{\alpha} q'}{p|q \xrightarrow{\alpha} p|q'}$$

$$\frac{p \xrightarrow{\alpha} p' \qquad q \xrightarrow{\bar{\alpha}} q'}{p|q \xrightarrow{\tau} p'|q'} \qquad \frac{p_j \xrightarrow{\alpha} p' \qquad j \in I}{\sum_{i \in I} p_i \xrightarrow{\alpha} p'} \qquad \frac{\Delta(C) \xrightarrow{\alpha} p'}{C \xrightarrow{\alpha} p'}$$

$$\frac{p \xrightarrow{\alpha} p' \qquad \alpha \notin M \cup \overline{M}}{p \setminus M \xrightarrow{\alpha} p' \setminus M} \qquad \frac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$$

I say a deduction sequence s is a sequence of $\mathcal{C}_\Delta$-deductions, and I write $s : p \xrightarrow{t} p'$ if $s = \Sigma_1 \cdot \ldots \cdot \Sigma_n$, and $\Sigma_1 \vdash p \xrightarrow{\alpha_1} p_1$; for all $1 < i \leq n$, $\Sigma_i \vdash p_{i-1} \xrightarrow{\alpha_i} p_i$; $t = \alpha_1 \cdot \ldots \cdot \alpha_n$ and $p_n = p'$; and no occurrence appears more than once in the sequence.

6.1.3 Well-terminating processes

We use a slightly different definition from that of page 59: we use Milner's original definition to reuse as many of his results as possible.

A process p is *well-terminating* if, for every derivative p' of p, $p' \stackrel{\text{done}}{\to}$ is impossible, and also if $p' \stackrel{\overline{\text{done}}}{\to}$ then $p' \sim \overline{\text{done}}.0$.

Proposition 6.0 *If p is well-terminating and $p \stackrel{\alpha}{\to} p'$ then p' is well-terminating. If also $p \sim q$ then q is well-terminating.* □

We define the following combinators:

$$
\begin{array}{rcl}
\mathit{Done} & \stackrel{\text{def}}{=} & \overline{\text{done}}.0 \\
p;q & \stackrel{\text{def}}{=} & (p[\text{done} \mapsto b] \mid b.q) \setminus \{b\} \\
p \| q & \stackrel{\text{def}}{=} & (p[\text{done} \mapsto d_1] \mid q[\text{done} \mapsto d_2] \mid \\
 & & \qquad d_1.d_2.\mathit{Done} + d_2.d_1.\mathit{Done}) \setminus \{d_1, d_2\}
\end{array}
$$

where b, d_1 and d_2 are new names.

Proposition 6.1 *If p and q are well-terminating, so are $p;q$ and $p\|q$.* □

Proposition 6.2

$$
p;(q;r) \;=\; (p;q);r \qquad p\|q \;=\; q\|p \qquad p\|(q\|r) \;=\; (p\|q)\|r
$$
$$
\mathit{Done};p \;\approx\; p \qquad \mathit{Done}\|p \;\approx\; p
$$

where $=$ (respectively $\approx$) denotes CCS *observation congruence (respectively, equivalence) [Mil89, ch 7].* □

6.1.4 Formalising scoping

My intention is to relate the scoping side-condition of section 5.2.1 to restriction in CCS. This means that when I interpret rules I shall need to interpret such side-conditions too. This in turn means that I need a formal way to represent scoping. The following is meant to capture it in a simpleminded manner.

SDQI-rules An X-scoped SDQI-rule is a triple (a, D, sc) where (a, D) is a DQI-rule and $sc \subseteq X$ is the set of labels which its premises are meant to scope. Thus scoping is restricted to rule atoms. This means that the await rule in section 5.2.2 is no longer acceptable. However, it can be rewritten into the following acceptable form:

$$
\frac{R : \mathit{scope.await}(b, c, \sigma, \sigma')}{R : \texttt{await}\ b\ \texttt{then}\ c} \qquad \overline{} \qquad \frac{}{\sigma \rightsquigarrow \sigma'}
$$

$$
\frac{\underbrace{R : b \to tt \ \ \blacktriangleright\ \ R : c \qquad \sigma \rightsquigarrow \sigma'}_{\mathit{STORE}}}{R : \mathit{scope.await}(b, c, \sigma, \sigma')}
$$

Where $STORE$ is as defined in section 5.2.2. Thus we have split the previous definition into two parts.

An SDQI-system is a pair $(\mathcal{L}, \mathcal{R})$ where $\mathcal{L}$ is a language of formulae, and $\mathcal{R}$ is a set of $\mathcal{L}$-scoped SDQI-rules mentioning only formulae in $\mathcal{L}$.

Definition 6.3 (SDQI-deduction) *Let $\mathcal{T}$ be a* SDQI-*system. Then* $\mathbf{SDQI}(\mathcal{T})$ *is the least set of quadruples $(F, I, \sqsubset, D)$ such that*

1. $0 = (\emptyset, \emptyset, \emptyset, \emptyset) \in \mathbf{SDQI}(\mathcal{T})$.
2. *If* $\Sigma_1, \Sigma_2 \in \mathbf{SDQI}(\mathcal{T})$ *then* $\Sigma_1 \otimes_f \Sigma_2 \in \mathbf{SDQI}(\mathcal{T})$.
3. *If* (a) $(F, I, \sqsubset, D) \in \mathbf{SDQI}(\mathcal{T})$ *and* (a, D_a, sc) *matches a rule of* $\mathcal{T}$ *such that* (a, D_a) *can be applied to* $(F, I, \sqsubset, D)$ *to yield* $(F', I', \sqsubset', D')$ *and*
   (b) $D \cap (O(F) \times sc^{\pm}) = \emptyset$

   then $(F', I', \sqsubset', D') \in \mathbf{SDQI}(\mathcal{T})$

Thus an SDQI-deduction is just a DQI-deduction that has satisfied some scoping constraints. I say that a deduction Π is *proper* if $D(\Pi) = \emptyset$.

6.2 The interpretation

Let $(\mathcal{L}, \mathcal{R})$ be a DQI-system. The interpretation is going to assign to each formula $A \in \mathcal{L}$ a well-terminating process $\underline{A}$ such that A is deducible if and only if $\underline{A}$ reduces silently to a terminal state (i.e., its only visible action being $\overline{\text{done}}$).

Thus our interpretation bears some resemblence to the quantifier-free case of Gödel's functional interpretation of Heyting Arithmetic (HA) [Göd58]. (The *Dialectica interpretation* is also explained clearly in [HS86, Chapter 18].) In the quantifier-free case, logical formulae A (with free variables $x_1, \ldots, x_n$) are interpreted by programs A^* such that if A is provable in HA then for all closed programs $X_1, \ldots, X_n$, $A[X_1/x_1, \ldots, X_n/x_n]$ beta-reduces to $\bar{0}$ (the church numeral zero).

However, the resemblance ends there. Gödel's interpretation is for a specific proof-system (for extensions to other systems see [Tro73]), and it proceeds by induction on the structure of the formula being interpreted. We shall be interpreting arbitrary SDQI-systems $\mathcal{T}$, and therefore do not have an inductive structure of formulae to work with. Instead, we use the idea that the meaning of a formula is captured by the rule that introduces it [Sun84a]. We define the interpretation of $\mathcal{T}$-formulae directly from the structure of the $\mathcal{T}$-rules.

We are going to interpret rules as process constant bindings. A rule is seen as a procedure for establishing its conclusion. Formulae correspond to process constants. Therefore, a rule will typically be bound to a restriction of some parallel composition ($||$) of sequenced process constants. Now, for this to make sense, every process constant mentioned in a rule interpretation ought to be bound to some process. This corresponds to the proof-theoretic condition that every premise of a rule is potentially deducible (i.e., there exists some rule which concludes it). I call these systems *sensible*.

There are situations where unsensible systems are useful: for instance, the propagation-free rules for abortion in section 5.1.4 were unsensible. But there we had the pruning rule to restore sensibility. In this section I shall only interpret sensible systems, which give rise to well-terminating processes. Pruning corresponds to unnatural, premature termination, which suggests a different treatment. In the interests of simplicity, I shall not provide such a treatment. Nevertheless, if one so desires, one can regard the pruning rule as an ordinary rule with a lot of instances.

Let us fix an arbitrary SDQI-system $\mathcal{T} = (\mathcal{L}, \mathcal{R})$, and occurrence ordering $\trianglelefteq$. We lift $\trianglelefteq$ to order sets of occurrences: $X_1 \trianglelefteq X_2$ if $(\min X_1) \trianglelefteq (\min X_2)$ (where $\min X$ is the minimum occurrence of X with respect to $\trianglelefteq$).

We define an injection $\underline{\cdot} : \mathcal{L} \to \mathit{Const}$ that maps formulae to process constants. We extend $\underline{\cdot}$ to occurrences: $\underline{(A, n)} = \underline{A}$.

We interpret $\mathcal{T}$ as a set of CCS constant bindings $\Delta_{\mathcal{T}}$ in the following way. Each rule atom is interpreted as a process that forms a partial definition of the constant associated to its conclusion. Then we define each conclusion with the sum of the bodies that partially define it (so if there are three rules concluding C, process constant $\underline{C}$ will be defined as the sum of three bodies).

When D is a set of actions, let me write

$$D.p = \begin{cases} \tau.p & \text{if } D = \emptyset \\ \alpha_1.\cdots.\alpha_n.p & \text{if } D = \{\alpha_1, \ldots, \alpha_n\} \end{cases}$$

(remembering the convention that $\alpha_1, \ldots, \alpha_n$ lists the elements of D using the fixed order). For sequences of occurrences, let me write $\underline{A_1 \cdot \ldots \cdot A_n}$ for $\underline{A_1}; \ldots; \underline{A_n}$. Then the interpretation of an SDQI-rule is given by

$$\underline{((\{P_1, \ldots, P_n\}, C), D, sc)} = D.(\underline{P_1} \| \ldots \| \underline{P_n}) \setminus sc$$

and by convention, the parallel composition of zero processes is simply *Done*, the null well-terminating process.

Rule systems to constant bindings Finally, let $\mathcal{R}(C)$ be the set of rules in $\mathcal{R}$ that conclude C. Then, we define the constant binding environment by $\Delta_{\mathcal{T}}$ as follows:

$$\Delta_{\mathcal{T}}(\underline{C}) = \sum_{r \in R(C)} \underline{r}$$

Some examples

For example, I give the interpretation of the QI-system $\mathcal{P}^{(;)}_{QI}$, the I-system $\mathcal{P}_{SOS}$ and the `await` rule. I abbreviate the interpretation by giving only equation schemes. Technically there should be different constants for different instantiations of process variables p and q. I label the communication interactions $comm(a)$ when a is the name of the communicating action.

$$
\begin{array}{rcl@{\qquad}rcl}
\underline{0\surd} & \stackrel{\text{def}}{=} & \tau.Done & \underline{a.p \xrightarrow{a} p} & \stackrel{\text{def}}{=} & +comm(a).Done \\
\underline{a.p\surd} & \stackrel{\text{def}}{=} & +comm(a).\underline{p\surd} & \underline{\bar{a}.p \xrightarrow{\bar{a}} p} & \stackrel{\text{def}}{=} & -comm(a).Done \\
\underline{\bar{a}.p\surd} & \stackrel{\text{def}}{=} & -comm(a).\underline{p\surd} & \underline{p|q \xrightarrow{\alpha} p'|q} & \stackrel{\text{def}}{=} & \tau.\underline{p \xrightarrow{\alpha} p'} \\
\underline{p|q\surd} & \stackrel{\text{def}}{=} & \tau.(\underline{p\surd} \,\|\, \underline{q\surd}) & \underline{p|q \xrightarrow{\alpha} p|q'} & \stackrel{\text{def}}{=} & \tau.\underline{q \xrightarrow{\alpha} q'} \\
\underline{p;q\surd} & \stackrel{\text{def}}{=} & \tau.(\underline{p\surd} \,;\, \underline{q\surd}) & \underline{p|q \xrightarrow{\tau} p'|q'} & \stackrel{\text{def}}{=} & \tau.(\underline{p \xrightarrow{\alpha} p'} \,\|\, \underline{q \xrightarrow{\bar{\alpha}} q'})
\end{array}
$$

As a more complex example, I give the schema for the interpretation of the modified `await` rule atom:

$$\underline{R : \texttt{await} \;\; b \;\texttt{do} \;\; c} \;\stackrel{\text{def}}{=}\; +atomic(\sigma, \sigma').\underline{R : scope.await(b, c, \sigma, \sigma')}$$

$$\underline{R : scope.await(b, c, \sigma, \sigma')} \;\stackrel{\text{def}}{=}\; \tau.((\underline{b \to tt} \,;\, \underline{R : c}) \;\|\; \underline{\sigma \rightsquigarrow \sigma'}) \setminus STORE$$

The next result is that the interpretation binds every constant to a well-terminating process. Now well-terminating process need not terminate, therefore we are showing a safety property: no process terminates without signalling $\overline{\text{done}}$ first. This can be seen as a consistency result.

Proposition 6.4 *Let $\mathcal{T}$ be a sensible* SDQI-*system. Then every constant in $\Delta_{\mathcal{T}}$ is bound to a well-terminating process.*

Proof: Say a set $X \subseteq \text{dom}\ \Delta_{\mathcal{T}}$ *checks out* if for all $\underline{C} \in X$,

(1) $FV(\Delta_{\mathcal{T}}(\underline{C})) \subseteq X$
(2) if every $C' \in FV(\Delta_{\mathcal{T}}(\underline{C}))$ is well-terminating, then so is $\Delta_{\mathcal{T}}(\underline{C})$

We show by coinduction on the definition of the relation *checks out* that dom $\Delta_{\mathcal{T}}$ checks out. Let $\underline{C} \in \text{dom}\ \Delta_{\mathcal{T}}$ and $p = \Delta_{\mathcal{T}}(\underline{C})$. (1) $FV(p) \subseteq \text{dom}\ \Delta_{\mathcal{T}}$ by the sensibility of $\mathcal{T}$. (2) Then $p = \sum_{r \in \mathcal{R}(C)} \underline{r}$. Suppose every $C' \in FV(p)$ is well-terminating. To show p is well-terminating, we must show that every choice is well-terminating. Let q be one such choice. Then $q = D.(Done) \setminus sc$ or $q = D.(\underline{P_1}\| \ldots \|\underline{P_n}) \setminus sc$ (for $n \geq 1$). In the first case, q is well-terminating because *Done* is well-terminating, restriction preserves it, and so does prefixing with D, because done, $\overline{\text{done}} \notin D$ by definition. In the second, the same result holds because we know that restriction, D-prefixing and the ; and $\|$ operators preserve well-termination. □

Relating deductions and transition sequences

We are going to relate the deductions of a sensible SDQI-system $\mathcal{T}$ to terminating transition sequences of $\Delta_{\mathcal{T}}$ processes. The interesting point is that the order of the non-silent actions is related to the order of the dangling interactions given by some history. In fact, we use histories to sequentialize $\lesssim_\Pi$ (where $\Pi \in \mathbf{SDQI}(\mathcal{T})$) and obtain a linear order of the dangling interactions.

Let h be a history of Π. Then we define a lexicographic order over $D(\Pi)$: $(A, \alpha) \leq_h (B, \beta)$ if

$$h(A) > h(B) \text{ or } (h(A) = h(B) \text{ and } \alpha \trianglelefteq \beta)$$

Let $\Pi \in \mathbf{SDQI}(\mathcal{T})$ have history h. Let s be a $\mathcal{C}_{\Delta_\mathcal{T}}$-deduction sequence. We say s *traces* Π via h if there exists an injective function $f : D(\Pi) \to \mathrm{dom}\,(s)$ such that

$$\begin{array}{l} \text{if } (A,\alpha) \leq_h (B,\beta) \text{ then } f(A,\alpha) \leq f(B,\beta) \\ \text{for all } (A,\alpha), \mathit{act}(s(f(A,\alpha))) = \alpha \\ \text{for all } i \in \mathrm{dom}\,(s) \setminus \mathrm{im}\ f, \mathit{act}(s(i)) \in \{\tau, \overline{\mathrm{done}}\} \end{array}$$

where $\mathit{act}(\Sigma) = \alpha$ if $\Sigma \vdash p \xrightarrow{\alpha} p'$ for some p, p'. The first condition says that traces must respect the historical order of dangling interactions. The second states that every undischarged formula perspective is an action in the trace. The third states that the only non-$\overline{\mathrm{done}}$ actions in the trace are undischarged formula perspectives.

For convenience, we label the $\mathcal{C}_\Delta$-deductions that conclude with an instance of the communication rule *Communication deductions*.

6.2.1 The interpretation is sound

Figure 6–1 defines a partial function *sub* which strips out the evaluation transition sequence of an immediate subprocess of $p|q$. (This is called *projection* in [Bes83]) If $s : p|q \xrightarrow{\alpha}{}^* \Omega$ then $sub(1, s) : p \xrightarrow{\alpha}{}^* \Omega$ and $sub(2, s) : q \xrightarrow{\alpha}{}^* \Omega$ are the evaluations of the processes on the left and right of the parallel composition respectively.

Let h and h' be histories. Then we say h *subsumes* h' if $\mathrm{dom}\ h' \subseteq \mathrm{dom}\ h$ and for all $A, B \in \mathrm{dom}\ h'$, $h'(A) \leq h'(B)$ if and only if $h(A) \leq h(B)$. That is, one history subsumes another if it preserves the timestamping relationship between common formula occurrences. This timestamping relationship is used to determine the order in which we compute the transitions of the interpreting process.

Lemma X(i) (Assembly) *Suppose we have a deduction sequence s of $p_1|p_2 \xrightarrow{t} \Omega$, and suppose for $i = 1, 2$ that projection s_i of s of $p_i \xrightarrow{t_i} \Omega$ that there exists a Π_i with history h_i such that s_i traces Π_i via h_i. Then there exists a binary connector g of Π_1 and Π_2 and a history h of $\Pi = \Pi_1 \otimes_g \Pi_2$ such that s traces Π via h and h subsumes h_1 and h_2.*

Proof: let $f_i : D(\Pi_i) \to \mathrm{dom}\,(s_i)$ be the appropriate functions that show how s_i traces Π_i via h_i. We construct g according to the communications between s_1 and s_2 in s. Let $\mathit{inj}_i : \mathrm{dom}\,(s_i) \to \mathrm{dom}\,(s)$ be the injective function such that for all $j \in \mathrm{dom}\,(s_i)$, $O(s_i(j)) \subseteq O(s(\mathit{inj}_i(j)))$. That is, inj_i maps the index of deduction Σ of s_i to the index of the deduction in s that contains Σ: i.e., from that which it was projected. Since projection preserves the relative order of deductions, it is easy to show that inj_i is order-preserving too. Also, if $\mathit{inj}_1(i) = \mathit{inj}_2(j)$ then the deductions $s_1(i)$ and $s_2(j)$ were children of a communication rule in s. So we

$$
\begin{aligned}
& sub(x, \varepsilon) = \varepsilon \\[1em]
& sub\left(1,\ \dfrac{\overset{\Sigma}{p \xrightarrow{\alpha} p'}}{p|q \xrightarrow{\alpha} p'|q} \cdot s\right) = \left(\overset{\Sigma}{p \xrightarrow{\alpha} p'}\right) \cdot sub(1, s) \\[1em]
& sub\left(1,\ \dfrac{\overset{\Sigma}{q \xrightarrow{\alpha} q'}}{p|q \xrightarrow{\alpha} p|q'} \cdot s\right) = sub(1, s) \\[1em]
& sub\left(1,\ \dfrac{\overset{\Sigma_1}{p \xrightarrow{\alpha} p'} \quad \overset{\Sigma_2}{q \xrightarrow{\bar{\alpha}} q'}}{p|q \xrightarrow{\tau} p'|q'} \cdot s\right) = \left(\overset{\Sigma_1}{p \xrightarrow{\alpha} p'}\right) \cdot sub(1, s) \\[1em]
& sub\left(2,\ \dfrac{\overset{\Sigma}{q \xrightarrow{\alpha} q'}}{p|q \xrightarrow{\alpha} p|q'} \cdot s\right) = \left(\overset{\Sigma}{q \xrightarrow{\alpha} q'}\right) \cdot sub(2, s) \\[1em]
& sub\left(2,\ \dfrac{\overset{\Sigma}{p \xrightarrow{\alpha} p'}}{p|q \xrightarrow{\alpha} p'|q} \cdot s\right) = sub(2, s) \\[1em]
& sub\left(2,\ \dfrac{\overset{\Sigma_1}{p \xrightarrow{\alpha} p'} \quad \overset{\Sigma_2}{q \xrightarrow{\bar{\alpha}} q'}}{p|q \xrightarrow{\tau} p'|q'} \cdot s\right) = \left(\overset{\Sigma_2}{q \xrightarrow{\bar{\alpha}} q'}\right) \cdot sub(2, s)
\end{aligned}
$$

Figure 6–1: The partial function $sub : \{1, 2\} \times \mathbf{I}(\mathcal{C}_\Delta)^* \rightharpoonup \mathbf{I}(\mathcal{C}_\Delta)^*$

define $H_i = \{f_i^{-1}(j) \mid \exists k. inj_i(j) = inj_{3-i}(k)\}$ to be the set of dangling interactions of Π_i which are to be connected by g. Then $g : (H_1 \cup H_2) \leftrightarrow (H_1 \cup H_2)$ is defined by

$$g(A, \alpha) = f_{3-i}^{-1}(inj_{3-i}^{-1}(inj_i(f_i(A, \alpha))))$$

when $(A, \alpha) \in D(\Pi_i)$. It is straightforward to show that this is involutive. However, we still have to show that it is a connector of $\Pi_1 \cup \Pi_2$ with respect to some history h. If it is a connector, then it is obviously binary.

First note that $g(A, \alpha) = (A', \bar{\alpha})$. This follows because if $s(inj_i(f_i(A, \alpha)))$ is a communication of two transitions, then one transition must be labelled α and the other $\bar{\alpha}$. Since s_{3-i} must contain the other transition, it must be labelled $\bar{\alpha}$. Therefore if $inj_{3-i}(f_{3-i}(A', \beta)) = inj_i(f_i(A, \alpha))$ then $\beta = \bar{\alpha}$ by the properties of f_i and f_{3-i}.

Next, suppose that $g(A, \alpha) = (A', \bar{\alpha})$, $g(B, \beta) = (B', \bar{\beta})$, and $h_1(A) \leq h_1(B)$. We want to show that $(A, \alpha) \leq_h (B, \beta)$ implies $f(A, \alpha) \leq f(B, \beta)$. There are two cases. First, suppose $h_1(A) < h_1(B)$. Then $(B, \beta) <_{h_1} (A, \alpha)$ and so $f_i(B, \beta) < f_i(A, \alpha)$. Therefore $inj_i(f_i(B, \beta)) < inj_i(f_i(A, \alpha))$ which in turn means $inj_{3-i}^{-1}(inj_i(f_i(B, \beta))) < inj_{3-i}^{-1}(inj_i(f_i(A, \alpha)))$, and so $(B', \bar{\beta}) <_{h_2} (A', \bar{\alpha})$, and so $h_2(A') < h_2(B')$. When $h_1(A) = h_1(B)$ the reasoning is similar, except that we have to consider the relative order of α and β. The result goes through because the relative order between perspectives is not affected by taking complements..

With this in mind, we can construct a history of $\Pi_1 \cup \Pi_2$ such that for all $A, B \in O(\Pi_i)$, $h_i(A) \leq h_i(B)$ iff $h(A) \leq h(B)$, and if $g(A, \alpha) = (A', \bar{\alpha})$ then $h(A) = h(A')$. These properties will ensure that g is a connector with respect to h.

