ECOLE DES MINES
DE PARIS

Thèse

pour obtenir le grade de

Docteur de l'Ecole des Mines de Paris

Spécialité : Informatique Temps réel, Robotique, Automatique

présentée et soutenue publiquement par

Massimo MERRO

le 6 Octobre 2000

Localité dans le π-calcul et applications aux objets distribués

Locality in the π-calculus and applications to distributed objects

Directeur de Thèse : Davide Sangiorgi

Frédéric Boussinot	....	Président
Rocco De Nicola	....	Rapporteur
Joachim Parrow	....	Rapporteur
Giuseppe Castagna	....	Examinateur
Davide Sangiorgi	....	Examinateur

Abstract

This thesis is concerned with the π*-calculus*, the prototypical name-passing *calculus of mobile processes*, i.e. processes with a dynamically changing communication topology. Through the years, several variants and/or subcalculi of the π-calculus have been proposed to naturally model, and prove properties of, important classes of *distributed concurrent systems.*

We introduce the *Local* π*-calculus*, Lπ, an asynchronous variant of the π-calculus, where the process receiving a name may only use it in output actions. Lπ can be seen as a simple basis of concurrent and/or distributed languages such as *Pict* and *Join*.

We study the foundational theory of Lπ. The behavioural equivalence we adopt in the calculus is *barbed congruence*. We give two *labelled bisimilarities* which characterise barbed congruence in Lπ. The first is based on an embedding of Lπ into a subcalculus where all names emitted are private. The second is based on a new labelled transition system which reveals what is observable in Lπ. Our bisimilarities form congruence relations and can be enhanced by means of up-to proof techniques. In the characterisation proofs, no matching construct for testing equality between names is used.

One of the main motivations of Lπ is its rich algebraic theory. A certain number of applications of this theory is presented, including a fully-abstract encoding of polyadic Lπ into monadic Lπ.

Much of the theory of Lπ is generalised to the asynchronous π, π_{a}, in which there is no constraint on received names. Applications of this new theory of π_{a} include: (i) a fully-abstract encoding of *external mobility* (global names are communicated) in terms of *internal mobility* (only private names are communicated); (ii) a fully-abstract encoding of an asynchronous variant of the *Fusion calculus* into π_{a}.

Finally, we use the theory of Lπ to study *object migration* in the distributed object-based programming language *Obliq*. We give a *translational semantics* of (an appropriate abstraction of) Obliq into (an appropriate extension of) Lπ. In Obliq, objects are not physically moved during migration, rather a clone of the object is created at the target site and the original (local) object is turned into an alias for the (remote) clone. We study the correctness of this form of migration, more precisely, we investigate if an object's migration is transparent to the object's clients. We formally show that this is not the case for the original semantics. For this reason we propose an amended semantics of Obliq and give a proof of the correctness of object migration with respect to this new semantics.

To my parents

Acknowledgements

I would like to thank my supervisor, Davide Sangiorgi. He was always encouraging me and ready to give enlightening suggestions all over these years. His constant support has been crucial for the development of this thesis.

I am also indebted to my former italian supervisor, Andrea Maggiolo-Schettini, for having introduced me to research, and for his precious "life lessons".

The meije (now mimosa) project at INRIA Sophia-Antipolis was a stimulating place to do research. In particular, I would like to thank Gérard Boudol, Ilaria Castellani, and Silvano Dal-Zilio for both their friendship and the many interesting discussions I had with them.

The fruitful collaboration with Hans Hüttel, Josva Kleist, and Uwe Nestmann at BRICS in Aalborg was a very exciting and rewarding experience. I really want to thank all of them for the nice time I had in Aalborg during my several visits.

Many thanks to Gérard Boudol, Ilaria Castellani, Silvano Dal-Zilio, Matthew Hennessy, Josva Kleist, Guy McCusker, Julian Rathke, António Ravara, Bernhard Reus, and Christine Röckl, who each read (part of) a draft of this thesis. A special thank to Uwe Nestmann who helped me very much in writing down the (non-easy) part of the thesis concerning our joint work. Finally, thank to Pascal Zimmer which helped me in writing the french résumé.

The research has been supported by scholarships from "Ministero Italiano dell'Università e della Ricerca Scientifica e Tecnologica", INRIA Sophia-Antipolis, and EU Marie Curie TMR.

This is a revised version of my thesis; I have benefited a lot from the criticisms of my examiners.

Contents

Résumé en français **1**

1 Introduction **15**
1.1 Algebraic models for concurrent languages . . . 17
1.2 The fusion of names . . . 20
1.3 Mobile objects as mobile processes . . . 21
1.4 Related work . . . 23
1.5 Outline of the thesis . . . 25

2 The π-calculus **27**
2.1 Syntax . . . 27
2.2 Operational semantics . . . 28
2.2.1 Reduction semantics . . . 28
2.2.2 Labelled transition semantics . . . 29
2.3 Behavioural equivalences . . . 30
2.4 Up-to proof techniques . . . 33
2.5 Variants of the π-calculus . . . 34
2.5.1 The asynchronous π-calculus . . . 34
2.5.2 The πI-calculus . . . 37
2.5.3 The Join calculus . . . 37
2.5.4 The Fusion Calculus . . . 38
2.6 Correctness criteria for encodings . . . 39

3 The Local π-calculus **41**
3.1 The syntax . . . 41
3.2 The link process . . . 42
3.3 A "translation-based" proof technique . . . 43
3.3.1 Background on $[\![\cdot]\!]$ . . . 44
3.3.2 The proof technique . . . 45
3.4 A "run-time" proof technique . . . 48
3.5 Two characterisations of barbed congruence . . . 51
3.6 Up-to link proof techniques . . . 54

4 Applications of the theory of Local π 57
4.1 Expressing substitution in Local π 57
4.2 The replication theorems 58
4.3 The delayed input 59
4.4 Encodings of the λ-calculus 64
4.5 Some properties of the Join calculus 64
4.6 External versus internal mobility 66
4.7 Operational soundness of CPS axioms 68

5 Encoding polyadicity in Local π 71
5.1 The polyadic Local π 71
5.2 Encoding polyadicity 72
5.3 Full abstraction of the encoding 75
5.4 Encoding polyadicity in Fusion calculus 77
5.5 Discussion 78

6 From Local π to asynchronous π 81
6.1 Asynchronous π and equators 81
6.1.1 A characterisation of barbed congruence in π_{a} 83
6.1.2 Internal mobility in asynchronous π 86
6.2 Fusions of names versus equators 87
6.2.1 The asynchronous monadic fusion calculus 88
6.2.2 The encoding 89
6.2.3 Two applications of the encoding 91

7 Migrating objects as mobile processes 93
7.1 The calculus $\mathrm{L}\pi^{+}$ 93
7.1.1 Terms and types 93
7.1.2 Operational and behavioural semantics 96
7.2 Øjeblik: syntax and informal semantics 102
7.3 Towards a formal semantics for Øjeblik 105
7.3.1 On the stability of alias chains 105
7.3.2 Cyclic alias chains 106
7.3.3 On forwarding requests along the alias chain 107
7.4 A translational semantics for Øjeblik 108
7.5 Behavioural equivalence for Øjeblik 114
7.6 Safety of surrogation as an equation 115
7.7 Unsafe surrogations 116
7.7.1 Countering self-inflicted surrogations 116
7.7.2 Countering external surrogations 119
7.8 The amended semantics 121
7.9 Properties of object managers 122
7.10 On the absence of internal surrogation 125

7.11 On committing external surrogations . . . 127
7.12 External Surrogation is Safe . . . 129

8 Conclusions **133**

A Proofs **135**
A.1 Proofs of Chapter 3 . . . 135
A.1.1 Proof of Lemma 3.3.12 . . . 135
A.1.2 Proof of Lemma 3.4.3 . . . 137
A.1.3 Complement to the Proof of Lemma 3.5.1 . . . 140
A.1.4 Proof of Theorem 3.6.4 . . . 142
A.2 Proof of Lemma 4.3.2 . . . 145
A.3 Proof of Section 5.3 . . . 147
A.3.1 Proof of Lemma 5.3.2 . . . 147
A.3.2 Proof of Lemma 5.3.7 . . . 149
A.4 Proofs of Chapter 7 . . . 150
A.4.1 Proof of Lemma 7.1.10 . . . 150
A.4.2 Proof of Lemma 7.9.2 . . . 151
A.4.3 Proof of Lemma 7.11.2 . . . 152
A.4.4 Proof of Lemma 7.11.3 . . . 155
A.4.5 Proof of Lemma 7.11.4 . . . 156
A.4.6 Proof of Lemma 7.11.5 . . . 157
A.4.7 Proof of Lemma 7.11.6 . . . 159

Résumé en français

Dans cette thèse, nous allons étudier les *systèmes parallèles*, c'est-à-dire les systèmes dans lesquels un certain nombre d'entités indépendantes – les processus – interagissent entre elles.

Les *calculs de processus* (parfois appelés *algèbres de processus*) sont l'un des formalismes les plus étudiés pour décrire les systèmes parallèles. Un grand nombre de calculs de processus différents ont été proposés dans la littérature, par exemple CCS par Milner [50], CSP par Hoare [32] et ACP par Bergstra et Klop [8]. Tous sont conçus autour de l'idée centrale d'*interaction* ou de *communication* entre processus. Ces formalismes proposent généralement un petit ensemble d'opérateurs qui correspondent chacun à une manière primitive de construire un système à partir de ses sous-composants. Parmi les opérateurs classiques, on trouve: la *composition parallèle*, le *séquencement* et le *choix non-déterministe*. Les calculs de processus permettent de décrire des systèmes à différents niveaux d'abstraction. De façon générale, une description abstraite peut être vue comme une *spécification* du système désiré; une version plus détaillée, comme une *implémentation*.

Une notion critique pour un calcul de processus est celle d'*égalité comportementale* entre processus. Par exemple, une préoccupation importante est de vérifier qu'une implémentation respecte sa spécification. Grossièrement, deux processus sont comportementalement égaux si on ne peut les distinguer en interagissant avec eux. L'égalité comportementale est une relation d'équivalence entre processus. De préférence, elle doit être préservée par tous les opérateurs du calcul. Ceci permet d'interchanger deux processus équivalents dans n'importe quel système construit à partir des opérateurs du calcul, sans modifier le comportement de ce système.

Une limitation des calculs de processus comme CCS, CSP et ACP est que l'ensemble des canaux reliant les processus est fixé d'après leur syntaxe. Par conséquent, ces calculs ne peuvent pas être utilisés pour décrire des *systèmes mobiles*, c'est-à-dire des systèmes dont la topologie des liaisons change dynamiquement. L'une des premières tentatives pour modéliser la mobilité dans un cadre algébrique est le langage ECCS d'Engberg et Nielsen [23]. Une simplification élégante d'ECCS est le *π-calcul* de Milner, Parrow et Walker [57], que l'on considère maintenant comme le *calcul prototype pour les processus mobiles*. En π-calcul, un nom reçu sur un canal peut être lui-même utilisé comme un nom de canal en émission ou en réception. De plus, un nom privé peut être envoyé au-delà de sa portée initiale (extrusion de portée). La différence essentielle entre le π-calcul et CCS réside dans la combinaison de l'envoi de noms de canaux et de l'extrusion de portée.

Le π-calcul est un formalisme très expressif. Il permet de coder: (i) les structures de données, comme les booléens ou les listes [51]; (ii) le λ-calcul [56, 74]; (iii) les communications d'ordre supérieur (lorsqu'on peut transmettre des processus à la place des noms [72]). En outre, le π-calcul peut être utilisé comme un outil de raisonnement sur les *langages orientés objet* [90, 39, 75, 42, 40].

Diverses variantes du π-calcul furent étudiées afin de mieux reproduire certaines caractéristiques spécifiques aux systèmes parallèles. Dans cette thèse, nous considérerons quelques-unes de ces variantes.

- Honda et Tokoro [34], et indépendamment Boudol [13], ont défini le *π-calcul asynchrone*, un sous-calcul du π-calcul qui modélise les communications asynchrones. Ces dernières sont plus faciles à implémenter et plus proches des primitives de communication proposées par les *systèmes répartis*. A partir du π-calcul asynchrone, Pierce et Turner ont développé le langage de programmation parallèle *Pict* [68].
- Sangiorgi a défini un sous-calcul du π-calcul, appelé πI, dans lequel seuls des noms privés peuvent être émis. Ce calcul πI garde la quasi totalité de la puissance expressive du π-calcul et possède une théorie plus simple [77]. Il est intéressant de noter que le π-calcul asynchrone peut être encodé en πI [10]. Une légère variante de cet encodage est cruciale pour la théorie développée dans cette thèse.
- Fournet et Gonthier ont proposé le *Join-calcul* [24], un descendant du π-calcul asynchrone conçu spécialement pour faciliter l'implémentation répartie des mécanismes de canaux.
- Tout récemment, Parrow et Victor ont défini le *calcul Fusion* [65], une simplification du π-calcul ne contenant qu'un seul opérateur de liaison. Dans le calcul Fusion, les noms de canaux peuvent être "fusionnés", c'est-à-dire qu'on peut les utiliser de façon interchangeable durant l'exécution.

Une méthode standard pour comprendre de nouveaux calculs est de les coder à partir d'un calcul plus familier ou mieux établi. Lorsqu'on étudie un encodage entre deux calculs, il est nécessaire d'avoir quelques *critères de correction* pour cet encodage. Le critère de correction le plus satisfaisant pour les encodages est la *complète adéquation* qui relie les calculs source et destination au niveau de l'équivalence comportementale. Plus précisément, deux termes d'un calcul source sont comportementalement équivalents si et seulement si leurs traductions sont comportementalement équivalentes. L'adéquation complète peut être enrichie par une *correspondance opérationnelle* entre un terme et sa traduction. Une correspondance opérationnelle met en relation les pas d'exécution, tels qu'ils sont définis par la sémantique opérationnelle des calculs source et destination. L'adéquation complète d'un encodage possède deux parties: l'*adéquation*, qui affirme que l'équivalence entre les traductions de deux termes source implique celle des termes source eux-mêmes; et la *complétude*, qui affirme la réciproque. L'adéquation est parfois suffisante pour affirmer que le calcul destination est au moins aussi expressif que le calcul source, mais elle ne dit pas grand chose sur la “qualité” de l’encodage. La complétude est beaucoup plus difficile à obtenir, mais combinée avec l’adéquation, on obtient un résultat beaucoup plus précis. Par exemple, la complétude garantit que toutes les lois algébriques dans le calcul source ont leurs équivalents dans le calcul destination.

Modèles algébriques pour les langages parallèles

Le π-calcul Les processus en π-calcul sont construits à partir des opérateurs d’*inaction*, de *choix gardé non-déterministe*, de *composition parallèle*, de *restriction* (pour restreindre la portée d’un nom), de *comparaison* (pour tester l’égalité entre deux noms) et de *réplication* [1] (pour créer autant de copies en parallèle que nécessaire) d’après la grammaire suivante:

$$P ::= \mathbf{0} \;\Big|\; \Sigma_{i\in I}\alpha_i.\,P_i \;\Big|\; P \mid P \;\Big|\; (\boldsymbol{\nu} a)P \;\Big|\; [a=b]P \;\Big|\; !P$$

où

$$\alpha ::= a(b) \;\Big|\; \overline{a}b$$

désignent respectivement les préfixes de réception et d’émission.

En π-calcul, la communication entre deux processus est synchrone, autrement dit les actions d’envoi et de réception d’un message ont lieu au même moment. Par exemple, considérons le processus suivant:

$$(\boldsymbol{\nu} b)(\overline{a}b.\,P \mid R) \mid a(x).\,Q \longrightarrow (\boldsymbol{\nu} b)(P \mid R \mid Q\{^b\!/\!_x\}).$$

Le canal privé b est transmis suivant a. Par suite de quoi, la portée de b doit être étendue à $Q\{^b\!/\!_x\}$, à savoir le processus Q dans lequel le paramètre formel x a été instantié par b (en évitant les captures de noms). Le calcul est synchrone parce que les continuations P et Q ne sont libres de continuer à évoluer qu’après avoir consommé leurs préfixes.

Le π-calcul asynchrone Lorsqu’on modélise les *systèmes répartis*, on ne peut faire aucune hypothèse sur les vitesses relatives des composants. Dans ce cas, il est plus naturel d’adopter les *communications asynchrones* qui n’obligent pas l’émetteur et le récepteur à se synchroniser quand ils échangent des messages.

En *π-calcul asynchrone*, π_{a} en abrégé, on impose une forte distinction entre les primitives de réception et d’émission. L’opérateur d’émission n’est pas bloquant, c’est-à-dire que le préfixe d’émission $\overline{a}b.\,P$ est remplacé par une particule d’émission plus simple $\overline{a}b$ qui n’a pas de continuation. Le calcul est asynchrone dans le sens que la lecture d’un message n’a pas d’effet sur le processus qui a créé ce message. Par conséquent, en π_{a}, la réception n’est pas observable.

[1]Parfois, la réplication est remplacée par la récursion.

Le π-calcul Local On considère que le π-calcul asynchrone est à la base des langages expérimentaux de programmation parallèle et/ou répartie (ou des propositions de langages de programmation) comme Pict [68], Join [24], π_1 [3] ou le calcul Bleu [15]. Cependant, en y regardant de plus près, les langages de programmation ci-dessus sont basés sur un calcul encore plus simple, dans lequel:

(a) le récepteur d'un nom ne peut l'utiliser que pour des actions d'émission; autrement dit, seule la *capacité à émettre* des noms peut être transmise;
(b) il n'y a pas de constructeur de *comparaison* pour tester l'égalité entre noms (ou de constructeurs similaires comme le test d'inégalité). [2]

Ces restrictions sont explicites en Join et dans les variantes récentes du calcul Bleu. En Pict, (b) est explicite; (a) ne l'est pas, mais la plupart des programmes Pict la respectent.

On appelle *π-calcul Local*, Lπ en abrégé, le π-calcul asynchrone avec les simplifications supplémentaires (a) et (b). En Lπ, d'après la restriction (a), les récepteurs d'un canal sont *locaux* au processus qui a créé ce canal. Plus précisément, dans un processus $(\boldsymbol{\nu}a)P$, les réceptions sur le canal a sont déterminées statiquement: aucune réception sur a ne peut être créée par la suite, ni à l'intérieur ni à l'extérieur de P. Par exemple, le processus:

$$(\boldsymbol{\nu}a)(\bar{b}a.\,P \mid a(x).\,Q) \;\mid\; b(z).\,z(y).\,R$$

n'est pas dans Lπ parce qu'après une réduction suivant b, une nouvelle réception sur a est créée.

La *propriété de localité* sur les canaux rend Lπ particulièrement adapté pour donner une sémantique et raisonner sur les langages parallèles ou répartis orientés objet. Par exemple, la propriété de localité peut garantir l'unicité des objets – une caractéristique fondamentale des objets: dans les langages orientés objet, le nom a d'un objet peut être transmis; le récepteur peut utiliser a pour accéder à ses méthodes, mais il ne peut pas créer de nouvel objet appelé a. Lorsqu'on représente les objets en π-calcul, ceci se traduit généralement par la contrainte que le processus qui reçoit le nom de l'objet ne peut l'utiliser qu'en émission.

Dans cette thèse, nous étudions la théorie opérationnelle et algébrique de Lπ. Nous nous intéressons aux équivalences comportementales basées sur la bisimulation, plus précisément à la *congruence à barbes* [58]. La congruence à barbes identifie des processus qui, très grossièrement, génèrent le même ensemble d'*actions observables* dans tous les contextes. Comme les autres formes de bisimulation définies suivant le contexte, la congruence à barbes est sensible à l'ensemble des opérateurs du calcul. Lπ est un sous-calcul de π_{a} et du π-calcul, il possède donc moins de contextes. Ceci nous permet de gagner des égalités de processus utiles. A cet égard, la plus importante loi algébrique du π-calcul asynchrone qui ne soit pas dans la théorie du π-calcul synchrone est la *loi*

[2] On peut voir (b) comme une conséquence de (a) puisque tester l'identité d'un nom demande plus que la capacité à l'émettre.

d'asynchronie:

$$a(x).\, \overline{a}x \;=\; \mathbf{0}.$$

En gros, cette loi est vraie parce qu'en π_a les réceptions ne sont pas observables. Bien que la loi d'asynchronie soit utile (elle est par exemple utilisée par Nestmann et Pierce pour prouver la correction de l'encodage de la somme gardée [62]), il semble juste de dire que la restriction aux contextes asynchrones ne permet pas de gagner beaucoup plus.

Par contraste, l'asynchronie a de fortes conséquences sémantiques avec les simplifications (a) et (b). Considérons les lois suivantes:

$$\overline{a}b \;=\; (\boldsymbol{\nu}c)(\overline{a}c \mid !c(x).\, \overline{b}x) \tag{1}$$

$$(\boldsymbol{\nu}a)(!a(x).\, R \mid P \mid Q) = (\boldsymbol{\nu}a)(!a(x).\, R \mid P) \mid (\boldsymbol{\nu}a)(!a(x).\, R \mid Q) \tag{2}$$

$$(\boldsymbol{\nu}a)(!a(x).\, R \mid C[\overline{a}b]) \;=\; (\boldsymbol{\nu}a)(!a(x).\, R \mid C[R\{^b\!/\!_x\}])\ ^3 \tag{3}$$

$$(\boldsymbol{\nu}c)(\overline{a}c \mid c(x).\, \mathbf{0}) \;=\; (\boldsymbol{\nu}c)(\overline{a}c) \;=\; (\boldsymbol{\nu}c)(\overline{a}c \mid \overline{c}b) \tag{4}$$

La Loi 1, dans laquelle $c{\neq}b$, identifie des processus qui peuvent réaliser syntaxiquement des émissions différentes: le processus du membre gauche émet un nom global b, tandis que celui du membre droit émet un nom privé c. Le processus réexpéditeur $!c(x).\, \overline{b}x$ rend les deux processus impossible à distinguer. La Loi 2 est une loi de distributivité pour des ressources répliquées connue comme l'un des *théorèmes de réplication* de Milner [51]. En général, cette loi est vraie à condition que le nom a ne soit jamais transmis et jamais utilisé en réception par les processus P, Q et R. En Lπ, la Loi 2 est encore vraie lorsque le nom a est transmis par P, Q ou R. La Loi 3 fait penser à l'*expansion en ligne*, une technique d'optimisation pour les langages fonctionnels qui remplace les appels de fonction (la particule $\overline{a}b$) par des copies du corps de la fonction (le processus $R\{^b\!/\!_x\}$). La Loi 3 est vraie en Lπ à condition que le nom a n'apparaisse pas libre en réception dans le processus R ou le contexte $C[\cdot]$. Enfin, la Loi 4 modélise une forme de ramasse-miettes. Il faut noter qu'après l'émission initiale, les processus dérivés sont très différents.