> This is achieved by an iterative method, where we consider each element (A, α) of H_1 (mapped to $(A', \bar{\alpha})$ of H_2) in turn and add displacements in the appropriate places to ensure $h(A) = h(A')$. We define a sequence of histories $h^0, \ldots, h^n$ where $n = |H_1| = |H_2|$ and $h^0 = h_1 \oplus h_2$. We order the elements of H_1 according to $\leq_{h_1}$. To construct h^{i+1}, we consider the $i+1$-th dangling interaction (A_i, α_i) of H_1, and its mate $(B_i, \bar{\alpha_i})$ in H_2. Suppose $h^i(A) < h^i(B)$. Then we compute the difference d between the two, and alter h^{i+1} by:
>
> $$h^{i+1}(A) = \begin{cases} h^i(A) & \text{if } A \in O(\Pi_2) \text{ or } h^i(A) \leq h^i(A_i) \\ h^i(A) + d & \text{otherwise} \end{cases}$$
>
> When $h^i(B) < h^i(A)$ we compute h^i in the symmetric way. It is not hard to see that h^{i+1} subsumes h^i. We set $h = h^n$.

Thus g is a connector of Π_1 and Π_2 with respect to h. Let $\Pi = \Pi_1 \otimes_g \Pi_2$, then h is a history of Π.

We have to modify h again to ensure that dangling interactions unrelated in Π are ordered by h according to the positions of their corresponding transition deductions in s. This will show that s traces Π via the modified history.

> Once again, we use a displacement adding argument. We define $f : D(\Pi) \to \mathrm{dom}(s)$ by $f(A, \alpha) = inj_i(f_i(A, \alpha))$ when $(A, \alpha) \in D(\Pi_i)$. f induces a total order over the dangling interactions of Π: if $f(A, \alpha) <$

$f(B,\beta)$ then the transition corresponding to (A,α) occurs earlier in s than that of (B,β). Note that no two dangling interactions of Π will be assigned equal positions in s: by our construction, every pair of dangling interactions with equal positions in s were connected by g. So let us order the dangling interactions $(A_1,\alpha_1),\ldots,(A_n,\alpha_n)$ such that $f(A_k,\alpha_k) < f(A_{k+1},\alpha_{k+1})$ for all $k = 1,\ldots,n-1$. Then we define a sequence of histories $h^0,\ldots,h^n$ by $h^0 = h$ and for each $i+1$, let $1 \leq k_{i+1} \leq i$ be the least index such that such that $f(A_{k_{i+1}},\alpha_{k_{i+1}}) \geq f(A_{i+1},\alpha_{i+1})$. If it does not exist, then $h^{i+1} = h^i$. Otherwise, let $d = 1 + h^i(A_{i+1}) - h^i(A_{k_{i+1}})$ and

$$h^{i+1}(A) = \begin{cases} h^i(A) & \text{if } h^i(A) < h^i(A_{k_{i+1}}) \\ h^i(A) + d & \text{ow} \end{cases}$$

A simple inductive proof shows that for all i, h^i is a history of Π, h^{i+1} subsumes h^i and for $1 \leq k, k' \leq i$, $(A_k,\alpha_k) \leq_{h^i} (A_{k'},\alpha_{k'})$ if and only if $f(A_k,\alpha_k) \leq f(A_{k'},\alpha_{k'})$.

It is now straightforward to show that s traces Π via h^n. The first condition, that $(A,\alpha) \leq_{h^n} (B,\beta)$ if and only if $f(A,\alpha) \leq f(B,\beta)$ is the result of the aforementioned inductive proof.

For the second condition, since $(A,\alpha) \in D(\Pi)$, $(A,\alpha) \notin \mathrm{dom}\ g$. Therefore, $s(inj_i(f_i(A,\alpha)))$ is not a communication deduction. Therefore $act(s(inj(f_i(A,\alpha))))$ equals $act(s_i(f_i(A,\alpha)))$ which in turn equals α by the main induction. Finally, for the third condition, let $j \in \mathrm{dom}\,(s) \setminus \mathrm{im}\ f$. Suppose $act(s(j)) \notin \{\tau, \overline{\mathrm{done}}\}$. Then $s(j)$ cannot be a communication deduction, so there must be an i such that $act(sub(i,s(j))) = act(s(j))$. But this must mean that there exists a k such that $s_i(k) = sub(i,s(j))$, which means that $act(s_i(k)) \notin \{\tau, \overline{\mathrm{done}}\}$. But then, by induction, that means there must exist an $(A,\alpha) \in D(\Pi_i)$ such that $f_i(A,\alpha) = k$. Since $s(inj_i(k))$ is not a communication deduction, (A,α) is not connected to anything by g, so therefore $f(A,\alpha) = inj_i(k) = j$. Contradiction. □

The following result is an easy corollary of the above. Let us extend the notion of projection to $||$. We define two auxiliary functions, $unlabel, unrestrict : \mathbf{I}^*(\mathcal{C}_\Delta) \rightharpoonup \mathbf{I}^*(\mathcal{C}_\Delta)$ to remove the relabelling and restriction rules that occur as the lowest rule in each deduction of a sequence.

$$unlabel(\varepsilon) = unrestrict(\varepsilon) = \varepsilon$$

$$unlabel\left(\frac{\begin{array}{c}\Sigma \\ p \stackrel{\alpha}{\to} p'\end{array}}{p[f] \stackrel{f(\alpha)}{\to} p'[f]} \cdot s \right) = \Sigma \cdot unlabel(s)$$

$$unrestrict\left(\frac{\begin{array}{c}\Sigma \\ p \stackrel{\alpha}{\to} p'\end{array}}{p \setminus L \stackrel{\alpha}{\to} p' \setminus L} \cdot s \right) = \Sigma \cdot unrestrict(s)$$

Then when s is a deduction sequence of $p||q$, we define

$$sub(i,s) = unlabel(sub(i, unrestrict(s)))$$

The proof of the following result is easy.

Lemma X(ii) *Suppose p_1 and p_2 are well-terminating, and we have a deduction sequence s of $p_1 \| p_2 \xrightarrow{t} \Omega$, and suppose for $i = 1, 2$ that projection s_i of s of $p_i \xrightarrow{t_i} \Omega$ that there exists a Π_i with history h_i such that s_i traces Π_i via h_i. Then there exists a binary connector g of Π_1 and Π_2 and a history h of $\Pi = \Pi_1 \otimes_g \Pi_2$ such that s traces Π via h and h subsumes h_1 and h_2.*

Proof: By well-termination, s_1 and s_2 will perform only one $\overline{done}$ action, and no done actions. Let s' be the deduction sequence of $p_1|p_2$ obtained from the projection $p_1[\text{done} \mapsto d_1] \,|\, p_2[\text{done} \mapsto d_2]$ of s simply by removing the evident relabelling operators from each deduction. Then s_1 and s_2 are projections of s'. By lemma X(i), we find the binary connector g and the history h such that $\Pi = \Pi_1 \otimes_g \Pi_2$ is traced by s', and h subsumes h_1 and h_2. Let $f : D(\Pi) \to \text{dom}\,(s')$ be the witnessing function.

It remains to relate the original sequence s to Π. We use the witnessing function f again. Then we already know the first property is satisfied. Now, consider $act(s(f(A, \alpha)))$. Since α labels a dangling interaction, it is neither silent nor $\overline{done}$. Therefore it is not restricted nor a relabelling of $\overline{done}$ by $[\text{done} \mapsto d_i]$. This means that $act(s(f(A, \alpha))) = act(s'(f(A, \alpha))) = \alpha$. Last, let $i \in \text{dom}\,(s) \setminus \text{im}\, f$. Then either $i \in \text{dom}(s') \setminus \text{im}\, f$ or not. If so, then $act(s'(i)) \in \{\tau, \overline{\text{done}}\}$. But given the definition of s', this means that $act(s(i)) \in \{\tau, \overline{\text{done}}\}$ too (relabelling and restriction do not affect τ actions, and if $\overline{\text{done}}$ is relabelled to $\overline{d_i}$, then the restriction forces it to communicate with d_1 in s, which means that $act(s(i)) = \tau$.) If not, then it can only be an $\overline{\text{done}}$ action: the last action of s. $\square$

Lemma X(iii) (Sequencing) *Suppose p_1 and p_2 are well-terminating and that we have a deduction sequences s of $p_1;p_2 \xrightarrow{t} \Omega$, and suppose for $i = 1, 2$, there exists a deduction Π_i and a history h_i of Π_i such that $s_i = sub(i, s)$ traces Π_i via h_i. Then there exists a history h of $\Pi = \Pi_1 \otimes_0 \Pi_2$ such that s traces Π via h, and h subsumes h_1 and h_2.*

Proof: Given the definition of the ; operator and by well-termination we find that s is $s_1' \cdot \Sigma \cdot s_2'$, where s_1' is just s_1 all of whose deductions have had an extra rule applied to convert p_1's transitions into $p|b.q$'s transitions. Similarly, s_2' is s_2 modified to make p_2's transitions transitions of $\Omega|p_2$, and Σ is the communication of the $\overline{b}$ and b actions. We define

$$h(A) = \begin{cases} h_2(A) & \text{if } A \in O(\Pi_2) \\ h_1(A) + 1 + \max(\text{im } h_2) & \text{if } A \in O(\Pi_1) \end{cases}$$

(it is easy to see that h subsumes h_1 and h_2). To show that s traces Π via h, we define $f : D(\Pi) \to \text{dom}\,(s)$ by

$$f(A, \alpha) = \begin{cases} f_1(A, \alpha) & \text{if } (A, \alpha) \in D(\Pi_1) \\ f_2(A, \alpha) + |s_1| & \text{if } (A, \alpha) \in D(\Pi_1) \end{cases}$$

Once again, there are three things to show. First, suppose $(A, \alpha) \leq_h (B, \beta)$. Then either both (A, α) and (B, β) belong to $D(\Pi_i)$. By hypothesis, $f_i(A, \alpha) \leq f_i(B, \beta)$, and so $f(A, \alpha) \leq f(B, \beta)$. If $(A, \alpha) \in D(\Pi_1)$ and $(B, \beta) \in D(\Pi_2)$ then $f_1(A, \alpha) \leq |s_1| + f_2(B, \beta)$. The other two properties follow almost immediately given that the sequence s effectively appends s_2 after s_1. □

Theorem X (Soundness) *Let s be a deduction sequence of $\underline{A_1}\| \dots \|\underline{A_n} \xrightarrow{t} \Omega$. Then there exists a deduction $\Pi \vdash A_1, \dots, A_n$ with history h such that s traces Π via h.*

Proof: by induction on the length of s **Case $|s| = 0$:** We have $\Pi = 0$. **Case $|s| = k+1$:** by strong induction on n: **Case $n = 1$:** here the first transition must involve the expansion of the constant $\underline{A_1}$:

$$\dfrac{\dfrac{\dfrac{\Sigma}{p \xrightarrow{\alpha_1} p'}}{\sum_{r \in \mathcal{R}(A_1)} \underline{r} \xrightarrow{\alpha_1} p'}}{\underline{A_1} \xrightarrow{\alpha} p'}$$

Let $\underline{r} = D.(\underline{P_1}\| \cdots \|\underline{P_m}) \setminus sc$. Either $D = \emptyset$, in which case $\alpha_1 = \tau$ or $D = \{\alpha_1, \dots, \alpha_k\}$, so $\underline{r} = \alpha_1. \cdots .\alpha_k.(\underline{P_1}\| \cdots \|\underline{P_m}) \setminus sc$. Then $s = \Sigma_1 \cdot \dots \cdot \Sigma_k \cdot s'$ where Σ_i records the prefix transitions of α_i. Let s'' be s' after removing the lowermost restriction rules: s'' is a deduction sequence of $\underline{P_1}\| \cdots \|\underline{P_m} \xrightarrow{t''} \Omega$ where $t = \alpha_1 \cdot \dots \cdot \alpha_k \cdot t''$ if $D \neq \emptyset$ and $t = t''$ if $D = \emptyset$.

Let $s_1, \dots, s_m$ be the projections of s'' such that s_i is a deduction sequence of $p_i \xrightarrow{t_i} \Omega$. Suppose $p_i = \underline{A_{i1}}; \dots ; \underline{A_{ik_i}}$. Then by repeated application of lemma X(iii) we obtain $\Pi_i \vdash A_{i1}, \dots, A_{ik_i}$ with history h_i such that s_i traces Π_i via h_i. By repeated application of lemma X(ii) we obtain $\Pi'' \vdash A_{11}, \dots, A_{mk_m}$ with history h'' such that s'' traces Π'' via h''. We also know that no element of t'' is contained within sc (since every deduction of s' has the restriction rule applied to it). Therefore, $D(\Pi'') \cap (O(\Pi'') \times sc^{\pm}) = \emptyset$ by the definition of tracing. We know that h'' subsumes each of its component histories. Therefore the sequencing constraints can be satisfied. Thus we can apply the rule $\dfrac{P_1 \quad \cdots \quad P_m}{C} D$ to Π'' to yield $\Pi \vdash C$. We then define h to be the history of Π such that $h(A) = h''(A)$ if $A \in O(\Pi'')$ and $h(C) = 1 + \max(\text{im } h'')$ otherwise. Since the dangling interactions hanging off C will be ordered according to $\trianglelefteq$, as are the prefixed actions of $\Delta(\underline{C})$, it is easy to show that s traces Π via h.

Case $n > 1$: by induction and the application of lemma X(i). □

6.2.2 The interpretation is complete

The following lemma contains the heart of the proof. It shows the correspondence between the assembly of deductions and the communication of processes. It is quite lengthy, but not at all difficult.

Lemma XI(i) (Interleaving) *Suppose for all histories h_i of Π_i there exists a deduction sequence s_i of $p_i \xrightarrow{t_i} \Omega$ that traces Π_i via h_i (for $i = 1, 2$). Then for all g, there exists a t and a deduction sequence s of $p_1 | p_2 \xrightarrow{t} \Omega$ that traces $\Pi_1 \otimes_g \Pi_2$ via some history h.*

Proof: Let h be a history of Π. Let $i = 1, 2$. Then $h \upharpoonright O(\Pi_i)$ is a history of Π_i. Suppose $\Pi_i \vdash A_{i1}, \ldots, A_{in_i}$. Then by induction we get a deduction sequence s_i of $p_i \xrightarrow{t_i} \Omega$ that traces Π_i via h_i (and h) for some t_i. Let f_i be the associated injective functions.

We construct a deduction sequence s of $p_1 | p_2 \xrightarrow{t} \Omega$. To obtain the result we may have to reorganize the parallel composition (for instance it is possible that $p_1 = \underline{A_{11}} | \ldots | \underline{A_{1n_1}}$ and $p_2 = \underline{A_{21}} | \ldots | \underline{A_{2n_2}}$ and $A_{1j} \not\trianglelefteq A_{2k}$ for some j and k). This is quite trivial: it follows from the commutativity and associativity of $|$ with respect to strong bisimulation [Mil89, Proposition 4.8].

We are going to interleave s_1 and s_2 with respect to h. First we must isolate those actions of s_i which will communicate with s_{3-i}. Now there are $m = |\text{dom } g|/2$ pairs of interaction links between Π_1 and Π_2, and so there shall be m pairs of deductions from s_1 and s_2 which will communicate.

These are specified by the binary connector g. Let

$$\{c_{i1}, \ldots, c_{im}\} = \{f_i(A, \alpha) \mid (A, \alpha) \in (D(\Pi_i) \cap \text{dom } g)\}$$

be the set of indices of s_i which correspond to the dangling interactions of Π_i which are connected by g. Now, it follows immediately from the definition of connector that if $g(A, \alpha) = (A', \bar{\alpha})$ and $g(B, \beta) = (B', \bar{\beta})$ then $(A, \alpha) \leq_h (B, \beta)$ if and only if $(A', \bar{\alpha}) \leq_h (B', \bar{\beta})$. This means that for $1 \leq j \leq m$, $c_{11}, \ldots, c_{1m_1}$ occur in s_1 in the same order as $c_{21}, \ldots, c_{2m_2}$ occur in s_2. Thus to build s, we shall apply the communication rule to deductions $s_1(c_{1j})$ with $s_2(c_{2j})$. This works because connectors also map formula perspectives into their opposite perspectives. Now we can describe the interleaving of the two sequences. Let us assume that the deductions of $s_1 \cdot s_2$ are disjoint, and let X be the set of these deductions. Then we define the following relation. We say $\Sigma_1 \preceq \Sigma_2$ if and only if there exists $i, j, k, (A, \alpha)$ and (B, β) such that

$$\begin{array}{ll} (1) & \Sigma_1 = s_i(f_i(A, \alpha)),\ \Sigma_2 = s_j(f_j(B, \beta)) \text{ and } (A, \alpha) \leq_h (B, \beta) \quad \text{or} \\ (2) & \Sigma_1 = s_i(j),\ \Sigma_2 = s_i(k) \text{ and } j \leq k \end{array}$$

We write $\Sigma_1 \prec \Sigma_2$ if $\Sigma_1 \preceq \Sigma_2$ and not $\Sigma_2 \preceq \Sigma_1$. We write $\Sigma_1 \asymp \Sigma_2$ if both $\Sigma_1 \preceq \Sigma_2$ and vice-versa. Now $\preceq$ is a preorder: reflexivity follows trivially. Transitivity follows by a straightforward case analysis. It is not a partial order because for each $1 \leq j \leq m$, $c_{1j} = (A, \alpha)$ and $c_{2j} = (B, \bar{\alpha})$ implies that $(A, \alpha) =_h (B, \bar{\alpha})$ because $h(A) = h(B)$ by the definition of connector and $\bar{\alpha} \trianglelefteq \alpha$ and vice-versa.

$$IL^p_q(\varepsilon,\varepsilon) = \varepsilon$$

$$IL^p_q\left(\varepsilon, \left(\stackrel{\Sigma}{q \xrightarrow{\alpha} q'}\right)\cdot s\right) = \left(\frac{\stackrel{\Sigma}{q \xrightarrow{\alpha} q'}}{p|q \xrightarrow{\alpha} p|q'}\right)\cdot IL^p_{q'}(\varepsilon, s)$$

$$IL^p_q\left(\left(\stackrel{\Sigma}{p \xrightarrow{\alpha} p'}\right)\cdot s, \varepsilon\right) = \left(\frac{\stackrel{\Sigma}{p \xrightarrow{\alpha} p'}}{p|q \xrightarrow{\alpha} p'|q}\right)\cdot IL^{p'}_q(s, \varepsilon)$$

$$IL^p_q\left(\left(\stackrel{\Sigma}{p \xrightarrow{\alpha} p'}\right)\cdot s_1, \left(\stackrel{\Sigma}{q \xrightarrow{\beta} q'}\right)\cdot s_2\right) = \begin{cases} \left(\frac{\stackrel{\Sigma}{p \xrightarrow{\alpha} p'}}{p|q \xrightarrow{\alpha} p'|q}\right)\cdot IL^{p'}_q(s_1, \Sigma_2\cdot s_2) & \text{if } \Sigma_1 \prec' \Sigma_1 \\ \left(\frac{\stackrel{\Sigma}{q \xrightarrow{\beta} q'}}{p|q \xrightarrow{\alpha} p|q'}\right)\cdot IL^{p}_{q'}(\Sigma_1\cdot s_1, s_2) & \text{if } \Sigma_2 \prec' \Sigma_2 \\ \left(\frac{\stackrel{\Sigma}{p \xrightarrow{\alpha} p'} \quad \stackrel{\Sigma}{q \xrightarrow{\beta} q'}}{p|q \xrightarrow{\tau} p'|q'}\right)\cdot IL^{p'}_{q'}(s_1, s_2) & \text{if } \Sigma_1 \asymp' \Sigma_2 \end{cases}$$

Figure 6–2: Interleaving two interacting sequences

Therefore $s_1(c_{1j}) \asymp s_2(c_{2j})$. Now, let $\preceq'$ be any extension of $\preceq$ such that for all $\Sigma_1, \Sigma_2 \in X$,

$$\begin{array}{l} \Sigma_1 \preceq' \Sigma_2 \text{ or } \Sigma_1 \asymp' \Sigma_2 \text{ or } \Sigma_2 \preceq' \Sigma_1 \\ \text{if } \Sigma_1 \asymp' \Sigma_2 \text{ then } \Sigma_1 \asymp \Sigma_2 \end{array}$$

Thus it is a total order which preserves the order of $\preceq$, and which adds no new equalities. Then we define the interleaving of s_1 and s_2 with respect to $\preceq'$ to be $IL^{p_1}_{p_2}(s_1, s_2)$ where $IL^p_q(s_1, s_2)$ is defined in figure 6–2.

Note that from what has gone before, in the last case, $\Sigma_1 \asymp' \Sigma_2$ implies $\Sigma_1 \asymp \Sigma_2$ which implies that for some i, (A,α) and (B,β) that $s_i(f_i(A,\alpha)) = \Sigma_1$ and $s_{3-i}(f_{3-i}(B,\beta)) = \Sigma_2$ and $g(A,\alpha) = (B,\beta)$, whence $\bar{\alpha} = \beta$ as g is a connector. Thus $act(\Sigma_1) = \alpha$ and $act(\Sigma_2) = \bar{\alpha}$ and so we can apply the communication rule.

Finally we show that s traces Π via h. Let $f : D(\Pi) \to \mathrm{dom}\,(s)$ be defined by $f(A,\alpha) = j$ where if $(A,\alpha) \in D(\Pi_i)$ then $O(s(j)) \supseteq O(s_i(f_i(A,\alpha)))$. To show injectivity, suppose $f(A,\alpha) = f(B,\beta) = k$ and $(A,\alpha) \neq (B,\beta)$ where $(A,\alpha) \in D(\Pi_i)$ and $(B,\beta) \in D(\Pi_j)$. Then $O(s(k)) \supseteq O(s_i(f_i(A,\alpha))$ and $O(s(k)) \supseteq O(s_j(f_j(B,\beta)))$. From the definition of s, $s(k)$ is built from one or two deductions in X. In the first case, $s_i(f_i(A,\alpha)) = s_j(f_j(B,\beta))$ from which $i = j$ and so $(A,\alpha) = (B,\beta)$ by the injectivity of f_i. Contradiction. In the second case, $s(k)$ concludes a τ transition. But then by the translation, this only occurs when (A,α) is connected to (B,β) which means that the two do not occur in $D(\Pi)$, and hence do not appear in the domain of f. Contradiction.

Second, let $(A,\alpha), (B,\beta) \in D(\Pi)$ be such that $(A,\alpha) \leq_h (B,\beta)$. Suppose $(A,\alpha) \in D(\Pi_i)$ and $(B,\beta) \in D(\Pi_j)$. Then we know that $s_i(f_i(A,\alpha)) \preceq' s_j(f_j(B,\beta))$, and therefore the first deduction will occur earlier in s than the second. Hence $f(A,\alpha) \leq f(B,\beta)$.

Third, we show $act(s(f(A,\alpha))) = \alpha$. From the definition of s, there must exist a j such that $O(s(f(A,\alpha))) \supseteq O(s_j(f_j(A,\alpha)))$. By induction, $act(s_j(f_j(A,\alpha))) = \alpha$. The result follows from the definition of s: $act(s(f(A,\alpha))) = act(s_j(f_j(A,\alpha)))$.

Fourth and last, we have to show that for all $i \in \mathrm{dom}\,(s) \setminus \mathrm{im}\ f$, $act(s(i)) \in \{\tau, \overline{\mathrm{done}}\}$. Suppose not: let $i \in \mathrm{dom}\,(s) \setminus \mathrm{im}\ f$ be such that $act(s(i)) \notin \{\tau, \overline{\mathrm{done}}\}$. Then from the definition of s, there must exist a j and k such that $O(s(i)) \supseteq O(s_j(k))$, and moreover, since $act(s_j(k)) \neq \tau$, it cannot be a communication deduction. By induction, $k = f_j(A,\alpha)$ for some $(A,\alpha) \in D(\Pi_j)$. Since $s(k)$ is not a communication of two deductions, it must be that $(A,\alpha) \notin \mathrm{dom}\ g$ and so $(A,\alpha) \in D(\Pi)$, which means by the definition of f that $i \in \mathrm{im}\ f$. Contradiction.