Toutes les lois précédentes sont valides (sous réserve des conditions spécifiées) en Lπ, mais fausses en π_a et en π-calcul. D'autres lois ainsi que leurs applications seront présentées dans la suite de cette thèse.

Les techniques de preuve pour Lπ peuvent être exploitées pour raisonner sur des langages comme Pict, Join, le calcul Bleu et π_1, soit en adaptant directement ces techniques à ces langages, soit au moyen d'encodages en Lπ. De plus, la théorie de Lπ (par exemple, ses propriétés algébriques et ses bisimulations étiquetées) est aussi utile pour les calculs dans lesquels l'utilisation de certains noms va au-delà de la syntaxe de Lπ. Par exemple, on pourrait envisager un ensemble distinct de noms synchrones, ou de noms qui pourraient être testés pour l'identité (voir le Chapitre 7). On pourrait utiliser un *système de types* pour distinguer les "noms Lπ" et les autres noms, et appliquer la théorie de Lπ dans le premier cas.

[3] $C[\cdot]$ est un contexte.

Comme toujours dans les calculs par passage de noms, les développements théoriques de Lπ ont été menés dans un cadre monadique, dans lequel un et un seul nom est échangé durant chaque communication. Par ailleurs, la plupart des applications des calculs par passage de noms utilisent des communications polyadiques, c'est-à-dire des communications avec des tuples de noms. Un problème intéressant est donc de chercher si les calculs par passage de noms monadiques et polyadiques ont le même degré d'expressivité. Dans cette thèse, nous donnons un encodage du calcul Lπ polyadique dans le calcul Lπ monadique, inspiré par l'encodage de Milner du π-calcul polyadique dans le π-calcul monadique [51]. Contrairement à l'encodage de Milner, notre encodage est complètement adéquat par rapport à la congruence à barbes. Les résultats de complète adéquation de ce genre ne sont pas triviaux. Des résultats montrant que les calculs monadiques sont aussi expressifs que leurs extensions polyadiques sont utiles pour les développements théoriques car il est techniquement plus simple de travailler sur un calcul monadique.

La partie de cette thèse concernant le calcul Lπ est basée sur des idées de [49] (travail commun avec Davide Sangiorgi) et [47].

La fusion de noms

En π-calcul Local, les noms reçus ne peuvent être utilisés que dans des actes d'émission. Dans la théorie de Lπ, cette inversion de polarité sur les noms est modélisée par l'utilisation de processus spécialisés, appelés des *liens* [77, 10] (ou des *réexpéditeurs*), qui se comportent comme des buffers recevant des noms sur un canal et les renvoyant sur un autre canal.

En π-calcul asynchrone, il n'y a aucune contrainte sur les noms reçus, autrement dit le récepteur d'un nom peut l'utiliser à la fois dans des actes de réception et d'émission. Cette utilisation simple des noms peut être modélisée dans la théorie de π_a en faisant appel à des processus spéciaux appelés *égalisateurs*, étudiés pour la première fois dans [37]. Tandis que les processus liens réexpédient l'information d'un canal a vers un autre canal b, les égalisateurs retransmettent l'information dans les deux directions, à savoir de a vers b et vice versa. Honda et Yoshida ont montré qu'en π-calcul asynchrone, les égalisateurs induisent une relation d'équivalence sur les noms [37].

Le calcul Fusion Dans le calcul Fusion de Parrow et Victor [65], on peut modéliser directement les relations d'équivalence sur les noms. L'idée est que deux noms peuvent être "fusionnés" et utilisés de façon interchangeable par la suite.

Le calcul Fusion a principalement deux motivations: (i) proposer une variante non bloquante de la primitive de réception; une telle forme de réception permet de modéliser un *λ-calcul avec appel par nom fort*, c'est-à-dire un λ-calcul avec appel par nom dans lequel il est possible de réduire sous les lambda-abstractions; (ii) modéliser l'affectation sur des structures partagées dans les langages par contraintes répartis [86].

La principale nouveauté du calcul Fusion est que la primitive de réception ne lie pas les noms, autrement dit qu'il n'existe qu'un seul opérateur de liaison, noté $(a)P$, signifiant que le nom a est privé dans P. Par conséquent, dans le calcul Fusion et à la différence du π-calcul, la communication a un "effet global". Afin de mieux comprendre ce que cela signifie, comparons les règles de réduction en π-calcul et en Fusion (monadique) [4]:

$$\pi: \qquad \overline{a}b.\, P \mid a(x).\, Q \mid R \longrightarrow P \mid Q\{^b\!/\!_x\} \mid R$$

$$\text{Fusion:} \quad (x)(\overline{a}b.\, P \mid ax.\, Q \mid R) \longrightarrow (P \mid Q \mid R)\{^b\!/\!_x\}$$

En π-calcul, les effets de la communication (c.-à-d. la substitution $\{^b\!/\!_x\}$) sont confinées au processus récepteur Q, tandis que dans le calcul Fusion tous les processus dans la portée de (x) sont affectés.

L'encodage du π-calcul en Fusion est très simple [28]. Cet encodage justifie l'affirmation que le calcul Fusion hérite de toute la puissance expressive du π-calcul. Il reste à déterminer si le calcul Fusion est plus expressif que le π-calcul. Dans cette thèse, nous montrons que cela n'est pas le cas pour le calcul asynchrone. Plus précisément, nous présentons un encodage du calcul Fusion asynchrone dans le π-calcul asynchrone, basé sur les égalisateurs. Nous prouvons que cet encodage est complètement adéquat relativement à la *congruence à barbes close*, une variante de la congruence à barbes, introduite dans cette thèse, dans laquelle les contextes lient tous les noms des processus testés. Tout comme dans la définition des *équivalences de test* [21], ces contextes signalent leur succès en émettant sur des noms qui n'apparaissent pas dans les processus testés. Dans le calcul Fusion, la transmission de noms entre deux processus ne peut avoir lieu qu'avec des noms restreints, c'est-à-dire des noms dont la portée est limitée. Pour cette raison, nous pensons que la congruence à barbes close représente une équivalence comportementale plus satisfaisante pour le calcul Fusion.

Cette partie de la thèse présente des idées reprises de [45, 46].

Les objets mobiles en tant que processus mobiles

Les systèmes à base d'objets mobiles ont été étudiés depuis plus d'une décennie, depuis leurs premiers représentants, comme *Emerald* [41]. Récemment, toute une série d'ateliers scientifiques furent dédiés aux systèmes à base d'objets mobiles [88]. Mais, tandis que la mobilité dans les systèmes à objets possède des avantages bien connus, par exemple comme support pour l'équilibrage de charge ou les agents mobiles, personne ne s'accorde sur la façon dont les objets mobiles devraient être vraiment réalisés.

Le travail présenté au Chapitre 7 est dans la lignée des recherches qui tendent à utiliser le π-calcul comme un outil de raisonnement pour les langages de programmation orientée objet. De précédents travaux sur la sémantique des objets en tant que

[4]Afin de rester simple, nous considérons le calcul Fusion monadique, la généralisation au cas polyadique est légèrement plus complexe.

processus ont montré la valeur de cette approche: tandis que [90, 39, 75, 42] se sont intéressés seulement à donner une sémantique formelle aux langages orientés objet et aux caractéristiques de ces langages, d'autres travaux [66, 79] furent guidés par un problème de programmation spécifique.

Migration = Clonage ; Aliasing Notre travail s'attaque à un problème d'*Obliq*, le langage de programmation *répartie à portée lexicale* de Cardelli [18]. Dans ce cadre, Cardelli proposa de dériver la migration des objets à partir de deux autres primitives, le *clonage* et l'*aliasing*, en les appliquant l'une après l'autre.

Obliq est bien adapté à ce style de migration. La *limitation de portée lexicale* dans un cadre réparti rend l'analyse de programmes plus aisée puisque la liaison des variables est complètement déterminée par leur emplacement dans le texte du programme, et non par le site d'exécution. En Obliq, les valeurs immuables peuvent être librement copiées d'un site à l'autre, tandis que les valeurs modifiables restent fixes. Seules les références aux valeurs modifiables peuvent être transmises entre les différents sites. Par conséquent, puisque les objets sont modifiables, la migration d'un objet ne déplace pas physiquement l'objet, mais crée à la place un *clone* de l'objet sur le site destination et transforme ensuite l'objet original (local) en un *alias* – parfois appelé un *proxy* – pour le nouvel objet (distant).

Dans les programmes parallèles et répartis, il est important que certains changements d'état, dans certaines parties du système en fonctionnement, aient lieu de façon transparente du point de vue du reste du système. S'assurer que l'implémentation de tels changements d'état est bien transparente peut s'avérer une tâche difficile puisque le programmeur doit en principe anticiper tous les scénarios d'exécution possibles. La question se pose donc naturellement de savoir si la migration des objets en Obliq est transparente pour les clients de l'objet, et comment ceci peut être prouvé formellement.

Quand la migration des objets est-elle correcte ? Intuitivement, la migration d'un objet a vers un autre site a lieu de façon transparente, ou sure, si (1) après la migration, un client de a ne peut pas dire si a est maintenant un alias, et si (2) pendant la migration, il n'est pas possible d'interagir avec a d'une manière qui empêche l'opération de migration de se terminer complètement. C'est pourquoi, en Obliq, les objets mobiles doivent être *protégés* et *sérialisés*: la protection garantit que les alias sont persistants; la sérialisation garantit l'atomicité de l'opération de migration en deux phases. Malheureusement, Obliq ne possède pas de sémantique formelle, sauf dans une note non publiée de Talcott [83], qui propose une sémantique par configuration pour un sous-ensemble d'Obliq n'incluant pas la migration; aucun traitement formel de la sûreté de la migration n'a donc été possible jusqu'à présent. Le but de notre travail est de remédier à cette lacune de façon à pouvoir raisonner sur la migration.

De la migration à la délégation Puisqu'Obliq est à *portée lexicale*, nous pouvons ignorer les aspects liés à la répartition, au moins en ce qui concerne les résultats des exécutions en Obliq, sauf en cas de panne de site. Poursuivant cette idée, intéressons-nous à Øjeblik [61], un langage à objets qui représente le noyau parallèle d'Obliq, mais que l'on peut aussi voir comme une extension parallèle du Calcul d'Objets Impératif [1]. Øjeblik possède une abstraction sans répartition de la migration appelée *délégation*. De même que la migration, la *délégation* d'un objet a est définie comme la création d'un clone b de a et la transformation de a en un proxy pour b, qui réexpédie à b les futures requêtes pour des méthodes de a. La principale différence par rapport à la migration est que ni a ni b ne sont rattachés à un site.

La correction sous la forme d'une équation Nous donnons une définition formelle de la *correction* pour la délégation des objets en Øjeblik, que l'on peut très facilement adapter à la migration des objets en Obliq. Intuitivement, afin d'être correct, la délégation (resp. la migration) d'un objet doit être transparente pour les clients de cet objet, autrement dit l'objet doit se comporter de la même manière avant et après la délégation (resp. la migration). Nous formalisons ce concept au moyen d'une simple équation:

$$a.\mathsf{ping} \doteq a.\mathsf{surrogate}$$

où le membre gauche représente l'objet a avant la délégation ($a.\mathsf{ping}$ retourne l'objet résultant de l'évaluation de a), le membre droit représente l'objet a après la délégation ($a.\mathsf{surrogate}$ retourne l'objet délégué), et $\doteq$ est une équivalence contextuelle appropriée, basée sur la convergence.

Contribution Dans cette thèse, nous présentons une sémantique $\mathrm{L}\pi$ pour Øjeblik correspondant à l'implémentation d'Obliq de Cardelli [17, 18]. Nous donnons également une notion d'équivalence contextuelle, définie en termes de convergence sur les π-processus, correspondant à l'équivalence $\doteq$.

Nous montrons que cette sémantique ne respecte pas l'équation ci-dessus. Ceci sera formellement démontré par la définition de contextes en Øjeblik capables de distinguer les termes $a.\mathsf{ping}$ et $a.\mathsf{surrogate}$. Des contre-exemples similaires s'appliquent à la migration des objets en Obliq, comme nous avons pu le tester en utilisant l'interpréteur Obliq [17].

Afin d'aboutir à la correction de la délégation, nous proposons une sémantique modifiée pour Øjeblik. Nous prouvons que cette sémantique permet la délégation correctement. La preuve est divisée en une partie algébrique, dans laquelle nous pouvons utiliser de puissantes adaptations des techniques de preuve du π-calcul standard et de $\mathrm{L}\pi$, et en une partie itérative, dans laquelle nous simulons d'une manière constructive des séquences finies arbitrairement longues "jusqu'à aboutir" aux équations algébriques. Cette preuve est loin d'être évidente, et nous donnons (à notre connaissance) la première preuve formelle qu'il est possible d'implémenter la migration des objets à partir du clonage et de l'aliasing.

Nous voulons faire remarquer que la plupart des contre-exemples concernant l'implémentation d'Obliq de Cardelli furent découverts en utilisant la sémantique $\mathrm{L}\pi$ pour prouver la correction de la migration. Enfin, la grande leçon à retenir de ce travail est que *les objets parallèles nécessitent une analyse formelle* – non seulement parce qu'on peut prouver des propriétés pour le plaisir, mais aussi parce que l'analyse formelle est un bon outil de débogage.

Cette partie de la thèse est basée sur un travail en commun avec Hans Hüttel, Josva Kleist, et Uwe Nestmann [48, 61].

Travaux voisins

π-calcul Local Des calcul similaires au π-calcul Local furent étudiés dans [35, 10, 3, 92]. On peut trouver des caractérisations de la congruence à barbes pour les calculs asynchrones sur les processus mobiles dans [4, 11, 3]. Cependant, pour ces bisimilarités, les transitions correspondantes des processus possèdent les mêmes étiquettes, par conséquent on ne peut pas prouver les lois comme 1.1−1.4. Un grand défi en Lπ, au contraire des calculs de [4, 11, 3], est qu'aucun constructeur pour tester l'égalité entre noms n'est disponible. Ceci est dû au fait que le test de noms, tout comme le test d'égalité entre pointeurs dans les langages impératifs, empêche un grand nombre d'optimisations et de transformations de programmes.

D'autres études de congruences à barbes pour les processus mobiles, ou de bisimulations similaires basées sur le contexte, se trouvent dans [37, 43, 91, 28, 30] (et en ce qui concerne un langage pour la coordination dans [16]). L'article [12] étudie la congruence à barbes dans un π-calcul synchrone avec des types capacités et pas de comparaison, dans lequel Lπ peut être traité comme un cas particulier. Nos caractérisations sont plus simples que celles de [12], mais ces dernières sont plus générales car on peut les appliquer à divers dialectes du π-calcul (bien que l'extension aux langages asynchrones ne soit pas triviale). L'approche technique est différente: dans [12], les bisimulations possèdent un environnement de types (en fait, des clôtures), tandis que dans cette thèse les bisimulations sont définies directement sur les processus.

Certaines des techniques utilisées dans la Section 3.3 sont inspirées par les techniques de [78].

L'encodage de Lπ polyadique en Lπ monadique est inspiré par l'encodage de Milner du π-calcul polyadique en π-calcul monadique [51]. Il faut également mentionner Yoshida [91], et Quaglia et Walker [70], qui ont introduit deux *systèmes de types* différents pour les processus monadiques afin de faire ressortir le protocole de communication à la base de l'encodage de Milner. Plus précisément, dans [91] est définie et étudiée une notion de type de graphe. Les noeuds d'un type de graphe représentent les actions atomiques, et les arêtes un ordre d'activation entre elles. L'approche de [70] est similaire mais le système de types est plus simple. Les deux articles démontrent un résultat de complète adéquation par rapport aux équivalences contextuelles typées qui rejettent tous les contextes qui ne respectent pas le protocole imposé par l'encodage. Tandis que [91, 70] travaillent sur le π-calcul complet, notre résultat ne s'applique qu'à Lπ. D'un autre côté, nous prouvons un résultat plus fin puisque nous obtenons la complétude de l'encodage par rapport à tous les contextes monadiques sans rejet des contextes "hostiles". Gonthier et Fournet suivent une approche différente pour le *Join-calcul* [24]. Dans [24], parmi d'autres résultats, figure un encodage direct, bien que complexe, et complètement adéquat des processus polyadiques vers leur version monadique. Mis à part les différences entre les deux calculs de processus, l'encodage de [24] est techniquement assez différent du nôtre; par exemple, comme les auteurs le disent eux-mêmes, leur traduction code puis ensuite décode les tuples deux fois.

La théorie de Lπ a déjà été utilisée par Sangiorgi pour donner en encodage complètement adéquat du π-calcul Local *d'ordre supérieur* (dans lequel on peut transmettre des processus) vers le π-calcul Local [80]. Dans cet article, Sangiorgi montre que le π-calcul Local permet d'écrire des preuves plus simples que dans d'autres π-calculs [72].

Enfin, en Join et en calcul Bleu, des systèmes de types polymorphes à la ML furent introduits [27, 20]. La contrainte sur la capacité à émettre des noms est cruciale dans les deux cas. Ceci nous encourage à penser qu'il est aussi possible de définir un système de types polymorphes en π-calcul Local.

Egalité/fusion de noms dans les calculs par passage de noms Une méthode (implicite) pour modéliser les relations d'équivalence sur les noms dans les calculs par passage de noms a été proposée par Honda et Yoshida dans [37], au moyen de leurs processus égalisateurs. Une méthode plus directe a aussi été proposée dans [38], où la composition résulte de ce que Honda appelle une "substitution ouverte". Indépendamment, Parrow et Victor développèrent le calcul Fusion [65] comme une évolution polyadique de leur calcul Update [64]. La théorie du calcul Fusion fut étudiée en détail dans [87]. Le calcul Chi, indépendamment introduit par Fu [28], peut être vu comme une variante monadique du calcul Fusion.

Les objets mobiles sous forme de processus mobiles Le travail le plus proche du nôtre est [42], dans lequel une interprétation en π-calcul typé du Calcul des Objets Impératif d'Abadi et Cardelli est présentée. Au contraire de [42], nous nous intéressons dans cette thèse à un calcul d'objets parallèle. [29] par Gordon et Hankin, et [22] par Di Blasio et Fisher, décrivent deux *calculs d'objets parallèles* différents, mais n'y considèrent pas la migration des objets.

Plan de la thèse

Au Chapitre 2, nous détaillerons le π-calcul de Milner, Parrow et Walker, accompagné de ses règles de réduction et de sa sémantique opérationnelle étiquetée. Nous rappellerons les principales égalités comportementales basées sur la bisimulation et quelques techniques de preuves "modulo" une autre relation (*up-to*). Puis, nous décrirons brièvement quelques variantes du π-calcul: le π-calcul asynchrone, le πI-calcul, le Join-calcul et le calcul Fusion. Finalement, étant donné un encodage entre deux calculs de processus différents, nous rappellerons brièvement les notions de correspondance opérationnelle et de complète adéquation.

Au Chapitre 3, nous introduirons le π-calcul Local. Nous donnerons deux techniques de preuve pour prouver la congruence à barbes en $\mathrm{L}\pi$. La première est basée sur un encodage qui, très grossièrement, remplace les émissions de noms libres par des émissions de noms privés. Cet encodage utilise les processus liens. La seconde technique de preuve est basée sur un nouveau *système de transition étiquetée* (LTS pour *labelled transition system*) qui montre ce qui est observable en π-calcul Local, à savoir ce qu'un observateur extérieur se comportant comme un processus en π-calcul Local peut voir en interagissant avec un processus en π-calcul Local. Cette technique de preuve repose également sur les processus liens. Nous prouverons que ces deux techniques de preuve sont adéquates et complètes, c'est-à-dire qu'elles caractérisent complètement la congruence à barbes en $\mathrm{L}\pi$. Enfin, nous montrerons comment renforcer la technique de preuve basée sur LTS grâce à quelques techniques de preuve non-standard up-to.

Au Chapitre 4, nous montrerons quelques applications de la théorie du π-calcul Local: (i) nous donnerons une méthode alternative pour exprimer la substitution de noms à partir des autres opérateurs du calcul; (ii) nous prouverons qu'en π-calcul Local, la *réception retardée* (une forme de réception non bloquante) peut être dérivée, et nous présenterons quelques-unes de ses propriétés algébriques; (iii) nous prouverons une forme plus forte des *théorèmes de réplication* de Milner [51]; (iv) nous donnerons une optimisation de l'encodage du λ-calcul avec appel par nom, et, grâce à la réception retardée, nous en déduirons un encodage du *passage par nom fort*; (v) nous prouverons quelques lois pour le Join-calcul; (vi) nous prouverons quelques résultats d'adéquations complètes et incomplètes pour l'encodage de Boreale [10] de la *mobilité externe* (la communication de noms libres) à partir de la *mobilité interne* (la communication de noms privés); (vii) nous prouverons que la sémantique axiomatique de Thielecke du *calcul de continuations* [84] est opérationnellement adéquate.

Au Chapitre 5, nous utiliserons l'encodage de la substitution de noms présentée au Chapitre 4 pour prouver l'adéquation complète d'un encodage du π-calcul polyadique Local dans le π-calcul monadique Local. Nous proposerons également un encodage du calcul Fusion polyadique dans sa version monadique, basée sur le même principe. Nous conjecturons que l'encodage du calcul Fusion est complètement adéquat par rapport à la congruence à barbes close.

Au Chapitre 6, nous étendrons la théorie du π-calcul Local au π-calcul asynchrone. Nous donnerons une caractérisation étiquetée de la congruence à barbes en π-calcul asynchrone en termes d'égalisateurs. Nous prouverons la complète adéquation de l'encodage du calcul Fusion asynchrone en π-calcul asynchrone comme une application de la théorie développée pour le π-calcul asynchrone. Nous démontrerons que cet encodage est complètement adéquat par rapport à la congruence à barbes asynchrone et close. Etant donné qu'un encodage du π-calcul vers le calcul Fusion est déjà connu depuis [28], notre résultat montre que le calcul Fusion asynchrone et le π-calcul asynchrone ont la même puissance expressive.

Au Chapitre 7, nous donnerons une sémantique pour Øjeblik à partir de $\mathrm{L}\pi$ (ou plutôt d'une variante étendue) correspondant à la sémantique informelle d'Obliq. Nous donnerons une notion d'équivalence contextuelle, définie en termes de convergence sur les π-processus. Nous donnerons un critère formel de correction pour la délégation des objets en Øjeblik. Nous montrerons que cette sémantique ne vérifie pas ce critère de correction pour la délégation des objets. Puis, nous proposerons une variante de la sémantique et nous prouverons qu'elle respecte le critère de correction pour la délégation des objets.

La thèse conclura au Chapitre 8 par un résumé des contributions et une discussion des problèmes ouverts ainsi que des futurs travaux possibles.

Chapter 1

Introduction

In this thesis we study *concurrent systems*, i.e., systems where a number of independent entities – the processes – interact with each other.

Process calculi (sometimes called *process algebras*) are one of the most studied formalisms for describing concurrent systems. A large number of different process calculi have been proposed in the literature, such as Milner's CCS [50], Hoare's CSP [32], and Bergstra and Klop's ACP [8]. All of them are designed around the central idea of *interaction* or *communication* between processes. These formalisms usually provide a small set of operators, each of which corresponds to a primitive way of building a system from subcomponents. Classical operators are: *parallel composition*, *sequentialization*, and *non-deterministic choice*. Process calculi permit describing systems at different levels of abstraction. Generally, an abstract description may be viewed as a *specification* of the wanted system; a more detailed one may be viewed as an *implementation*.

A crucial notion in a process calculus is that of *behavioural equality* between processes; it can be used to verify that an implementation respects its specification. Roughly, two processes are behaviourally equivalent if they cannot be distinguished by interacting with them. Behavioural equalities are equivalence relations over processes. Preferably, they should be preserved by all operators of the calculus. This permits replacing two equivalent processes in any system we can build with the operators of the calculus, without altering the behaviour of the system.

A limitation of process calculi like CCS, CSP, and ACP is that the set of channels linking processes is fixed by their syntax. As a consequence, these calculi cannot be used to describe *mobile systems*, i.e., systems with a dynamically changing linkage topology. One of the first attempts to model mobility in an algebraic setting is Engberg and Nielsen's ECCS [23]. An elegant simplification of ECCS is Milner, Parrow, and Walker's *π-calculus* [57], which, by now, is considered the prototypical *calculus of mobile processes*. In the π-calculus, a name received on a channel can be used as a channel name for output or input. Furthermore, a private name can be sent outside its original scope (scope extrusion). The combination of sending channel names and scope extrusion is the essential difference between the π-calculus and CCS.

The π-calculus is a very expressive formalism. It allows to encode: (i) data structures, such as booleans or lists [51]; (ii) the λ-calculus [56, 74]; (iii) higher-order communications (where processes can be transmitted instead of names [72]). Furthermore, the π-calculus has been used as a toolbox for reasoning about *object-oriented languages* [90, 39, 75, 42, 40].

Several variants of the π-calculus have been studied to better capture specific features of concurrent systems. In this thesis we will consider some of these variants.

- Honda and Tokoro [34], and independently Boudol [13], have defined the *asynchronous π-calculus*, a subcalculus of π that models asynchronous communications. Asynchronous communications are easier to implement and closer to the communication primitives offered by *distributed systems*. On top of asynchronous π, Pierce and Turner have developed the concurrent programming language *Pict* [68].
- Sangiorgi has defined a subcalculus of π, called πI, where only private names may be emitted. The calculus πI retains much of the expressive power of the π-calculus and has a simpler theory [77]. Interestingly, the asynchronous π-calculus can be encoded in πI [10]. A slight variant of this encoding is crucial for the theory developed in this thesis.
- Fournet and Gonthier have proposed the *Join calculus* [24], an offspring of the asynchronous π-calculus specifically designed to facilitate distributed implementations of channel mechanisms.
- Most recently, Parrow and Victor have defined the *Fusion calculus* [65], a simplification of the π-calculus containing only one binding operator. In Fusion calculus, names of channels can be "fused", that is, they can be used interchangeably during the computation.

A standard manner of understanding new calculi is to code them up in terms of a more familiar or better established calculus. When studying an encoding between two calculi it is necessary to have some *correctness criteria* for the encoding. The most satisfactory correctness criterion for encodings is *full abstraction* which relates source and target calculi at the level of behavioural equivalences. Precisely, two source calculus terms are behaviourally equivalent if and only if their translations are behaviourally equivalent. Full abstraction should be enriched by an *operational correspondence* between a term and its translation. An operational correspondence relates execution steps, as defined by an operational semantics of the source and target calculi. Full abstraction of an encoding has two parts: *soundness*, which says that the equivalence between the translations of two source terms implies that of the source terms themselves; and *completeness*, which says the converse. Soundness is sometime enough to assert that the target calculus is at least as expressive as the source calculus, but it doesn't say much about the "quality" of the encoding. Completeness is much harder to achieve, but together with soundness, it provides a much finer result. For instance, completeness guarantees that all algebraic laws in the source calculus are reflected in the target calculus.

1.1 Algebraic models for concurrent languages

The π-calculus Processes in π-calculus are built from the operators of *inaction*, *non-deterministic guarded choice*, *parallel composition*, *restriction* (to restrict the scope of a name), *matching* (for testing equality between two names), and *replication* [1] (to create as many parallel replicas as needed) according to the following grammar:

$$P ::= \mathbf{0} \;\Big|\; \Sigma_{i\in I}\alpha_i.\, P_i \;\Big|\; P \mid P \;\Big|\; (\boldsymbol{\nu} a)P \;\Big|\; [a=b]P \;\Big|\; !P$$

where

$$\alpha ::= a(b) \;\Big|\; \overline{a}b$$

denote input and output prefixes, respectively.

In π-calculus, the communication between two processes is synchronous, that is, the acts of sending and receiving a message happen at the same time. For instance, consider the example below:

$$(\boldsymbol{\nu} b)(\overline{a}b.\, P \mid R) \mid a(x).\, Q \longrightarrow (\boldsymbol{\nu} b)(P \mid R \mid Q\{^{b}\!/\!_{x}\}).$$

The private channel b is transmitted along a. As a consequence, the scope of b is extended to $Q\{^{b}\!/\!_{x}\}$, that is, the process Q where the formal parameter x has been instantiated with b (with Q possibly modified to avoid name capture). The calculus is synchronous because the continuations P and Q can evolve only after they have consumed their prefixes.

The asynchronous π-calculus When modelling *distributed systems*, no assumptions can be made about the relative speeds of the components. In this case, it is more natural to adopt *asynchronous communications* which do not oblige sender and receiver to synchronise when exchanging messages.

In the *asynchronous π-calculus*, abbreviated π_{a}, a sharp distinction between input and output primitives is imposed. The output operator is non-blocking, that is, the output prefix $\overline{a}b.\, P$ is replaced by the simpler output particle $\overline{a}b$, which has no continuation. The calculus is asynchronous in the sense that consuming a message has no effect on the process that created that message. As a consequence, in π_{a}, inputs are not observable.

The Local π-calculus Asynchronous π is considered the basis of experimental concurrent and/or distributed programming languages (or proposal of programming languages) like Pict [68], Join [24], π_1 [3], and the Blue calculus [15]. However, at a closer inspection, the programming languages above are based on an even simpler calculus, where:

[1] Sometimes, replication is replaced by recursion.

(a) the recipient of a name may only use it in output actions; that is, only the *output capability* of names may be transmitted;
(b) there is no *matching* construct (or similar constructs like mismatching) for testing equality between names. [2]

These restrictions are explicit in Join and in recent variations of the Blue calculus. In Pict, (b) is explicit; (a) is not, but most Pict programs comply with it.

We call *Local* π, abbreviated $\mathrm{L}\pi$, the asynchronous π-calculus with the additional simplifications (a) and (b). In $\mathrm{L}\pi$, by restriction (a), the recipients of a channel are *local* to the process that has created the channel. More precisely, in a process $(\boldsymbol{\nu} a)P$ the inputs at channel a are statically determined: no further inputs on a may be created, inside or outside P. For instance, the process:

$$(\boldsymbol{\nu} a)(\overline{b}a.\, P \mid a(x).\, Q) \;\mid\; b(z).\, z(y).\, R$$

is not in $\mathrm{L}\pi$ because, after a reduction along b, a new input along a is created.

The *locality property* on channels makes $\mathrm{L}\pi$ particularly suitable for giving semantics to, and reasoning about, concurrent or distributed object-oriented languages. For instance, the locality property can guarantee unique identity of objects – a fundamental feature of objects: in object-oriented languages, the name a of an object may be transmitted; the recipient may use a to access its methods, but it cannot create a new object called a. When representing objects in the π-calculus, this usually translates into the constraint that the process receiving the object name may only use it in output.

In this thesis, we study the operational and algebraic theory of $\mathrm{L}\pi$. We focus on bisimulation-based behavioural equivalences, precisely on *barbed congruence* [58]. Barbed congruence equates processes that, very roughly, in all contexts give rise to the same set of *observable actions*. Like other contextually-defined forms of bisimulation, barbed congruence is sensitive to the set of operators of a calculus. $\mathrm{L}\pi$ is a sub-calculus of π_{a} and π-calculus, and therefore has fewer contexts. This allows us to gain useful process equalities. In this respect, the most important algebraic law of asynchronous π that is not in the theory of the synchronous π-calculus is the *asynchrony law*:

$$a(x).\, \overline{a}x \;=\; \mathbf{0}.$$

Roughly, this law is true because in π_{a} inputs are not observable. Although the asynchrony law is useful (it is used for instance by Nestmann and Pierce to prove the correctness of an encoding of guarded sum [62]), it seems fair to say that the restriction to asynchronous contexts does not allows to gain much more than that.

[2] We may also view (b) as a consequence of (a), since testing the identity of a name requires more than the output capability.

By contrast, asynchrony has strong semantic consequences under simplifications (a) and (b). Consider the following laws:

$$\overline{a}b \;=\; (\boldsymbol{\nu}c)(\overline{a}c \mid !c(x).\,\overline{b}x) \tag{1.1}$$

$$(\boldsymbol{\nu}a)(!a(x).\,R \mid P \mid Q) = (\boldsymbol{\nu}a)(!a(x).\,R \mid P) \mid (\boldsymbol{\nu}a)(!a(x).\,R \mid Q) \tag{1.2}$$

$$(\boldsymbol{\nu}a)(!a(x).\,R \mid C[\overline{a}b]) \;=\; (\boldsymbol{\nu}a)(!a(x).\,R \mid C[R\{^{b}\!/\!_{x}\}]) \tag{1.3}$$

$$(\boldsymbol{\nu}c)(\overline{a}c \mid c(x).\,\mathbf{0}) \;=\; (\boldsymbol{\nu}c)(\overline{a}c) \;=\; (\boldsymbol{\nu}c)(\overline{a}c \mid \overline{c}b) \tag{1.4}$$

Law 1.1, where $c{\neq}b$, equates processes that may perform syntactically different outputs: the process on the left performs the output of a global name b, whereas that on the right the output of a private name c. The forwarder process $!c(x).\,\overline{b}x$ makes the two processes indistinguishable. Law 1.2 is a distributivity law for replicated resources known as one of Milner's *replication theorems* [51]. In general, this law is true under the hypothesis that name a is never transmitted and never used in input by processes P, Q and R. In $\mathrm{L}\pi$, Law 1.2 is still true when name a is transmitted by P, Q or R. Law 1.3 is true in $\mathrm{L}\pi$ provided that name a does not appear free in input in process R and context $C[\cdot]$. This law is reminiscent of *inline expansion*, an optimisation technique for functional languages which replaces a function call (the particle $\overline{a}b$) with a copy of the function body (the process $R\{^{b}\!/\!_{x}\}$). Finally, Law 1.4 models a form of garbage collection. Notice that, after the initial output, the derivatives are very different.

All the laws above are valid (under the specified conditions) in $\mathrm{L}\pi$, but are false in π_{a} and in π-calculus. Further laws together with their applications will be presented in the remainder of the thesis.

Proof techniques for $\mathrm{L}\pi$ can be exploited to reason about languages such as Pict, Join, Blue and π_1, either by directly adapting the techniques to these languages, or by means of encodings into $\mathrm{L}\pi$. Moreover, the theory of $\mathrm{L}\pi$ (for instance, its algebraic properties and labelled bisimulations) is also useful in calculi where the usage of some names goes beyond the syntax of $\mathrm{L}\pi$. For instance, there could be a distinct set of synchronous names, or names that can be tested for identity (see Chapter 7). A *type system* could be used to distinguish between "$\mathrm{L}\pi$ names" and the other names, and the theory of $\mathrm{L}\pi$ can then be applied to the formers.

As usual in name-passing calculi, the theoretical developments on $\mathrm{L}\pi$ have been conducted in a monadic setting in which exactly one name is exchanged during each communication. On the other hand, most applications in name-passing calculi use polyadic communications, i.e., communications involving tuples of names. So, an interesting issue is to investigate whether monadic and polyadic name-passing calculi have the same expressive power. In this thesis, we give an encoding of polyadic $\mathrm{L}\pi$ into monadic $\mathrm{L}\pi$ inspired by Milner's encoding of polyadic π into monadic π [51]. Unlike Milner's encoding, our encoding is fully-abstract with respect to barbed congruence. Full-abstraction results of this kind are nontrivial. Results showing that monadic calculi are as expressive as their polyadic extensions are useful in theoretical developments, since it is technically simpler to work with a monadic calculus.

The part of this thesis concerning the calculus Lπ is based on material from [49] (joint work with Davide Sangiorgi) and [47].

1.2 The fusion of names

In Local π, received names may be used only in output actions. In the theory of Lπ, this inversion of polarity on names is modelled by making use of special processes, called *links* [77, 10] (or *forwarders*), which behave as name buffers receiving names at one channel and retransmitting them at another channel.

In asynchronous π, there is no constraint on received names, that is, the recipient of a name may use it in both input and output actions. This plain use of names can be modelled in the theory of π_a by making use of special processes called *equators*, first studied in [37]. While link processes forward the information from one channel a to another channel b, equators forward the information in both directions, that is, from a to b and vice versa. Honda and Yoshida have showed that, in asynchronous π-calculus, equators induce an equivalence relation over names [37].

The Fusion calculus In Parrow and Victor's Fusion calculus [65], equivalence relations over names can be directly modelled. The idea is that two names may be "fused" and afterwards used interchangeably.

The Fusion calculus has essentially two motivations: (i) to support a non-blocking variant of the input primitive; such a form of input allows to model *strong call-by-name λ-calculus*, that is, call-by-name λ-calculus in which it is possible to reduce under lambda abstractions; (ii) to model assignment to shared structures in concurrent constraint languages [86].

The main novelty of Fusion calculus is that the input primitive does not bind names, that is, there exists only one binding operator, written $(a)P$, meaning that name a is private to P. As a consequence, in Fusion calculus, unlike π-calculus, communication has a "global effect". To better understand what this means, let us compare the reduction rules in π and (monadic) Fusion, respectively[3]:

$$\pi: \qquad \overline{a}b.\, P \mid a(x).\, Q \mid R \longrightarrow P \mid Q\{b/x\} \mid R$$

$$\text{Fusion:} \quad (x)(\overline{a}b.\, P \mid ax.\, Q \mid R) \longrightarrow (P \mid Q \mid R)\{b/x\}$$

In π-calculus the effect of the communication (i.e. the substitution $\{b/x\}$) is confined to the receiver process Q; while in Fusion calculus all the processes in the scope of (x) are affected.

The encoding from π into Fusion is straightforward [28]. This encoding justifies the claim that the Fusion calculus inherits all π-calculus' expressive power. It remains

[3] For simplicity, we consider monadic Fusion calculus, the generalisation to the polyadic case is slightly more complex.

to be understood whether the Fusion calculus is more expressive than the π-calculus. In this thesis, we show that this is not the case for asynchronous calculi. Precisely, we exhibit an encoding of asynchronous Fusion calculus into asynchronous π-calculus, based on equators. We prove this encoding fully-abstract with respect to *closed barbed congruence*, a variant of barbed congruence, introduced in this thesis, where contexts bind all names of the tested processes. As in the definition of *testing equivalences* [21], these contexts signal success by emitting along names that do not appear in the tested processes. In Fusion calculus, the transmission of names between two processes may arise only when involving restricted names, that is, names with a delimited scope. For this reason we believe that closed barbed congruence represents a more satisfactory behavioural equivalence in Fusion calculus.

This part of the thesis contains revised material from [45, 46].

1.3 Mobile objects as mobile processes

Mobile object systems have been investigated for more than one decade, starting out with early representatives, like *Emerald* [41]. Recently, a whole series of scientific workshops has been dedicated to mobile object systems [88]. While mobility in object systems has well-known advantages, for example as support for load balancing or mobile agents, there is no common agreement on how mobile objects should actually be obtained.

The work presented in Chapter 7 is in line with the research activity to use the π-calculus as a toolbox for reasoning about object-oriented programming languages. Former works on the semantics of objects as processes have showed the value of this approach: while [90, 39, 75, 42] have focused on just providing formal semantics to object-oriented languages and language features, the work of others [66, 79] has been driven by a specific programming problem.

Migration = Cloning ; Aliasing Our work tackles a problem in Cardelli's *lexically-scoped distributed* programming language *Obliq* [18]. Cardelli proposed to derive object migration from two other primitives, *cloning* and *aliasing*, by performing one after the other.

Obliq is well-suited for the above style of migration. *Lexical scoping* in distributed settings makes program analysis easier since the binding of variables is completely determined by their location in the program text, and not by the execution site. In Obliq, immutable values can be freely copied from site to site, whereas mutable values are stationary. Only references to mutable values may be transmitted between different sites. Accordingly, since objects are mutable, the migration of an object does not physically move the object, but instead creates a *clone* of the object at the target site and then turns the original (local) object into an *alias* – sometimes called a *proxy* – for the new (remote) object.

In concurrent and distributed programs, it is important that certain state changes, in parts of the running system, may happen transparently from the point of view of the rest of the system. Ensuring that the implementation of such state changes is in fact transparent can be a difficult task since the programmer must in principle anticipate all possible execution scenarios. So the question arises naturally, whether object migration in Obliq is transparent to the object's clients, and how that can be stated formally.

When is object migration correct? Intuitively, migration of an object a to some other site works transparently, or safely, if (1) after the migration a client of a cannot tell that a is now an alias, and if (2) during migration it is not possible to interact with a in a way that prevents the migration operation from proper completion. In Obliq, mobile objects are therefore required to be *protected* and *serialised*: protection guarantees that aliases are persistent; serialisation guarantees atomicity of the two-phase migration operation. Unfortunately, Obliq is not equipped with a formal semantics, except for an unpublished note by Talcott [83], which provides a configuration-style semantics for a subset of Obliq excluding migration, so no formal treatment of the safety of migration has been possible up to now. The aim of our work is to remedy this lack of formality in order to reason about migration.

From Migration to Surrogation Since Obliq is *lexically scoped*, we ignore the aspects of distribution, at least when regarding the results of Obliq computations, unless site failures. Following this idea, we focus on Øjeblik [61], an object-based language that represents Obliq's concurrent core, but can also be seen as a concurrent extension of the Imperative Object Calculus [1]. Øjeblik supports a distribution-free abstraction of migration called *surrogation*. Similarly to migration, the *surrogation* of an object a is described as the creation of a clone b of a and then turning a itself into a proxy for b, which forwards future request for methods of a to b. The main difference with respect to migration is that neither a nor b are attached to any site.

Correctness as an equation We give a formal definition of *correctness* for object surrogation in Øjeblik which can be straightforwardly adapted to object migration in Obliq. The intuition is that, in order to be correct, the surrogation (resp. migration) of an object must be transparent to the clients of that object, i.e., the object must behave the same before and after surrogation (resp. migration). We formalise this concept by means of a simple equation:

$$a.\mathsf{ping} \doteq a.\mathsf{surrogate}$$

where the left side represents the object a before surrogation ($a.\mathsf{ping}$ returns the object resulting from the evaluation of a), the right side represents the object a after surrogation ($a.\mathsf{surrogate}$ returns the surrogated object), and $\doteq$ is an appropriate contextual equivalence, based on the possibility of convergence.

Contribution In this thesis, we present an Lπ-semantics for Øjeblik corresponding to Cardelli's Obliq implementation [17, 18]. We also give a notion of contextual equivalence, defined in terms of may-convergence on π-processes, corresponding to the equivalence $\doteq$.

We show that this semantics does not respect the equation above. This will be formally proved by exhibiting Øjeblik contexts that are able to distinguish the terms $a.\mathsf{ping}$ and $a.\mathsf{surrogate}$. Similar counterexamples apply to object migration in Obliq, as we have tested using the Obliq interpreter [17].