□

Lemma XI(ii) *Suppose for all histories h_i of Π_i there exist well-terminating processes p_i and deduction sequences s_i of $p_i \xrightarrow{t_i} \Omega$ that traces Π_i via h_i (for $i = 1, 2$). Then for all g, there exists a t and a deduction sequence s of $p_1 \| p_2 \xrightarrow{t} \Omega$ that traces $\Pi_1 \otimes_g \Pi_2$ via some history h.*

Proof: By lemma XI(i), we obtain a t, an h and a deduction sequence s that traces $\Pi = \Pi_1 \otimes_g \Pi_2$ via h. We construct a deduction sequence of $p_1 \| p_2 \xrightarrow{t'} \Omega$ which also traces Π via h. Since p_1 and p_2 are well-terminating s performs exactly two $\overline{\mathrm{done}}$ actions, one of which must occur at the end. We construct the deduction sequence $s' = par(s, d_1.d_2.Done + d_2.d_1.Done)$. The function par is defined in figure 6–3. We have to show that s exists par is well-defined. This follows by induction on the length of prefixes of s. The shortest possible length of s must be two, when it performs the two $\overline{done}$ actions. A simple check shows that s' exists in this case. If the size is $k+1$, then induction tells us that k-length prefix of s' exists. Consider the $k+1$th deduction. By symmetry there are three cases, given by the three cases in figure 6–3. The first and third case are trivial. The second case is the most interesting: it exists if r can perform the d_1 transition. Suppose it cannot. r is a derivative of $d_1.d_2.Done + d_2.d_1.Done$, so the only case when it cannot make a d_1 transition is if it has already performed one. But this is impossible, because s_1 only performs one $\overline{\mathrm{done}}$ action by well-terminatedness, hence the k-prefix of s cannot have performed it. By induction then, s' exists.

The last thing to do is construct s'' by appending a deduction to perform the last $\overline{\mathrm{done}}$ action of $p\|q$. An easy induction shows that s'' traces Π via h. □

Let $\Pi_1, \ldots, \Pi_n$ be a set of disjoint deductions, and let $\Pi = \bigotimes_0 \{\Pi_1, \ldots, \Pi_n\}$. Then a history h of Π *sequences* $\Pi_1, \ldots, \Pi_n$ in Π if for all $1 \leq i < j \leq n$, for all $A \in O(\Pi_i)$ and $B \in O(\Pi_j)$, $h(B) > h(A)$.

Lemma XI(iii) (Sequencing) *Suppose that for all histories h_i of $\Pi_i \vdash A_i$ (for $i = 1, \ldots, n$) there exists a deduction sequence s_i of $\underline{A_i} \xrightarrow{t_i} \Omega$ that traces Π_i via h_i. Then for all histories h that sequence $\Pi_1, \ldots, \Pi_n$ in $\Pi = \bigotimes_0 \{\Pi_1, \ldots, \Pi_n\}$ there exists a deduction sequence s of $\underline{A_1}; \ldots; \underline{A_n} \xrightarrow{t} \Omega$ that traces Π via h for some t.*

Let λ_1 be the relabelling function [done $\to d_1$] and λ_2 be the function [done $\to d_2$]. Last, let us write $p\|_r q$ for the process $(p[\lambda_1] \,|\, q[\lambda_2] \,|\, r) \setminus \{d_1, d_2\}$.

$$
\begin{aligned}
&par(\varepsilon, r) = \varepsilon \\
&par(\Sigma \cdot s, r) = \Sigma' \cdot par(s, r')
\end{aligned}
$$

where Σ' and r' are defined by cases as follows.

$$
\text{When } \Sigma = \frac{\overset{\Sigma}{p \xrightarrow{\alpha} p'}}{p|q \xrightarrow{\alpha} p'|q}\,, \qquad \Sigma' = \cfrac{\cfrac{\cfrac{\cfrac{\overset{\Sigma}{p \xrightarrow{\alpha} p'}}{p[\lambda_1] \xrightarrow{\alpha} p'[\lambda_1]}}{p[\lambda_1] \,|\, q[\lambda_2] \xrightarrow{\alpha} p'[\lambda_1] \,|\, q[\lambda_2]}}{(p[\lambda_1] \,|\, q[\lambda_2] \,|\, r) \xrightarrow{\alpha} (p'[\lambda_1] \,|\, q[\lambda_2] \,|\, r)}}{p\|_r q \xrightarrow{\alpha} p'\|_r q} \qquad r' = r
$$

where $\alpha \neq \overline{\text{done}}$, and so $\lambda_1(\alpha) = \alpha$. When $\alpha = \overline{\text{done}}$ then

$$
\Sigma' = \cfrac{\cfrac{\cfrac{\cfrac{\overset{\Sigma}{p \xrightarrow{\alpha} p'}}{p[\lambda_1] \xrightarrow{\overline{d_1}} p'[\lambda_1]}}{p[\lambda_1] \,|\, q[\lambda_2] \xrightarrow{\overline{d_1}} p'[\lambda_1] \,|\, q[\lambda_2]} \qquad \overset{\Sigma'}{r \xrightarrow{d_1} r'}}{(p[\lambda_1] \,|\, q[\lambda_2] \,|\, r) \xrightarrow{\tau} (p'[\lambda_1] \,|\, q[\lambda_2] \,|\, r)}}{p\|_r q \xrightarrow{\tau} p'\|_r q}
$$

The cases for when q makes the transition are symmetric. Last, when

$$
\Sigma = \frac{\overset{\Sigma_1}{p \xrightarrow{\alpha} p'} \qquad \overset{\Sigma_2}{q \xrightarrow{\bar{\alpha}} q'}}{p|q \xrightarrow{\tau} p'|q'} \qquad \Sigma' = \cfrac{\cfrac{\cfrac{\cfrac{\overset{\Sigma_1}{p \xrightarrow{\alpha} p'}}{p[\lambda_1] \xrightarrow{\alpha} p'[\lambda_1]} \qquad \cfrac{\overset{\Sigma_2}{q \xrightarrow{\bar{\alpha}} q'}}{q[\lambda_2] \xrightarrow{\bar{\alpha}} q'[\lambda_2]}}{(p[\lambda_1] \,|\, q[\lambda_2]) \xrightarrow{\tau} (p'[\lambda_1] \,|\, q'[\lambda_2])}}{(p[\lambda_1] \,|\, q[\lambda_2] \,|\, r) \xrightarrow{\tau} (p'[\lambda_1] \,|\, q'[\lambda_2] \,|\, r)}}{p\|_r q \xrightarrow{\tau} p'\|_r q'} \qquad r' = r
$$

Figure 6–3: The definition of $par : \mathbf{I}^*(\mathcal{C}_\Delta) \rightharpoonup \mathbf{I}^*(\mathcal{C}_\Delta)$

Proof: If $A \in O(\Pi_i)$ and $B \in O(\Pi_j)$ where $i < j$ then $h(A) > h(B)$. Therefore for all $(A, \alpha) \in D(\Pi_i)$ and $(B, \beta) \in D(\Pi_j)$, $(A, \alpha) <_h (B, \beta)$. Let q_i be the terminal state of p_i, and λ be the relabelling function that acts as the identity everywhere except at done, where $\lambda(\text{done}) = b$. Then we construct s to be $s'_1 \cdot \ldots \cdot s'_n$ where for each i and $j \in \text{dom}\,(s_i)$,

$$s'_i(j) = \dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c} s_i(j) \\ p \stackrel{\alpha}{\to} p' \end{array}}{p[\lambda] \stackrel{\lambda(\alpha)}{\to} p'[\lambda]}}{p[\lambda] | b.(\underline{A_{i+1}}; \ldots ;\underline{A_n}) \stackrel{\lambda(\alpha)}{\to} p'[\lambda] | b.(\underline{A_{i+1}}; \ldots ;\underline{A_n})}}{p;\underline{A_{i+1}}; \ldots ;\underline{A_n} \stackrel{\lambda(\alpha)}{\to} p';\underline{A_{i+1}}; \ldots ;\underline{A_n}}}{q_1 | \ldots | q_{i-1} | p;\underline{A_{i+1}}; \ldots ;\underline{A_n} \stackrel{\lambda(\alpha)}{\to} q_1 | \ldots | q_{i-1} | p;\underline{A_{i+1}}; \ldots ;\underline{A_n}}$$

(The exception is the s'_n, which does not use the relabelling function λ), and $s'_i(|s_i| + 1)$ is the deduction

$$\dfrac{\dfrac{\dfrac{\begin{array}{c} \Sigma_1 \\ p[\lambda] \stackrel{\overline{b}}{\to} q_i \end{array} \qquad b.(\underline{A_{i+1}}; \ldots ; \underline{A_n}) \stackrel{b}{\to} \underline{A_{i+1}}; \ldots ; \underline{A_n}}{p[\lambda] | b.(\underline{A_{i+1}}; \ldots ; \underline{A_n}) \stackrel{\tau}{\to} q_i | \underline{A_{i+1}}; \ldots ; \underline{A_n}}}{p;\underline{A_{i+1}}; \ldots ;\underline{A_n} \stackrel{\tau}{\to} q_i | \underline{A_{i+1}}; \ldots ;\underline{A_n}}}{q_1 | \ldots | q_{i-1} | p;\underline{A_{i+1}}; \ldots ;\underline{A_n} \stackrel{\tau}{\to} q_1 | \ldots | q_i | \underline{A_{i+1}}; \ldots ;\underline{A_n}}$$

We show that s traces Π via h. Define $f : D(\Pi) \to \text{dom}\,(s)$ by $f(A, \alpha) = f_i(A, \alpha) + \sum_{j=i}^{i-1} |s_j|$. There are three properties of f to show. First, suppose $(A, \alpha) \leq_h (B, \beta)$. There are two cases. First, suppose $(A, \alpha), (B, \beta) \in D(\Pi_i)$ for some i. Then by induction $f_i(A, \alpha) \leq f_i(B, \beta)$, and so $f(A, \alpha) \leq f(B, \beta)$. Second, suppose $(A, \alpha) \in D(\Pi_i)$ and $(B, \beta) \in D(\Pi_j)$ where $i \neq j$. Then by the definition of $\leq_h$, $h(A) > h(B)$ and so $i < j$. Thus $f(A, \alpha) < f(B, \beta)$.

Second, we get $act(s(f(A, \alpha))) = act(s_i(f_i(A, \alpha))) = \alpha$ by the construction and induction. Third, we know that for all $i \in \text{dom}\,(s) \setminus \text{im}\, f$, $act(s(i)) \in \{\tau, \overline{\text{done}}\}$, from induction. $\square$

Theorem XI (Completeness) *For all* $\Pi \in \mathbf{SDQI}(\mathcal{T})$, *if* $\Pi \vdash A_1, \ldots, A_n$ *then for every history* h *of* Π, *there exists a* $\mathcal{C}_{\Delta_\mathcal{T}}$*-deduction sequence* s *of* $\underline{A_1} \| \ldots \| \underline{A_n} \stackrel{t}{\to} \Omega$ *that traces* Π *via* h *for some* t.

Proof: By induction on the depth of inference of Π. **Case $\boldsymbol{\Pi = 0}$:** we set $s = \varepsilon$, which is a deduction sequence attributed to 0. **Case $\boldsymbol{\Pi = \Pi_1 \otimes_g \Pi_2}$:** follows immediately from lemma XI(ii). **Case $\boldsymbol{\Pi =}$ $\dfrac{\boxed{\boldsymbol{\Pi'}}_{\boldsymbol{sc}}}{\boldsymbol{C}}\,\boldsymbol{D}$:** Let $\Pi' \vdash A_1, \ldots, A_n$, and let h be a history of Π. Let r be the rule applied to Π_1 to achieve Π. **Sequencing:** consider the premises of r: $P_1, \ldots, P_m$. Suppose $O(P_i) = \{A_{i1}, \ldots, A_{in_i}\}$ where the occurrences are ordered according to their sequencing. Then by the associativity and commutativity of binary assembly (propositions 4.3 and 4.4), Π' can be assembled from $\Pi_{11} \vdash A_{11}$, $\ldots$, $\Pi_{mn_m} \vdash A_{mn_m}$ by:

$$\Pi_1 \otimes_{g_1} (\Pi_2 \otimes_{g_2} (\cdots (\Pi_{m-1} \otimes_{g_{m-1}} \Pi_m) \cdots))$$

where $\Pi_i = \Pi_{i1} \otimes_{g_{i1}} (\Pi_{i2} \otimes_{g_{i2}} (\cdots (\Pi_{i(n_i-1)} \otimes_{g_{i(n_i-1)}} \Pi_{in_i}) \cdots))$. Now, by induction, for all i, j and histories h_{ij} of Π_{ij}, we get deduction sequences s_{ij} of $\underline{A_{ij}} \xrightarrow{t_{ij}} \Omega$ that traces Π_{ij} via h_{ij}. Now since we can apply the sequencing constraints specified by $P_1, \ldots, P_m$, it follows that there can be no interactions between any deduction in $\{\Pi_{i1}, \ldots, \Pi_{im_i}\}$. This means that $g_{i1} = g_{i2} = \ldots = g_{in_i} = 0$ for each i. Therefore, $\Pi_i = \bigotimes_0\{\Pi_{i1}, \ldots, \Pi_{in_i}\}$. By lemma XI(iii), we get for all histories h_i that sequence $\Pi_{i1}, \ldots, \Pi_{in_i}$ in Π_i a deduction sequence s_i of $\underline{A_{i1}}; \ldots; \underline{A_{in_i}} \xrightarrow{t_i} \Omega$ that traces Π_i via h_i. By definition, history h of Π must satisfy the sequencing constraints imposed by r, and therefore $h \upharpoonright O(\Pi_i)$ will sequence $\Pi_{i1}, \ldots, \Pi_{in_i}$ in Π_i. Therefore, by lemma XI(i) for every history h of Π restricted to Π', we know that there exists a deduction sequence s' of $\underline{P_1} \| \ldots \| \underline{P_m} \xrightarrow{t} \Omega$ that traces Π' via h.

Restriction: let $f : D(\Pi') \to \mathrm{dom}\,(s')$ be the associated injective function. Now, from the definition of deduction, we know $D(\Pi') \cap (O(\Pi') \times sc) = \emptyset$. Therefore by the properties of f, for no $\Sigma \in s'$, $act(\Sigma) \in sc$. Thus we can construct a deduction sequence s'' of $(\underline{P_1} \| \ldots \| \underline{P_m}) \backslash sc \xrightarrow{t} \Omega$ simply by applying the restriction rule to each deduction in s'. Moreover, s'' traces Π' via h.

Prefixing: Let $D = \{\alpha_1, \ldots, \alpha_n\}$. (By convention, $\alpha_1 \trianglelefteq \alpha_2 \trianglelefteq \ldots \trianglelefteq \alpha_n$). Thus we obtain a deduction sequence s''' by prefixing the following deduction sequence to s'':

$$\frac{}{\alpha_1.\alpha_2.\cdots.\alpha_n.p \xrightarrow{\alpha_1} \alpha_2.\cdots.\alpha_n.p} \cdot \frac{}{\alpha_2.\alpha_3.\cdots.\alpha_n.p \xrightarrow{\alpha_2} \alpha_3.\cdots.\alpha_n.p} \cdot \ldots \cdot \frac{}{\alpha_n.p \xrightarrow{\alpha_n} p}$$

Where $p = (\underline{P_1} \| \ldots \| \underline{P_m}) \setminus sc$. **Constants:** last we construct the deduction sequence s:

$$\cfrac{\cfrac{\begin{array}{c} s'''(1) \\ p \xrightarrow{\alpha_1} p' \end{array}}{\sum_{r \in \mathcal{R}(C)} \underline{r} \xrightarrow{\alpha_1} p'}}{\underline{C} \xrightarrow{\alpha_1} p'} \cdot s'''(2) \cdot \ldots \cdot s'''(|s'''|)$$

where p' is the right-hand side of the conclusion of $s'''(1)$. It should be clear from the interpretation that $D.(\underline{P_1} \| \ldots \| \underline{P_m}) \backslash sc$ is one of the summands of $\underline{C}$ in $\mathbb{E}(\mathcal{T})$.

Last, we have to show that s traces Π via history h. Since s'' traces Π' via h there must exist an injective function $f'' : D(\Pi') \to \mathrm{dom}\,(s'')$ with the required properties. We construct an injective function $f : D(\Pi) \to \mathrm{dom}(s)$ with the three properties. Let $f(C, \alpha_n) = n$ and $f(A, \alpha) = f''(A, \alpha) + m$ when $A \in O(\Pi')$, where $m = 1$ if $D = \emptyset$ or $m = n$ if $D = \{\alpha_1, \ldots, \alpha_n\}$. m counts the number of elements stuck on the front of s'' in the construction of s. Now there are three things to show. First, let $(A, \alpha), (B, \beta) \in D(\Pi)$ be such that $(A, \alpha) \leq_h (B, \beta)$. If both are also in $D(\Pi')$ then the result follows by induction: $f''(A, \alpha) + m < f''(B, \beta) + m$. If neither are, then $A = B = C$ and $\alpha \trianglelefteq \beta$ (but not vice-versa). Then $\alpha = \alpha_i$ and $\beta = \alpha_j$ where $i < j$. Hence $f(A, \alpha) < f(B, \beta)$. Otherwise, suppose that $(A, \alpha) = (C, \alpha_i)$ and $(B, \beta) \in D(\Pi')$. Then $f(A, \alpha) = i$ and $f(B, \beta) > m \geq i$.

The other two properties are even more straightforward by induction and the construction of s. $\square$

6.2.3 The reverse interpretation

We can also go in the reverse direction, and show that every CCS constant environment Δ corresponds to a DQI-system $\mathcal{T}_\Delta$ such that whenever process p reduces to 0, $\mathcal{T}_\Delta \vdash f(p)$ for some translation function f. The difficult approach would be to attempt to translate the constant equations in Δ into inference rules. This could be done. However, the easy way is just to add a cut rule

$$\frac{p \xrightarrow{t_1} p'' \qquad p'' \xrightarrow{t_2} p'}{p \xrightarrow{t_1 \cdot t_2} p'}$$

to the theory $\mathcal{C}_\Delta$. This translation is obviously correct. In the following, define the function f by $f(p,t,p') = p \xrightarrow{t} p'$.

Theorem XII *For every* $\Delta : Const \rightharpoonup \mathrm{CCS}$ *there exists a* DQI-*system* $\mathcal{T}_\Delta = (\mathcal{L}_\Delta, \mathcal{R}_\Delta)$ *and a function* $f : CCS \times Act^* \times \mathrm{CCS} \rightharpoonup \mathcal{L}_\Delta$ *such that* $p \xrightarrow{t} p'$ *if and only if* $\mathcal{T}_\Delta \vdash f(p,t,p')$ □

6.3 Some consequences

6.3.1 Sequencing is not primitive

It is easy to show that sequencing is not primitive. Let $\mathcal{T}$ be any QI-system, and $\mathcal{T}^*$ an equivalent DQI-encoding. Then $\mathcal{C}_{\Delta_{\mathcal{T}^*}}$ is an I-system. Let us extend it to $\mathcal{D}$ by adding judgments of form $p\surd$ and rules

$$\frac{p \xrightarrow{\overline{\mathrm{done}}} \Omega}{p\surd} \qquad\qquad \frac{p \xrightarrow{\tau} p' \qquad p'\surd}{p\surd}$$

Then we get that $\mathcal{T} \vdash A_1, \ldots, A_n$ if and only if $\mathcal{T}^* \Vdash A_1, \ldots, A_n$ if and only if there exists a $\mathcal{C}_{\Delta_{\mathcal{T}^*}}$-deduction sequence of $\underline{A_1} \| \ldots \| \underline{A_n} \xrightarrow{\overline{\mathrm{done}}} \Omega$, if and only if $\mathcal{D} \vdash \underline{A_1} \| \ldots \| \underline{A_n}\surd$. Therefore, if we define $\mathcal{T}^* = D$ and the family of functions f_n such that $f_n(A_1, \ldots, A_n) = \underline{A_1} \| \ldots \| \underline{A_n}\surd$ we obtain

Theorem IV *For all* QI-*systems* $\mathcal{T} = (\mathcal{L}, \mathcal{R})$, *there exists an* I-*system* $\mathcal{T}^* = (\mathcal{L}^*, \mathcal{R}^*)$ *and a family of injective maps* $f_n : \mathcal{L}^n \to \mathcal{L}^*$ *such that* $\mathcal{T} \vdash A_1, \ldots, A_n$ *if and only if* $\mathcal{T}^* \vdash f_n(A_1, \ldots, A_n)$ □

6.3.2 Deducibility is undecidable

The previous proof has the side-effect that for every p and every $i, j \geq 1$, every well-behaved (with respect to *Act*) terminating trace of p performs signal a_i as its last action. We can use this fact to help prove that there is no procedure for deciding if a formula is deducible in an arbitrary DQI-system. The proof goes via a reduction from the question "is a trace t a trace of process p?", which is undecidable in CCS. From the interpretation, we know that $\Pi \vdash A_1, \ldots, A_n$ if and only if there exists a deduction sequence s of $\underline{A_1} \| \ldots \| \underline{A_n} \stackrel{t \cdot \overline{\mathrm{done}}}{\to} \Omega$ for some t. When $t = \alpha_1 \cdot \ldots \cdot \alpha_m$ (for $m \geq 0$), this means that $A_1, \ldots, A_n$ is deducible iff $t \cdot \overline{\mathrm{done}}$ is a trace of $\underline{A_1} \| \ldots \| \underline{A_n}$.

Theorem XIII *There is no procedure to decide whether or not an arbitrary trace t is a trace of an arbitrary* CCS *process p.*

Proof: By a reduction from the halting problem for Turing Machines. □

Theorem XIV *Let $\mathcal{T}$ be an arbitrary* DQI*-system. Then there is no procedure that decides whether or not a set of occurrences is deducible or properly deducible in $\mathcal{T}$.*

Proof: Suppose not, i.e., that there is such a decision procedure. Then we construct a decision procedure to test whether a given trace is a trace of a given process given some constant environment Δ. Let $t \in Act^*$ and $p \in$ CCS. Then there must exist a p' such that $p \stackrel{t}{\to} p'$ in Δ. By theorem XII, this is true iff there exists an f such that $\mathcal{T}_\Delta \vdash f(p, t, p')$. By supposition, this is decidable. Contradiction. □

Corollary XIVa *There is no procedure to decide whether a given set of formulae is deducible in a given* I-*,* QI- *or* SDQI*-system.*

Proof: There cannot be such a procedure for SDQI-systems because the DQI-systems can be easily coded as SDQI-systems: we simply scope every rule by the empty set. By theorem VI, we have that QI-deduction is just proper DQI-deduction. By theorem IV, this in turn implies that there is no descision procedure for I-deduction either. □

6.3.3 Models

Another consequence of the interpretation is that we can use models of CCS to give a model-theoretic account of our notion of deduction, and therefore establish a model-theoretic account of the formulae of particular systems. For example, we have *event structures* [Win82,Win88,NPW81]. For other kinds of models, see [WN94].

What does our interpretation actually interpret? There are two related answers. The first is that it interprets deductions: a deduction is a terminating transition sequence (or an *evaluation*) of a particular process. Thus a denotational semantics of a deduction would be a trace, or set of traces of a particular process. The denotation of a formula A (via the Heyting semantics) would then be the set of all traces of the process $\underline{A}$.

The second answer is that it interprets formulae directly: the meaning of formula A is the process $\underline{A}$. Here, a denotational semantics for CCS (e.g., using labelled event structures [Win82]) gives a denotation to formulae directly: a formula is a labelled event structure.