In order to achieve the correctness of surrogation, we propose an amended semantics for Øjeblik. We prove that this semantics supports surrogation correctly. The proof is divided into an algebraic part, where we can use powerful adaptations of proof techniques, from standard π-calculus and Lπ, and an iterative part, where we constructively simulate arbitrarily long finite sequences "up to" the algebraic equations. The proof is non-trivial, and we give (to our knowledge) the first formal proof that object migration can be correctly implemented in terms of cloning and aliasing.

We want to remark that most counterexamples concerning Cardelli's Obliq implementation have been discovered while using Lπ-semantics to prove the correctness of migration. Finally, the major lesson learned from this work, is that *concurrent objects need formal analysis* – not only because one should prove properties for their own sake, but because formal analysis is a good debugging tool.

This part of the thesis is mostly based on [48] (joint work with Josva Kleist and Uwe Nestmann). It also contains some revised material from [60, 61] (joint work with Hans Hüttel, Josva Kleist, and Uwe Nestmann).

1.4 Related work

Local π Calculi similar to Local π are discussed by Amadio, Boreale, Honda, Tokoro, and Yoshida in [35, 10, 3, 92]. Characterisations of barbed congruence on asynchronous calculi for mobile processes have been investigated by Amadio, Boreale, Castellani, Fournet, Laneve, and Sangiorgi in [4, 11, 3]. However, in these bisimilarities, matching transitions of processes have the same labels, therefore laws like 1.1$-$1.4 cannot be proved. A major challenge in Lπ is that, unlike the calculi in [4, 11, 3], no construct for testing equality between names is provided. This because name-testing, like testing equality between pointers in imperative languages, prevents many useful program optimisations and transformations.

Other studies on barbed congruence, or similar context-based bisimulations, for mobile processes have been conducted by Hennessy, Honda, Kobayashi, Pierce, Riely, Turner, and Yoshida in [37, 43, 91, 30] (and, for a coordination language, by Busi, Gorrieri, and Zavattaro [16]). Boreale and Sangiorgi [12] have studied barbed congruence in synchronous π-calculus with capability types and no matching, where Lπ can be treated as a special case. Our characterisations are simpler than those in [12], but the latter are more general, in that they can be applied to several π-calculus languages (although the extension to asynchronous languages is not straightforward). The technical approaches are different: in [12] bisimilarities have a type environment (in fact, closures) whereas our bisimilarities are directly defined on processes.

Some of the techniques we use in Section 3.3 are inspired by those in [78] introduced by Sangiorgi.

The encoding of polyadic Lπ into monadic Lπ is inspired by Milner's encoding of polyadic π into monadic π [51]. We must also mention Yoshida [91], and Quaglia and Walker [70], which have introduced two different *type systems* for monadic processes to capture the communication protocol underlying Milner's encoding. More precisely, in [91] a notion of graph type is introduced and studied. Nodes of a graph type represent atomic actions, and edges represent an activation ordering between them. The approach in [70] is similar but the type system is simpler. Both papers show a full abstraction result with respect to typed contextual equivalences that reject all contexts which do not respect the protocol imposed by the encoding. While [91, 70] work on the full π-calculus our result only applies in Lπ. On the other hand, we prove a sharper result because we get the completeness of the encoding with respect to all monadic contexts without rejecting "hostile" contexts. A different approach has been followed by Gonthier and Fournet in the *Join-calculus* [24]. In [24], among other results, a direct, although complex, fully-abstract encoding of polyadic processes into monadic ones is proposed. Apart from the differences among the two process calculi, the encoding in [24] is technically quite different from ours; for instance, as the authors themselves say, their translation encodes and then subsequently decodes tuples twice.

The theory of Lπ developed in this thesis, has already been used by Sangiorgi to give a fully-abstract encoding of *higher-order* Local π (where processes can be transmitted) into Local π [80]. In that paper, Sangiorgi shows that Local π allows simpler proofs than those for other π-calculi [72].

Finally, in Join and Blue calculus, polymorphic type systems à la ML have been introduced [27, 20]. The constraint on the output capability of names is crucial in both cases. This encourages us to believe that it is possible to define a polymorphic type system in Local π as well.

Equating/fusing names in name-passing calculi An (implicit) way to model equivalence relations over names in name-passing calculi has been proposed by Honda and Yoshida in [37] by means of their equator processes. A more direct way has been proposed in [38], where the composition results in what Honda calls an "open substitution". Independently, Parrow and Victor have developed the Fusion calculus [65] as a polyadic evolution of their Update calculus [64]. The theory of Fusion calculus has been studied in detail in [87]. The Chi calculus, independently introduced by Fu [28], can be seen as a monadic variant of the Fusion calculus.

Mobile objects as mobile processes The work closest to our is [42] where an interpretation of Abadi and Cardelli's Imperative Object Calculus into typed π-calculus is presented. Unlike [42], in this thesis we focus on a concurrent object calculus. Gordon and Hankin's [29], and Di Blasio and Fisher's [22] describe two different *concurrent object calculi*, but no account of object migration is given for them.

1.5 Outline of the thesis

In Chapter 2 we describe Milner, Parrow, and Walker's π-calculus with both reduction and labelled operational semantics. We recall the main bisimulation-based behavioural equalities and some up-to proof techniques. Then, we briefly describe a few variants of the π-calculus: the asynchronous π-calculus, the πI-calculus, the Join calculus, and the Fusion calculus. Finally, given an encoding between two different process calculi, we briefly recall the notions of operational correspondence and full abstraction.

In Chapter 3 we introduce Local π. We give two proof techniques for proving barbed congruence in Lπ. The first is based on an encoding which, very roughly, replaces outputs of free names with outputs of private names. This encoding uses link processes. The second proof technique is based on a new *labelled transition system* (LTS) which reveals what is observable in Local π, that is, what an external observer that behaves like a Local π process can see by interacting with a Local π process. Also this proof technique relies on link processes. Both proof techniques are proved sound and complete, that is, they completely characterise barbed congruence in Lπ. Finally, we show how to strengthen the LTS-based proof technique by means of some non-standard up-to proof techniques.

In Chapters 4 we show some applications of the theory of Local π: (i) we give an alternative way to express name substitution in terms of the other operators of the calculus; (ii) we prove that in Local π the *delayed input* (a form of non-blocking input prefixing) is derivable, and present some of its algebraic properties; (iii) we prove a stronger form of Milner's *replication theorems* [51]; (iv) we give an optimisation of the encoding of call-by-name λ-calculus and, exploiting delayed input, we derive an encoding of *strong call-by-name*; (v) we prove some laws for the Join-calculus; (vi) we prove some non-full abstraction and full abstraction results for Boreale's encoding [10] of *external mobility* (communication of free names) in terms of *internal mobility* (communication of private names); (vii) we prove that Thielecke's axiomatic semantics of the *Continuation Passing Style calculus* [84] is operationally sound.

In Chapter 5, we use the encoding of name substitution presented in Chapter 4 to prove the full abstraction of an encoding of polyadic Local π into monadic Local π. We also propose an encoding from polyadic Fusion calculus into its monadic variant, based on the same idea. We conjecture that the encoding for Fusion calculus is fully-abstract with respect to closed barbed congruence.

In Chapter 6 we extend the theory of Local π to asynchronous π. We give a labelled characterisation of barbed congruence in asynchronous π in terms of equators. As an application of the theory developed for asynchronous π, we prove the full abstraction of an encoding of asynchronous Fusion calculus into asynchronous π. The encoding is proved fully-abstract with respect to asynchronous closed barbed congruence. Since an encoding of π-calculus into Fusion is already known from [28], our result shows that asynchronous Fusion calculus and asynchronous π-calculus have the same expressive power.

In Chapter 7 we give a semantics for Øjeblik in terms of (an extended variant of) $\mathrm{L}\pi$ according to Obliq informal semantics. We give a notion of contextual equivalence, defined in terms of may convergence on π-processes. We give a formal correctness criterion for object surrogation in Øjeblik. We show that the above semantics does not respect the correctness criterion for object surrogation. Then, we propose a variant of the semantics and we prove that it respects the correctness criterion for object surrogation.

The thesis concludes in Chapter 8 with a summary of the contributions and a discussion of open ends and possible future work.

Chapter 2

The π-calculus

In this chapter we recall *syntax*, *operational semantics*, the main *bisimulation-based equivalences*, and some *up-to proof techniques* of the π-calculus.

The π-calculus presented here is essentially that of [57], to which the reader is referred to for additional details. The only differences with respect to [57] is that, as in [51], we consider guarded choice instead of full choice, and replication instead of recursion.

In this chapter, we also present a few variants of the π-calculus: the *asynchronous π-calculus*, the *πI-calculus*, the *Join calculus*, and the *Fusion calculus*. Such variants will be discussed and/or studied later on.

Finally, we recall the notions of *operational correspondence* and *full abstraction* for encodings between process calculi.

2.1 Syntax

Let $\mathcal{N}$ be a countable infinite set of names ranged over by the metavariables $a, b, c, \ldots, x$, and y. Let $\mathcal{P}$ be the set of processes, ranged over by $P, Q, \ldots$, built from the operators of *inactions*, *non-deterministic guarded choice*, *parallel composition*, *restriction*, *match*, and *replication* (or bang) according to the following grammar:

$$P ::= \Sigma_{i \in I} \alpha_i.\, P_i \;\Big|\; P \mid P \;\Big|\; (\boldsymbol{\nu} a)P \;\Big|\; [a = b]P \;\Big|\; !P$$

Here I is a finite indexing set; in the case $I = \emptyset$ we write the choice as $\mathbf{0}$. In a term $\alpha.\, P$ the prefix α represents either input or output:

$$\alpha ::= a(b) \mid \overline{a}b.$$

Input prefix $a(b).\, P$ and restriction $(\boldsymbol{\nu} b)P$ acts as *binders* for name b. *Free names* and *bound names* of a process P, written $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$ respectively, arise as expected (in analogy with the λ-calculus); the *names* of P, written $\mathrm{n}(P)$, are given by $\mathrm{fn}(P) \cup \mathrm{bn}(P)$. Sometimes, $\mathrm{fn}(P, Q)$ is used as a shorthand for $\mathrm{fn}(P) \cup \mathrm{fn}(Q)$. Similar notations hold for the functions $\mathrm{n}(\cdot)$ and $\mathrm{bn}(\cdot)$. A name is *fresh* if it is different from any other free name.

Substitutions, ranged over by $\sigma, \sigma', \ldots$ are functions from $\mathcal{N}$ to $\mathcal{N}$; for any process P, we write $P\sigma$ for the process obtained by applying σ to P with renaming possibly involved to avoid capture of free names.

The following order precedence when writing processes is assumed: substitution $>$ { restriction, prefix, match, replication } $>$ parallel composition $>$ choice. We write $\overline{a}.\, P$ and $a.\, P$ when the name transmitted at a is not important. We write $\tau.\, P$ as an abbreviation for $(\boldsymbol{\nu} a)(\overline{a}.\, \mathbf{0} \mid a.\, P)$ where $a \notin \mathrm{fn}(P)$. We write $\widetilde{a}$ to denote a tuple of names $a_1, \ldots, a_n$. We write $(\boldsymbol{\nu}\widetilde{a})P$ for $(\boldsymbol{\nu} a_1) \ldots (\boldsymbol{\nu} a_n)P$.

2.2 Operational semantics

The original semantics of the π-calculus [57] has been given in terms of a *labelled transition system* (LTS) in the SOS style of [69], where transitions are of kind $P \xrightarrow{\mu} Q$ for some set of actions ranged over by μ. A special action, denoted by τ, model the communication between two processes.

Alternatively, the behaviour of π-processes can be described by means of a *reduction semantics* [51]. The idea is that that $P \longrightarrow Q$, pronounced "P reduces to Q", models the internal evolution of the process P. The reduction semantics is simpler than the labelled one but records less information. This is because reduction semantics ignores the capability of a process to interact with the environment. In the π-calculus, labelled and reduction semantics coincide, that is, the relations $\longrightarrow$ and $\xrightarrow{\tau}$ coincide.

2.2.1 Reduction semantics

The reduction semantics of the π-calculus is based on an auxiliary relation called *structural congruence* which brings the participants of a potential communication into contiguous positions.

Definition 2.2.1 (Structural congruence) Structural congruence *is the smallest congruence relation satisfying the axioms below:*

- $P \;\; \alpha\mathrm{-convertible} \;\; Q \;\; \mathrm{implies} \;\; P \equiv Q$;
- $\Sigma_{i \in I}\alpha_i.\, P_i \equiv \Sigma_{i \in J}\alpha_i.\, P_i$ *if* J *is a permutation of* I;
- $P \mid \mathbf{0} \equiv P,\;\; P \mid Q \equiv Q \mid P,\;\; P \mid (Q \mid R) \equiv (P \mid Q) \mid R$;
- $(\boldsymbol{\nu} a)\mathbf{0} \equiv \mathbf{0},\;\; (\boldsymbol{\nu} a)(\boldsymbol{\nu} b)P \equiv (\boldsymbol{\nu} b)(\boldsymbol{\nu} a)P,\;\; (\boldsymbol{\nu} a)(P \mid Q) \equiv P \mid (\boldsymbol{\nu} a)Q$ *if* $a \notin \mathrm{fn}(P)$;
- $!P \equiv P \mid !P$;
- $[a = a]P \equiv P$.

The reduction rules in Table 2.1 model the evolution of a process. Note that reduction is not allowed underneath a prefix. Prefixing represents sequentialization. Hence, nothing underneath a prefix should occur before this has fired.

$$\texttt{com:}\frac{}{(\ldots + a(x).\,P) \mid (\ldots + \overline{a}b.\,Q) \longrightarrow P\{^b\!/\!_x\} \mid Q} \qquad \texttt{par:}\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

$$\texttt{str:}\frac{Q \equiv P \qquad P \longrightarrow P' \qquad P' \equiv Q'}{Q \longrightarrow Q'} \qquad \texttt{res:}\frac{P \longrightarrow P'}{(\boldsymbol{\nu} a)P \longrightarrow (\boldsymbol{\nu} a)P'}$$

Table 2.1: Reduction rules for the π-calculus

2.2.2 Labelled transition semantics

In Table 2.2, we give a *labelled transition system* for the π-calculus, in the *late* style [57]. The rules `com`, `close`, `par`, and `sum` have a symmetric version which is omitted. *Transitions* are of the form $P \xrightarrow{\mu} P'$, where *action* μ can be: τ (interaction); $a(b)$ (input); $\overline{a}b$ (free output) and $\overline{a}(b)$ (bound output, that is the transmission of a private name b at a). In these actions, a is the *subject* and b the *object*. Functions $\mathrm{bn}(\cdot)$, $\mathrm{fn}(\cdot)$ and $\mathrm{n}(\cdot)$ are extended to actions in the following way: $\mathrm{fn}(\overline{a}(b)) = \mathrm{fn}(a(b)) = \{a\}$, $\mathrm{fn}(\overline{a}b) = \{a, b\}$, $\mathrm{fn}(\tau) = \mathrm{bn}(\tau) = \mathrm{bn}(\overline{a}b) = \emptyset$, $\mathrm{bn}(\overline{a}(b)) = \mathrm{bn}(a(b)) = \{b\}$. We write $\xrightarrow{\hat{\mu}}$ to mean $P \xrightarrow{\mu} Q$, if $\mu \neq \tau$, and either $P = Q$ or $P \xrightarrow{\tau} Q$, if $\mu = \tau$. Relation $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$; moreover, $\stackrel{\mu}{\Longrightarrow}$ stands for $\Longrightarrow \xrightarrow{\mu} \Longrightarrow$, and $\stackrel{\hat{\mu}}{\Longrightarrow}$ for $\stackrel{\mu}{\Longrightarrow}$ if $\mu \neq \tau$, and for $\Longrightarrow$ if $\mu = \tau$.

The labelled transition system of Table 2.2 is called late because, in the input rule, the instantiation is performed only when a communication occurs. An input transition $P \xrightarrow{a(x)} Q$ means that P may receive some name b via a, and then evolve to $Q\{^b\!/\!_x\}$. In that action, x does not represent the value received, rather it is a reference to the places in Q where the received name will appear.

Another possibility is to have an *early semantics* where the received name is instantiated as soon as the rule for input is used. The rule for early inputs is the following:

$$\texttt{e-inp:}\frac{}{a(x).\,P \xrightarrow{ab} P\{^b\!/\!_x\}}.$$

In $P \xrightarrow{ab} Q$, the *free input action* ab means that P has received the name b along a and then continues as Q. Free and bound names of the action ab are as expected: $\mathrm{fn}(ab) = \{a, b\}$ and $\mathrm{bn}(ab) = \emptyset$. In the early semantics the rules `com` and `close` are slightly different: late inputs are replaced by early inputs, and substitutions on received names are omitted.

The τ-actions that can be inferred with the early semantics are exactly those that can be inferred with the late semantics. Also input and output actions coincide. In the case

$$\texttt{alpha:}\frac{P \xrightarrow{\mu} P' \qquad P \text{ and } Q \text{ are } \alpha\text{−convertible}}{Q \xrightarrow{\mu} P'}$$

$$\texttt{inp:}\frac{}{a(x).\,P \xrightarrow{a(x)} P} \qquad \texttt{out:}\frac{}{\overline{a}b.\,P \xrightarrow{\overline{a}b} P}$$

$$\texttt{open:}\frac{P \xrightarrow{\overline{a}b} P' \qquad a \neq b}{(\boldsymbol{\nu} b)P \xrightarrow{\overline{a}(b)} P'} \qquad \texttt{close:}\frac{P \xrightarrow{\overline{a}(c)} P' \quad Q \xrightarrow{a(c)} Q'}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} c)(P' \mid Q')}$$

$$\texttt{com:}\frac{P \xrightarrow{\overline{a}b} P' \qquad Q \xrightarrow{a(x)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{b/x\}} \qquad \texttt{par:}\frac{P \xrightarrow{\mu} P' \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\mu} P' \mid Q}$$

$$\texttt{new:}\frac{P \xrightarrow{\mu} P' \qquad a \notin \mathrm{n}(\mu)}{(\boldsymbol{\nu} a)P \xrightarrow{\mu} (\boldsymbol{\nu} a)P'} \qquad \texttt{match:}\frac{P \xrightarrow{\mu} P'}{[a = a]P \xrightarrow{\mu} P'}$$

$$\texttt{sum:}\frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'} \qquad \texttt{rep-a:}\frac{P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P' \mid !P}$$

$$\texttt{rep-b:}\frac{P \xrightarrow{\overline{a}b} P' \quad P \xrightarrow{a(x)} P''}{!P \xrightarrow{\tau} (P' \mid P''\{b/x\}) \mid !P} \qquad \texttt{rep-c:}\frac{P \xrightarrow{\overline{a}(c)} P' \quad P \xrightarrow{a(c)} P''}{(\boldsymbol{\nu} c)(P' \mid P'') \mid !P}$$

Table 2.2: Late labelled transition system for the π-calculus

of input, it holds that:

$$P \xrightarrow{ab} P' \;\text{ iff }\; \exists P'', x : P \xrightarrow{a(x)} P'' \;\text{ and }\; P' = P''\{b/x\}.$$

So, the choice between early and late semantics is a matter of taste. In this thesis we assume a late semantics but we sometimes use early inputs as an abbreviation for the definition above.

2.3 Behavioural equivalences

A crucial notion in a process calculus is that of *behavioural equality* between processes. We focus on bisimulation-based behavioural equivalences, precisely on (weak) *barbed congruence* [58]. Barbed congruence can be defined in any calculus possessing: (i) an *interaction relation* (the τ-steps in the π-calculus), modelling the evolution of the system; and (ii) an *observability predicate* $\downarrow_a$ for each name a, which detects the possibility of a process of accepting a communication with the environment at a.

Definition 2.3.1 (Synchronous observability) *We write* $P \downarrow_a$ *if there is a derivative* P'*, and an action* μ *with subject* a*, such that* $P \xrightarrow{\mu} P'$*. We write* $P \Downarrow_a$ *if there is* P' *such that* $P \Longrightarrow P'$ *and* $P' \downarrow_a$*.*

Like other contextually-defined behavioural equivalences, barbed congruence is sensitive to the set of operators of the calculus. We recall the notion of context.

Definition 2.3.2 (Context) *A (monadic)* context $C[\cdot]$ *is a process expression with a single hole in it, written* $[\cdot]$*. Given a process* P*,* $C[P]$ *is the process obtained by plugging the process* P *in the hole. A context* $C[\cdot]$ *is* static *if it is pointwise structural congruent to* $(\boldsymbol{\nu}\widetilde{a})(P \mid [\cdot])$*, for some* P *and* $\widetilde{a}$*.*

Definition 2.3.3 (Synchronous barbed bisimilarity) *A symmetric relation* $\mathcal{S}$ *on processes is a* synchronous barbed bisimulation *if* $P \;\mathcal{S}\; Q$ *implies:*

1. *If* $P \xrightarrow{\tau} P'$ *then there exists* Q' *such that* $Q \Longrightarrow Q'$ *and* $P' \;\mathcal{S}\; Q'$*.*
2. *If* $P \downarrow_a$ *then* $Q \Downarrow_a$*.*

Two processes P *and* Q *in a set of processes* $\mathcal{L}$ *are:*

- synchronous barbed bisimilar*, written* $P \stackrel{\bullet}{\approx}^{\mathrm{s}} Q$*, if* $P \;\mathcal{S}\; Q$ *for some synchronous barbed bisimulation* $\mathcal{S}$*;*
- synchronous barbed equivalent in $\mathcal{L}$*, written* $P \simeq^{\mathrm{s}}_{\mathcal{L}} Q$*, if for all static contexts* $C[\cdot]$ *in* $\mathcal{L}$ *it holds that* $C[P] \stackrel{\bullet}{\approx}^{\mathrm{s}} C[Q]$*;*
- synchronous barbed congruent in $\mathcal{L}$*, written* $P \cong^{\mathrm{s}}_{\mathcal{L}} Q$*, if for all contexts* $C[\cdot]$ *in* $\mathcal{L}$ *it holds that* $C[P] \stackrel{\bullet}{\approx}^{\mathrm{s}} C[Q]$*.*

Context-based behavioural equalities like barbed congruence suffer from the universal quantification on contexts. This quantification makes very hard to prove process equalities, and makes mechanical checking impossible. Simpler proof techniques are based on *labelled bisimulations* whose definitions do not use context quantification. These bisimulations should imply, or (better) coincide with, barbed congruence. In CCS barbed congruence coincides with observation congruence [58]. In the π-calculus equipped with *name testing* (also called *matching*), barbed congruence coincides with the closure under substitutions of synchronous early bisimilarity [72].