Another model is the *synchronization tree*. If we attempt to evaluate process $\underline{A}$, we are attempting to find a deduction of A. Thus the *synchronization tree* of a process corresponds to the tableaux of a formula. The idea that the tableaux method for a deductive system gives its computational semantics is also found in more standard proof theory. Underwood [Und95] uses tableaux in this way to give semantics to intuitionistic predicate logic: "a bounded tableau search procedure can be interpreted as the computational content about prefixes of (possibly infinite) Kripke countermodels".

6.4 Chapter Summary

We have seen that DQI-deduction corresponds to the termination of CCS processes. This is philosophically appealing: it confirms that evaluation semantics judgments really do concern process evaluation. We can also reinterpret standard results about CCS as results about DQI-deduction. For instance, the undecidability of termination for CCS programs implies the undecidability of DQI-deduction. Moreover, we can use process calculus tools to manipulate deductions.

Chapter 7

Conclusions

In this thesis I have investigated a way of improving the pragmatics of operational semantics. Particularly, I have presented a way of decreasing the syntactic complexity of and improving the modularity of both semantic definitions and proofs about them. I achieved this by increasing the functionality of the notion of deduction. Thus I have traded off the complexity of the metatheory with the complexity of semantic definitions. However, the increase in complexity of the metatheory has been slight, and the result corresponds to a well-understood notion of computation. This seems a small price to pay.

7.1 The theory of interacting deductions

Standard proof theories are all essentially functional in flavour. An inference rule is seen as a function that delivers a proof of some result given proofs of its arguments. One strand of this thesis has been to ask what a concurrent flavour of proof theory would look like.

The basic theory The novel idea was to introduce the notion of interaction between inference trees. Thus a deduction would be a forest of interacting inference trees. In the basic theory of interacting deductions, a deduction would be represented by a pair (F, I), where F is the forest and I the interactions — pairs of formula occurrences. We saw that they obeyed a simple temporal property, that one cannot interact with a previous state. Deductions were built from simple interacting rules: finite sets of ordinary rules, each of which are meant to be applied to different parts of the forest simultaneously.

We saw that we could indeed improve the modifiability and modularity of definitions and proofs using the *proof fragmentation theorem*, which allows one to isolate those parts of a deduction relevant to the result at hand.

Extensions We saw some simple examples based upon a transition semantics. We gave some evaluation semantics too. However, to do so simply we needed to introduce a notion of sequentiality into the metatheory. The result, the theory of sequential deductions was easily codable in the basic theory. It simply provided a way to give evaluation semantics easily.

We also saw the method of pruning deductions. This was introduced to recover the notion of trace from the evaluation semantics. We also saw that it could be used for a number of purposes, for instance to characterize possible deadlock and nontermination, and to provide propagation-free abort rules. Another proof-theoretic feature used to capture various different semantic features was the scoping side-condition.

The feeling is that although the basic theory is expressive enough, sometimes simple features are so cumbersome to express that we need to add extensions ("tweaks") to the basic theory for the sake of simplicity. Whereas the notions of sequencing, pruning and scoping are straightforward, one wonders if they cover everything. Perhaps we shall need an infinite progression of ever more refined tweaks.

The fact that we require tweaks suggests that our proof theory is not quite right for our application. On the other hand, the fact that these tweaks seem to be ubiquitous suggests that we have discovered a standard repertoire of semantic techniques.

The content of interaction The basic theory is certainly flawed on two counts. First, when we wished to prove results by induction on the depth of inference of inference trees, we could not use the definition of deduction directly: we had to fragment deductions manually to obtain the individual trees. This occurred in proofs about both transition and evaluation semantics. Second, sometimes we had to develop a semantics-specific notion of visible occurrence to aid our proofs. In fact, the notion of visibility was actually a property of the structure of particular deductions than the semantics. An occurrence was visible if it should have been part of an interaction link, but was not. So we introduced the notion of *dangling interaction*, which gave a formal treatment of the content of interactions.

This extension proved fundamental. It achieved an elegant and usable definition of deduction which — like Natural Deduction — included the notions of inference (rule application) and the mechanism of assumption and discharge (assembly). This refinement of deduction also gave rise to the technique of *proof assembly*, which was dual to proof fragmentation. Fragmentation showed how we could break deductions to reason about individual parts. Assembly showed how we could construct proofs about deductions from proofs of individual parts.

What deductions are Last, we saw that our resultant theory was not *ad hoc*: the computational semantics of a deductive system was a set of CCS constant definitions. Formulae corresponded to processes and deductions corresponded to terminating evaluations of processes. In this framework, we saw that dangling interactions were just action prefixes (and thus assembly was communication), scoping was restriction, sequencing was sequential composition and pruning was just unfinished evaluation.

CCS is well-understood and foundational in the theory of concurrency. Therefore the correspondence between CCS and interacting deductions lends credence to our belief that we have a metatheory suitable for giving semantics to concurrent languages. It is also philosophically attractive for at least two other reasons. First, in terms of our application, it showed that since deductions were process evaluations, then evaluation judgments really were about evaluation. Second, it allows us to reinterpret results about CCS in our setting. For example, we saw that deducibility was undecidable. Another result was the reinterpretation of models of processes as models of deduction. Practically, the correspondence also suggests how we may use process calculus simulators to provide prototype interpreters for languages.

7.2 Application to operational semantics

Our initial examples concerned transition semantics of a simple process calculus P. We saw that we did not need to propagate action information to obtain a definition of processes. However, we also saw some limitations of not doing so: we could not limit the number of interactions to one per transition, and anyway action information is very useful in process calculus applications.

Nevertheless, propagation-freeness can be useful. We saw that we could add a notion of store to the various semantics of P almost uniformly. Moreover we obtained a modular proof about the stores which held true without alteration for each of the semantics.

We then considered evaluation semantics. We saw that we could give evaluation semantics, but not in as simple a way as we might like. Along the way we discovered a distributed control-stack semantics which could be useful for languages with first-class continuations.

We introduced the notion of sequentiality to permit simple presentations of evaluation semantics. The resulting evaluation semantics of P(;) (P extended with sequential composition) was much more concise than the standard operational semantics. However, the results were not so clear in the example translation correctness result. Negatively, the only equivalence easily definable in the evaluation semantics was a trace equivalence (and even then we needed the notion of pruning). This equivalence is acceptable for deterministic languages, and indeed many "real" programs are deterministic. Positively, we saw that the evaluation semantics proof was less structurally complex than the transition semantics proof. It contained no subinductions on the length of transition sequences.

We saw another, more positive use of evaluation semantics in the soundness proof of our Hoare Logic for the partial correctness of CSP. The evaluation judgments were ideally suited to interpret Hoare triples $\{\phi\}\ p\ \{\psi\}$ which concern the effect of the evaluation of p. The proof was easy, and it could have been made even easier if we had not fragmented stores from evaluation judgments!

We also considered how many different linguistic features we could capture in an evaluation semantics. Most of the features we considered could be captured:

shared variables, dynamic process creation, nested parallelism, multicasting and procedures. This latter case showed that while we could add procedures in a modular fashion, the result was a mess. In particular, it feels as if environments ought always to be bound to program judgments.

One kind of feature we cannot expect to model are pre-emptive actions that must occur instantaneously. Another kind of feature we cannot model easily are those that require a snapshot of the entire system (e.g., broadcasting).

7.3 Further work

In the body of the thesis, we noted several possible avenues of further research. Chapter two suggested that we might be able to make a more careful analysis of the kinds of propositions provable via proof fragmentation (and in retrospect also proof assembly). Perhaps there are further techniques to be discovered. Chapter three suggested that the coinductive notion of deduction might be useful in the field of abstract interpretation [CC77,Sch95] for concurrent languages. Chapter five noted some improvements to our Hoare Logic. Last, chapter six suggested that we could use process calculus simulators to implement semantic definitions, and also to describe models of deduction.

In addition, there are at least four other topics which suggest themselves: scoping, static semantics, exploration of further proof-theoretic extensions and the relationship to linear logic.

7.3.1 Scope Extrusion

Our scoping side-condition corresponds to CCS-like restriction. However, there are languages (such as the π-calculus [MPW92], CONCURRENT ML [Rep91a]) which require a more liberal kind of scoping, which permits *scope-extrusion*, where scoped channels may be communicated out of scope.

We have stressed the idea that deduction order is opposite to evaluation order. To this way of thinking, scope extrusion (at least for the appropriate variant of P) is not a problem: when we come to apply the (appropriate variant of) the scoping condition, we shall already know how the scoped channels have been used, and therefore we can widen the forest of trees which are scoped to include those parts of those trees to which scope has been extruded.

For functional languages, where the evaluation judgment can return a value (including a channel) the situation becomes more complex. There scope can be extruded in two ways: first by communication, and second by function value. For example, *channel x in x* will return channel x, even though it will be out of scope whenever it is subsequently used.

To capture scope extrusion in this setting would require a finer notion of judgment and preorder between judgments. For instance, an evaluation judgment would be split into a simple kind of sequent: $A \Rightarrow B$, where A was an expression and B its value. The point here would be that the meaning of dependency would alter. If $A \Rightarrow B$ occurred above $A' \Rightarrow B'$ in a forest F, then A' would precede A, and both A and B would precede B'. Thus evaluation would be seen as "going up" the left-hand side of judgments and "going down" the right-hand side. To capture scope extrusion, we would have to scope left-hand sides only, and ensure that every occurrence that came after it in the preorder obeyed the scoping conditions. Of course, we should always have to make sure that rules applied subsequently which used a scoped channel did not violate the required condition.

7.3.2 Static semantics

In this thesis we have concentrated primarily on the use of interaction in dynamic semantics. It seems harder to use interaction usefully in static semantics. One exception would be *name servers*. In the definition of STANDARD ML, the static semantics have to pass around the set of currently generated type names so that whenever a new name is required (e.g., rules (19) and (20) on page 25), it is guaranteed not to clash with a previous choice of name.

Another way to capture this idea of "generativity" would be to include special "fresh" judgments:

$$\frac{}{\mathrm{fresh}(\alpha)} \quad \rule{2em}{0.4pt} \quad \frac{\mathrm{free.names}(X \setminus \{\alpha\})}{\mathrm{free.names}(X \cup \{\alpha\})} \qquad \frac{}{\mathrm{free.names}(X)}$$

The idea would be that each invocation of the fresh judgment would pick a different name from the global set of free names. Of course, such a scheme would require that only one name server existed per deduction.

7.3.3 More Extensions

Relabelling

We consider the use of scoping to extend P with relabelling. The syntax is extended to get P($[f]$):

$$p ::= \cdots \mid p[f]$$

where $f : \mathcal{A} \to \mathcal{A}$ is a relabelling function (see section 6.1.2). The simplest way to model CCS-like relabelling would be to propagate the relabelling function f upwards, and apply it directly at the communication rule, for instance as in:

$$\frac{p\surd_f}{a.p\surd_f} \quad \rule{2em}{0.4pt} \quad \frac{q\surd_g}{\bar{b}.q\surd_g}\ f(a) = g(b)$$

However, we can also find DQI-rules which avoid propagation. These make use of dangling interactions in an essential way. The idea is that each communication signal is routed through a series of "relabelling stations" until it is received.

$$\frac{p\surd}{a.p\surd}\ {+comm(p,a)} \qquad \frac{q\surd}{\bar{a}.q\surd}\ {-comm(p,a)} \qquad \frac{Rel(q,f) \quad -comm(p,a)}{Rel(q,f) \quad +comm(q,f(a))}$$

$$\frac{\underbrace{p\surd \quad Rel(p,f)}_{}\ {}_{REL(p,f)}}{p[f]\surd} \qquad \frac{}{Rel(p,f)}$$

Where $REL(p, f) = \{comm(q, a) \mid q \in \mathrm{P}([f]),\ p \neq q,\ a \in Act\}$. The label $comm(p, a)$ is meant to denote that the action a has been performed, and is associated with the process p. This process tag is meant to distinguish different occurrences of the same relabelling function, e.g., so that $p[f][f]$ is guaranteed to relabel every action of p via f twice.

Once again, we have found that the evaluation semantics of a straightforward concurrency primitive is not so straightforward. One obvious proposal would be then to extend the metatheory of deduction to contain relabelling. At first glance, this appears quite attractive. First, it would not ruin our interpretation of deduction — relabelling is already a feature of CCS. Second, it would improve the modularity of DQI-deduction again: e.g., we might be able to reuse the algebraic definition of stores for some other storage device.

However, it is hard to see what relabelling corresponds to logically. If Roger participated in a discussion about A, then his interlocutor was not discussing $f(A)$ (unless f is the identity of course). Talk of "translation functions" is fatuous: one cannot guarantee $+A$ with $-\neg A$. The lack of a deduction-theoretic explanation of relabelling is a serious obstacle to its institution in the meta-theory of deduction.

Metatheories ad nauseam

We introduced sequencing into the metatheory of deduction because although we could capture sequential composition in an evaluation semantics I-system, we could not capture it simply. We have also seen several other similar proposals. For example, we have the above relabelling proposal. Another example is the sequent-style rules for scope extrusion. Another concerns the propagation-free abort rules which made use of pruning rules. The problem here is that the result was not neat. We might be tempted to introduce the notion of escaping into our metatheory, perhaps using the idea of *continuation proof* [Und93, §5.3.5]).

There are two questions that concern this drive for better metatheories of evaluation semantics. The first is how many different extensions such a philosophy would need. At some point we have to draw the line and say "this is our theory of deduction". To some extent I have done this already: the fact that the theory of interacting deductions corresponds to the well-known calculus CCS suggests that we have a reasonable notion of deduction which is expressive enough.

The second question concerns the pragmatic value of such a philosophy. The law of diminishing returns tells us that the more we tweak our metatheory the less gains we shall achieve. We have already seen that evaluation semantics has limited value for concurrency: the only equivalence we can talk easily about is trace equivalence. This suggests that it is not a suitable medium for analysing nondeterminism. Of course, that is not to say that evaluation semantics for concurrency has no merit. Similarly, more complex theories of deduction (provided they are comprehensible) may also have meritorious applications.

There will always be semantic features that are hard to express in metatheories that are not tailored to express them. Perhaps one solution to the problem might be to consider a universe of metatheories of deduction and their relationships, perhaps in a manner similar to Moggi's use of monads for modularity in denotational semantics [Mog91]. Maybe Milner's work on action calculi and control structures [Mil,MRP] is a suitable framework.

7.3.4 The relationship to Linear Logic

Our notion of deduction is inherently linear: each formula occurrence in a deduction is the conclusion of one rule and the premise of at most one rule. In this section we consider the relationship between linear logic [Gir87,Gir89] and interacting deductions. It turns out that I-deduction is fairly easy to characterize in linear logic, using just tensor product and linear implication. QI-deduction is harder to characterize: it requires two tensors, one commutative and the other non-commutative. There does not seem to be any simple logical account of DQI-deduction.

A logical treatment of I-deduction The following account is a logical perspective of I-deduction owing to Gordon Plotkin. Let Γ, Δ range over multisets of formulae (see figure 2–2 for a formal account of multisets). In this context, I write Γ, Δ for the multiset union of Γ and Δ and 1 for the empty multiset. Then we define $\vdash \Delta$ inductively using the following rules:

$$\vdash 1$$

$$\frac{\vdash \Gamma_1, \ldots, \Gamma_n, \Delta}{\vdash A_1, \ldots, A_n, \Delta} \qquad \text{when} \qquad \left\{ \frac{\Gamma_1}{A_1}, \ldots, \frac{\Gamma_n}{A_n} \right\} \qquad \text{is an I-rule.}$$

Thus $\vdash \Gamma$ if and only if Γ is deducible in the ambient system of rules. This characterization is logically straightforward, but it is not amenable to the technique of fragmentation.

Now, if we code multisets of formulae $\Gamma = A_1, \ldots, A_n$ by the intuitionistic linear formula $\phi_\Gamma = A_1 \otimes \ldots \otimes A_n$, and we code rules $r = \{\frac{\Gamma_1}{A_1}, \ldots, \frac{\Gamma_n}{A_n}\}$ by the formula

$$\phi_r = (\phi_{\Gamma_1} \multimap A_1) \otimes \ldots \otimes (\phi_{\Gamma_n} \multimap A_n)$$

Then I believe that $\vdash \Gamma$ (under ambient rules $r_1, \ldots, r_n$) if and only if $!\phi_{r_1}, \ldots, !\phi_{r_n} \vdash \phi_\Gamma$ is provable in intuitionistic linear logic. Apart from helping us to understand I-deduction better, another consequence of this result would be to use linear logic programming languages such as FORUM [Mil94] or LOLLI [HM94] or others [Milar] to build prototype interpreters. Perhaps also it could inspire a coherence-space semantics [GLT89].

A logical treatment of QI-deduction The main problem in giving a simple inductive account of QI-deduction is that sequencing and rule application are not distinct operations. In our definition, we chose to say that premises were sequenced after conclusions (given that the evaluation order seems to be opposite to deduction order). Any definition of QI-deduction must take this into account. I cannot think of a simple way to do this here, although I can for the DQI-deductions.

The main problem in giving a logical account is that we require two tensors: one commutative tensor as before and one non-commutative tensor for sequencing. Retoré [Ret93] studied such a linear logic, introducing the *précède* tensor $<$ which was both self-dual (i.e., $(A < B)^\perp = (A^\perp < B^\perp)$) and in-between tensor and par: $A \otimes B \multimap A < B$ and $A < B \multimap A \parr B$. Reddy [Red93b,Red93a] uses this work to help him develop a logic for reasoning about state in imperative programming languages. Here, sequent calculus contexts correspond not to multisets but to *partially-ordered multisets*. Reddy writes the non-commutative tensor $\triangleright$. Using his sequent calculus LLMS, we could code a Q-atom

$$\frac{P_{11} \blacktriangleright \ldots \blacktriangleright P_{1n_1} \quad \cdots \quad P_{m1} \blacktriangleright \ldots \blacktriangleright P_{mn_m}}{C}$$

as

$$[(P_{11} \triangleright \ldots \triangleright P_{1n_1}) \otimes \ldots \otimes (P_{m1} \triangleright \ldots \triangleright P_{mn_m})] \multimap C$$

I do not know, but I think it would not be hard to prove that in this setting QI-deducibility corresponded to provability in LLMS. I do not know of a logic programming language that could be used to build prototype interpreters, although we may be able to appropriate Retoré's coherence space semantics for models.

A logical treatment of DQI-deduction We have suggested that I-deducibility can be captured in intuititionistic linear logic and QI-deducibility can be captured in the more exotic linear logic model of state. However, it is not clear what kind of logic could capture DQI-deduction. The problem is that it is not clear how to treat dangling interactions and assembly. One solution might be to use a classical linear logic (i.e., introduce linear negation). This would use linear negation to distinguish opposite perspectives of dangling interactions, which would be treated as logical hypotheses. Thus we may have a sequent calculus style presentation of DQI-deduction (in much the same way that we can get sequent calculus style presentations of natural deduction [Sun84b]). For this we need a grammar of contexts which allows arbitrarily deep nesting of sequential and parallel composition of formulae:

$$\begin{array}{l} \Gamma ::= 1 \;\mid\; A \;\mid\; \Gamma, \Gamma \;\mid\; \Gamma; \Gamma \\ \Gamma[\cdot] ::= [\cdot] \;\mid\; \Gamma, \Gamma[\cdot] \;\mid\; \Gamma[\cdot], \Gamma \;\mid\; \Gamma; \Gamma[\cdot] \;\mid\; \Gamma[\cdot]; \Gamma \end{array}$$

and the following rules:

$$\frac{}{\vdash 1} \qquad \frac{\Gamma \vdash \Delta \qquad \Gamma' \vdash \Delta'}{\Gamma; \Gamma' \vdash \Delta; \Delta'} \qquad \frac{\Gamma \vdash \Delta \qquad \Gamma' \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}$$

$$\frac{\Theta \vdash \Gamma}{D; \Theta \vdash A} \qquad \text{when} \qquad \frac{\Gamma}{A}\, D \qquad \text{is a DQI-rule.}$$

$$\frac{\Gamma[A, A^\perp] \vdash \Delta}{\Gamma[1] \vdash \Delta} \qquad \frac{\Gamma[(\Gamma_1; \Delta_1)\,,\, (\Gamma_2; \Delta_2)] \vdash \Delta}{\Gamma[(\Gamma_1, \Gamma_2)\,;\, (\Delta_1, \Delta_2)] \vdash \Delta}$$

plus rules for the commutativity and associativity of ',', the associativity of ';' and the unity of 1. There are three interesting rules. The first is the coding of DQI-rules: this rule clearly shows how rule application corresponds to sequencing. The left hand side of the conclusion, $D; \Theta$ shows that interactions occur before premises are evaluated. The other choice, represented by $\Theta; D$, shows that interactions occur after premise evaluation.

The other two interesting rules are the assembly rules (the left-hand side rules). The first "assembles" one interaction link. The second schedules interaction links.

We could, perhaps, use higher-order linear formulae to encode the sequents $\Gamma \vdash \Delta$ as $\phi_\Gamma \multimap \phi_\Delta$ (where ϕ_Γ is some suitable encoding of context Γ). But then it is hard to see how to code rules: how do we distinguish rule premises and dangling interactions?

Appendix A

Languages of operational judgments

In this appendix, we give a brief account of the notion of languages of terms and formulae. In section A.1 we give an axiomatic description. In section A.2 we give an algebraic account, and sketch its use in operational semantics.

A.1 A formal definition of language

This definition comes from [Gar95]. A (many-sorted) language $\mathcal{L}$ is a quintuple (S, L, V, FV, sub) where S is a set of sorts, L is an S-sorted family of sets of *formulae* (ranged over by $A, B, C, \ldots$), V is an S-sorted family of sets of *variables* that appear in L (ranged over by x, y), $FV : L_\top \to \wp(V_\top)$ is an S-sorted function that returns the *free variables* of a formula, and $sub : (V \to L) \to L \to L$ is a function that applies a *substitution* $\theta : V \to L$ to a formula, such that

1. If $A \in L_s$ then $(sub\ \theta\ A) \in L_s$
2. If $x \in V_s$ then $(sub\ \theta\ x) = \theta_s x$
3. $sub\ id = id_L$
4. $(sub\ \theta_0) \circ (sub\ \theta_1) = (sub\ \theta_3)$ where for $x \in V_\top$, $\theta_3 x = sub\ \theta_0\ (\theta_1 x)$
5. $FV(x) = \{x\}$
6. $FV(sub\ \theta\ A) = \bigcup_{x \in FV(A)} FV(\theta x)$
7. If $\theta_0 \restriction (FV(A)) = \theta_1 \restriction (FV(A))$ then $sub\ \theta_0\ A = sub\ \theta_1\ A$

We abbreviate $sub\ \theta\ A$ by θA, and the composition of substitution θ_1 after θ_0 by $\theta_1\theta_0$. If $\mathcal{L}$ is a language, we write $A \in \mathcal{L}$ if A is a formula of $\mathcal{L}$. Similarly, we often treat $\mathcal{L}$ as if it were just a(n S-sorted) set of formulae.

A.2 Algebras and operational semantics

An operational semantics consists of a language of judgments that relate to the evaluation of programs, and a set of rules that build proofs of these judgments. In this section, we describe how one can use many-sorted algebraic signatures with equations to specify an arbitrary language of judgments.

A judgment consists of some evaluation symbol (e.g., $\Rightarrow$) and one or two *machine configurations*. That is, a judgment will be a term of the algebraic signature (i.e., an element of the term algebra).

A configuration consists of a program and a machine state, which may be composed of various different entities (e.g., a store, an environment or an input-output buffer). For want of a better term, we call these entities *machine components*.

We summarize Wechler's [Wec92, ch4] account of Σ-algebras here. The reader familiar with universal algebra may safely skip this section. We use it only to describe judgment-formulae and the notions of substitution and instantiation which make up the notion of language described in section 2.1. Further details can be found in [Coh65,Grä79,GB90,ST99].

Sorted families of sets Let S be a set of sorts, and X a set. Then an *S-sorted family of subsets of X* is a function $F : S \to \wp(X)$. For $s \in S$, we write F_s for $F(s)$. We also define $F_\top = \bigcup_{s \in S} F_s$. When F and G are S-sorted families of sets, an *S-sorted function* from F to G (written $f : F \to G$) is an S-sorted family of sets of functions $f_s : F_s \to G_s$. When F is an S-sorted family of sets, the *identity* function id_F is the S-sorted family of identity functions: $id_s : F_s \to F_s$.

When S is a set, an *S-sorted* (or *S-indexed*) family A of sets of type T is a function $A : S \to \wp(T)$. We write A_s for the set of type T corresponding to sort s. We define the union, intersection, powerset operations over indexed families of sets pointwise. E.g., $(A \cup B)_s = A_s \cup B_s$. We use $\wp_{fin}$ for the finite powerset operation.

Σ-Algebras. A *(many sorted) algebraic signature* Σ is a pair $\langle S, F \rangle$ where S is a set of *sorts*, and F is an $S^* \times S$-sorted family of sets of *function names*. We abbreviate $f \in F_{s_1,\ldots,s_n,s}$ as $f : s_1 \cdots s_n \to s$. When $n = 0$ we write $f : s$ and f is said to be a *constant*.

A Σ-*algebra* is a pair $\langle A, F \rangle$ where A is an S-sorted set and F is a family of operations such that each operation symbol $f : s_1 \cdots s_n \to s$ of Σ is realized as an operation $f^A : A_{s_1} \times \cdots \times A_{s_n} \to A_s$.

An algebraic signature Σ defines a set of Σ-*terms* that may be constructed from metavariables and function names. Let $\Sigma = \langle S, F \rangle$ be an algebraic signature, let V be an S-sorted family of countably infinite sets of metavariable universes, and X an S-sorted set of metavariables from V. Then, for $s \in S$, we say $T_\Sigma(X)_s$ is the set of terms of sort s with metavariables in X. It is defined inductively:

- if $x \in X_s$ then $x \in T_\Sigma(X)_s$

- if $f \in F_{s_1,\ldots,s_n,s}$ and $t_1, \ldots, t_n$ are in $T_\Sigma(X)_{s_1}, \ldots, T_\Sigma(X)_{s_n}$ respectively, then $f(t_1, \ldots, t_n) \in T_\Sigma(X)_s$.

The set of terms $T_\Sigma(X)$ forms the *term-algebra over* X whose operations are just the term constructors. We define the *ground* terms to be members of the set $T_\Sigma(\emptyset)$ which is written T_Σ.

A *substitution* θ is an S-sorted family of total functions such that $\theta_s : X_s \to T_\Sigma(X)_s$. The θ-*instance* of term t is given by the recursively defined function $\mathcal{I}_s : (X \to T_\Sigma(X)) \to T_\Sigma(X)_s \to T_\Sigma(X)_s$:

- $\mathcal{I}_s\ \theta\ x = \theta_s(x)$ when $x \in X_s$
- $\mathcal{I}_s\ \theta\ f(t_1, \ldots, t_n) = f(\mathcal{I}_{s_1}\ \theta\ t_1, \ldots, \mathcal{I}_{s_n}\ \theta\ t_n)$ when $f : s_1 \cdots s_n \to s$.

For brevity, we write θt for $\mathcal{I}_s\ \theta\ t$.

We show how Σ-algebras with equations can describe the syntax of programming languages, the structure of and operations over machine components, and hence the structure of semantic judgments.

Program Syntax We use Σ-algebras to give the syntax of programming languages. The sorts are the syntactic classes and the operations the productions of a grammar. For instance, the simple language with grammar

$$\begin{aligned} e &::= x \mid n \mid e_0 + e_1 \\ c &::= x := e \mid c_1 ; c_2 \end{aligned}$$

is represented by the algebra with sorts $S = \{e, c, Var, \mathbb{Z}\}$ where Var is a set of variable names and $\mathbb{Z}$ is the set of integers, and functions

$$\begin{aligned} &+ : e \times e \to e \\ &:=: Var \times e \to c \\ &; : c \times c \to c \end{aligned}$$

The set of terms over this signature corresponds to the set of syntax trees of the language, which may contain metavariables. One does not usually identify non-identical terms, although it can sometimes be useful to do so (see, for example [Mil91, p25]).

Machine components We use many-sorted algebraic signatures with equations to define the various components that record the state of an interpreter. We use equations to define various primitive operations over them. For example, simple stores can be defined by the following signature:

srts:	$Store, Var, \mathbb{Z}$	metavars:	$\sigma, \sigma', \ldots, \sigma_0, \sigma_1, \ldots$
fns:	$\mathtt{empty} : Store$	eqns:	$\mathtt{empty}(x) = 0$
	$\cdot[\cdot/\cdot] : Store \times \mathbb{Z} \times Var \to Store$		$\sigma[n/x](x) = n$
	$\cdot(\cdot) : Store \times Var \to \mathbb{Z}$		$\sigma[n/x](y) = \sigma(y)$ if $y \neq x$

Many sorted equations Given an algebraic signature Σ, an *equation of sort s* is a triple $\langle X, t_0, t_1\rangle$ usually written $\forall X(t_0 = t_1)$ or just $\forall x_1 : s_1, \ldots, x_n : s_n . t_0 = t_1$ when $x_i \in X_{s_i}$ for all $i = 1, \ldots, n$, or even $t_0 = t_1$ when the universal quantification of the variables is understood.

We can consider a set of equations E as a family $\bar{E}$ of $\wp_{fin}(V)$-indexed family of S-sorted families $E(X)$ of sets $E(X)_s = \{(t_0, t_1) \mid \forall X(t_0 = t_1) \in E_s\}$.

A *clone congruence* R is a family of S-sorted families of relations $R(X)$ where X_s is a finite subset of V_s for all s, which are congruences invariant under α-conversion. The full details are in [Wec92, §4.1]. The importance of clone congruences come from the *generalized Birkhoff criterion for equational theories*: A set E of equations is an equational theory (i.e., describes some abstract class of algebras) if and only if $\bar{E}$ is a clone congruence. Of course, not every set of equations E will describe a clone congruence in this way until we close them with respect to reflexivity, symmetry, transitivity, congruence and α-equivalence.

The ground equational theory of E, written $G(E)$ is an S-sorted relation such that $G(E)_s = \{(\theta t_0, \theta t_1) \mid \forall X(t_0 = t_1) \in \bar{E}_s, \theta : X \to T_\Sigma\}$. That is, it is the set of pairs of equal ground terms (terms without variables).

Then the initial semantics $T_{\langle\Sigma,E\rangle}$ of $\langle\Sigma, E\rangle$ is the class $T_\Sigma/{=_{G(E)}}$, where $=_{G(E)}$ is the clone congruence generated by the ground equational theory of E. The initial semantics of an abstract syntax (without equations) will be a set of equivalence classes, each of which being singleton terms (representing programs). The initial semantics for the store will be sets of equivalence classes of stores and of values. Each store equivalence class will be singleton since distinct stores are not identified. For each value (here, integer) there will be an equivalence class containing every store lookup expression that equals it. So, for instance, the equivalence class of the value 3 will contain the values 3, $\sigma[3/x](x)$, $\sigma[3/y][4/x](y)$ etc.

Directed equations We can read the equations for stores operationally by directing the equations to make a *term rewriting system*:

$$\begin{array}{rll} \forall x : Var. & \mathtt{empty}(x) \to 0 & \\ \forall x : Var, n : \mathbb{Z}, \sigma : Store. & \sigma[n/x](x) \to n & \\ \forall x : Var, y : Var, n : \mathbb{Z}, \sigma : Store. & \sigma[n/x](y) \to \sigma(y) & \text{if } y \neq x \end{array}$$

We do not bother to give the details here: these may be found in [Wec92, §4.1.3].

Appendix B

The correctness of $\mathcal{CSP}$

In this appendix, we outline the main details of an equivalence between the semantic definition of CSP given in chapter 5. This should increase our confidence that that semantics for CSP is correct.

This equivalence shows how terminating sequences of a cut-down version of Plotkin's Structural Operational Semantics for CSP [Plo83] corresponds to the $\mathcal{CSP}$ evaluation judgments. The proof proceeds via three intermediate semantic definitions. Section B.1 presents the SOS, $\mathcal{CSP}_{SOS}$, which gives an interleaving semantics. Section B.2 presents a more truly concurrent semantics $\mathcal{CSP}_{MSOS}$, and indicates their equivalence. Section B.3 presents an interacting transition semantics, $\mathcal{CSP}_I$, and indicates its equivalence with $\mathcal{CSP}_{MSOS}$. These results mirror those in section 2.5.3. Section B.4 is the main part which shows how an extenstion of $\mathcal{CSP}_I$ (called $\mathcal{CSP}_{CUT}$) relates to $\mathcal{CSP}$.

We shall assume all of the auxiliary definitions presented in chapter 5, as well as some new ones, defined shortly. In particular, the grammar is as before, as is the static semantics.

In this appendix, we shall use Plotkin's convention for abbreviating families of transition rules (also in [Plo83]. When $n \geq 1$,

$$\frac{\gamma \xrightarrow{\lambda} \gamma_1 \mid \ldots \mid \gamma_n}{\gamma' \xrightarrow{\lambda'} \gamma'_1 \mid \ldots \mid \gamma'_n}$$

abbreviates the n rules

$$\frac{\gamma \xrightarrow{\lambda} \gamma_1}{\gamma' \xrightarrow{\lambda'} \gamma'_1} \qquad \ldots \qquad \frac{\gamma \xrightarrow{\lambda} \gamma_n}{\gamma' \xrightarrow{\lambda'} \gamma'_n}$$

and for $m, n \geq 1$,

$$\frac{\gamma' \xrightarrow{\lambda'} \gamma'_1 \mid \ldots \mid \gamma'_m \qquad \gamma'' \xrightarrow{\lambda''} \gamma''_1 \mid \ldots \mid \gamma''_n}{\gamma \xrightarrow{\lambda} \gamma_{11} \mid \ldots \mid \gamma_{1n} \mid \ldots \mid \gamma_{m1} \mid \ldots \mid \gamma_{mn}}$$

abbreviates the $m \times n$ rules

$$\frac{\gamma' \xrightarrow{\lambda'} \gamma'_1 \qquad \gamma'' \xrightarrow{\lambda''} \gamma''_1}{\gamma \xrightarrow{\lambda} \gamma_{11}} \quad \cdots \quad \frac{\gamma' \xrightarrow{\lambda'} \gamma'_1 \qquad \gamma'' \xrightarrow{\lambda''} \gamma''_n}{\gamma \xrightarrow{\lambda} \gamma_{1n}}$$

$$\cdots$$

$$\frac{\gamma' \xrightarrow{\lambda'} \gamma'_m \qquad \gamma'' \xrightarrow{\lambda''} \gamma''_1}{\gamma \xrightarrow{\lambda} \gamma_{m1}} \quad \cdots \quad \frac{\gamma' \xrightarrow{\lambda'} \gamma'_m \qquad \gamma'' \xrightarrow{\lambda''} \gamma''_n}{\gamma \xrightarrow{\lambda} \gamma_{mn}}$$

In this appendix, we treat I-systems as special cases of DQI-systems. This is purely to facilitate the equivalence proof.

B.1 A Structural Operational Semantics of CSP

This section gives a transition semantics $\mathcal{CSP}_{SOS}$ for CSP. It is based on Plotkin's Structured Operational Semantics for CSP [Plo83], but differs in three ways. First, the parallel composition of processes occurs only at the top level. Second, because of this, we have to model partially aborted networks of processes differently. When one process aborted, Plotkin simply sequenced another `abort` command after the rest of the network. Here we use a different trick: we extend the syntax of programs to include a special "aborted process" production:

$$p ::= R :: c \mid p \parallel p \mid \texttt{aborted}$$

together with the equations $\texttt{aborted} \parallel \texttt{aborted} = \texttt{aborted}$. This addition is a convenience to allow us to represent partially aborted networks of processes.

The third difference is that expressions (and hence guarded commands) cannot abort.

B.1.1 The Labelled transition system

The following labelled transition systems are required for CSP. We require one each for expressions, guarded commands, commands and programs. These make use of the following communication labels.

Communication labels

The first definition comes straight from [Plo83]. To model communication, we define the set of labels

$$\Lambda = \{P?v, Q :: P?v, Q!v, P :: Q!v \mid P, Q \in Pid, P \neq Q, v \in Val\} \cup \{\varepsilon\}$$

ranged over by λ. Labels consist of three parts. The first (optional) part denotes the name of the process that has performed the communication (the *agent*). The second denotes the name of the other process involved in the communication. The third denotes the message that was sent. No label has form $P :: P!v$ (or $P :: P?v$). Such labels would correspond to the action of self-communication, which is impossible for synchronized communication.

Configurations

We require four transition systems: one each for the expressions, guarded commands, commands and programs. From the informal account of the semantics of CSP, an expression configuration can either be an expression waiting to be evaluated or its value. A guarded command can either fail or select a command that can proceed. Selecting a command involves making its first step, which may abort. Commands may either terminate or abort, as may programs. The following table expresses this more formally.

Syntactic class	(L)TS	Terminal Config'ns (Ω)	Config'ns (Γ)
expressions	$\langle \Gamma_e, \Omega_e, \longrightarrow_e \rangle$	Val	$Exp \times Store \ \cup \ \Omega_e$
g'd commands	$\langle \Gamma_g, \Omega_g, \Lambda, \longrightarrow_g \rangle$	$Com \times Store \cup Store$ $\cup \{\texttt{failure}, \texttt{aborted}\}$	$GCom \times Store \cup \Omega_g$
commands	$\langle \Gamma_c, \Omega_c, \Lambda, \longrightarrow_c \rangle$	$Store \cup \{\texttt{aborted}\}$	$Com \times Store \ \cup \ \Omega_c$
programs	$\langle \Gamma_p, \Omega_p, \Lambda, \longrightarrow_p \rangle$	$Store \cup \{\texttt{aborted}\}$	$Prog \times Store \cup \Omega_p$

The transition rules can be found in figures B–1 and B–2, where we have dropped the subscripts from the transition relations.

Before we give the rules for the programs, we describe a partial function over labels, $R :: (\cdot) : \Lambda \rightharpoonup \Lambda$, which takes a label of form $P?v$ or $P!v$ and adds $R ::$ in front if $R \neq P$. If the label is empty, it is undefined — thus $\{\!| R :: \varepsilon |\!\} = \emptyset$.

$$R :: (\lambda) = \begin{cases} R :: P?v & \text{if } \lambda = P?v \text{ and } P \neq R \\ R :: Q!v & \text{if } \lambda = Q!v \text{ and } Q \neq R \end{cases}$$

Figure B–2 lists the transition rules for programs. We do not discuss them here: the details are just those of [Plo83].

B.2 Stage one: from interleaving to true concurrency

The first step of the equivalence proof is essentially just that of section 2.5.3 — where we present a transition I-system for CSP, and prove it equivalent to the previous structured operational semantics. Just as before, this proof is factored into two: first between $\mathcal{CSP}_{SOS}$ and a semantics $\mathcal{CSP}_{MSOS}$ which allows multiple communications to occur simultaneously; second, between this intermediate language and the I-system $\mathcal{CSP}_I$. In this section we give $\mathcal{CSP}_{MSOS}$ and state the equivalence with $\mathcal{CSP}_{SOS}$. Modulo some difficulty with stores, it is no more exciting than before.

The two systems only differ in the rules for programs. Figure B–3 lists the rules for programs. Again, each program rule is decorated with multisets of labels from Λ. We define complementation over Λ by

$$\overline{R :: P!v} = P :: R?v \quad \text{and} \quad \overline{R :: P?v} = P :: R!v$$

$\boxed{\text{Rules for expressions}}$

$$\langle v,\sigma\rangle \longrightarrow v \qquad \langle x,\sigma\rangle \longrightarrow \sigma(x) \qquad \frac{\langle e_1,\sigma\rangle \longrightarrow v_1 \qquad \langle e_2,\sigma\rangle \longrightarrow v_2}{\langle e_1\ op\ e_2,\sigma\rangle \longrightarrow app(op,v_1,v_2)}$$

$\boxed{\text{Rules for guarded commands}}$

$$\frac{\langle b,\sigma\rangle \longrightarrow tt \qquad \langle c,\sigma\rangle \xrightarrow{\lambda} \langle c',\sigma'\rangle \mid \sigma' \mid \texttt{aborted}}{\langle b\Rightarrow c,\sigma\rangle \xrightarrow{\lambda} \langle c',\sigma'\rangle \mid \sigma' \mid \texttt{aborted}}$$

$$\frac{\langle g_i,\sigma\rangle \xrightarrow{\lambda} \langle c',\sigma'\rangle \mid \sigma' \mid \texttt{aborted}}{\langle g_1 \mathbin{[\!]} g_2,\sigma\rangle \xrightarrow{\lambda} \langle c',\sigma'\rangle \mid \sigma' \mid \texttt{aborted}}\ i=1,2$$

$$\frac{\langle b,\sigma\rangle \longrightarrow f\!f}{\langle b\Rightarrow c,\sigma\rangle \xrightarrow{\varepsilon} \texttt{failure}} \qquad \frac{\langle g_1,\sigma\rangle \xrightarrow{\varepsilon} \texttt{failure} \qquad \langle g_2,\sigma\rangle \xrightarrow{\varepsilon} \texttt{failure}}{\langle g_1 \mathbin{[\!]} g_2,\sigma\rangle \xrightarrow{\varepsilon} \texttt{failure}}$$

$\boxed{\text{Rules for commands}}$

$$\langle \texttt{skip},\sigma\rangle \xrightarrow{\varepsilon} \sigma \qquad \langle \texttt{abort},\sigma\rangle \xrightarrow{\varepsilon} \texttt{aborted}$$

$$\frac{\langle e,\sigma\rangle \longrightarrow v}{\langle Q!e,\sigma\rangle \xrightarrow{Q!v} \sigma} \qquad \langle Q?x,\sigma\rangle \xrightarrow{Q?v} \sigma[v/x]$$

$$\frac{\langle e,\sigma\rangle \longrightarrow v}{\langle x := e,\sigma\rangle \xrightarrow{\varepsilon} \sigma[v/x]} \qquad \frac{\langle c_1,\sigma\rangle \xrightarrow{\lambda} \langle c_1',\sigma'\rangle \mid \sigma' \mid \texttt{aborted}}{\langle c_1;c_2,\sigma\rangle \xrightarrow{\lambda} \langle c_1';c_2,\sigma'\rangle \mid \langle c_2,\sigma'\rangle \mid \texttt{aborted}}$$

$$\frac{\langle g,\sigma\rangle \xrightarrow{\lambda} \langle c',\sigma'\rangle \mid \sigma' \mid \texttt{failure} \mid \texttt{aborted}}{\langle \texttt{if}\ g\ \texttt{fi},\sigma\rangle \xrightarrow{\lambda} \langle c',\sigma'\rangle \mid \sigma' \mid \texttt{aborted} \mid \texttt{aborted}}$$

$$\frac{\langle g,\sigma\rangle \xrightarrow{\lambda} \langle c',\sigma'\rangle \mid \sigma' \mid \texttt{failure} \mid \texttt{aborted}}{\langle \texttt{do}\ g\ \texttt{od},\sigma\rangle \xrightarrow{\lambda} \langle c';\texttt{do}\ g\ \texttt{od},\sigma'\rangle \mid \langle \texttt{do}\ g\ \texttt{od},\sigma'\rangle \mid \sigma \mid \texttt{aborted}}$$

Figure B–1: SOS transition rules for CSP excluding programs

Rules for communication

$$\frac{\langle c,\sigma\rangle \xrightarrow{\lambda} \langle c',\sigma'\rangle \mid \sigma' \mid \texttt{aborted}}{\langle R :: c,\sigma\rangle \xrightarrow{R::\lambda} \langle R :: c',\sigma'\rangle \mid \sigma' \mid \langle \texttt{aborted},\sigma\rangle}$$

$$\frac{\langle p_1,\sigma\rangle \xrightarrow{\lambda} \langle p_1',\sigma'\rangle \mid \sigma'}{\langle p_1 \parallel p_2,\sigma\rangle \xrightarrow{\lambda} \langle p_1' \parallel p_2,\sigma'\rangle \mid \langle p_2,\sigma'\rangle} \qquad \frac{\langle p_2,\sigma\rangle \xrightarrow{\lambda} \langle p_2',\sigma'\rangle \mid \sigma'}{\langle p_1 \parallel p_2,\sigma\rangle \xrightarrow{\lambda} \langle p_1 \parallel p_2',\sigma'\rangle \mid \langle p_1,\sigma'\rangle}$$

$$\frac{\langle p_1,\sigma\rangle \xrightarrow{Q::P?v} \langle p_1',\sigma'\rangle \mid \sigma' \qquad \langle p_2,\sigma\rangle \xrightarrow{P::Q!v} \langle p_2',\sigma\rangle \mid \sigma}{\langle p_1 \parallel p_2,\sigma\rangle \xrightarrow{\varepsilon} \langle p_1' \parallel p_2',\sigma'\rangle \mid \langle p_1,\sigma'\rangle \mid \langle p_2,\sigma'\rangle \mid \sigma'}$$

$$\frac{\langle p_1,\sigma\rangle \xrightarrow{P::Q!v} \langle p_1',\sigma\rangle \mid \sigma \qquad \langle p_2,\sigma\rangle \xrightarrow{Q::P?v} \langle p_2',\sigma'\rangle \mid \sigma'}{\langle p_1 \parallel p_2,\sigma\rangle \xrightarrow{\varepsilon} \langle p_1' \parallel p_2',\sigma'\rangle \mid \langle p_1,\sigma'\rangle \mid \langle p_2,\sigma'\rangle \mid \sigma'}$$

Figure B–2: SOS transition rules for CSP programs

Rules for programs

$$\frac{\langle c,\sigma\rangle \xrightarrow{\lambda} \langle c',\sigma'\rangle \mid \sigma' \mid \texttt{aborted}}{\langle R :: c,\sigma\rangle \xrightarrow{\{\!|\, R::\lambda \,|\!\}} \langle R :: c',\sigma'\rangle \mid \sigma' \mid \langle \texttt{aborted},\sigma\rangle}$$

$$\frac{\langle p_1,\sigma\rangle \xrightarrow{m} \langle p_1',\sigma'\rangle \mid \sigma'}{\langle p_1 \parallel p_2,\sigma\rangle \xrightarrow{m} \langle p_1' \parallel p_2,\sigma'\rangle \mid \langle p_2,\sigma'\rangle} \qquad \frac{\langle p_2,\sigma\rangle \xrightarrow{m} \langle p_2',\sigma'\rangle \mid \sigma'}{\langle p_1 \parallel p_2,\sigma\rangle \xrightarrow{m} \langle p_1 \parallel p_2',\sigma'\rangle \mid \langle p_1,\sigma'\rangle}$$

$$\frac{\langle p_1,\sigma\rangle \xrightarrow{m_1} \langle p_1',\sigma_1\rangle \mid \sigma_1 \qquad \langle p_2,\sigma\rangle \xrightarrow{m_2} \langle p_2',\sigma_2\rangle \mid \sigma_2}{\langle p_1 \parallel p_2,\sigma\rangle \xrightarrow{m} \langle p_1' \parallel p_2',\sigma'\rangle \mid \langle p_1,\sigma'\rangle \mid \langle p_2,\sigma'\rangle \mid \sigma'} \; *$$

* where $\exists m' \subseteq (m_1 \cap \overline{m_2}).m = (m_1 \setminus m') \cup (m_2 \setminus \overline{m'})$ and $\sigma' = \sigma \oplus (\sigma_1 \restriction WV(p_1)) \oplus (\sigma_2 \restriction WV(p_2))$

Figure B–3: MSOS transition rules for CSP programs

Note that these rules make use of the WV function defined in section 5.1.2. It also makes use of two new functions on stores. These are defined as follows:

$$\sigma \oplus 0 = \sigma \qquad\qquad 0 \restriction X = 0$$
$$\sigma \oplus (\sigma'[v/x]) = (\sigma \oplus \sigma')[v/x] \qquad\qquad (\sigma[v/x]) \restriction X = \sigma \restriction X \text{ if } x \notin X$$
$$(\sigma[v/x]) \restriction X = (\sigma \restriction X)[v/x] \text{ if } x \in X$$

The first defines the operation of overwriting one store with another, and the other restricts a store to consist of only those bindings to variables that belong to a specified set X.