Definition 2.3.4 (Synchronous early bisimilarity) *A symmetric relation* $\mathcal{S}$ *on processes is a* synchronous early bisimulation *if whenever* $P \;\mathcal{S}\; Q$*:*

1. *If* $P \xrightarrow{\mu} P'$*,* μ *is not an input and* $\mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$*, then there exists* Q' *such that* $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ *and* $P' \;\mathcal{S}\; Q'$*.*
2. *If* $P \xrightarrow{a(x)} P'$*, then for all* b *there exists* Q' *such that* $Q \stackrel{a(x)}{\Longrightarrow} Q'$ *and* $P'\{b/x\} \;\mathcal{S}\; Q'\{b/x\}$*.*

Two processes P *and* Q *are* synchronous early bisimilar*, written* $P \approx^{\mathrm{e}} Q$*, if* $P \;\mathcal{S}\; Q$ *for some synchronous early bisimulation* $\mathcal{S}$*.*

Remark 2.3.5 *The input clause in the above definition could be rewritten in terms of early input as follows:*

$$\text{If } P \xrightarrow{ab} P' \text{ then there exists } Q' \text{ such that } Q \stackrel{ab}{\Longrightarrow} Q' \text{ and } P' \ \mathcal{S}\ Q'.$$

The choice between the two formulations is only a matter of taste.

The proof that the closure under substitutions of early bisimilarity coincides with barbed congruence is usually given on the class of the *image-finite* processes (to which most of the processes one would like to write belongs) by exploiting the n-approximants of the labelled equivalences.

Definition 2.3.6 *The class of* image-finite processes *is the largest subset $\mathcal{I}$ of π-processes which is derivation closed and such that $P \in \mathcal{I}$ implies that, for all μ, the set $\{P' : P \stackrel{\mu}{\Longrightarrow} P'\}$, quotiented by alpha conversion, is finite.*

In the proof that the closure under substitutions of early bisimilarity coincides with barbed congruence, a central role is played by the *matching* construct for testing equalities between names. If matching is removed from the language, then the closure under substitutions of early bisimilarity still implies barbed congruence, but the vice versa does not hold.

In π-calculus, early bisimilarity is strictly contained in a simpler form of bisimilarity, called *ground bisimilarity* [74], which differs from the early one in that it has no universal quantification on the received names in the input clause.

Definition 2.3.7 (Synchronous ground bisimilarity) *A symmetric relation $\mathcal{S}$ on processes is a* synchronous ground bisimulation *if whenever $P\ \mathcal{S}\ Q$:*

1. *If $P \xrightarrow{\mu} P'$, μ is not an input and $\mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$, then there is Q' such that $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P'\ \mathcal{S}\ Q'$.*
2. *If $P \xrightarrow{a(b)} P'$, and $b \notin \mathrm{fn}(Q)$, then there is Q' such that $Q \stackrel{a(b)}{\Longrightarrow} Q'$ and $P'\ \mathcal{S}\ Q'$.*

Two processes P and Q are synchronous ground bisimilar, *written $P \approx Q$, if $P\ \mathcal{S}\ Q$ for some synchronous ground bisimulation $\mathcal{S}$.*

Remark 2.3.8 *As for early bisimilarity, the input clause in the above definition could be rewritten in terms of early input as follows:*

$$\text{If } P \xrightarrow{ab} P', \text{ and } b \notin \mathrm{fn}(P, Q), \text{ then } \exists\, Q' \text{ s.t. } Q \stackrel{ab}{\Longrightarrow} Q' \text{ and } P'\ \mathcal{S}\ Q'.$$

Since $\approx^{\mathrm{e}} \subset \approx \subset \dot{\approx}^{\mathrm{s}}$, in presence of matching, the closures under substitutions of early bisimilarity and the closure under contexts of ground bisimilarity coincide with barbed congruence (on image-finite processes).

Sometimes, it may be useful to count the number of silent moves performed by a process. The *synchronous early expansion* [6], written $\lesssim^{\mathrm{e}}$, is an asymmetric variant of $\approx^{\mathrm{e}}$ such that $P \lesssim^{\mathrm{e}} Q$ holds if $P \approx^{\mathrm{e}} Q$, and Q has at least as many τ-moves as P.

Definition 2.3.9 (Synchronous early expansion) *A relation $\mathcal{S}$ on processes is a* synchronous early expansion *if $P \ \mathcal{S} \ Q$ implies:*

1. *If $P \xrightarrow{\mu} P'$, $\mu \in \{\tau, \overline{a}b, \overline{a}(c), ab\}$ and $\mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$, then there exists Q' such that $Q \overset{\mu}{\Longrightarrow} Q'$ and $P' \ \mathcal{S} \ Q'$.*
2. *If $Q \xrightarrow{\mu} Q'$, $\mu \in \{\tau, \overline{a}b, \overline{a}(c), ab\}$ and $\mathrm{bn}(\mu) \cap \mathrm{fn}(P) = \emptyset$, then there exists P' such that $P \xrightarrow{\hat{\mu}} P'$ and $P' \ \mathcal{S} \ Q'$.*

We write $P \lesssim^{\mathrm{e}} Q$ if $P \ \mathcal{S} \ Q$ for some synchronous early expansion $\mathcal{S}$.

In a similar manner we can define: (i) *synchronous ground expansion*, denoted by $\lesssim$, and (ii) *barbed expansion* as the expansion variant of barbed bisimulation. From barbed expansion we can derive an expansion variant of barbed congruence obtained by replacing in Definition 2.3.3 the relation $\dot{\approx}$ with its expansion variant. Note that all expansion variants imply their corresponding bisimilarities.

2.4 Up-to proof techniques

The *up-to* proof techniques allow us to reduce the size of the relation $\mathcal{S}$ to exhibit for proving that two processes are bisimilar. We consider two up-to proof techniques. The first is the the *up-to expansion* technique [81], where the expansion relation (Definition 2.3.9) is used to achieve the closure of $\mathcal{S}$. The other is the *up-to context* technique [76], which admits the use of contexts to achieve the closure of $\mathcal{S}$. These two techniques can also be merged.

Definition 2.4.1 ($\approx^{\mathrm{e}}$-bisimulation up-to $\gtrsim^{\mathrm{e}}$ and $\approx^{\mathrm{e}}$) *A relation $\mathcal{S}$ is a* $\approx^{\mathrm{e}}$-bisimulation up to $\gtrsim^{\mathrm{e}}$ and $\approx^{\mathrm{e}}$ *if $P \ \mathcal{S} \ Q$ implies, for all μ:*

1. *whenever $P \xrightarrow{\mu} P'$ and $\mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$, then Q' exists s.t. $Q \overset{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \gtrsim^{\mathrm{e}} \mathcal{S} \approx^{\mathrm{e}} Q'$;*
2. *whenever $Q \xrightarrow{\mu} Q'$ and $\mathrm{bn}(\mu) \cap \mathrm{fn}(P) = \emptyset$, then P' exists s.t. $P \overset{\widehat{\mu}}{\Longrightarrow} P'$ and $P' \approx^{\mathrm{e}} \mathcal{S} \lesssim^{\mathrm{e}} Q'$.*

Lemma 2.4.2 ([81]) *If $\mathcal{S}$ is a $\approx^{\mathrm{e}}$-bisimulation up to $\gtrsim^{\mathrm{e}}$ and $\approx^{\mathrm{e}}$, then it holds that $\mathcal{S} \subseteq \approx^{\mathrm{e}}$.*

Definition 2.4.3 ($\approx^{\mathrm{e}}$-bisimulation up-to context) *A symmetric relation $\mathcal{S}$ is an* $\approx^{\mathrm{e}}$-bisimulation up to context *if $P \ \mathcal{S} \ Q$ and $P \xrightarrow{\mu} P''$ imply that there are a static context $C[\cdot]$ and processes P' and Q' such that $P'' = C[P']$, $Q \overset{\widehat{\mu}}{\Longrightarrow} C[Q']$ and $P' \ \mathcal{S} \ Q'$.*

Lemma 2.4.4 ([76]) *If $\mathcal{S}$ is a $\approx^{\mathrm{e}}$-bisimulation up to context, then $\mathcal{S} \subseteq \approx^{\mathrm{e}}$.*

Definition 2.4.5 ($\approx^{\mathrm{e}}$-bisimulation up-to context and up to $\gtrsim^{\mathrm{e}} \approx^{\mathrm{e}}$) *A relation $\mathcal{S}$ is an* $\approx^{\mathrm{e}}$-bisimulation up to context and up to $\gtrsim^{\mathrm{e}} \approx^{\mathrm{e}}$ *if $P \ \mathcal{S} \ Q$ implies, for all μ:*

1. *whenever* $P \xrightarrow{\mu} P''$, *with* $\mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$, *there are a static context* $C[\cdot]$ *and processes* P' *and* Q' *such that* $P'' \gtrsim^{\mathrm{e}} C[P']$, $Q \stackrel{\widehat{\mu}}{\Longrightarrow} \approx^{\mathrm{e}} C[Q']$ *and* $P' \,\mathcal{S}\, Q'$;
2. *whenever* $Q \xrightarrow{\mu} Q''$, *with* $\mathrm{bn}(\mu) \cap \mathrm{fn}(P) = \emptyset$, *there are a static context* $C[\cdot]$ *and processes* Q' *and* P' *such that* $Q'' \gtrsim^{\mathrm{e}} C[Q']$, $P \stackrel{\widehat{\mu}}{\Longrightarrow} \approx^{\mathrm{e}} C[P']$ *and* $P' \,\mathcal{S}\, Q'$.

Lemma 2.4.6 ([76]) *If* $\mathcal{S}$ *is a* $\approx^{\mathrm{e}}$*-bisimulation up to context and up to* $\gtrsim^{\mathrm{e}} \approx^{\mathrm{e}}$, *then* $\mathcal{S} \subseteq \approx^{\mathrm{e}}$.

2.5 Variants of the π-calculus

In this section, we present a few variants of the π-calculus.

2.5.1 The asynchronous π-calculus

Asynchronous π, written π_{a}, is a subcalculus of synchronous π without choice and match, where message emission is non-blocking [34, 13]. Formally, the output prefix $\overline{a}b.\,P$ is replaced by the simpler output particle $\overline{a}b$, which has no continuation. In this manner, consuming a message has no effect on the process that created that message. The asynchronous π-calculus is defined by the following grammar:

$$P ::= \mathbf{0} \;\Big|\; a(b).\,P \;\Big|\; \overline{a}b \;\Big|\; P \mid P \;\Big|\; (\boldsymbol{\nu}a)P \;\Big|\; !P$$

The reduction semantics and the labelled semantics are similar to those seen for the synchronous π-calculus in Sections 2.2.1 and 2.2.2.

In asynchronous π, a message is sent simply when it is unguarded, i.e., when it is not underneath an input prefix. Note that the coincidence between "message sent" and "message unguarded" would break if the calculus has the choice operator. In a process $\overline{a}b + c(x).\,P$, message $\overline{a}b$, although unguarded, is not really sent because its offer can vanish at any moment. The availability of $\overline{a}b$ only depends on whether the other summand $c(x).\,P$ can performs its input at c. An asynchronous process that non-deterministically can chose to send $\overline{a}b$ or to receive at $c(x).\,P$ should be written $\tau.\,\overline{a}b + c(x).\,P$ (see also Castellani and Hennessy's asynchronous choice [19]). Nestmann and Pierce have showed that this form of choice, in which each summand is guarded by a τ or by an input prefix, can be encoded in the asynchronous π-calculus [62].

In the next section we show how the constraint on output actions in π_{a} reflects on the behavioural equivalences seen in Section 2.3

Asynchronous behavioral equivalences

In this paragraph we report the asynchronous variant of the behavioural equivalences seen in Section 2.3.

Unlike synchronous π-calculus, in asynchronous π-calculus it is more natural to restrict the observation to output actions. The reason is that in an asynchronous π-calculus the observer has no direct way of knowing if the message it has sent has been received.

Definition 2.5.1 (Asynchronous observability) *We write $P \downarrow_a$ if there is a derivative P', and an output action μ with subject a, such that $P \xrightarrow{\mu} P'$. We write $P \Downarrow_a$ if there is P' such that $P \Longrightarrow P'$ and $P' \downarrow_a$.*

Definition 2.5.2 (Asynchronous barbed bisimilarity) *A symmetric relation $\mathcal{S}$ on processes is an* asynchronous barbed bisimulation *if $P \;\mathcal{S}\; Q$ implies:*

1. *If $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \Longrightarrow Q'$ and $P' \;\mathcal{S}\; Q'$.*
2. *If $P \downarrow_a$ then $Q \Downarrow_a$.*

Two processes P and Q in a set of processes $\mathcal{L}$ are:

- asynchronous barbed bisimilar, *written $P \stackrel{\bullet}{\approx} Q$, if $P \;\mathcal{S}\; Q$ for some asynchronous barbed bisimulation $\mathcal{S}$;*
- asynchronous barbed equivalent in $\mathcal{L}$, *written $P \simeq_{\mathcal{L}} Q$, if for all static contexts $C[\cdot]$ in $\mathcal{L}$ it holds that $C[P] \stackrel{\bullet}{\approx} C[Q]$;*
- asynchronous barbed congruent in $\mathcal{L}$, *written $P \cong_{\mathcal{L}} Q$, if for all contexts $C[\cdot]$ in $\mathcal{L}$ it holds that $C[P] \stackrel{\bullet}{\approx} C[Q]$.*

In synchronous π-calculus, synchronous and asynchronous barbed equivalence coincide. A similar result holds for barbed congruence. This is not the case in π_{a}. As a counterexample, take $P = \mathbf{0}$ and $Q = a(x).\,\overline{a}x$, then $P \simeq_{\pi_{\mathrm{a}}} Q$ and $P \cong_{\pi_{\mathrm{a}}} Q$, while $P \not\simeq^{\mathrm{s}}_{\pi_{\mathrm{a}}} Q$ and $P \not\cong^{\mathrm{s}}_{\pi_{\mathrm{a}}} Q$.

Definition 2.5.3 (Asynchronous early bisimilarity) *A symmetric relation $\mathcal{S}$ on processes is an* asynchronous early bisimulation *if whenever $P \;\mathcal{S}\; Q$:*

1. *If $P \xrightarrow{\mu} P'$, μ is not an input and $\mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$, then there exists Q' such that $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \;\mathcal{S}\; Q'$.*
2. *If $P \xrightarrow{a(x)} P'$, then for all names b there exists Q' such that:*
   (a) *either $Q \stackrel{a(x)}{\Longrightarrow} Q'$ and $P'\{b/x\} \;\mathcal{S}\; Q'\{b/x\}$;*
   (b) *or $Q \Longrightarrow Q'$ and $P'\{b/x\} \;\mathcal{S}\; Q' \mid \overline{a}b$.*

Two processes P and Q are asynchronous early bisimilar, *written $P \approx^{\mathrm{e}}_{\mathrm{a}} Q$, if $P \;\mathcal{S}\; Q$ for some asynchronous early bisimulation $\mathcal{S}$.*

In asynchronous π with matching, the closure under substitutions of asynchronous early bisimilarity coincides with asynchronous barbed congruence and with the closure under contexts of asynchronous ground bisimilarity (on image-finite processes) [4].

Definition 2.5.4 (Asynchronous ground bisimilarity) *A symmetric relation $\mathcal{S}$ on processes is an* asynchronous ground bisimulation *if whenever $P \ \mathcal{S} \ Q$:*

1. *If $P \xrightarrow{\mu} P'$, μ is not an input and $\mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$, then there exists Q' such that $Q \overset{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \ \mathcal{S} \ Q'$.*
2. *If $P \xrightarrow{a(b)} P'$, and $b \notin \mathrm{fn}(Q)$, then there is Q' such that:*
   (a) either $Q \overset{a(b)}{\Longrightarrow} Q'$ and $P' \ \mathcal{S} \ Q'$;
   (b) or $Q \Longrightarrow Q'$ and $P' \ \mathcal{S} \ (Q' \mid \overline{a}b)$.

Two processes P and Q are asynchronous ground bisimilar, *written $P \approx_{\mathrm{a}} Q$, if $P \ \mathcal{S} \ Q$ for some asynchronous ground bisimulation $\mathcal{S}$.*

In asynchronous π-calculus without matching, both synchronous and asynchronous early bisimilarities are congruences and they coincide with their ground variants, respectively [33, 74] [1]. As a consequence, all the up-to proof techniques seen in Section 2.4 are valid in asynchronous π-calculus when considering ground relations instead of early ones. Since in the thesis we work on asynchronous calculi without matching, we focus on the simpler ground bisimilarities distinguishing only between synchronous and asynchronous bisimilarity. Of course, synchronous bisimilarity implies asynchronous bisimilarity.

Finally, we give the the asynchronous variant of synchronous ground expansion (c.f. Definition 2.3.9).

Definition 2.5.5 *A relation $\mathcal{S}$ on processes is an* asynchronous ground expansion *if $P \ \mathcal{S} \ Q$ implies:*

1. *If $P \xrightarrow{\mu} P'$, μ is not an input, and $\mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$, then there exists Q' such that $Q \overset{\mu}{\Longrightarrow} Q'$ and $P' \ \mathcal{S} \ Q'$.*
2. *If $P \xrightarrow{a(b)} P'$, $b \notin \mathrm{fn}(Q)$, there exists Q' such that:*
   (a) either $Q \overset{a(b)}{\Longrightarrow} Q'$ and $P' \ \mathcal{S} \ Q'$
   (b) or $Q \overset{\tau}{\Longrightarrow} Q'$ and $P' \ \mathcal{S} \ (Q' \mid \overline{a}b)$.
3. *If $Q \xrightarrow{\mu} Q'$, μ is not an input, and $\mathrm{bn}(\mu) \cap \mathrm{fn}(P) = \emptyset$, then there exists P' such that $P \xrightarrow{\hat{\mu}} P'$ and $P' \ \mathcal{S} \ Q'$;*
4. *if $Q \xrightarrow{a(b)} Q'$, $b \notin \mathrm{fn}(P)$, then there exists P' such that:*
   (a) either $P \xrightarrow{a(b)} P'$ and $P' \ \mathcal{S} \ Q'$
   (b) or $P \xrightarrow{\hat{\tau}} P'$ and $(P \mid \overline{a}b) \ \mathcal{S} \ Q'$.

We write $P \lesssim_{\mathrm{a}} Q$, if $P \ \mathcal{S} \ Q$ for some asynchronous ground expansion $\mathcal{S}$.

[1] They also coincide with other form of bisimilarity like *late* and *open* [74].

2.5.2 The πI-calculus

The mobility mechanism of π-calculus can be divided into *internal* mobility and *external* mobility [77]. The former arises when an input meets the output of a private name, the latter when an input meets the output of a free name. The πI-calculus is a subcalculus of π, where only private names may be transmitted. The syntax is given by the following grammar:

$$P ::= \mathbf{0} \;\Big|\; a(x).\,P \;\Big|\; (\boldsymbol{\nu} b)(\overline{a}b.\,P) \;\Big|\; P \mid P \;\Big|\; (\boldsymbol{\nu} a)P \;\Big|\; A\langle a_1, \ldots, a_n\rangle$$

In πI, recursive definitions are more appropriate than replication. This because, when only internal mobility is allowed, recursion is strictly more expressive than replication [77]. Each constant A has a unique defining equation of the form $A \stackrel{\text{def}}{=} (\tilde{x})P$. In constant definition $A \stackrel{\text{def}}{=} (\tilde{x})P$ and constant application $A\langle\tilde{a}\rangle$, the parameters $\tilde{x}$ and $\tilde{a}$ are tuples of all distinct names whose length equals the arity of A. In a constant definition $A \stackrel{\text{def}}{=} (\tilde{x})P$ all free occurrences of names $\tilde{x}$ in P are bound and $\mathrm{fn}(P) \subseteq \tilde{x}$. The transition rule for constants is the standard one [57].

In πI, as well as in asynchronous π-calculus, ground and early bisimilarity are congruence relation and coincide with barbed congruence [2]. Furthermore, since only fresh names can be exchanged, the match operator becomes superfluous. For instance, in $a(x).\,b(y).\,[x = y]P$ the variables x and y will be always instantiated with different names. The expressive power of πI is quite surprising: both λ-calculus and higher-order communications can be faithfully encoded.

Finally, an asynchronous variant of πI can be easily defined by replacing in the grammar the blocking output processes $(\boldsymbol{\nu} b)(\overline{a}b.\,P)$ with $(\boldsymbol{\nu} b)(\overline{a}b \mid P)$. We denote this calculus with $\pi\mathrm{I_a}$.

2.5.3 The Join calculus

The Join calculus [24] is an off-spring of the asynchronous π-calculus specifically designed to facilitate distributed implementations of channels mechanisms. The syntax of the (core) Join calculus is given by the following grammar:

$$P ::= a\langle b\rangle \;\Big|\; P_1 \mid P_2 \;\Big|\; \texttt{def}\ a\langle x\rangle \,|\, b\langle y\rangle = P_1\ \texttt{in}\ P_2.$$

The particle $a\langle b\rangle$ denotes the asynchronous output of name b at channel a. $P_1 \mid P_2$ denotes two processes P_1 and P_2 running in parallel. The construct $\texttt{def}\ a\langle x\rangle \,|\, b\langle y\rangle = P_1\ \texttt{in}\ P_2$ is a sort of amalgamation of the operators of replication, parallel composition, and restriction, which allows to model the joint reception of values from different channels.