When the MSOS semantics describes a transition of p, it may write to the store many times. However, each process in a transition may write to the store at most once. Since processes do not share variables, this means that each variable is updated only once. This explains the side condition

$$\sigma' = \sigma \oplus (\sigma_1 \restriction WV(p_1)) \oplus (\sigma_2 \restriction WV(p_2))$$

in figure B–3. We need the following relation

$$\sigma_1 \simeq_X \sigma_2 \quad \text{iff} \quad \text{for all } x \in X.\ \sigma_1(x) = \sigma_2(x)$$

and we write $\sigma_1 \simeq \sigma_2$ for $\sigma_1 \simeq_{Var} \sigma_2$. This allows us to relate the stores obtained after executing a series of $\mathcal{CSP}_{SOS}$ transitions and one $\mathcal{CSP}_{MSOS}$ transition. Modulo this caveat, we obtain the "boring"[sic] proofs again:

Proposition B.0 *For all $p, p', \sigma, \sigma', \lambda$, and for each row in the following table, if $\mathcal{CSP}_{SOS}$ deduces the left hand entry, then $\mathcal{CSP}_{MSOS}$ deduces the right hand entry.*

$$\begin{array}{ll} \langle p, \sigma\rangle \xrightarrow{\lambda} \langle p', \sigma'\rangle & \langle p, \sigma\rangle \xrightarrow{m} \langle p', \sigma'\rangle \\ \langle p, \sigma\rangle \xrightarrow{\lambda} \sigma' & \langle p, \sigma\rangle \xrightarrow{m} \sigma' \end{array}$$

where in both cases $m = \{\!|\, \lambda \,|\!\}$ if $\lambda \neq \varepsilon$ and $m = \emptyset$ otherwise. □

The other direction requires the notion of t-traced deduction sequences again, which can be straightforwardly copied from page 35. Then we get

Proposition B.1 *For all $p, p', \sigma, \sigma', m$ and traces t of m, and for each row in the following table, if $\mathcal{CSP}_{MSOS}$ deduces the left-hand entry then $\mathcal{CSP}_{SOS}$ deduces the middle entry such that the right hand-hand entry is true.*

$$\begin{array}{lll} \langle p, \sigma\rangle \xrightarrow{m} \langle p', \sigma'\rangle & \langle p, \sigma\rangle \xrightarrow{t} \langle p', \sigma''\rangle & \sigma' \simeq \sigma'' \\ \langle p, \sigma\rangle \xrightarrow{m} \sigma' & \langle p, \sigma\rangle \xrightarrow{t} \sigma'' & \sigma' \simeq \sigma'' \end{array}$$

□

B.3 Stage two: from ordinary to interacting transition rules

In this section, we introduce a transition I-semantics for CSP, called $\mathcal{CSP}_I$. (Actually, it is really a DQI-system bu we call it $\mathcal{CSP}_I$ because the only sequencing constraints occur in the store rule.) The syntax and static semantics and auxiliary definitions are as before. The main change lies in the actual transition systems defined. First, they are labelled with process identifiers rather than from Λ. Second, the transition systems for commands,guarded commands and programs contain an extra item `done` in the set of terminal states. This token is intended to represent successful termination.

The following table lists the transition systems. Note that the sets should be primed in order to distinguish them from the previous transition systems. We do not do so here to ease legibility.

Syntactic class	(L)TS	Terminal Config'ns (Ω)	Config'ns (Γ)
expressions	$\langle \Gamma_e, \Omega_e, \longrightarrow_e \rangle$	Val	$Exp \supseteq \Omega_e$
g'd commands	$\langle \Gamma_g, \Omega_g, \Lambda, \longrightarrow_g \rangle$	$Com \cup \{\mathtt{done}, \mathtt{failure}, \mathtt{aborted}\}$	$GCom \cup \Omega_g$
commands	$\langle \Gamma_c, \Omega_c, \Lambda, \longrightarrow_c \rangle$	$\{\mathtt{done}, \mathtt{aborted}\}$	$Com \cup \Omega_c$
programs	$\langle \Gamma_p, \Omega_p, \Lambda, \longrightarrow_p \rangle$	$\{\mathtt{done}, \mathtt{aborted}\}$	$Prog \cup \Omega_p$
stores	$\langle Store, \emptyset, \rightsquigarrow \rangle$		$Store$

And for convenience once again, we add some equations for the new `done` production:

$$\mathtt{done} \parallel p \;=\; p \parallel \mathtt{done} \;=\; p \qquad \mathtt{done} \parallel \mathtt{done} \;=\; \mathtt{done}$$

In addition to the transition rules found in figures B–4 and B–5, we require the following rules for stores:

$$\sigma \rightsquigarrow \sigma \qquad \frac{\sigma \rightsquigarrow \sigma' \quad \blacktriangleright \quad \sigma' \rightsquigarrow \sigma''}{\sigma \rightsquigarrow \sigma''}$$

The first is the dummy transition rule of page 119 and the second is actually the store composition rule of $\mathcal{CSP}$, found in figure 5–2.

The proofs of equivalence between $\mathcal{CSP}_{MSOS}$ and $\mathcal{CSP}_I$ is almost exactly identical to that between $\mathcal{P}_{MSOS}$ and $\mathcal{P}_I$ found in section 2.5.3, modulo some difficulty with stores. In $\mathcal{CSP}_I$ we have to accept that processes other than those mentioned in the judgment currently of interest can write to the store; interference freedom guarantees that they will not affect those parts relevant to the processes at hand.

We label the communication interactions $comm(P, Q, v)$. In the following, we associate $P :: Q!v$ with $+comm(P, Q, v)$ and $Q :: P?v$ with $-comm(P, Q, v)$.

Let us write $\Pi[m] \vdash \Delta$ if m is the multiset of labels of Λ associated to the communicative dangling interactions of Π, which deduces the set of conclusions Δ. Note that Π may or may not have store dangling interactions. Let us write

Rules for expressions

$$v \xrightarrow{P} v \qquad\quad x \xrightarrow{P} v \;\text{———}\; \frac{\sigma(x) = v}{\sigma \rightsquigarrow \sigma}$$

$$\frac{e_1 \xrightarrow{P} v_1 \qquad e_2 \xrightarrow{P} v_2}{e_1 \; op \; e_2 \xrightarrow{P} app(op, v_1, v_2)}$$

Rules for guarded commands

$$\frac{b \xrightarrow{P} f\!f}{b \Rightarrow c \xrightarrow{P} \texttt{failure}} \qquad\quad \frac{b \xrightarrow{P} tt \quad \blacktriangleright \quad c \xrightarrow{P} c' \mid \texttt{done} \mid \texttt{aborted}}{b \Rightarrow c \xrightarrow{P} c' \mid \texttt{done} \mid \texttt{aborted}}$$

$$\frac{g_i \xrightarrow{P} c \mid \texttt{done} \mid \texttt{aborted}}{g_1 \, [\!] \, g_2 \xrightarrow{P} c \mid \texttt{done} \mid \texttt{aborted}} \; i = 1, 2$$

$$\frac{g_1 \xrightarrow{P} \texttt{failure} \qquad g_2 \xrightarrow{P} \texttt{failure}}{g_1 \, [\!] \, g_2 \xrightarrow{P} \texttt{failure}}$$

Rules for commands

$$\texttt{skip} \xrightarrow{P} \texttt{done} \qquad\quad \texttt{abort} \xrightarrow{P} \texttt{aborted}$$

$$\frac{e \xrightarrow{P} v \quad \blacktriangleright \quad P : set(x, v)}{x := e \xrightarrow{P} \texttt{done}} \qquad\quad \frac{e \xrightarrow{R} v \quad \blacktriangleright \quad R : out(Q, v)}{Q!e \xrightarrow{R} \texttt{done}}$$

$$\overline{P : set(x, v)} \text{——} \overline{\sigma \rightsquigarrow \sigma[v/x]} \qquad\quad \overline{R : out(Q, v)} \text{——} \overline{R?x \xrightarrow{Q} \texttt{done}} \text{——} \overline{\sigma \rightsquigarrow \sigma[v/x]}$$

$$\frac{g \xrightarrow{P} c \mid \texttt{done} \mid \texttt{failure} \mid \texttt{aborted}}{\texttt{if } g \texttt{ fi} \xrightarrow{P} c \mid \texttt{done} \mid \texttt{aborted} \mid \texttt{aborted}}$$

$$\frac{g \xrightarrow{P} c \mid \texttt{done} \mid \texttt{failure} \mid \texttt{aborted}}{\texttt{do } g \texttt{ od} \xrightarrow{P} c; \texttt{do } g \texttt{ od} \mid \texttt{do } g \texttt{ od} \mid \texttt{done} \mid \texttt{aborted}}$$

Figure B–4: Transition (Q)I-rules for CSP except programs

Rules for programs

$$\frac{c \xrightarrow{R} c' \mid \texttt{done} \mid \texttt{aborted}}{R :: c \longrightarrow R :: c' \mid \texttt{done} \mid \texttt{aborted}}$$

$$\frac{p_1 \longrightarrow p_1' \qquad p_2 \longrightarrow p_2'}{p_1 \parallel p_2 \longrightarrow p_1' \parallel p_2'}$$

$$\frac{p_1 \longrightarrow p_1'}{p_1 \parallel p_2 \longrightarrow p_1' \parallel p_2} \qquad \frac{p_2 \longrightarrow p_2'}{p_1 \parallel p_2 \longrightarrow p_1 \parallel p_2'}$$

Figure B–5: Transition I-rules for CSP programs

$\mathcal{CSP}_I[m] \vdash \Delta$ if there exists a $\Pi \in \mathbf{I}(\mathcal{CSP}_I)$ such that $\Pi[m] \vdash \Delta$, in which case we shall say $[m] \vdash \Delta$ is deducible in $\mathcal{CSP}_I$.

Then the following propositions are easy to prove by induction and analogously to proposition 2.13 (although we do not need to use fragmentation).

Proposition B.2 *For all e in Exp, v in Val, κ, κ' in $Comm \cup GComm \cup Prog$ which occur as subphrases of an R-labelled process in p, $\sigma, \sigma', \sigma''$ in $Store$, $m \in \mathbb{N}^\Lambda$, for each row in the following table, if the left hand entry is deducible in $\mathcal{CSP}_{MSOS}$ then the middle entry is deducible in $\mathcal{CSP}_I$ such that the condition in the right hand entry is satisfied.*

$\langle e, \sigma\rangle \to v$	$e \xrightarrow{R} v, \sigma \rightsquigarrow \sigma'$	$\sigma' \simeq_X \sigma$
$\langle \kappa, \sigma\rangle \overset{m}{\to} \langle \kappa', \sigma'\rangle$	$[m] \vdash \kappa \xrightarrow{R} \kappa', \sigma \rightsquigarrow \sigma''$	$\sigma' \simeq_X \sigma''$
$\langle \kappa, \sigma\rangle \overset{m}{\to} \sigma'$	$[m] \vdash \kappa \xrightarrow{R} \texttt{done}, \sigma \rightsquigarrow \sigma''$	$\sigma' \simeq_X \sigma''$
$\langle \kappa, \sigma\rangle \overset{m}{\to} \texttt{aborted}$	$[m] \vdash \kappa \xrightarrow{R} \texttt{aborted}, \sigma \rightsquigarrow \sigma'$	$\sigma \simeq_X \sigma'$

where $X = Acc(R, p)$ if κ is not a program, and $X = FV(\kappa)$ otherwise. □

Proposition B.3 *For all e in Exp, v in Val, κ, κ' in $Comm \cup GComm \cup Prog$ which occur as subphrases of an R-labelled process in p, $\sigma, \sigma', \sigma''$ in $Store$, $m \in \mathbb{N}^\Lambda$, for each row in the following table, if the middle entry is deducible in $\mathcal{CSP}_I$ then the left hand entry is deducible in $\mathcal{CSP}_{MSOS}$ such that the condition in the right hand entry is satisfied.* □

In order to take stock of where we are at the moment, we define the full transition system of $\mathcal{CSP}_I$ as $\langle \Gamma_I, \Omega_I, \longrightarrow_I\rangle$ where $\Gamma_I = \Gamma_p \times Stores$, $\Omega_I = \{\texttt{done}, \texttt{aborted}\} \times Stores$ and $\longrightarrow_I \subseteq \Gamma_I \times \Gamma_I$ is defined by

$$\langle p, \sigma\rangle \longrightarrow \langle p', \sigma'\rangle \quad \text{iff} \quad \mathcal{CSP}_I \Vdash p \longrightarrow p', \sigma \rightsquigarrow \sigma'$$

Note the use of proper deducibility: it ensures that every alteration to the store is accounted for, and moreover so is every interaction of the program.

Then so far we have asserted that for all $p, p' \in \mathit{Prog}$, $\sigma, \sigma', \sigma'' \in \mathit{Store}$ and each row in the following table, if $\mathcal{CSP}_{SOS}$ (sequentially) deduces the left-hand entry then $\mathcal{CSP}_I$ (sequentially) deduces the middle entry such that the right-hand entry is true. Moreover, if $\mathcal{CSP}_I$ deduces the middle entry, then $\mathcal{CSP}_{SOS}$ deduces the left-hand entry such that the right-hand entry is true.

$$\begin{array}{lll}
\langle p, \sigma\rangle (\xrightarrow{\varepsilon})^+ \langle p', \sigma'\rangle & \langle p, \sigma\rangle \longrightarrow^+ \langle p', \sigma'\rangle & \sigma' \simeq_{FV(p)} \sigma'' \\
\langle p, \sigma\rangle (\xrightarrow{\varepsilon})^+ \sigma' & \langle p, \sigma\rangle \longrightarrow^+ \langle \mathtt{done}\ , \sigma'\rangle & \sigma' \simeq_{FV(p)} \sigma'' \\
\langle p, \sigma\rangle (\xrightarrow{\varepsilon})^+ \mathtt{aborted} & \langle p, \sigma\rangle \longrightarrow^+ \langle \mathtt{aborted}, \sigma'\rangle &
\end{array}$$

B.4 Stage three: from interacting transitions to evaluations

We introduce yet another intermediate semantics, but this time we simply introduce the following three rules to $\mathcal{CSP}_I$:

$$\frac{\gamma \xrightarrow{\langle R\rangle} \gamma' \quad \blacktriangleright \quad \gamma' \xrightarrow{\langle R\rangle} \gamma''}{\gamma \xrightarrow{\langle R\rangle} \gamma''}$$

$$\frac{g \xrightarrow{R} \mathtt{done} \quad \blacktriangleright \quad \mathtt{do}\ g\ \mathtt{od} \xrightarrow{R} \gamma}{\mathtt{do}\ g\ \mathtt{od} \xrightarrow{R} \gamma} \qquad \frac{c_1 \xrightarrow{R} \mathtt{done} \quad \blacktriangleright \quad c_2 \xrightarrow{R} \gamma}{c_1; c_2 \xrightarrow{R} \gamma}$$

where $\gamma, \gamma', \gamma'' \in \Gamma_c \cup \Gamma_g \cup \Gamma_p$ and the angle brackets denote options — either the R labels all exist in a rule instance or they do not. (See the options convention of the definition of standard ML [HMT90, §4.10].) We shall refer to the first rule as the "cut" rule, and the system as a whole as $\mathcal{CSP}_{CUT}$. We refer to the last two rules as *evaluation* rules, because they are similar to the sequential composition and repetition rules of $\mathcal{CSP}$. We refer to a deduction as *cut-free* if it contains no instance of a cut rule, and *eval-free* if it contains no instance of an evaluation rule. $\mathcal{CSP}_I$ transitions are both cut and eval-free. We can cut sequences of $\mathcal{CSP}_I$ transitions together to get eval-free deductions. The key idea behind the following proof is that $\mathcal{CSP}$ program transitions correspond to cut-free deductions of terminating transition sequences.

First, we state a cut-elimination result for $\mathcal{CSP}_{CUT}$, which uses the eval rules to remove cuts. The proof itself is standard, but depends on specialized notions of *cut-formula* and *degree* of cut-formula and deductions.

A cut-formula is the configuration that occurs on the left-hand side of the left-hand premise of a cut-rule. (This is in contrast with the usual, logical notion of cut-rule where the cut formula occurs on the right-hand side of the left-hand premise. The reason for this is that the semantic rules are syntax-directed according to the left-hand sides of transition rules.) In the above cut rule schema, the cut-formula is the configuration γ. Note that no cut formula has form `done` or `aborted`

$$\dfrac{\dfrac{\genfrac{}{}{0pt}{}{\Sigma_1}{b \xrightarrow{R} tt} \quad \blacktriangleright \quad \genfrac{}{}{0pt}{}{\Sigma_2}{c \xrightarrow{R} c'}}{b \Rightarrow c \xrightarrow{R} c'} \quad \blacktriangleright \quad \genfrac{}{}{0pt}{}{\Sigma_3}{c' \xrightarrow{R} \gamma}}{b \Rightarrow c \xrightarrow{R} \gamma} \quad \rightsquigarrow \quad \dfrac{\genfrac{}{}{0pt}{}{\Sigma_1}{b \xrightarrow{R} tt} \quad \blacktriangleright \quad \dfrac{\genfrac{}{}{0pt}{}{\Sigma_2}{c \xrightarrow{R} c'} \quad \blacktriangleright \quad \genfrac{}{}{0pt}{}{\Sigma_3}{c' \xrightarrow{R} \gamma}}{c \xrightarrow{R} \gamma}}{b \Rightarrow c \xrightarrow{R} \gamma}$$

$$\dfrac{\dfrac{\genfrac{}{}{0pt}{}{\Sigma_1}{g_i \xrightarrow{R} c}}{g_1 \,[\!]\, g_2 \xrightarrow{R} c} \quad \blacktriangleright \quad \genfrac{}{}{0pt}{}{\Sigma_2}{c \xrightarrow{R} \gamma}}{g_1 \,[\!]\, g_2 \xrightarrow{R} \gamma} \quad \rightsquigarrow \quad \dfrac{\dfrac{\genfrac{}{}{0pt}{}{\Sigma_1}{g_i \xrightarrow{R} c} \quad \blacktriangleright \quad \genfrac{}{}{0pt}{}{\Sigma_2}{c \xrightarrow{R} \gamma}}{g_i \xrightarrow{R} \gamma}}{g_1 \,[\!]\, g_2 \xrightarrow{R} \gamma}$$

Figure B–6: Elimination rules for guarded commands

The degree of a cut formula A, $\partial(A)$ is defined inductively as follows:

$$\begin{array}{l}
\partial(\texttt{abort}) = \partial(\texttt{skip}) = \partial(x := e) = \partial(Q!e) = \partial(Q?x) = 0 \\
\partial(\texttt{if}\ g\ \texttt{fi}) = \partial(\texttt{do}\ g\ \texttt{od}) = 1 + \partial(g) \\
\partial(b \Rightarrow c) = 1 + \partial(c) \\
\partial(g_1 \,[\!]\, g_2) = 1 + \max\{\partial(g_1), \partial(g_2)\} \\
\partial(c_1; c_2) = 1 + \max\{\partial(c_1), \partial(c_2)\} \\
\partial(R :: c) = 1 + \partial(c) \\
\partial(p_1 \parallel p_2) = 1 + \max\{\partial(p_1), \partial(p_2)\}
\end{array}$$

The degree of a deduction is the maximum of the degrees of its cut formulae.

Now we can prove the following result in exactly the same way as one would prove an ordinary cut-elimination theorem (see, e.g., [GLT89, ch 13]), using the elimination rules found in figures B–6, B–7, B–8 and B–9.

Proposition B.4 *If $\mathcal{CSP}_{CUT} \vdash p \longrightarrow p'$ then there exists a $\mathcal{CSP}_{CUT}$-deduction of degree 0 which also deduces it.* □

An important consequence of a deduction being cut-free is that it will contain no instance of either of the rules

$$\frac{p_1 \longrightarrow p_1'}{p_1 \parallel p_2 \longrightarrow p_1' \parallel p_2} \qquad \frac{p_2 \longrightarrow p_2'}{p_1 \parallel p_2 \longrightarrow p_1 \parallel p_2'}$$

because these are elminated by the second rule (and its symmetric counterpart) in figure B–8.

$$\dfrac{\dfrac{\overset{\Sigma_1}{c_1 \xrightarrow{R} c_1'}}{c_1;c_2 \xrightarrow{R} c_1';c_2} \quad \blacktriangleright \quad \dfrac{\overset{\Sigma_2}{c_1' \xrightarrow{R} \gamma}}{c_1';c_2 \xrightarrow{R} \gamma'}}{c_1;c_2 \xrightarrow{R} \gamma'} \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Sigma_1}{c_1 \xrightarrow{R} c_1'} \quad \blacktriangleright \quad \overset{\Sigma_2}{c_1' \xrightarrow{R} \gamma}}{c_1 \xrightarrow{R} \gamma}}{c_1;c_2 \xrightarrow{R} \gamma'}$$

$$\dfrac{\dfrac{\overset{\Sigma_1}{c_1;c_2 \xrightarrow{R} c_2} \quad \blacktriangleright \quad \overset{\Sigma_2}{c_2 \xrightarrow{R} c_2'}}{c_1;c_2 \xrightarrow{R} c_2'} \quad \blacktriangleright \quad \overset{\Sigma_3}{c_2' \xrightarrow{R} \gamma}}{c_1;c_2 \xrightarrow{R} \gamma} \quad \rightsquigarrow$$

$$\dfrac{\overset{\Sigma_1}{c_1;c_2 \xrightarrow{R} c_2} \quad \blacktriangleright \quad \dfrac{\overset{\Sigma_2}{c_2 \xrightarrow{R} c_2'} \quad \blacktriangleright \quad \overset{\Sigma_3}{c_2' \xrightarrow{R} \gamma}}{c_2 \xrightarrow{R} \gamma}}{c_1;c_2 \xrightarrow{R} \gamma}$$

$$\dfrac{\dfrac{\overset{\Sigma_1}{g \xrightarrow{R} c}}{\texttt{if}\ g\ \texttt{fi} \xrightarrow{R} c} \quad \blacktriangleright \quad \overset{\Sigma_2}{c \xrightarrow{R} \gamma}}{\texttt{if}\ g\ \texttt{fi} \xrightarrow{R} \gamma} \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Sigma_1}{g \xrightarrow{R} c} \quad \blacktriangleright \quad \overset{\Sigma_2}{c \xrightarrow{R} \gamma}}{g \xrightarrow{R} c}}{\texttt{if}\ g\ \texttt{fi} \xrightarrow{R} \gamma}$$

$$\dfrac{\dfrac{\overset{\Sigma_1}{g \xrightarrow{R} c}}{\texttt{do}\ g\ \texttt{od} \xrightarrow{R} c;\texttt{do}\ g\ \texttt{od}} \quad \blacktriangleright \quad \dfrac{\overset{\Sigma_2}{c \xrightarrow{R} \gamma}}{c;\texttt{do}\ g\ \texttt{od} \xrightarrow{R} \gamma'}}{\texttt{do}\ g\ \texttt{od} \xrightarrow{R} \gamma'} \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Sigma_1}{g \xrightarrow{R} c} \quad \blacktriangleright \quad \overset{\Sigma_2}{c \xrightarrow{R} \gamma}}{g \xrightarrow{R} \gamma}}{\texttt{do}\ g\ \texttt{od} \xrightarrow{R} \gamma'}$$

$$\dfrac{\dfrac{\overset{\Sigma_1}{c \xrightarrow{R} c'}}{R :: c \longrightarrow R :: c'} \quad \blacktriangleright \quad \dfrac{\overset{\Sigma_2}{c' \xrightarrow{R} \gamma}}{R :: c' \longrightarrow \gamma'}}{R :: c \longrightarrow \gamma'} \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Sigma_1}{c \xrightarrow{R} c'} \quad \blacktriangleright \quad \overset{\Sigma_2}{c' \xrightarrow{R} \gamma}}{c \xrightarrow{R} \gamma}}{R :: c \longrightarrow \gamma'}$$

Figure B–7: Elimination rules for commands and atomic programs

$$\dfrac{\dfrac{\overset{\Sigma_1}{p_1 \longrightarrow p_1'}}{p_1 \parallel p_2 \longrightarrow p_1' \parallel p_2} \quad \blacktriangleright \quad \dfrac{\overset{\Sigma_2}{p_1' \longrightarrow p_1''}}{p_1' \parallel p_2 \longrightarrow p_1'' \parallel p_2}}{p_1 \parallel p_2 \longrightarrow p_1'' \parallel p_2} \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Sigma_1}{p_1 \longrightarrow p_1'} \quad \blacktriangleright \quad \overset{\Sigma_2}{p_1' \longrightarrow p_1''}}{p_1 \longrightarrow p_1''}}{p_1 \parallel p_2 \longrightarrow p_1'' \parallel p_2}$$

and symmetrically for p_2

$$\dfrac{\dfrac{\overset{\Sigma_1}{p_1 \longrightarrow p_1'}}{p_1 \parallel p_2 \longrightarrow p_1' \parallel p_2} \quad \blacktriangleright \quad \dfrac{\overset{\Sigma_2}{p_2 \longrightarrow p_2'}}{p_1' \parallel p_2 \longrightarrow p_1' \parallel p_2'}}{p_1 \parallel p_2 \longrightarrow p_1' \parallel p_2'} \quad \rightsquigarrow \quad \dfrac{\overset{\Sigma_1}{p_1 \longrightarrow p_1'} \qquad \overset{\Sigma_2}{p_2 \longrightarrow p_2'}}{p_1 \parallel p_2 \longrightarrow p_1' \parallel p_2'}$$

and symmetrically for p_2

$$\dfrac{\dfrac{\overset{\Sigma_1}{p_1 \longrightarrow p_1'}}{p_1 \parallel p_2 \longrightarrow p_1' \parallel p_2} \quad \blacktriangleright \quad \dfrac{\overset{\Sigma_2}{p_1' \longrightarrow p_1''} \qquad \overset{\Sigma_3}{p_2 \longrightarrow p_2'}}{p_1' \parallel p_2 \longrightarrow p_1'' \parallel p_2'}}{p_1 \parallel p_2 \longrightarrow p_1'' \parallel p_2'} \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\overset{\Sigma_1}{p_1 \longrightarrow p_1'} \quad \blacktriangleright \quad \overset{\Sigma_2}{p_1' \longrightarrow p_1''}}{p_1 \longrightarrow p_1''} \qquad \overset{\Sigma_3}{p_2 \longrightarrow p_2'}}{p_1 \parallel p_2 \longrightarrow p_1'' \parallel p_2'}$$

and symmetrically for p_2

$$\dfrac{\dfrac{\overset{\Sigma_1}{p_1 \longrightarrow p_1'} \qquad \overset{\Sigma_2}{p_2 \longrightarrow p_2'}}{p_1 \parallel p_2 \longrightarrow p_1' \parallel p_2'} \quad \blacktriangleright \quad \dfrac{\overset{\Sigma_3}{p_1' \longrightarrow p_1''}}{p_1' \parallel p_2' \longrightarrow p_1'' \parallel p_2'}}{p_1 \parallel p_2 \longrightarrow p_1'' \parallel p_2'} \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\overset{\Sigma_1}{p_1 \longrightarrow p_1'} \quad \blacktriangleright \quad \overset{\Sigma_3}{p_1' \longrightarrow p_1''}}{p_1 \longrightarrow p_1''} \qquad \overset{\Sigma_2}{p_2 \longrightarrow p_2'}}{p_1 \parallel p_2 \longrightarrow p_1'' \parallel p_2'}$$

and symmetrically for p_2

$$\dfrac{\dfrac{\overset{\Sigma_1}{p_1 \longrightarrow p_1'} \qquad \overset{\Sigma_2}{p_2 \longrightarrow p_2'}}{p_1 \parallel p_2 \longrightarrow p_1' \parallel p_2'} \quad \blacktriangleright \quad \dfrac{\overset{\Sigma_3}{p_1' \longrightarrow p_1''} \qquad \overset{\Sigma_4}{p_2' \longrightarrow p_2''}}{p_1' \parallel p_2' \longrightarrow p_1'' \parallel p_2''}}{p_1 \parallel p_2 \longrightarrow p_1'' \parallel p_2''} \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\overset{\Sigma_1}{p_1 \longrightarrow p_1'} \quad \blacktriangleright \quad \overset{\Sigma_3}{p_1' \longrightarrow p_1''}}{p_1 \longrightarrow p_1''} \qquad \dfrac{\overset{\Sigma_2}{p_2 \longrightarrow p_2'} \quad \blacktriangleright \quad \overset{\Sigma_4}{p_2' \longrightarrow p_2''}}{p_2 \longrightarrow p_2''}}{p_1 \parallel p_2 \longrightarrow p_1'' \parallel p_2''}$$

Figure B–8: Elimination rules for parallel composition

$$\frac{\dfrac{\overset{\Sigma_1}{c_1 \xrightarrow{R} \mathtt{done}}}{c_1; c_2 \xrightarrow{R} c_2} \quad \blacktriangleright \quad \overset{\Sigma_2}{c_2 \xrightarrow{R} \gamma}}{c_1; c_2 \xrightarrow{R} \gamma} \quad \rightsquigarrow \quad \frac{\overset{\Sigma_1}{c_1 \xrightarrow{R} \mathtt{done}} \quad \blacktriangleright \quad \overset{\Sigma_2}{c_2 \xrightarrow{R} \gamma}}{c_1; c_2 \xrightarrow{R} \gamma}$$

$$\frac{\dfrac{\overset{\Sigma_1}{g \xrightarrow{R} \mathtt{done}}}{\mathtt{do}\ g\ \mathtt{od} \xrightarrow{R} \mathtt{do}\ g\ \mathtt{od}} \quad \blacktriangleright \quad \overset{\Sigma_2}{\mathtt{do}\ g\ \mathtt{od} \xrightarrow{R} \gamma}}{\mathtt{do}\ g\ \mathtt{od} \xrightarrow{R} \gamma} \quad \rightsquigarrow \quad \frac{\overset{\Sigma_1}{g \xrightarrow{R} \mathtt{done}} \quad \blacktriangleright \quad \overset{\Sigma_2}{\mathtt{do}\ g\ \mathtt{od} \xrightarrow{R} \gamma}}{\mathtt{do}\ g\ \mathtt{od} \xrightarrow{R} \gamma}$$

Figure B–9: Elimination rules for evaluations

Proposition B.5 *For each of the rows in the following table, if $\mathcal{CSP}_{CUT}$ deduces the left-hand column, then $\mathcal{CSP}$ deduces the right-hand column:*

$p \longrightarrow \mathtt{done}$	p
$p \longrightarrow \mathtt{aborted}$	$p\dagger$
$c \xrightarrow{R} \mathtt{done}$	$R : c$
$c \xrightarrow{R} \mathtt{aborted}$	$R \dagger c$
$g \xrightarrow{R} \mathtt{done}$	$R : g$
$g \xrightarrow{R} \mathtt{aborted}$	$R \dagger g$
$g \xrightarrow{R} \mathtt{failure}$	$R * g$
$e \xrightarrow{R} v$	$R : e \longrightarrow v$

Proof: (Sketch) For each case, we take a cut-free deduction of the required judgment, and then recursively replace each of the above $\mathcal{CSP}_{CUT}$ judgments with their $\mathcal{CSP}$ equivalents. □

The last step in the proof of this direction is to construct the corresponding $\mathcal{CSP}$-deduction of the store transitions, and then assemble it back together with the newly constructed constructed program $\mathcal{CSP}$-deduction. This is easily done: first, it is quite straightforward to notice that every $\mathcal{CSP}_{CUT}$ store deduction is also a $\mathcal{CSP}$-deduction. Second, it is easy to show that the constructed $\mathcal{CSP}$ program deduction is simulated by the original $\mathcal{CSP}_{CUT}$-deduction (shown by rule-induction, and the fact that the elimination rules actually removes dependencies between dangling interactions. Therefore we can use the Interaction Reflection theorem to assemble the program deduction and the store deduction to make a proper $\mathcal{CSP}$-deduction.

Proposition B.6 *For all p, σ, σ', and each row of the following table, if $\mathcal{CSP}_{CUT}$ properly deduces the left hand column, then $\mathcal{CSP}$ properly deduces the right hand* *column.*

$$\begin{array}{ll} p \longrightarrow \texttt{done}\ , \sigma \rightsquigarrow \sigma' & p, \sigma \rightsquigarrow \sigma' \\ p \longrightarrow \texttt{aborted}, \sigma \rightsquigarrow \sigma' & p\dagger, \sigma \rightsquigarrow \sigma' \end{array}$$

□

B.4.1 The other direction: from evaluations to transitions

We have outlined how to show that every terminating sequence of transitions of $\mathcal{CSP}_I$ corresponds to a $\mathcal{CSP}$ evaluation deduction. Now we outline the reverse, that every $\mathcal{CSP}$ deduction corresponds to a terminating sequence of transitions of $\mathcal{CSP}_I$.

This direction is slightly more difficult in that we have to both discover the individual transitions of each process, and then schedule them. This process is complicated by the fact that the store interactions of the various processes may be interleaved, and therefore have to be swapped around to accord with the chosen scheduling.

Nevertheless, despite this difficulty, the idea of the proof is quite straightforward: we simply identify those parts of the evaluation deduction that correspond to a "first transition", and then peel it off to make one. This peeling is essentially the cut-elimination process in reverse.

Throughout the rest of this appendix, let Π be a proper $\mathcal{CSP}$-deduction concluding $p, \sigma \rightsquigarrow \sigma'$. Let us identify it with its cut-free $\mathcal{CSP}_{CUT}$ analogue for convenience. Let Π_p be the program fragment of Π and Π_s be the store fragment. We assume that the only dangling interactions of Π_p are those relating to the store (i.e., Π is a binary assembly of Π_p and Π_s), so that communicating commands are still connected. We use Σ to range over arbitrary deductions, which in the following will always be trees.

In the following definition, we use the following abbreviations. We write $\Sigma \vdash \Sigma'$ when Σ' is obtained from applying a single rule to Σ. We write $\Sigma_1 \blacktriangleright \Sigma_2 \vdash \Sigma$ when Σ is the result of applying a single rule with two sequenced premises to Σ_1 and Σ_2 in order. Last, we write $\Sigma_1 \parallel \Sigma_2 \vdash \Sigma$ when Σ is the result of applying a single rule with two unsequenced premises to Σ_1 and Σ_2.

When Σ is not a store deduction, let $PP(\Sigma)$ be the *principal phrase* of Σ: the left-hand side configuration of the single conclusion of Σ. Let us fix program deduction Π_p. Then we define the *first transition* of command and guarded command subdeductions Σ of Π_p to be the set $\mathit{first}(\Sigma)$ defined inductively by

$$\mathit{first}(\Sigma) = \begin{cases} O(\Sigma) & \text{if } PP(\Sigma) \in \{\texttt{abort}, \texttt{skip}, x := e, Q!e, Q?x\} \\ O(\Sigma) & \text{if } \Sigma \vdash g \xrightarrow{R} \texttt{failure} \\ \mathit{first}(\Sigma') & \text{if } PP(\Sigma) \in \{\texttt{if}\ g\ \texttt{fi}, \texttt{do}\ g\ \texttt{od}\} \text{ and } \Sigma' \vdash \Sigma \\ \mathit{first}(\Sigma_1) & \text{if } \Sigma_1 \blacktriangleright \Sigma_2 \vdash \Sigma \\ O(\Sigma_1) \cup \mathit{first}(\Sigma_2) & \text{if } PP(\Sigma) = b \Rightarrow c \text{ and } \Sigma_1 \blacktriangleright \Sigma_2 \vdash \Sigma \end{cases}$$

Note that $\mathit{first}(\Sigma)$ is upwards-closed with respect to *aboveness*: any occurrence strictly above a member of the set is also in the set. We define the downwards closure of $\mathit{first}(\Sigma)$ to be $\downarrow \mathit{first}(\Sigma) = \{A \mid \exists B \in \mathit{first}(\Sigma). A \text{ is below } B \text{ in } \Sigma\}$. This is important because it contains all the formulae down which we will propagate cuts.

Now, let us write $\Sigma_1 <_{\Pi_p} \Sigma_2$ to abbreviate the condition that there exists $A_i \in \mathit{first}(\Sigma_i)$ such that $A_1 <_{\Pi_p} A_2$. We get the important result for commands:

Lemma B.7(i) *If disjoint subdeductions Σ_1 and Σ_2 of Π_p deduce behaviour about a guarded command or command, and moreover $\Sigma_1 <_{\Pi_p} \Sigma_2$ then $\Sigma_2 \not<_{\Pi_p} \Sigma_1$*

Proof: Essentially because $\mathit{first}(\Sigma)$ contains at most one occurrence which interacts with another part of Π_p. □

Then the first transition of deductions of program behaviour can be determined by

$$\mathit{first}(\Sigma) = \begin{cases} \mathit{first}(\Sigma') & \text{if } PP(\Sigma) = R :: c \text{ and } \Sigma' \vdash \Sigma \\ \mathit{first}(\Sigma_1) & \text{if } \Sigma_1 \blacktriangleright \Sigma_2 \vdash \Sigma \\ \mathit{first}(\Sigma_1) & \text{if } PP(\Sigma) = p_1 \parallel p_2,\ \Sigma_1 \parallel \Sigma_2 \vdash \Sigma \text{ and } \Sigma_1 <_{\Pi_p} \Sigma_2 \\ \mathit{first}(\Sigma_2) & \text{if } PP(\Sigma) = p_1 \parallel p_2,\ \Sigma_1 \parallel \Sigma_2 \vdash \Sigma \text{ and } \Sigma_2 <_{\Pi_p} \Sigma_1 \\ \mathit{first}(\Sigma_1) \cup \mathit{first}(\Sigma_2) & \text{if } PP(\Sigma) = p_1 \parallel p_2 \text{ and neither of the above} \\ & \qquad\qquad\qquad\qquad \text{two cases} \end{cases}$$

Proposition B.7 *Let disjoint subdeductions Σ_1 and Σ_2 of Π_p be such that $\Sigma_1 <_{\Pi_p} \Sigma_2$ then $\Sigma_2 \not<_{\Pi_p} \Sigma_1$.*

Proof: By lemma B.7(i) and induction. □

We use this relation to schedule the transitions. $\mathit{first}(\Sigma_1)$ and $\mathit{first}(\Sigma_2)$ will be disjoint since $\mathit{first}(\Sigma_i) \subseteq O(\Sigma_i)$ and Σ_1 and Σ_2 are disjoint trees in the cases for parallel composition above. The only way for an occurrence of $\mathit{first}(\Sigma_1)$ is via a sequence of interaction links. So if $\mathit{first}(\Sigma_1)$ is sequenced after $\mathit{first}(\Sigma_2)$, it means that something in Σ_1 communicates with a process sequenced after the first transition of Σ_2.

Proposition B.8 *For all transition judgments $\gamma \xrightarrow{\langle R \rangle} \gamma'$ in* first(Σ)*, γ is a subphrase of $PP(\Sigma)$.*

Proof: Essentially because the only rule of $\mathcal{CSP}_{CUT}$ in which the left-hand side of a premise is not a proper subphrase of the left-hand side of the conclusion is the repetition rule. But the definition of $\mathit{first}(\Sigma)$ ignores the problematic premise. □

Let us call a *cut-formula* the left-hand side of the conclusion of an instance of the cut rule. This lemma allows us to define the *codegree* of a cut-formula as the difference in size between the whole program and the program phrase where the cut occurs: when we come to propagate cuts down the deduction (the process of "peeling off the first transition"), the codegree will decrease until the cut formula is the entire program (when it will have codegree zero).

We define the codegree of γ with respect to superphrase γ', written $\mathcal{C}(\gamma, \gamma')$, inductively:

$$\mathcal{C}(\gamma,\gamma') = \begin{cases} 0 & \text{if } \gamma = \gamma' \\ 1 + \mathcal{C}(\gamma, g) & \text{if } \gamma' \in \{\texttt{if}\ g\ \texttt{fi}, \texttt{do}\ g\ \texttt{od}\} \\ 1 + \mathcal{C}(\gamma, c) & \text{if } \gamma' \in \{R :: c, c; c', b \Rightarrow c\} \\ 1 + \mathcal{C}(\gamma, p_i) & \text{if } \gamma' = p_1 \parallel p_2 \text{ and } \gamma \text{ occurs in } p_i \\ 1 + \mathcal{C}(\gamma, g_i) & \text{if } \gamma' = g_1 \mathbin{[\!]} g_2 \text{ and } \gamma \text{ occurs in } g_i \end{cases}$$

We define the codegree of an eval-free $\mathcal{CSP}_{CUT}$ deduction Σ to be the maximum of its codegrees that occur in $\downarrow$ *first*(Σ). (We are only interested in propagating cuts to peel off the first transition.)

Then the following result can be proved in a similar way to that of proposition B.4, using the elimination rules found in figures B–6, B–7 and B–8 in reverse. Note that we choose which parallel composition rules to "unapply" according to our scheduling. For instance, we unapply the last rule of figure B–8 only when we know that the first transitions of p_1 and p_2 belong to the first transition of their parallel composition in Π_p.

Proposition B.9 *If $\mathcal{CSP}_{CUT} \vdash p \longrightarrow p'$ then there exists a $\mathcal{CSP}_{CUT}$-deduction of codegree 0 which also deduces it.* $\square$

In actual fact, given the above definitions, we can also show that if Π'_p has codegree 0 then there exists only one cut-rule in $\downarrow$ *first*(Π'_p), and moreover it occurs at the bottom. This follows because after we first introduce cuts by eliminating the eval-rules from the $\mathcal{CSP}_{CUT}$ analogue of the $\mathcal{CSP}$-deduction, there will be exactly one cut per process whose first transition occurs during the first transition of the whole program. The process of propagating cuts down programs will merge the cuts of parallel programs.

Note that the second rule of figure B–8 and its symmetric counterpart actually introduce cuts, and may therefore increase the codegree. In this case we have to do a separate proof to reduce this codegree to 0 before we can proceed with the overall proof. This works because we can only introduce a bounded number of such cuts.

Another useful property is that the cut propagation rules do not alter the relative dependencies between the dangling interactions attached to occurrences in *first*(Σ), nor to dangling interactions within the rest of Σ. Instead, all it does is introduce dependencies between these two groups. Thus when we strip away the single cut rule at the bottom, we shall have a cut-free and eval-free $\mathcal{CSP}_{CUT}$-deduction (i.e., a $\mathcal{CSP}_I$-deduction) corresponding to the first transition of Π_p and another deduction corresponding to all the subsequent transitions, both of which reflect dependencies of Π_p. Hence we shall be able to use the Interaction Reflection theorem to glue interactions back with the store.

The problem with stores

Unfortunately, it is not quite so simple: in an evaluation deduction, transitions of different processes which are scheduled one after the other may interleave their store writes (e.g., when evaluating expressions). This small subsection outlines the way in which we can reshuffle these store transitions to segregate the transistions induced by the first transition of the program from all the rest.

When $X \subseteq O(\Sigma)$ and Σ is not a store deduction, we define $\mathit{stores}(X)$ to be the set

$$\{A \in O(\Pi_s) \mid \exists B \in X. A \sim_{\Pi} B\}$$

We define the *convex closure* of $\mathit{stores}(X)$ in Π_s, written $\mathit{stores}^{\langle \Pi_s \rangle}(X)$, to be the set

$$\{A \mid \exists B, C \in \mathit{stores}(X). B \lesssim_{\Pi_s} A \lesssim_{\Pi_s} C\}$$

that is, the set of all store transitions that occur interleaved with those associated with X. What we want in order to peel off the store transitions of the first transitions of Π_p is the situation where

$$\mathit{stores}(\mathit{first}(\Pi_p)) = \mathit{stores}^{\langle \Pi_s \rangle}(\mathit{first}(\Pi_p))$$

that is, when there are no extra transitions interleaved with the first ones.

The following proof is not difficult, just tedious. It depends on proposition B.7 and the Store Reshuffling proposition (5.1). It involves repeatedly swapping the order of store transitions, gradually migrating the unwanted transitions after those in $\mathit{stores}(\mathit{first}(\Pi_p))$. The main difficulty is that we have to ensure we do not introduce any sequencing errors — i.e., that we can still reassemble the altered store deduction to the original program deduction. This boils down to the fact that we can only commute the order of two transitions if their respective program occurrences are unrelated by $\lesssim_{\Pi_p}$. It can be shown that this is the case.

Proposition B.10 *Let $\Pi \Vdash p, \sigma \rightsquigarrow \sigma'$ be a $\mathcal{CSP}_{CUT}$ deduction. with fragments $\Pi_p \vdash p$ and $\Pi_s \vdash \sigma \rightsquigarrow \sigma'$. Then there exists a $\mathcal{CSP}_{CUT}$-deduction Π'_s such that $\Pi_p \otimes_f \Pi'_s \Vdash p, \sigma \rightsquigarrow \sigma''$ where $\sigma' \simeq \sigma''$ and* $\mathrm{stores}(\mathrm{first}(\Pi_p)) = \mathrm{stores}^{\langle \Pi'_s \rangle}(\mathrm{first}(\Pi_p))$. □

We can use the Store Reshuffling proposition to partition the store deduction Π'_s into two pieces, joined by a single cut rule. The first piece contains just those transitions induced by $\mathit{first}(\Pi_p)$, and the second contains the rest. Again, the internal dependencies between the interactions of both pieces reflect the dependencies of Π'_s. So therefore, we can simply remove the first piece and use the Interaction Reflection theorem to assemble it with the first transition.

By repeating this process, we build up a sequence of interactions. Therefore

Proposition B.11 *For all $p, \sigma, \sigma', \sigma''$, and each row of the following table, if $\mathcal{CSP}$ properly deduces the left hand column, then $\mathcal{CSP}_{CUT}$ properly deduces the middle column such that the right-hand column is true.*

$p, \sigma \rightsquigarrow \sigma'$	$p \longrightarrow \texttt{done}\ , \sigma \rightsquigarrow \sigma''$	$\sigma' \simeq \sigma''$
$p\dagger, \sigma \rightsquigarrow \sigma'$	$p \longrightarrow \texttt{aborted}, \sigma \rightsquigarrow \sigma''$	$\sigma' \simeq \sigma''$

□

Therefore, we have shown that

Proposition B.12 *For all p, σ, σ' there exists a σ'', and each row of the following table, if $\mathcal{CSP}_{SOS}$ deduces the left hand column, then $\mathcal{CSP}$ properly deduces the middle column such that the right hand column is true. Moreover, for all p, σ, σ'' there exists a σ' such that if $\mathcal{CSP}$ properly deduces the middle column then $\mathcal{CSP}_{SOS}$ deduces the left hand column such that the right-hand column is true.*

$\langle p, \sigma\rangle (\xrightarrow{\varepsilon})^{+} \sigma'$	$p, \sigma \rightsquigarrow \sigma''$	$\sigma' \simeq \sigma''$
$\langle p, \sigma\rangle (\xrightarrow{\varepsilon})^{+} \texttt{aborted}$	$p\dagger, \sigma \rightsquigarrow \sigma''$	

$\square$

Bibliography

[ABNB+86] E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, P. Inverardi, E. Karlsen, F. Mazzanti, J. Storbank Pedersen, G. Reggio, and E. Zucca. The draft formal definition of ANSI/MIL-STD 1815 Ada. Deliverable 7 of the CEC MAP project, 1986.