Free names and *bound names* of a process P, are defined as follows:

- $\mathrm{fn}(a\langle b\rangle) = \{a, b\}$,

[2]They also coincide with other form of bisimilarity like *late* and *open* [74].

- $\mathrm{fn}(P_1 \mid P_2) = \mathrm{fn}(P_1) \cup \mathrm{fn}(P_2)$,
- $\mathrm{fn}(\mathtt{def}\ a\langle x\rangle \,|\, b\langle y\rangle = P_1\ \mathtt{in}\ P_2) = ((\mathrm{fn}(P_1) \setminus \{x, y\}) \cup \mathrm{fn}(P_2)) \setminus \{a, b\}$.

The semantics of the Join calculus was originally given using the framework of reflexive chemical abstract machine [9], but one can also resort to a simple reduction semantics based on a straightforward structural equivalence relation $\equiv$, as suggested by Lévy [44]. With J abbreviating a join pattern $a\langle x\rangle \mid b\langle y\rangle$, the main (simplified) reduction rule is:

$$\mathtt{def}\ J = P\ \mathtt{in}\ J\sigma \mid Q \quad\rightarrow\quad \mathtt{def}\ J = P\ \mathtt{in}\ P\sigma \mid Q$$

where the substitution σ does not affect channels a and b.

The intuition is that if an instantiation $J\sigma$ of a definition join-patter J can be found at top-level in the scope of the definition, then this instance may be replaced by the corresponding instantiation $P\sigma$ of the defined continuation P.

The construct $\mathtt{def}\ a\langle x\rangle \,|\, b\langle y\rangle = P_1\ \mathtt{in}\ P_2$ has the following properties:

1. channels a and b are *locally defined*, that is, they can be accessed only from within P_1 and P_2;
2. channels a and b are *uniquely defined*, that is, they appear only in one definition (see also [3, 78]).

A derived construct in Join is the *single pattern definition*

$$\mathtt{def}\ a\langle x\rangle = P_1\ \mathtt{in}\ P_2$$

which can be seen as an abbreviation for $\mathtt{def}\ a\langle x\rangle \,|\, b\langle y\rangle = P_1 \,|\, y\langle y\rangle\ \mathtt{in}\ P_2 \,|\, b\langle b\rangle$, where b is not free in P_1 and P_2 and y is not free in P_1.

The *observability predicate* for the Join calculus is similar to that in Definition 2.5.1. Finally, *barbed congruence* can be defined in the standard manner (see Section 2.3).

2.5.4 The Fusion Calculus

The Fusion calculus [65] is a simplification of polyadic π-calculus [51] where the input operator is not binding, that is, there is only one binding operator, written $(a)P$, meaning that name a is private to P. In this section, we give a calculus which is a slight simplification of that in [65], and we only recall the notions which are essential for the thesis, details on the Fusion calculus can be found in [87].

Processes are given by the following grammar:

$$P ::= \Sigma_{i\in I}\alpha_i.\, P_i \;\Big|\; P \mid P \;\Big|\; (a)P \;\Big|\; [a = b]P \;\Big|\; !P$$

Here I is a finite indexing set; in the case $I = \emptyset$ we write the choice as $\mathbf{0}$. In a term $\alpha.\, P$ the prefix α represents either input or output:

$$\alpha ::= a\langle\widetilde{b}\rangle \mid \overline{a}\langle\widetilde{b}\rangle.$$

The angle brackets in the input prefix mean that, like output, input is not binding. Conventions about names are as in π, except for the input prefix for which we have:

$$\mathrm{fn}(a\langle\widetilde{b}\rangle.\,P) \stackrel{\mathrm{def}}{=} \{a\} \cup \widetilde{b} \cup \mathrm{fn}(P) \quad \text{and} \quad \mathrm{bn}(a\langle\widetilde{b}\rangle.\,P) \stackrel{\mathrm{def}}{=} \mathrm{bn}(P).$$

We write $(\widetilde{a})P$ for $(a_1)\ldots(a_n)P$.

The original Fusion calculus is equipped with a further action/process $\{\widetilde{a} = \widetilde{b}\}$, called *fusion*, which represents the effect of a communication, that is, the identification of output and input object parts. Fusion is not a primitive construct because it can be derived by the grammar above as follows:

$$\{\widetilde{a} = \widetilde{b}\}.\,P \stackrel{\mathrm{def}}{=} (u)(\overline{u}\langle\widetilde{a}\rangle.\,\mathbf{0} \mid u\langle\widetilde{b}\rangle.\,P).$$

A fusion represents an equivalence relation over names. Precisely, the fusion $\{\widetilde{a} = \widetilde{b}\}$, where $|\widetilde{a}| = |\widetilde{b}|$, denotes the smallest equivalence relation relating each a_i with b_i.

The original Fusion calculus is equipped with both a *reduction* and *labelled semantics*. The reduction semantics is quite similar to that seen in Section 2.2.1 for the π-calculus, except for a crucial point: in Fusion calculus, the transmission of names between two processes may arise only when carrying restricted names, that is, names with a delimited scope. Like π-calculus, the Fusion calculus is equipped with a notion of *structural congruence* which is essentially the same as in Definition 2.2.1. Also the *reduction rules* are the same as those of the π-calculus (see Table 2.1), except for the rule of communication between two processes. Given a fusion $\{\widetilde{a} = \widetilde{b}\}$, we say that a substitution σ *agrees* with $\{\widetilde{a} = \widetilde{b}\}$ if $\sigma(a_i) = \sigma(b_i)$ for all i. The reduction rule modelling the communication between two processes is the following:

$$\texttt{com:}\ \frac{}{(\widetilde{z})\big(R \mid (\ldots + a\langle\widetilde{x}\rangle.\,P) \mid (\ldots + \overline{a}\langle\widetilde{b}\rangle.\,Q)\big) \longrightarrow (R \mid P \mid Q)\sigma}$$

where $|\,\widetilde{x}\,|=|\,\widetilde{b}\,|$, σ *agrees* with $\{\widetilde{x} = \widetilde{b}\}$, $\mathrm{ran}(\sigma) = \emptyset$, and $\mathrm{dom}(\sigma) = \widetilde{z}$.

The *observability predicate* for the Fusion calculus is similar to that in Definition 2.3.1. Finally, *barbed congruence* can be defined in the standard manner (see Section 2.3).

2.6 Correctness criteria for encodings

A standard manner of understanding a new calculus is to code it up in terms of a more familiar or better established calculus. When studying an encoding between two calculi it is necessary to have some *correctness criteria* for the encoding. The most satisfactory correctness criterion for encodings is *full abstraction* which relates source and target calculi at the level of behavioural equivalences. Precisely, two source calculus terms are behavioural equivalent if and only if their translations are behavioural equivalent. Full abstractions should be completed by an *operational correspondence* between a term and its translation. An operational correspondence relates execution steps, as defined by an operational semantics of the source and target calculi.

More formally, let us denote with $(\mathcal{S}, \asymp_{\mathrm{s}}, \longrightarrow_{\mathrm{s}})$ and $(\mathcal{T}, \asymp_{\mathrm{t}}, \longrightarrow_{\mathrm{t}})$ two process calculi equipped with behavioural equivalences $\asymp_{\mathrm{s}}$ and $\asymp_{\mathrm{t}}$, and transition relations $\longrightarrow_{\mathrm{s}}$ and $\longrightarrow_{\mathrm{t}}$, respectively. Let $[\![\cdot]\!] : \mathcal{S} \longmapsto \mathcal{T}$ be an encoding from $\mathcal{S}$ to $\mathcal{T}$. A formal definition of operational correspondence is the following:

Definition 2.6.1 (operational correspondence) *Given two process calculi* $(\mathcal{S}, \asymp_{\mathrm{s}}, \longrightarrow_{\mathrm{s}})$ *and* $(\mathcal{T}, \asymp_{\mathrm{t}}, \longrightarrow_{\mathrm{t}})$*, an encoding* $[\![\cdot]\!]{:}\mathcal{S} \mapsto \mathcal{T}$ *enjoys a (strong)* operational correspondence *if for each* $S{\in}\mathcal{S}$ *the following two properties holds:*

1. *If* $S \longrightarrow_{\mathrm{s}} S'$ *then* $[\![S]\!] \longrightarrow_{\mathrm{t}} \asymp_{\mathrm{t}} [\![S']\!]$.
2. *If* $[\![S]\!] \longrightarrow_{\mathrm{t}} T$ *then there is* S' *such that* $S \longrightarrow_{\mathrm{s}} S'$ *and* $T \asymp_{\mathrm{t}} [\![S']\!]$.

Requirements 1 and 2 assert that all possible executions of S may be simulated, up to behavioural equivalence, by its translation, and vice-versa. A notion of *weak operational correspondence* can be easily derived from Definition 2.6.1 by simply replacing $\longrightarrow_{\mathrm{s}}$ and $\longrightarrow_{\mathrm{t}}$ with their reflexive transitive closure in requirements 1 and 2.

Full abstraction has two parts: *soundness*, which says that the equivalence between the translations of two source terms implies that of the source terms themselves; and *completeness*, which says the converse. Soundness is sometime enough to assert that the target calculus is at least as expressive as the source calculus, but it doesn't say anything about the quality of the encoding. Completeness is in general much harder to achieve, but together with soundness, it provides a much finer result. For instance, completeness guarantees that all algebraic laws in the source calculus are reflected in the target calculus.

Definition 2.6.2 *Given two process calculi* $(\mathcal{S}, \asymp_{\mathrm{s}}, \longrightarrow_{\mathrm{s}})$ *and* $(\mathcal{T}, \asymp_{\mathrm{t}} \longrightarrow_{\mathrm{t}})$*. An encoding* $[\![\cdot]\!] : \mathcal{S} \longmapsto \mathcal{T}$ *is* sound *if* $[\![S_1]\!] \asymp_{\mathrm{t}} [\![S_2]\!]$ *implies* $S_1 \asymp_{\mathrm{s}} S_2$ *for each* $S_1, S_2 \in \mathcal{S}$*; it is* complete *if* $S_1 \asymp_{\mathrm{s}} S_2$ *implies* $[\![S_1]\!] \asymp_{\mathrm{t}} [\![S_2]\!]$ *for each* $S_1, S_2 \in \mathcal{S}$*; it is* fully-abstract *if it is sound and complete.*

Full abstraction will be our main correctness criterion for the encodings we are going to present in this thesis.

Chapter 3

The Local π-calculus

In this chapter, we study the foundational theory of Lπ.

We give two labelled characterisations of barbed congruence for image-finite processes. The first is based on an embedding of Lπ into a an asynchronous variant of Sangiorgi's πI (c.f. Section 2.5.2) where only the output capability on names is transmitted. Barbed congruence between processes of Lπ coincides, on their images, with a variant of asynchronous early bisimulation. The second characterisation is based on a new labelled transition system (LTS) which modifies the standard LTS to reveal what is observable in Lπ, that is, what an external observer that behaves like a Lπ-process can see by interacting with an Lπ-process. Barbed congruence in Lπ coincides with the standard asynchronous ground bisimilarity defined on the new LTS. The resulting coinductive proof method can be enhanced by means of both standard and new "bisimulation up–to" proof techniques.

3.1 The syntax

Lπ is a subset of asynchronous π-calculus. Lπ has operators of inaction, input prefix, asynchronous output, parallel composition, restriction and replicated input.

Definition 3.1.1 *The grammar of* Lπ*-processes is*

$$P ::= \mathbf{0} \;\Big|\; a(b).\,P \;\Big|\; \overline{a}b \;\Big|\; P \mid P \;\Big|\; (\boldsymbol{\nu} a)P \;\Big|\; !a(b).\,P$$

with the syntactic constraint *that in processes $a(b).\,P$ and $!a(b).\,P$ name b may not occur free in P in input position.*

The calculus has been equipped with replicated input because: (i) replicated input has the same expressive power as full replication [36] and recursion [51, 82]; (ii) replicated input has a simple semantics and is handy for implementations [68]. Nevertheless, we can replace in Lπ replicated input with recursion maintaining theory and results.

$\mathrm{L}\pi$ is a monadic calculus in the sense that only a single name at a time may be transmitted. The theory of $\mathrm{L}\pi$ can be easily adapted to the *polyadic* case, where tuples of names may be transmitted. We denote with $\overline{a}\langle\widetilde{b}\rangle$ and $a(\widetilde{x}).\, P$ polyadic output and polyadic input, respectively.

The *labelled transition system* is as in Table 2.2 except for the replicated input for which we have the simpler rule:

$$\texttt{rep-i:}\ \frac{}{!a(x).\,P \xrightarrow{a(x)} P \mid !a(x).\,P}$$

3.2 The link process

The theory of Local π is based on special processes called *links*. Links are processes which behave as name buffers receiving names at one end-point and retransmitting them at the other end-point (in the π-calculus literature, links are sometimes called forwarders [37] or wires [82]).

Definition 3.2.1 (Static link) *Given two names a and b, we call* static link *the process below:*

$$a \triangleright b \stackrel{\mathrm{def}}{=} !a(x).\,\overline{b}x.$$

We sometimes use a more sophisticated form of link $a \to b$, which does not perform free outputs: the name sent at b is not x, but a link to x (this is the definition of links in calculi where all outputs are bound [77]).

Definition 3.2.2 (Dynamic link) *Given two names a and b, we call* dynamic link *the process defined by the following recursive definition:*

$$a \to b \stackrel{\mathrm{def}}{=} !a(x).\,(\boldsymbol{\nu}c)(\overline{b}c \mid c \to x).$$

Being recursively defined, the process $a \to b$ is not in $\mathrm{L}\pi$. In [51], Milner shows that recursive definitions can be encoded, up to bisimilarity, in terms of replication. When applying Milner's encoding to a dynamic link we get a processes which does not respect the $\mathrm{L}\pi$ constraint on the output capability. This problem can be avoided by rewriting the definition of dynamic links as:

$$a \to b \stackrel{\mathrm{def}}{=} !a(x).\,\mathrm{out}(b, x)$$

where $\mathrm{out}(b, x)$ is recursively defined as:

$$\mathrm{out}(b, x) \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}c)(\overline{b}c \mid !c(y).\,\mathrm{out}(x, y)).$$

Process $\mathrm{out}(b, x)$ can be expressed in (polyadic) $\mathrm{L}\pi$ in terms of replication as follows:

$$\mathrm{out}(b, x) \approx (\boldsymbol{\nu}o)(\overline{o}\langle b, x\rangle \mid !o(b, x).\,(\boldsymbol{\nu}c)(\overline{b}c \mid !c(y).\,\overline{o}\langle x, y\rangle)).$$

$$[\![P_1 \mid P_2]\!] \stackrel{\text{def}}{=} [\![P_1]\!] \mid [\![P_2]\!] \quad [\![a(x).\,P]\!] \stackrel{\text{def}}{=} a(x).\,[\![P]\!] \qquad [\![!a(x).\,P]\!] \stackrel{\text{def}}{=} !a(x).\,[\![P]\!]$$

$$[\![(\boldsymbol{\nu} a)P]\!] \stackrel{\text{def}}{=} (\boldsymbol{\nu} a)[\![P]\!] \qquad [\![\overline{a}b]\!] \stackrel{\text{def}}{=} (\boldsymbol{\nu} c)(\overline{a}c \mid c \to b) \quad [\![\mathbf{0}]\!] \stackrel{\text{def}}{=} \mathbf{0}$$

Table 3.1: The encoding of free outputs in terms of bound outputs

As already pointed out in Section 2.5.1, in asynchronous calculi, the relation $\approx$ is a congruence. So, the process $a \to b$ can be expressed in (polyadic) Lπ in terms of replication. Furthermore, by exploiting a slight variant of Milner's encoding of polyadic processes into monadic ones [51], we can prove that there is a process in Lπ which is bisimilar to the recursive process $a \to b$ [1]. The encoding is an homomorphism on all operators except input and output for which we have:

- $\{\!| a(x_1, x_2).\, P |\!\} \stackrel{\text{def}}{=} a(w).\,(\boldsymbol{\nu} c_1 c_3)(\overline{w}c_1 \mid c_1(c_2).\,(\overline{c_2}c_3 \mid c_3(x_1).\,c_1(x_2).\,\{\!| P |\!\}))$
- $\{\!| \overline{a}\langle b_1, b_2\rangle |\!\} \stackrel{\text{def}}{=} (\boldsymbol{\nu} w)(\overline{a}w \mid w(c_1).\,(\boldsymbol{\nu} c_2)(\overline{c_1}c_2 \mid c_2(c_3).\,(\overline{c_3}b_1 \mid \overline{c_1}b_2)))$.

Finally, we show how links can be joined with each other.

Proposition 3.2.3 *Let a and b be names different from c. Then*

1. $(\boldsymbol{\nu} b)(a \rhd b \mid b \rhd c) \gtrsim a \rhd c$
2. $(\boldsymbol{\nu} b)(a \to b \mid b \to c) \gtrsim a \to c$
3. $(\boldsymbol{\nu} b)(a \rhd b \mid b \to c) \gtrsim a \to c$
4. $(\boldsymbol{\nu} b)(a \to b \mid b \rhd c) \gtrsim a \to c$

Proof: Parts 1, 3, 4 are proved by simply exhibiting the appropriate expansion relations. Part 2 has been already proved in [10] and requires up-to context proof techniques. $\Box$

3.3 A "translation-based" proof technique

In this section, we give a proof technique for barbed congruence in Lπ based on an encoding of free outputs in terms of bound outputs (see Table 3.1). The encoding $[\![\cdot]\!]$ is an homomorphism on all operators except output. The output particle $\overline{a}b$ is mapped into the $\pi\mathrm{I_a}$-process $(\boldsymbol{\nu} c)(\overline{a}c \mid c \to b)$ where $c \notin \{a, b\}$. Our encoding is the asynchronous variant of Boreale's [10] which maps $\overline{a}b$ into the πI-process $(\boldsymbol{\nu} c)(\overline{a}c.\, c \to b)$. It is easy to see that the two encodings are equivalent because in both cases the link $c \to b$ is active only after performing the bound output along a.

[1] For our purposes it suffices to consider the encoding where only pairs are taken into account.

Remark 3.3.1 *Notice that the process $(\boldsymbol{\nu} c)(\overline{a}c \mid c \to b)$ coincides with the recursive definition $\mathrm{out}(a, b)$ seen in Section 3.2.*

In Section 3.3.1, we present some results on $[\![\cdot]\!]$ already proved by Boreale [10] for its encoding and that trivially apply to our encoding as well. Then, in Section 3.3.2 we give the proof technique.

3.3.1 Background on $[\![\cdot]\!]$

The encoding $[\![\cdot]\!]$ commutes with substitutions, i.e., for each substitution σ it holds that $[\![P\sigma]\!] = [\![P]\!]\sigma$. In [10] Boreale proves an operational correspondence between processes P and $[\![P]\!]$; he also gives an adequacy result for $[\![\cdot]\!]$ with respect to barbed bisimulation.

Lemma 3.3.2 (Boreale [10]) *Let P be a process in $\mathrm{L}\pi$.*

1. *Suppose that $P \xrightarrow{\mu} P'$. Then we have:*
   (a) *if $\mu = a(c)$ then $[\![P]\!] \xrightarrow{a(c)} \gtrsim [\![P']\!]$;*
   (b) *if $\mu = \overline{a}b$ then $[\![P]\!] \xrightarrow{\overline{a}(c)} \gtrsim (c \to b \mid [\![P']\!])$, with $c \notin \mathrm{fn}(P')$;*
   (c) *if $\mu = \overline{a}(b)$ then $[\![P]\!] \xrightarrow{\overline{a}(c)} \gtrsim (\boldsymbol{\nu} b)(c \to b \mid [\![P']\!])$, with $c \notin \mathrm{fn}(P')$;*
   (d) *if $\mu = \tau$ then $[\![P]\!] \xrightarrow{\tau} \gtrsim [\![P']\!]$.*
2. *Suppose that $[\![P]\!] \xrightarrow{\mu} P_1$. Then there exists $P' \in \mathrm{L}\pi$ such that:*
   (a) *if $\mu = a(c)$ then $P \xrightarrow{a(c)} P'$, with $P_1 \gtrsim [\![P']\!]$;*
   (b) *if $\mu = \overline{a}(c)$ then:*
      i. *either $P \xrightarrow{\overline{a}b} P'$, with $c \notin \mathrm{fn}(P')$ and $P_1 \gtrsim (c \to b \mid [\![P']\!])$,*
      ii. *or $P \xrightarrow{\overline{a}(b)} P'$, with $c \notin \mathrm{fn}(P')$ and $P_1 \gtrsim (\boldsymbol{\nu} b)(c \to b \mid [\![P']\!])$;*
   (c) *if $\mu = \tau$ then $P \xrightarrow{\tau} P'$ with $P_1 \gtrsim [\![P']\!]$.*

The proof of the lemma above relies on Proposition 3.2.3(2) and a technical but important lemma that we report below.

Lemma 3.3.3 (Boreale [10]) *Let P be an $\mathrm{L}\pi$ process, and a and b two names such that $a \neq b$ and a does not occur free in P in input-subject position. Then:*

$$(\boldsymbol{\nu} a)(a \to b \mid [\![P]\!]) \gtrsim [\![P]\!]\{^b\!/\!_a\}.$$

From Lemma 3.3.2 Boreale derives the following adequacy result for $[\![\cdot]\!]$.

Theorem 3.3.4 (Boreale [10]) *Let P and Q be two processes in $\mathrm{L}\pi$. Then:*

$$P \stackrel{\bullet}{\approx} Q \ \text{ iff } \ [\![P]\!] \stackrel{\bullet}{\approx} [\![Q]\!].$$

3.3.2 The proof technique

For convenience we define a calculus, called $\mathcal{L}\pi$, whose operators are the same as those of Lπ but with the output construct $\overline{a}b$ replaced by its translation $[\![\overline{a}b]\!]$. Formally, the syntax of $\mathcal{L}\pi$ is the following:

$$P ::= \mathbf{0} \;\Big|\; a(x).\,P \;\Big|\; [\![\overline{a}b]\!] \;\Big|\; P \mid P \;\Big|\; (\boldsymbol{\nu} a)P \;\Big|\; !a(x).\,P$$

with the same constraint on received names as in Lπ. $\mathcal{L}\pi$ is the image of the encoding $[\![\cdot]\!]$.

Lemma 3.3.5 *If P is an* Lπ*-process then $[\![P]\!]$ is an $\mathcal{L}\pi$-process. Vice versa, for each $\mathcal{L}\pi$-process Q there exists an* Lπ*-process P such that $[\![P]\!] = Q$.*

It is easy to see that $\mathcal{L}\pi$ is closed under under labelled transitions:

Lemma 3.3.6 *Let P be an $\mathcal{L}\pi$-process. If $P \xrightarrow{\mu} P_1$ then $P_1 \in \mathcal{L}\pi$.*

Proof: By transition induction. □

The proof technique to verify if two Lπ-processes are barbed congruent consists in translating them, by means of $[\![\cdot]\!]$, and then checking if their images are related by the bisimilarity $\asymp_{\mathrm{a}}$ defined below. The relation $\asymp_{\mathrm{a}}$ is a slight variant of asynchronous ground bisimilarity (c.f. Definition 2.5.4) in which the output particle $\overline{a}b$ in clause 2b is replaced by $[\![\overline{a}b]\!]$.

Definition 3.3.7 ($\asymp_{\mathrm{a}}$-bisimilarity) *A symmetric relation $\mathcal{S}$ on processes of $\mathcal{L}\pi$ is a* $\asymp_{\mathrm{a}}$-bisimulation *if whenever $P \;\mathcal{S}\; Q$ the following holds:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \Longrightarrow Q'$ and $P' \;\mathcal{S}\; Q'$.*
2. *If $P \xrightarrow{\overline{a}(b)} P'$, $b \notin \mathrm{fn}(Q)$, then there exists Q' such that $Q \stackrel{\overline{a}(b)}{\Longrightarrow} Q'$ and $P' \;\mathcal{S}\; Q'$.*
3. *If $P \xrightarrow{ab} P'$, $b \notin \mathrm{fn}(P, Q)$, then there exists Q' such that:*
   (a) either $Q \stackrel{a(b)}{\Longrightarrow} Q'$ and $P' \;\mathcal{S}\; Q'$
   (b) or $Q \Longrightarrow Q'$ and $P' \;\mathcal{S}\; (Q' \mid [\![\overline{a}b]\!])$.

Processes P and Q are $\asymp_{\mathrm{a}}$-bisimilar, written $P \asymp_{\mathrm{a}} Q$, if $P \,\mathcal{S}\, Q$ for some $\asymp_{\mathrm{a}}$-bisimulation $\mathcal{S}$.

$\asymp_{\mathrm{a}}$ is designed to be used on $\mathcal{L}\pi$-processes and therefore its definition does not contain clauses for free output actions. In Lemmas 3.3.8 and 3.3.12 we will show that $\asymp_{\mathrm{a}}$ is an equivalence relation and it is preserved by all operators of $\mathcal{L}\pi$.

Lemma 3.3.8 *Let P, Q and R be three $\mathcal{L}\pi$-processes. Then:*

1. *$P \asymp_{\mathrm{a}} P$;*
2. *$P \asymp_{\mathrm{a}} Q$ implies $Q \asymp_{\mathrm{a}} P$;*

3. *$P \precsim_a Q$ and $Q \precsim_a R$ implies $P \precsim_a R$.*

Proof: The only nontrivial property is transitivity. We essentially need to prove two results:

- $\precsim_a$ is preserved by injective substitutions. The proof is analogous to that for $\approx$ in π-calculus.
- $P \precsim_a Q$ implies $P \mid [\![\overline{a}b]\!] \precsim_a Q \mid [\![\overline{a}b]\!]$. This result is necessary in the asynchronous clause for input. The proof follows by Lemma 3.3.12(2). Notice that the proof of Lemma 3.3.12(2) does not rely on the transitivity of $\precsim_a$.

$\square$

In order to prove that $\precsim_a$ is a congruence on $\mathcal{L}\pi$, we adapt the up-to proof technique seen in Section 2.4.

Definition 3.3.9 ($\precsim_a$-bisimulation up to $\gtrsim$ and $\approx$) *A symmetric relation $\mathcal{S}$ is a* $\precsim_a$-bisimulation up to $\gtrsim$ and $\approx$ *if whenever $P \;\mathcal{S}\; Q$ the following holds:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \Longrightarrow Q'$ and $P' \gtrsim \mathcal{S} \approx Q'$.*
2. *If $P \xrightarrow{\overline{a}(b)} P'$, $b \notin \mathrm{fn}(Q)$, then there exists Q' such that $Q \stackrel{\overline{a}(b)}{\Longrightarrow} Q'$ and $P' \gtrsim \mathcal{S} \approx Q'$.*
3. *If $P \xrightarrow{ab} P'$, $b \notin \mathrm{fn}(P, Q)$, then there exists Q' such that:*
   (a) *either $Q \stackrel{ab}{\Longrightarrow} Q'$ and $P' \gtrsim \mathcal{S} \approx Q'$*
   (b) *or $Q \Longrightarrow Q'$ and $P' \gtrsim \mathcal{S} \approx (Q' \mid [\![\overline{a}b]\!])$.*

Lemma 3.3.10 *If $\mathcal{S}$ is a $\precsim_a$-bisimulation up to $\gtrsim$ and $\approx$ then $\mathcal{S} \subseteq \precsim_a$.*

Proof: The proof is analogous to that for standard bisimilarity [81]. Let $\mathcal{S}$ be a $\precsim_a$-bisimulation up to $\gtrsim$ and $\approx$, we show that the relation $\approx \mathcal{S} \approx$ is a $\precsim_a$-bisimulation. This follows from the transitivity of $\approx$ and the fact that $\approx$ is preserved by parallel composition. In the asynchronous clause for input, we need to know that $\approx$ is preserved by parallel composition. Finally, since $\mathcal{S} \subseteq \approx \mathcal{S} \approx \subseteq \precsim_a$, we can conclude. $\square$

The following lemma gives us some information about the structure of the $\mathcal{L}\pi$-processes which may perform an output action. Part 2 of Lemma 3.3.11 will be crucial when proving that $\precsim_a$ is preserved by parallel composition.

Lemma 3.3.11 *Let P be an $\mathcal{L}\pi$-process.*

1. *If $P \xrightarrow{\overline{a}(c)} P_1$ then $P \equiv (\boldsymbol{\nu}\widetilde{z})((\boldsymbol{\nu}c)(\overline{a}c \mid c \to b) \mid P_2)$ and $P_1 \equiv (\boldsymbol{\nu}\widetilde{z})(c \to b \mid P_2)$ for some $\widetilde{z}$, b and P_2 such that $\{a, c\} \cap \widetilde{z} = \emptyset$ and $c \notin \mathrm{fn}(P_2)$.*
2. *If $P \xrightarrow{\overline{a}(c)} P_1$ then $P \approx (\boldsymbol{\nu}c)([\![\overline{a}c]\!] \mid P_1)$.*

Proof:

1. By transition induction.
2. By part 1 $P \equiv (\boldsymbol{\nu}\widetilde{z})((\boldsymbol{\nu}c)(\overline{a}c \mid c \to b) \mid P_2)$ and $P_1 \equiv (\boldsymbol{\nu}\widetilde{z})(c \to b \mid P_2)$ for some $\widetilde{z}$, b and P_2 such that $\{a, c\} \cap \widetilde{z} = \emptyset$ and $c \notin \mathrm{fn}(P_2)$. By proposition 3.2.3, we can choose d fresh such that:
$$\begin{aligned} P &\equiv (\boldsymbol{\nu}\widetilde{z})((\boldsymbol{\nu}d)(\overline{a}d \mid d \to b) \mid P_2) \\ &\approx (\boldsymbol{\nu}\widetilde{z})((\boldsymbol{\nu}d)(\overline{a}d \mid (\boldsymbol{\nu}c)(d \to c \mid c \to b)) \mid P_2) \\ &\equiv (\boldsymbol{\nu}c)((\boldsymbol{\nu}d)(\overline{a}d \mid d \to c) \mid (\boldsymbol{\nu}\widetilde{z})(c \to b \mid P_2)) \\ &\equiv (\boldsymbol{\nu}c)([\![\overline{a}c]\!] \mid P_1). \end{aligned}$$

□

Lemma 3.3.12 *Let P and Q be two $\mathcal{L}\pi$-processes such that $P \asymp_{\mathrm{a}} Q$. Then:*

1. $(\boldsymbol{\nu}a)P \asymp_{\mathrm{a}} (\boldsymbol{\nu}a)Q$*;*
2. *for each $\mathcal{L}\pi$-process R it holds that $P \mid R \asymp_{\mathrm{a}} Q \mid R$;*
3. $a(x).\, P \asymp_{\mathrm{a}} a(x).\, Q$*;*
4. $!a(x).\, P \asymp_{\mathrm{a}} !a(x).\, Q$*.*

Proof: Here we give a sketch of the proof. A detailed proof can be found in Appendix A.1.1

1. The proof relies on the fact that all output actions performed by $\mathcal{L}\pi$ processes are bound. This implies that the restriction operator cannot bind names in output object position.
2. By applying both the up-to proof techniques of Definition 3.3.9 and Lemma 3.3.11. Note that in general to prove that a ground bisimulation is preserved by parallel composition is quite difficult. Here, it is easier because processes in $\mathcal{L}\pi$ never perform free output actions.
3. Straightforward.
4. The proof relies on the fact that $\asymp_{\mathrm{a}}$ is preserved by injective substitutions, parallel composition, and restriction. Lemma 3.3.8(3) is also necessary.

□

Finally, we can prove the soundness of our "translation-based" technique for barbed congruence in $\mathrm{L}\pi$.

Theorem 3.3.13 (Soundness) *Let P and Q be two processes in $\mathrm{L}\pi$. Then:*

$$[\![P]\!] \asymp_{\mathrm{a}} [\![Q]\!] \ \text{ implies } \ P \cong_{\mathrm{L}\pi} Q.$$

Proof: By Lemmas 3.3.5 and 3.3.12 it holds that $[\![C[P]]\!] \asymp_{\mathrm{a}} [\![C[Q]]\!]$ for any context $C[\cdot]$ in $\mathrm{L}\pi$. Since $\asymp_{\mathrm{a}} \subset \dot{\approx}$, it holds that $[\![C[P]]\!] \dot{\approx} [\![C[Q]]\!]$ and therefore, by Theorem 3.3.4, $C[P] \dot{\approx} C[Q]$. □

Corollary 3.3.14 *Let P and Q be two processes in* $\mathrm{L}\pi$. *Then:*

$$[\![P]\!] \approx [\![Q]\!] \text{ implies } P \cong_{\mathrm{L}\pi} Q.$$

Proof: $\approx$ implies $\precsim_{\mathrm{a}}$. By applying Theorem 3.3.13 we get the result. □

Remark 3.3.15 *Theorem 3.3.13 (resp. Corollary 3.3.14) does not hold when replacing $\precsim_{\mathrm{a}}$ (resp. $\approx$) and barbed congruence with their expansion variants, respectively. This because, as we will see in the next section, the encoding $[\![\cdot]\!]$ may introduce a few τ-steps changing the balance of silent moves between two processes. Had such an expansion variant of Theorem 3.3.13 (and Corollary 3.3.14) been available, then proof of a key result in Section 4.3 would have been much simpler (see Remark 4.3.7).*

The encoding $[\![\cdot]\!]$ is essentially based on the law below, relating free and bound output actions:

$$\overline{a}b \cong_{\mathrm{L}\pi} (\boldsymbol{\nu}c)(\overline{a}c \mid c \to b) \tag{3.1}$$

However, in $\mathrm{L}\pi$, it holds an even simpler law which relates free and bound output actions. We recall that $c \triangleright b$ denotes the static link of Definition 3.2.1.

Lemma 3.3.16 *Let a and b be two names different from c. Then:*

$$\overline{a}b \cong_{\mathrm{L}\pi} (\boldsymbol{\nu}c)(\overline{a}c \mid c \triangleright b).$$

Proof: We code up, via $[\![\cdot]\!]$, both members of the equation obtaining, respectively:

$$(\boldsymbol{\nu}d)(\overline{a}d \mid d \to b) \text{ and } (\boldsymbol{\nu}c)((\boldsymbol{\nu}d)(\overline{a}d \mid d \to c) \mid c \to b).$$

By Proposition 3.2.3(2) the two processes above are synchronous bisimilar. By Corollary 3.3.14, we can conclude. □

From Lemma 3.3.16 we can derive a simpler "translation based" proof technique for barbed congruence. A generalisation of Lemma 3.3.16 will be crucial to prove the main result of Chapter 7.

3.4 A "run-time" proof technique

The main drawback of the proof technique of Theorem 3.3.13 is that the encoding $[\![\cdot]\!]$ adds a dynamic link to all outputs in the source processes, slowing down computation. A process $[\![P]\!]$ will normally take more steps to simulate the computation of P. For instance, if P is

$$\overline{a}b \mid a(x).\,\overline{x}c \mid b(z).\,\overline{z}w$$

$$\texttt{free-out:}\frac{P \xrightarrow{\overline{a}b} P' \qquad p \notin \mathrm{fn}(P)}{P \overset{\overline{a}(p)}{\longmapsto} (p \rhd b \mid P')} \qquad \texttt{bound-out:}\frac{P \xrightarrow{\overline{a}(b)} P' \qquad p \notin \mathrm{fn}(P)}{P \overset{\overline{a}(p)}{\longmapsto} (\boldsymbol{\nu} b)(p \rhd b \mid P')}$$

$$\texttt{sync:}\frac{P \xrightarrow{\tau} P'}{P \overset{\tau}{\longmapsto} P'} \qquad \texttt{input:}\frac{P \xrightarrow{a(b)} P'}{P \overset{a(b)}{\longmapsto} P'}$$

Table 3.2: A new labelled transition system for $\mathrm{L}\pi$

then P needs two τ-steps before producing an output at c, whereas its image

$$[\![\overline{a}b]\!] \mid a(x).\,[\![\overline{x}c]\!] \mid b(z).\,[\![\overline{z}w]\!]$$

will take 5 steps. However, we conjecture that the encoding does not introduce divergences, that is, infinite internal computations. The proof of this statement seems to be quite complex and strictly related to the syntax of $\mathrm{L}\pi$. For instance, adding replicated output the encoding is not divergence-free. As an example, take the process

$$P \overset{\mathrm{def}}{=} \overline{a}b \mid a(x).\,!\overline{x}c$$

then, P does not diverge, whereas $[\![P]\!]$ does. We leave the proof (or disproof) of the divergence-freedom of $[\![\cdot]\!]$ in $\mathrm{L}\pi$ as an open problem.

In this section, we introduce a more powerful technique based on a new Labelled Transition System (LTS) $\overset{\mu}{\longmapsto}$ for $\mathrm{L}\pi$. The new LTS, reported in Table 3.2, is defined on top of the original one, and transforms the output of a name b into the output of a fresh *pointer* p to b. We call p a pointer to b because a link $p \rhd b$ is introduced through which any output along p is redirected onto b. The new LTS makes explicit the constraint that only the output capability of names may be transmitted by transforming the occurrence of a name in output object position with an occurrence of the same name in output subject position.

The weak transitions $\overset{\mu}{\Longmapsto}$ and $\Longmapsto$ for the new LTS are defined from the strong transitions $\overset{\mu}{\longmapsto}$ and $\overset{\tau}{\longmapsto}$ in the usual way. We write $\overset{\mapsto}{\approx}_{\mathrm{a}}$ to denote the relation obtained by replacing, in Definition 2.5.4, arrow $\longrightarrow$ with $\longmapsto$ and arrow $\Longrightarrow$ with $\Longmapsto$. We prove that asynchronous ground bisimulation defined on the new LTS implies barbed congruence. To avoid reasoning with two LTSs, we reformulate the definition of $\overset{\mapsto}{\approx}_{\mathrm{a}}$ on the original LTS.

Definition 3.4.1 (Link bisimilarity) *A symmetric relation $\mathcal{S}$ on processes of $\mathrm{L}\pi$ is a* link bisimulation *if whenever $P \mathrel{\mathcal{S}} Q$ the following holds:*

1. *If $P \xrightarrow{\tau} P'$ then $Q \Longrightarrow Q'$ and $P' \mathrel{\mathcal{S}} Q'$.*
2. *If $P \xrightarrow{a(p)} P'$, and $p \notin \mathrm{fn}(Q)$, then*

(a) *either* $Q \stackrel{a(p)}{\Longrightarrow} Q'$ *and* $P' \ \mathcal{S}\ Q'$
(b) *or* $Q \Longrightarrow Q'$ *and* $P' \mathcal{S}\ (Q' \mid \overline{a}p)$.

3. *If* $P \stackrel{\overline{a}b}{\longrightarrow} P'$, *then*
   (a) *either* $Q \stackrel{\overline{a}d}{\Longrightarrow} Q'$ *and* $\exists p \notin \mathrm{fn}(P, Q)$ *s.t.* $(p \rhd b \mid P')\ \mathcal{S}\ (p \rhd d \mid Q')$
   (b) *or* $Q \stackrel{\overline{a}(c)}{\Longrightarrow} Q'$, *with* $c \notin \mathrm{fn}(P)$, *and* $\exists p \notin \mathrm{fn}(P, Q)$ *s.t.* $(p \rhd b \mid P')\ \mathcal{S}\ (\boldsymbol{\nu} c)(p \rhd c \mid Q')$.
4. *If* $P \stackrel{\overline{a}(c)}{\longrightarrow} P'$, *with* $c \notin \mathrm{fn}(Q)$, *then*
   (a) *either* $Q \stackrel{\overline{a}b}{\Longrightarrow} Q'$ *and* $\exists p \notin \mathrm{fn}(P, Q)$ *s.t.* $(\boldsymbol{\nu} c)(p \rhd c \mid P')\ \mathcal{S}\ (p \rhd b \mid Q')$
   (b) *or* $Q \stackrel{\overline{a}(c)}{\Longrightarrow} Q'$, $\exists p \notin \mathrm{fn}(P, Q)$ *s.t.* $(\boldsymbol{\nu} c)(p \rhd c \mid P')\ \mathcal{S}\ (\boldsymbol{\nu} c)(p \rhd c \mid Q')$.

P and Q are link bisimilar, written $P \approx_{\mathrm{l}} Q$, *if* $P\ \mathcal{S}\ Q$ *for some link bisimulation* $\mathcal{S}$.

The only difference between link bisimilarity and asynchronous ground bisimilarity (Definition 2.5.4) is in the clauses for output actions: In the link bisimilarity the name emitted by the two processes may be different. In clause 3a , for instance, processes P and Q may emit along a two names b and d. To model the fact that b and d cannot be distinguished by the environment, the bisimulation game continues by adding in parallel two links $p \rhd b$ and $p \rhd d$ which make b and d "equal" to a fresh name p. In some sense, from the environment point of view, it is as if both processes P and Q had emitted the same fresh name p instead of b and d, respectively. Similarly, in clauses 3b and 4a, free output actions are matched by bound output actions. As in the previous case, we need to record, in the derivatives, that we are sending different names.

Lemma 3.4.2 *Let P and Q be two processes in* $\mathrm{L}\pi$. *Then:*

$$P \stackrel{\mapsto}{\approx}_{\mathrm{a}} Q \ \text{ iff } \ P \approx_{\mathrm{l}} Q.$$

The following lemma relates the translation-based and the run-time proof techniques.

Lemma 3.4.3 *Let P and Q be two processes in* $\mathrm{L}\pi$. *Then:*

$$P \approx_{\mathrm{l}} Q \ \text{ iff } \ [\![P]\!] \asymp_{\mathrm{a}} [\![Q]\!].$$

Proof: We give a sketch of the proof. A detailed proof can be found in Appendix A.1.2. In the implication from left to right, we use a weak variant of the operational correspondence between processes P and $[\![P]\!]$ to show that the relation

$$\mathcal{S} = \{([\![P]\!], [\![Q]\!]) \mid P, Q \in \mathrm{L}\pi \ \text{ and } \ P \approx_{\mathrm{l}} Q\}$$

is a $\asymp_{\mathrm{a}}$-bisimulation up-to $\gtrsim$ and $\approx$. In the implication from right to left, we show that the relation

$$\mathcal{S} = \{(P, Q) \mid P, Q \in \mathrm{L}\pi \ \text{ and } \ [\![P]\!] \asymp_{\mathrm{a}} [\![Q]\!]\}$$

is a link bisimulation. $\square$

Finally, we can prove the soundness of the "run-time" proof technique.

Theorem 3.4.4 (Soundness) *Let P and Q be two processes in* $\mathrm{L}\pi$*. Then*

$$P \overset{\mapsto}{\approx}_{\mathrm{a}} Q \text{ implies } P \cong_{\mathrm{L}\pi} Q.$$

Proof: By applying, in sequence, Lemma 3.4.2, Lemma 3.4.3, and Theorem 3.3.13. □

Both the "translation-based" and the "run-time" proof techniques are based on the use of links. In the former, links are added statically via an encoding (at "compile-time"); in the latter, they are added dynamically in the bisimulation game (at "run-time"). The advantages of the latter proof-technique are that: (i) it uses simpler links $p \triangleright b$ instead of links $p \to b$; (ii) links are not added in case of internal communications; (iii) in the input clause, it uses the particle $\overline{a}b$ instead of $[\![\overline{a}b]\!]$ (that produces links).