[Abr93] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993. Special issue for MFPS 1990.

[ACO93] I. Attali, D. Caromel, and M. Oudshoorn. A formal definition of the dynamic semantics of the Eiffel language. In *Proceedings of the sixteenth Australian Computer Science Conference (ACSC'93)*, 1993. WWW: http://zenon.intria.fr:8003/croap/centaur/papers.html.

[ACW95] I. Attali, D. Caromel, and A. Wendelborn. From a formal dynamic semantics of Sisal to a Sisal environment. In *Proceedings of the 28th Hawaiian conference of system sciences (HICSS)*. IEEE Computer Society Press, 1995.

[Acz77] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *The handbook of mathematical logic*, pages 739–782. North-Holland, 1977.

[AFdR80] K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, 1980.

[AGN94] S. Abramsky, S. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In *Proceedings of the 1994 Marktoberdorf Summer School*. Springer-Verlag, 1994. Available by anonymous ftp from `theory.doc.ic.ac.uk/papers/Abramsky/marktoberdorf.ps`.

[AI75] European Computer Manufacturers' Association and American National Standards Institute. *PL/I BASIS/1–12*, volume BSR X3.53. Computer and Business Equipment Manufacturers' Association, 1975.

[And91] James H. Andrews. *Logic programming: operational semantics and proof theory*. PhD thesis, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1991.

[AO91] K. R. Apt and E-R. Olderog. *Verification of sequential and concurrent programs.* Springer-Verlag, 1991.

[Apt81] K. R. Apt. Ten years of Hoare's logic: A survey — part 1. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[Apt83] K. R. Apt. Formal justification for a proof system for Communicating Sequential Processes. *Journal of the Association for Computing Machinery*, 30(1):197–216, 1983.

[AR87a] E. Astesiano and G. Reggio. Direct semantics for concurrent languages in the SMoLCS approach. *IBM journal of research and development*, 31(5):512–534, 1987.

[AR87b] E. Astesiano and G. Reggio. SMoLCS driven concurrent calculi. In *Proceedings TAPSOFT'87.* Springer-Verlag, 1987. LNCS 245.

[AS] Action semantics home page. WWW, `http://www.daimi.aau.dk/~thales/as/AS.html`.

[Ast91] E. Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts.* Springer Verlag, 1991.

[BCGL88] R. Bjornson, N. Carriero, D. Gelernter, and J. Leichter. Linda, the portable parallel. Technical Report Research Report 520, Yale University, 1988.

[Ber89] G. Berry. Real time programming: special purpose or general purpose languages. In G. X. Ritter, editor, *Information processing '89*, pages 11–18. North Holland, 1989.

[Ber91] Dave Berry. *Generating program animators from programming language semantics.* PhD thesis, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1991.

[Ber93a] G. Berry. Preemption in concurrent systems. In *Proceedings FSTTCS '93*, pages 72–93. Springer-Verlag, 1993. LNCS 761; WWW: http://cma.cma.fr/ftp/esterel/preemption.ps.gz.

[Ber93b] B. Berthomieu. Programming with behaviours in an ML framework: the syntax and semantics of LCS. Technical Report LAAS/CNRS Technical Report 93144, Laboratoire d'Automatique er d'Analyse des Systèmes du CNRS, Tolouse Cedex, France, 1993.

[Bes83] E. Best. Relational semantics of concurrent programs (with some applications). In D. Bjørner, editor, *Formal Description of Programming Concepts – II*, pages 431–452. North-Holland, 1983.

[BG92] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: design, semantics, implementation. *Science of Computer Programming*, 19(2):83–152, 1992. WWW: http://cma.cma.fr/ftp/esterel/BerryGonthierSCP.ps.gz.

[BH83] P. Brinch-Hansen. The programming language Concurrent PASCAL. In E. Horowitz, editor, *Programming languages: A Grand Tour*, pages 264–272. Springer-Verlag, 1983.

[BIM88] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced: preliminary report. In *ACM Symposium on the Principles of Programming Languages (POPL)*, volume 15, 1988.

[BJ78] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: the meta-language*. Springer-Verlag, 1978. LNCS 61.

[BJ82] D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall International, 1982.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *ACM Symposium on the Principles of Programming Languages (POPL)*, volume 4, pages 238–252. ACM Press, 1977.

[CEN] CENTAUR home page. WWW, http://zenon.inria.fr:8003/croap/centaur/centaur.html.

[Cen96] Pietro Cenciarelli. *Computational Applications of Calculi based on Monads*. PhD thesis, LFCS, department of computer science, the University of Edinburgh, Forthcoming, 1996.

[Cli85] W. D. Clinger et. al. The revised revised report on scheme. Technical report, Indiana University 174 and MIT Laboratory for Computer Science 848, 1985.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog,*2nd Edition. Springer-Verlag, 1984.

[Coh65] P. M Cohn. *Universal Algebra*. New York, Harper and Row, 1965.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based verification tool for finite-state systems. In *Proceedings of the Workshop on Automated Verification Methods for Finite-state Systems*. Springer-verlag, 1989. LNCS 407.

[CU73] J. Cleaveland and R. Uzgalis. What every programmer should know about grammar. Technical report, Dept. Computer Science, University of California, Los Angeles, 1973.

[CWB] Concurrency workbench home page. WWW, http://www.dcs.ed.ac.uk/packages/cwb.

[DBOM81] B. Du Boulay, T. O'Shea, and J. Monk. The black box inside the glass box: presenting concepts to novices. *International Journal of Man-Machine Studies*, 14:237–249, 1981.

[deB69] J. W. deBakker. Semantics of programming languages. *Advances in Information Systems Science*, 2, 1969.

[Des84] Th. Despeyroux. Executable specification of static semantics. In *Semantics of data types.* Springer-Verlag, 1984. LNCS 173.

[Des88] J. Despeyroux. TYPOL: a formalism to represent natural semantics. Technical Report Research report 94, INRIA, 1988.

[Dij65] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965. reprinted in [Gen68], pages 43–112.

[Dij76] E. W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.

[DJ86] M. Dam and F. Jensen. Compiler generation from relational semantics. In *European Symposium on Programming*, pages 1–30. Springer-Verlag, 1986. LNCS 213.

[dNH84] R. de Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics.* Springer-Verlag, 1990.

[DS92] Fabio Q. B. Da Silva. *Correctness proofs of compilers and debuggers: an approach based on structural operational semantics.* PhD thesis, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1992.

[EA] Evolving algebras home page. WWW, http://www.eecs.umich.edu/ealgebras.

[End77] H. B. Enderton. *Elements of Set Theory.* Academic Press, 1977.

[FC94] M. Felleisen and R. Cartwright. Extensible denotational language specifications. In *TACS '94.* Springer-Verlag, 1994. LNCS 789.

[FFHD86] M. Felleisen, D. P. Friedman, E. Hohlbecker, and B. Duba. Reasoning with continuations. *Symposium of Logic in Computer Science*, 1:131–141, 1986.

[FFHD87] M. Felleisen, D. P. Friedman, E. Hohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987. Preliminary version in [FFHD86].

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proceedings AMS Symposium of Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, 1967.

[Gar63] J. L. Garwick. The definition of programming languages. In T. B. Steel Jr., editor, *Formal language description languages*, pages 139–147. North-Holland publishing company, Amsterdam, The Netherlands, 1963.

[Gar95] P. Gardner. Equivalences between logics and their representing type theories. *Mathematical Structures in Computer Science*, 5:to appear, 1995.

[GB90] J. Goguen and R. Burstall. INSTITUTIONS: Abstract model theory for specification and programming. Technical Report ECS-LFCS-90-106, LFCS, Edinburgh University, 1990.

[Gen68] F. Genuys. *Programming Languages.* Academic Press, London, 1968.

[Gir87] J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir89] J-Y. Girard. Towards a geometry of interactions. In J. W. Gray and A. Scedrov, editors, *Categories in computer science and logic*, volume 92 of *Contemporary Mathematics*. American Mathematical Society, 1989.

[GK93] Y. Gurevich and Huggins J. K. The semantics of the C programming language. In *Proceedings of Computer Science Logic (CSL)'92*, pages 274–308. Springer Verlag, 1993. LNCS 702. Obtainable by anonymous FTP — ftp.eecs.umich.edu:groups/Ealgebras/calgebra.ps.

[GLT89] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989. ISBN 0-521-37181-3.

[GMP89] A. Giacalone, A. Mishra, and S. Prasad. Facile: a symmetric integration of concurrent and functional programming. *International journal of parallel programming*, 18(2):121–160, 1989.

[GMP90] A. Giacalone, A. Mishra, and S. Prasad. Operational and algebraic semantics for Facile: a symmetric integration of concurrent and functional programming. In *ICALP 90 (LNCS 443)*, pages 765–780. Springer-Verlag, 1990.

[Göd58] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958. English version in *Journal of Philosophic Logic* 9:133–142, 1980.

[Grä79] G. Grätzer. *Universal Algebra.* Springer-Verlag, 2 edition, 1979.

[GS86] S. Graf and J. Sifakis. A modal characterization of observational congruence on finite terms of CCS. *Information and Control*, 68:125–145, 1986.

[Gun91] C. Gunter. Forms of semantic specification. *Bulletin EATCS*, 45:98–113, 1991.

[Gur91] Y. Gurevich. Evolving algebras, a tutorial introduction. *Bulletin EATCS*, 43:254–284, Feb. 1991.

[Han93] J. Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, 1993.

[Har84] D. Harel. Dynamic logic. In *Handbook of philosophical logic vol. II*, pages 497–604. Kluwer academic publishers, 1984.

[Hen90] M. Hennessy. *The semantics of programming languages — an elementary introduction using Structural Operational Semantics.* Wiley, 1990.

[HK81] J. L. Hennessy and R. B. Kieburtz. The formal definition of a real-time language. *Acta Informatica*, 16:309–345, 1981.

[HL74] C. A. R. Hoare and P. E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3:135–153, 1974.

[HM94] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[HMT90] R. Harper, R. Milner, and M. Tofte. *The Definition of Standard ML.* MIT Press, 1990.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580,583, 1969.

[Hoa74] C. A. R. Hoare. Monitors: An operating systems structuring concept. *Communications of the ACM*, 17(10):549–557, 1974. Erratum in CACM 18(2), 1975, page 95.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[Hoo86] J. Hooman. The quest goes on: a survey of proofsystems for partial correctness of CSP. In J. W. de Bakker, W-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, pages 343–395. Springer-Verlag, 1986. LNCS 224.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ–Calculus.* London Mathematical Society Student Texts 1. Cambridge University Press, 1986. ISBN 0-521-31839-4.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison–Wesley, 1979. ISBN 0-201-02988-X.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programing language PASCAL. *Acta Informatica*, 2:335–355, 1973.

[IP91] P. Inverardi and C. Priami. Evaluation of tools for the analysis of communicating systems. *Bulletin of EATCS*, 45:158–185, 1991.

[Jab95] A. Jaber. Spécification de la sémantique dynamique du lanage Sisal. Technical report, DEA, Université de Nice-Sophia Antipolis, June 1995.

[Jon83] C. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP*, pages 321–332, 1983.

[Kah87] G. Kahn. Natural semantics. In G. Goos and J. Hartmanis, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag LNCS, Vol. 247, 1987.

[Kah93] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1993. WWW: http://www.dcs.ed.ac.uk/publications/lfcsreps/EXPORT/93/ECS-LFCS-93-257/.

[Kri68] G. Kriesel. A survey of proof theory. *Journal of Symbolic Logic*, 33(3), 1968.

[Kri71] G. Kriesel. A survey of proof theory II. In J.E. Fenstad, editor, *Proc. 2nd Scandinavian logic symposium*, pages 109–170. North Holland, 1971.

[KST94] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of extended ML. Technical Report ECS-LFCS-94-300, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1994.

[Lan64] P. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[LG81] G. M. Levin and D. Gries. A proof technique for Communicating Sequential Processes. *Acta Informatica*, 15:281–302, 1981.

[LS84] L. Lamport and F. Schneider. The “Hoare Logic” of CSP and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, 1984.

[LW69] P. Lucas and K. Walk. On the formal description of PL/I. *Annual Rev. Automatic Programming*, 6(3):105–182, 1969.

[MC81] J. Misra and K. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[Men79] E. Mendelson. *Introduction to mathematical logic 2nd edition*. Van Nostrand Co., 1979.

[MG95] M. Miculan and F. Gadducci. Modal μ-types for processes. In *Symposium of Logic in Computer Science*, volume 10, pages 221–231. IEEE, 1995.

[Mil] R. Milner. Calculi for interaction. WWW: ftp://ftp.cl.cam.ac.uk/users/rm135/ac9.ps.Z.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, The laboratory for the foundations of computer science, Edinburgh University, 1991.

[Mil94] D. Miller. Forum: a multiple-conclusion specification logic. In *Symposium of Logic in Computer Science*, pages 272–281, 94. To appear TCS, and
WWW: ftp://ftp.cis.upenn.edu/pub/papers/miller/tcs95.dvi.Z.

[Milar] D. Miller. A survey of linear logic programming. *Newsletter of the Network of Excellence on Computational Logic*, pages –, To Appear. WWW:
ftp://ftp.cis.upenn.edu/pub/papers/miller/ComputNet95/llsurvey.html.

[Mit] K. Mitchell. Language semantics and implementation. Lecture notes, The Department of Computer Science, Edinburgh University, ?

[Mit94] K. Mitchell. Concurrency in a natural semantics. Technical Report ECS-LFCS-94-311, The Laboratory for the Foundations of Computer Science, the University of Edinburgh, 1994. Available via the WWW: http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/94/ECS-LFCS-94-311.

[MLB76] M. Marcotty, H. F. Ledgard, and G. V. Bochmann. A sampler of formal definitions. *ACM Computing surveys*, 8(2):191–275, 1976.

[Mog90] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for the foundations of Computer Science, Edinburgh University., 1990.

[Mog91] E. Moggi. Notions of computations and monads. *Information and Control*, 93(1):55–92, July 1991.

[Mos74] P. D. Mosses. The mathematical semantics of Algol 60. Technical Report PRG–12, Programming research group, University of Oxford, 1974.

[Mos90] P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal*

Models and Semantics, chapter 11, pages 575–631. Elsevier Science Publishers, B.V., 1990.

[Mos91] P. D. Mosses. Action semantics. Technical Report DAIMI Fn-48, Aarhus University, Denmark, 1991. Now published as volume 26 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.

[MP93] M. C. Mayer and F. Pirri. First order abduction via tableau and sequent calculi. *Bulletin of the IGPL*, 1(1):99–117, 1993. WWW: http://www.mpi-sb.mpg.de:80/igpl/Bulletin/.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (parts I and II). *Information and Control*, 100:1–77, 1992.

[MRP] A. Mifsud, Milner R., and J. Power. Control structures I. WWW: ftp://ftp.cl.cam.ac.uk/users/rm135/cs1.ps.Z.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings of the 19th International Conference on Automata, Languages and Programming (ICALP)*, pages 685–695. Springer-Verlag, 1992. Lecture Notes in Computer Science vol. 623.

[MT90] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT press, 1990.

[MT92] R. Milner and M. Tofte. Co-induction in a relational semantics. *Theoretical Computer Science*, 17:209–220, 1992.

[NN92] H. R. Neilsen and F. Neilsen. *Semantics with applications: a formal introduction*. John Wiley and sons, 1992.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskell. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[OPTT95] P.W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Electronic notes in Theoretical computer science*, 1, 1995. On the WWW: `http://top.cis.syr.edu/users/ohearn/scir.ps`.

[Oss83] M. Ossefort. Correctness proofs of communicating sequential processes: three illustrative examples from the literature. *ACM Transactions on Programming Languages and Systems*, 5(4):620–640, 1983.

[PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *SIGPLAN '88 conference on programming language design and implementation*, pages 199–208, 1988.

[Pit91] A.M. Pitts. Evaluation logic. In G. Birtwistle, editor, *Workshops in Computing 283*. 4th Higher Order Workshop, Banf. 1990, Springer Verlag, 1991.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[Plo83] G. D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal description of programming concepts - II*, pages 199–225. North-Holland publishing company, 1983.

[Plo85] G. D. Plotkin. Denotational semantics with partial functions. Lecture at CSLI Summer School, 1985.

[Pol95] Robert Pollack. *The Theory of LEGO, a proof checker for the extended calculus of constructions*. PhD thesis, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1995.

[PR96] A. M. Pitts and J. R. X. Ross. Process calculus based upon evaluation to committed form. In U. Montanari and V. Sassone, editors, *CONCUR'96: Concurrency Theory, Proc. 7th Int. Conf. Pisa, 1996*, volume 1119 of *Lecture Notes in Computer Sceince*, pages 18–33. Springer-Verlag, Berlin, 1996.

[Pra65] D. Prawitz. *Natural Deduction*. Almquist and Wiksell, Stockholm, 1965.

[Pra71] D. Prawitz. Ideas and results in proof theory. In J.E. Fenstad, editor, *Proc. 2nd Scandinavian logic symposium*. North Holland, 1971.

[Pra91] K. V. S. Prasad. A calculus of broadcasting systems. In *TAPSOFT'91 Volume 1:CAAP*. Springer-Verlag, 1991. LNCS 493.

[Pra93] K. V. S. Prasad. Programming with broadcasts. In *CONCUR '93*. Springer-Verlag, 1993. LNCS 715.

[PT95] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus, 1995. Compiler, documentation, demonstration programs, and standard libraries; available electronically `http://www.cl.cam.ac.uk/users/bcp1000/ftp/pict`.

[Red93a] U. S. Reddy. Global state considered unnecessary: Semantics of interference-free imperative programming. In *ACM SIGPLAN Workshop. on State in Programming Languages*, pages 120–135. Technical Report YALEU/DCS/RR-968, June 1993.

[Red93b] U. S. Reddy. A linear logic model of state. Available by anonymous ftp from `theory.doc.ic.ac.uk` in /theory/papers/Reddy, 1993.

[Rep91a] J. H. Reppy. CML: a higher-order concurrent language. In *Proceedings of the SIGPLAN'91 conference on programming language design and implementation*, pages 293–305, 1991.

[Rep91b] J. H. Reppy. An operational semantics of first-class synchronous operations. Technical Report TR 91-1232, Dept. Computer Science, Cornell University, 1991.

[Ret93] C. Retoré. *Réseaux et Séquents Ordonnés.* PhD thesis, Equipe de Logique, Departement de Mathematiques, Université Paris, 1993.

[Rey81] J. C. Reynolds. *The Craft of Programming.* Prentice-Hall International, 1981.

[San93] Davide Sangiorgi. *Expressing mobility in process algebras: first-order and higher-order paradigms.* PhD thesis, Laboratory for the foundations of computer science, University of Edinburgh, 1993.

[San94] P. Sansom. *Execution profiling for non-strict functional languages.* PhD thesis, Department of Computing Science, University of Glasgow, 1994.

[Sch88] D. Schmidt. *Denotational Semantics — a methodology for language development.* Wm. C. Brown Publishers, 1988. ISBN 0-697-06849-8.

[Sch95] D. Schmidt. Natural-semantics-based abstract interpretation. In *Proceedings of the second international Static Analysis Symposium '95*, pages 1–18. Springer-Verlag, 1995. LNCS 983.

[Sco69] D. Scott. A type-theoretical alternative to CHUCH, ISWIM, OWHY, 1969. Later published in *Theoretical Computer Science* 121:411–440, 1993.

[SMo] The SMoLCS repository. FTP, ftp://ftp.disi.unige.it/pub/smolcs/.

[ST99] D. Sannella and A. Tarlecki. *Foundations of algebraic specifications and formal program development.* Cambridge University Press, To appear, 199?

[Sti88] C. Stirling. A generalization of Owicki-Gries's Hoare Logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.

[Sti92] C. Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science vol. 2*, pages 478–571. Oxford: Clarendon, 1992.

[Sun84a] G. Sundholm. Proof theory and meaning. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic III.* Kluwer Academic Publishers, 1984.

[Sun84b] G. Sundholm. Systems of deduction. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic I.* Kluwer Academic Publishers, 1984.

[Sve86] E. Svendsen. *The professional handbook of the donkey.* The donkey sanctuary, 1986.

[Tof87] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, The Laboratory for the Foundations of Computer Science, The University of Edinburgh, 1987.

[Tro73] A. S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Springer-Verlag, 1973. Lecture Notes in Mathematics 344.

[TS85] J-P. Tremblay and P. G. Sorenson. *The Theory and Practice of Compiler Writing*. McGraw-Hill, 1985.

[Und93] J. Underwood. On the computational content of classical sequent proofs. Technical Report ARO Report 94-1, U.S. Department of Defense Army Research Office, 1993.

[Und95] J. Underwood. Tableau for intuitionistic predicate logic as metatheory. Technical Report ECS-LFCS-95-321, Laboratory for the Foundations of Computer Science, Edinburgh University, 1995. WWW: http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/95/ECS-LFCS-95-321.

[VG90] J. Van Glabbeek. The linear-time branching-time spectrum. In *Proceedings CONCUR 90*, pages 278–297. Springer-Verlag, 1990. LNCS 458.

[vWMP+75] A. van Wijngaarden, B. J. Mailloux, J. E. Peck, C. H. A. Koster, and M. Sintzoff. ALGOL 68 revised report. *Acta Informatica*, 5:1–236, 1975.

[vWMPK69] A. van Wijngaarden, B. J. Mailloux, J. E. Peck, and C. H. A. Koster. Report on the algorithmic language ALGOL 68. Technical Report MR101, Mathematisch Centrum, Amsterdam, The Netherlands, 1969.

[Wat87] D. A. Watt. An action semantics of Standard ML. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *3rd Workshop on the Mathematical Foundations of Programming Language Semantics*, pages 572–598. Springer-Verlag, 1987. LNCS 298.

[Wec92] W. Wechler. *Universal Algebra for computer scientists*, volume 25 of *EATCS monographs on theoretical computer science*. Springer-Verlag, 1992.

[Weg72] P. Wegner. The Vienna definition language. *Computing Surveys*, 4(1):5–63, 1972.

[Weg76] P. Wegner. Programming languages — the first 25 years. *IEEE Transactions on computers*, pages 1207–1225, Dec 1976.

[Wex89] J. Wexler. *Concurrent programming in OCCAM 2*. Ellis Horwood (a division of John Wiley and sons), 1989.

[Win82] G. Winskel. Event structure semantics for CCS and related languages. In *Proceedings 9th ICALP*, pages 581–576. Springer-Verlag, 1982. LNCS 140.

[Win88] G. Winskel. An introduction to event structures. In *Proceedings, Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency workshop*, pages 364–397. Springer-Verlag, 1988. LNCS 354.

[Wir83] N. Wirth. MODULA: a language for modular multiprogramming. In E. Horowitz, editor, *Programming languages: A Grand Tour*, pages 273–305. Springer-Verlag, 1983.

[WN94] G. Winskel and M. Nielsen. Models for concurrency. Technical Report RS-94-12, BRICS Research Series, May 1994. To appear as a chapter in the Handbook of Logic in Computer Science.

[ZdBdR83] J. Zwiers, A. de Bruin, and W-P. de Roever. A proof system for partial correctness of dynamic networks of processes. In *Proceedings, Logics of programs*, pages 513–527. Springer-Verlag, 1983.

[ZdRvEB85] J. Zwiers, W-P. de Roever, and P. van Emde Boas. Compositionality and concurrent networks, soundness and completeness of a proof-system. In *International Conference on Automata, Languages and Programming '85*, pages 509–519. Springer-Verlag, 1985. LNCS 194.