3.5 Two characterisations of barbed congruence

In this section we show that the translation-based and the run-time proof techniques mentioned above are not only sound but also complete, that is, they completely characterise barbed congruence in $\mathrm{L}\pi$. As usual when proving labelled characterisations of barbed congruence [4, 72], we prove the completeness for image-finite processes (see Definition 2.3.6). The challenge here is that, unlike similar results in the literature, our calculus does not provide any construct for testing equality between names (such as matching).

Lemma 3.5.1 *Let P and Q be two image-finite* $\mathrm{L}\pi$*-processes. Then*

$$P \cong_{\mathrm{L}\pi} Q \text{ implies } P \approx_{\mathrm{l}} Q.$$

Proof: Let $\simeq_{\mathrm{l}}$ be the variant of $\approx_{\mathrm{l}}$ obtained by replacing $\xrightarrow{\mu}$ with $\overset{\mu}{\Longrightarrow}$ in the hypothesis of each clause in Definition 3.3.7. Let $\dot{\simeq}$ be the variant of $\dot{\approx}$ obtained by replacing $\xrightarrow{\tau}$ with $\overset{\tau}{\Longrightarrow}$ and $\downarrow_a$ with $\Downarrow_a$ in the hypothesis of each clause in Definition 2.5.2. It holds that $\simeq_{\mathrm{l}} = \approx_{\mathrm{l}}$ and $\dot{\simeq} = \dot{\approx}$. So, we prove that

$$(\forall\, R \in \mathrm{L}\pi \;\; P \mid R \,\dot{\simeq}\, Q \mid R) \implies P \simeq_{\mathrm{l}} Q.$$

Let $\mathcal{F}$ be the monotone operator over $\mathcal{P}(\mathrm{L}\pi \times \mathrm{L}\pi)$ associated with the definition of $\simeq_{\mathrm{l}}$. Suppose $\simeq_{\mathrm{l}}^{0} = \mathrm{L}\pi \times \mathrm{L}\pi$, $\simeq_{\mathrm{l}}^{k+1} = \mathcal{F}(\simeq_{\mathrm{l}}^{k})$, and $\simeq_{\mathrm{l}}^{\omega} = \bigcap_{k<\omega} \simeq_{\mathrm{l}}^{k}$. On image-finite LTS the operator $\mathcal{F}$ preserves co-directed sets. In particular, $\mathcal{F}(\simeq_{\mathrm{l}}^{\omega}) = \simeq_{\mathrm{l}}^{\omega}$. This means that on image-finite processes $\simeq_{\mathrm{l}} = \simeq_{\mathrm{l}}^{\omega}$. Therefore, we are left with proving:

$$(\forall\, R \in \mathrm{L}\pi \;\; P \mid R \,\dot{\simeq}\, Q \mid R) \text{ implies that } P \simeq_{\mathrm{l}}^{\omega} Q.$$

We define a collection of tests $R(n, \mathcal{L}, \mathcal{M})$ depending on the integer n and the finite sets of channel names $\mathcal{L}$, $\mathcal{M}$. Intuitively, $\mathcal{L}$ contains the free names along which the processes P and Q may perform some observable actions which have to be tested by $R(n, \mathcal{L}, \mathcal{M})$; $\mathcal{M}$ contains the names in $\mathcal{L}$ which cannot be used in input subject position by the test process. We show by induction on n that there exist $\mathcal{L}, \mathcal{M}, \mathcal{L}'$ such that $\mathcal{L} \supseteq \mathrm{fn}(P, Q)$, $\mathcal{L}' \subseteq \mathcal{L}$, $\mathcal{M} \subseteq \mathcal{L}$ and $(\boldsymbol{\nu}\mathcal{L}')(P \mid R(n, \mathcal{L}, \mathcal{M})) \stackrel{\bullet}{\simeq} (\boldsymbol{\nu}\mathcal{L}')(P \mid R(n, \mathcal{L}, \mathcal{M}))$ implies $P \simeq_1^n Q$.

If the property above holds then we can conclude the proof by observing that:

$$
\begin{array}{ll}
\forall\, R \in \mathrm{L}\pi\ (P \mid R \stackrel{\bullet}{\simeq} Q \mid R) & \text{implies } (\boldsymbol{\nu}\mathcal{L}')(P \mid R(n, \mathcal{L}, \mathcal{M})) \stackrel{\bullet}{\simeq} (\boldsymbol{\nu}\mathcal{L}')(Q \mid R(n, \mathcal{L}, \mathcal{M})) \\
& \qquad \text{for each } n \in \omega \text{ with } \mathcal{L} = \mathrm{fn}(P, Q), \\
& \qquad \mathcal{M} = \emptyset, \text{ and } \mathcal{L}' = \emptyset. \\
& \text{implies } \forall\, n \in \omega\ \ P \simeq_1^n Q \\
& \text{implies } P \simeq_1^\omega Q \\
& \text{implies } P \simeq_1 Q.
\end{array}
$$

For defining the tests $R(n, \mathcal{L}, \mathcal{M})$ we introduce an internal choice operator $\oplus$. This is a derived operator defined as follows:

$$
P_1 \oplus \ldots \oplus P_n \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}a)(a.\, P_1 \mid \ldots \mid a.\, P_n \mid \overline{a}) \text{ with } a \notin \mathrm{fn}(P_1, \ldots, P_n)
$$

We suppose that the collection of channel names Ch has been partitioned in two infinite well-ordered sets Ch' and Ch''. In the following we have $\mathcal{L}' \subseteq \mathcal{L}$, $\mathcal{M} \subseteq \mathcal{L}$, $\mathcal{L} \subseteq Ch''$. We also assume the sequences

$$
\{b_n, b'_n : n \in \omega\} \cup \{c_n^\beta : n \in \omega,\ \beta \in \{\tau, a, \overline{a} : a \in Ch''\}\}
$$

of distinct names in Ch'. The test $R(n, \mathcal{L}, \mathcal{M})$ is defined by induction on n as follows, where we pick a' to be the first name in the well-ordered set $Ch'' \setminus \mathcal{L}$. When emitting or receiving a name which is not in $\mathcal{L}$ we work up to injective substitution to show that $P \simeq_1^n Q$. The relation $R(n, \mathcal{L}, \mathcal{M})$ is defined as follows:

$$
R(0, \mathcal{L}, \mathcal{M}) = \overline{b_0} \oplus \overline{b'_0}
$$

$$
\begin{aligned}
R(n, \mathcal{L}) = {} & \overline{b_n} \oplus \overline{b'_n} \text{ (for } n > 0) \\
& \oplus(\overline{c_n^\tau} \oplus R(n-1, \mathcal{L}, \mathcal{M})) \\
& \oplus\{\overline{c_n^{\overline{a}}} \oplus ((\boldsymbol{\nu}a')(\overline{a}a' \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M}))) \mid a \in \mathcal{L}\} \\
& \oplus\{\overline{c_n^a} \oplus a(x).\, (a' \rhd x \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\})) \mid a \in \mathcal{L}, a \notin \mathcal{M}\}.
\end{aligned}
$$

The proof is by induction on n. Details can be found in Appendix A.1.3. □

By Lemma 3.5.1 we can derive the completeness of our two proof techniques.

Theorem 3.5.2 (First characterisation) *Let P and Q be two processes in $\mathrm{L}\pi$. Then:*

1. $P \cong_{\mathrm{L}\pi} Q$ implies $[\![P]\!] \asymp_{\mathrm{a}} [\![Q]\!]$, *for P and Q image-finite;*
2. $[\![P]\!] \asymp_{\mathrm{a}} [\![Q]\!]$ implies $P \cong_{\mathrm{L}\pi} Q$.

Proof: Part 1 follows by applying in sequence Lemmas 3.5.1 and 3.4.3. Part 2 follows from Theorem 3.3.13 □

As already pointed out in the end of Section 3.3, the encoding based on Lemma 3.3.16 can be seen as a simpler proof technique for barbed congruence in $\mathrm{L}\pi$. However, that proof technique is not complete. As a counterexample, take the processes:

$$P = \overline{a}b \text{ and } Q = (\boldsymbol{\nu} c)(\overline{a}c \mid c \to b).$$

Then, $P \cong_{\mathrm{L}\pi} Q$ (see Lemma 4.6.1); while if we replace in P and Q their free outputs with bound outputs, in according to Lemma 3.3.16, the resulting processes are not related by $\asymp_{\mathrm{a}}$ or any variant of it.

Theorem 3.5.3 (Second characterisation) *Let P and Q be two processes in $\mathrm{L}\pi$. Then:*

1. $P \cong_{\mathrm{L}\pi} Q$ implies $P \overset{\mapsto}{\approx}_{\mathrm{a}} Q$, *for P and Q image-finite processes;*
2. $P \overset{\mapsto}{\approx}_{\mathrm{a}} Q$ implies $P \cong_{\mathrm{L}\pi} Q$.

Proof: Part 1 follows by applying in sequence Lemmas 3.5.1 and 3.4.2. Part 2 follows from Theorem 3.4.4. □

Theorem 3.5.3 says that, on image-finite processes, asynchronous ground bisimilarity, defined on the new LTS, coincides with barbed congruence in $\mathrm{L}\pi$. We recall that, in presence of testing equality between names, the closure under contexts of asynchronous ground bisimilarity on the LTS $\xrightarrow{\mu}$ coincides with barbed congruence in π_{a} [4][2]. Therefore, the difference between the two LTSs essentially shows the difference between what is observable in $\mathrm{L}\pi$, and in π_{a} with matching.

Technical differences between the labelled characterisations of barbed congruence in $\mathrm{L}\pi$ and those in π_{a} and π are: (i) the labelled bisimilarities of $\mathrm{L}\pi$ are congruence relations and therefore do not have to be closed under substitutions to obtain barbed congruence; (ii) in $\mathrm{L}\pi$ the labelled bisimilarities are in ground style (without universal quantification on the received names); (iii) the characterisations in $\mathrm{L}\pi$ are proved without the matching construct for testing equality between names (which is essential in the proofs in π_{a} and π).

The absence of matching is crucial in $\mathrm{L}\pi$. Indeed, name-testing, like testing equality between pointers in imperative languages, prevents many useful program optimisations and transformations like the Laws 1.1−1.4 in Section 1.1. Also from a programming point of view the usefulness of matching is questionable. For instance, Join and Pict do not provide any construct for testing channels.

[2]In [4], Amadio, Castellani and Sangiorgi have proved the result for the closure under substitutions of asynchronous early bisimilarity that, as said in Section 2.5.1, coincides with the closure under contexts of asynchronous ground bisimilarity (on image-finite processes).

3.6 Up-to link proof techniques

As for standard bisimilarity, link bisimilarity can be enhanced by means of up to expansion techniques [81].

Definition 3.6.1 (Link bisimilarity up to expansion) *A symmetric relation $\mathcal{S}$ on* Lπ*-processes is a* link bisimulation up to $\gtrsim$ *if whenever $P\ \mathcal{S}\ Q$ the following holds:*

1. *If $P \xrightarrow{\tau} P'$, then $Q \Longrightarrow Q'$ and $P' \gtrsim \mathcal{S} \lesssim Q'$.*
2. *If $P \xrightarrow{a(p)} P'$, and $p \notin \mathrm{fn}(P \mid Q)$, then*
   (a) *either $Q \stackrel{a(p)}{\Longrightarrow} Q'$ and $P' \gtrsim \mathcal{S} \lesssim Q'$*
   (b) *or $Q \Longrightarrow Q'$ and $P' \gtrsim \mathcal{S} \lesssim (Q' \mid \overline{a}p)$.*
3. *If $P \xrightarrow{\overline{a}b} P'$, and $p \notin \mathrm{fn}(P, Q)$, then*
   (a) *either $Q \stackrel{\overline{a}d}{\Longrightarrow} Q'$ and $(p \triangleright b \mid P') \gtrsim \mathcal{S} \lesssim (p \triangleright d \mid Q')$*
   (b) *or $Q \stackrel{\overline{a}(c)}{\Longrightarrow} Q'$ with $c \notin \mathrm{fn}(P)$ and $(p \triangleright b \mid P') \gtrsim \mathcal{S} \lesssim (\boldsymbol{\nu} c)(p \triangleright c \mid Q')$.*
4. *If $P \xrightarrow{\overline{a}(c)} P'$, with $c \notin \mathrm{fn}(Q)$ and $p \notin \mathrm{fn}(P, Q)$, then*
   (a) *either $Q \stackrel{\overline{a}b}{\Longrightarrow} Q'$ and $(\boldsymbol{\nu} c)(p \triangleright c \mid P') \gtrsim \mathcal{S} \lesssim (p \triangleright b \mid Q')$*
   (b) *or $Q \stackrel{\overline{a}(c)}{\Longrightarrow} Q'$ and $(\boldsymbol{\nu} c)(p \triangleright c \mid P') \gtrsim \mathcal{S} \lesssim (\boldsymbol{\nu} c)(p \triangleright c \mid Q')$.*

Lemma 3.6.2 *If $\mathcal{S}$ is a link bisimilarity up to expansion then $\mathcal{S} \subseteq \approx_l$.*

Proof: The proof is analogous to that of standard bisimilarity [81]. Let $\mathcal{S}$ be a link bisimulation up to expansion, we show that the relation $\approx \mathcal{S} \approx$ is a link bisimulation. This follows from the transitivity of $\approx$ and the fact that $\approx$ is a congruence. □

However, link bisimilarity can be strengthened by means of a more powerful up-to proof technique which allows us to reduce the number of links introduced in the derivatives. Roughly speaking, when comparing two processes P and Q this technique permits cutting the *same* links from both processes, or to cut a *private* link from one process only.

Definition 3.6.3 (Link bisimulation up to link) *A symmetric relation $\mathcal{S}$ on* Lπ*-processes is a* link bisimulation up to link *if whenever $P\ \mathcal{S}\ Q$ the following holds:*

1. *If $P \xrightarrow{\tau} P'$ then $Q \Longrightarrow Q'$ and $P'\ \mathcal{S}\ Q'$.*
2. *If $P \xrightarrow{a(p)} P'$, and $p \notin \mathrm{fn}(P \mid Q)$, then*
   (a) *either $Q \stackrel{a(p)}{\Longrightarrow} Q'$ and $P'\ \mathcal{S}\ Q'$*
   (b) *or $Q \Longrightarrow Q'$ and $P' \mathcal{S}\ (Q' \mid \overline{a}p)$.*

3. *If* $P \xrightarrow{\overline{a}b} P'$, *and* $p \notin \mathrm{fn}(P \mid Q)$, *then*
   (a) *either* $Q \stackrel{\overline{a}b}{\Longrightarrow} Q'$ *and* $P' \mathcal{S} Q'$
   (b) *or* $Q \stackrel{\overline{a}d}{\Longrightarrow} Q'$, *with* $d \neq b$, *and* $(p \triangleright b \mid P') \mathcal{S} (p \triangleright d \mid Q')$
   (c) *or* $Q \stackrel{\overline{a}(c)}{\Longrightarrow} Q'$, *with* $c \notin \mathrm{fn}(P)$, *and*
      i. *either* $(p \triangleright b \mid P') \mathcal{S} Q'\{p/c\}$
      ii. *or* $(p \triangleright b \mid P') \mathcal{S} (\boldsymbol{\nu} c)(p \triangleright c \mid Q')$.
4. *If* $P \xrightarrow{\overline{a}(c)} P'$, *with* $c \notin \mathrm{fn}(Q)$ *and* $p \notin \mathrm{fn}(P \mid Q)$, *then*
   (a) *either* $Q \stackrel{\overline{a}b}{\Longrightarrow} Q'$ *and*
      i. *either* $P'\{p/c\} \mathcal{S} (p \triangleright b \mid Q')$
      ii. *or* $(\boldsymbol{\nu} c)(p \triangleright c \mid P') \mathcal{S} (p \triangleright b \mid Q')$
   (b) *or* $Q \stackrel{\overline{a}(c)}{\Longrightarrow} Q'$ *and*
      i. *either* $P' \mathcal{S} Q'$
      ii. *or* $P'\{p/c\} \mathcal{S} (\boldsymbol{\nu} c)(p \triangleright c \mid Q')$
      iii. *or* $(\boldsymbol{\nu} c)(p \triangleright c \mid P') \mathcal{S} Q'\{p/c\})$.
      iv. *or* $(\boldsymbol{\nu} c)(p \triangleright c \mid P') \mathcal{S} (\boldsymbol{\nu} c)(p \triangleright c \mid Q')$.

Two processes P *and* Q *are link bisimilar up to link, written* $P \approx_{\mathrm{lut}} Q$, *if* $P \mathcal{S} Q$ *for some link bisimulation up to link* $\mathcal{S}$.

The up to link technique is inspired by the well-known up to expansion and up to context techniques [81, 73]. However, it cannot be reduced to them because private links may be cut from only one process and not from both of them as required by the up to context technique.

As usual in up to techniques, we need not always apply a cut for reducing the size of the derivative processes. This explains why we have the subclauses marked by $i. \ldots iv$.

The proof of the completeness of the up to link technique is non trivial.

Theorem 3.6.4 *Let* P *and* Q *be two* $\mathrm{L}\pi$*-processes. Then:*

$$P \approx_{\mathrm{l}} Q \;\text{ iff }\; P \approx_{\mathrm{lut}} Q.$$

Proof: We give a sketch of the proof. Details can be found in Appendix A.1.4. In the implication from left to the right, we prove that the relation

$$\mathcal{S} = \{(P, Q) : P \approx_{\mathrm{l}} Q\}$$

is a link bisimulation up to link. The proof is easy. The only interesting point is when dealing with the clause in which a free output action $\overline{a}b$ is matched by the same free output action. In this case we need a result saying that if

$$p \notin \mathrm{fn}(P, Q) \text{ and } (p \triangleright b \mid P) \approx_{\mathrm{l}} (p \triangleright b \mid Q) \text{ then } P \approx_{\mathrm{l}} Q.$$

In the implication from right to left we prove that the relation

$$\mathcal{S} = \{(P, Q) : P \approx_{\mathrm{lut}} Q\}$$

is a link bisimulation. The proof requires the following results:

1. $P \approx_{\mathrm{lut}} Q$ and $p \notin \mathrm{fn}(P, Q)$ implies $(p \triangleright b \mid P) \approx_{\mathrm{lut}} (p \triangleright b \mid Q)$.
2. $P \approx_{\mathrm{lut}} Q$ and $p \notin \mathrm{fn}(P, Q)$ implies $(\boldsymbol{\nu} c)(p \triangleright c \mid P) \approx_{\mathrm{lut}} (\boldsymbol{\nu} c)(p \triangleright c \mid Q)$.
3. $P\{p/c\} \approx_{\mathrm{lut}} (\boldsymbol{\nu} c)(p \triangleright c \mid Q)$ and $p \notin \mathrm{fn}(P, Q)$ implies $(\boldsymbol{\nu} c)(p \triangleright c \mid P) \approx_{\mathrm{lut}} (\boldsymbol{\nu} c)(p \triangleright c \mid Q)$.
4. $P\{p/c\} \approx_{\mathrm{lut}} (p \triangleright b \mid Q)$, $c \notin \mathrm{fn}(Q)$ implies $(\boldsymbol{\nu} c)(p \triangleright c \mid P) \approx_{\mathrm{lut}} (p \triangleright b \mid Q)$.

$\square$

The up to link technique can be used in combination with up to expansion and up to context techniques [81, 76].

Chapter 4

Applications of the theory of Local π

In this chapter we show a certain number of applications of the theory of $\mathrm{L}\pi$.

4.1 Expressing substitution in Local π

In π-calculus, the process $P\{b/a\}$ can be expressed as $(\boldsymbol{\nu} u)(\overline{u}b \mid u(a).\, P)$. In $\mathrm{L}\pi$, there exists an alternative way to express name substitution which presents a few advantages (see Sections 4.5, 4.7, and 5.2).

Proposition 4.1.1 *Let P be an $\mathrm{L}\pi$-process and a and b two names such that $a \neq b$ and a does not appear free in P in input position. Then:*

$$(\boldsymbol{\nu} a)(P \mid a \triangleright b) \cong_{\mathrm{L}\pi} P\{b/a\}.$$

Proof: We code up, via $[\![\cdot]\!]$ (the encoding in Section 3.3), both members. By Lemma 3.3.3 we get

$$[\![(\boldsymbol{\nu} a)(P \mid a \triangleright b)]\!] = (\boldsymbol{\nu} a)([\![P]\!] \mid a \to b) \gtrsim [\![P]\!]\{b/a\} = [\![P\{b/a\}]\!].$$

Since $\gtrsim$ implies $\approx$, we use Corollary 3.3.14 to conclude. $\square$

The law in Proposition 4.1.1 is valid in $\mathrm{L}\pi$ but not in π_{a} and π-calculus. A similar law, but with the double link $a \triangleright b \mid b \triangleright a$ in place of $a \triangleright b$, is given in [37] for the asynchronous π-calculus.

We exploit Proposition 4.1.1 to give an easy proof that early and ground bisimilarities coincide in $\mathrm{L}\pi$.

Proposition 4.1.2 *Let $\approx_{\mathrm{l}}^{\mathrm{e}}$ be the early variant of the link bisimilarity $\approx_{\mathrm{l}}$. Then:*

$$P \approx_{\mathrm{l}}^{\mathrm{e}} Q \ \text{ iff } \ P \approx_{\mathrm{l}} Q.$$

Proof: The implication from left to right is easy. Indeed, by definition early bisimilarities have an universal quantification on the received names, whereas ground bisimilarities have an existential quantification. Since the set of received names is non-empty, early bisimilarities always imply their ground variant. As regards the implication from right to left, we prove that the relation

$$\mathcal{S} = \{(P, Q) : \ P, Q \in \mathrm{L}\pi, \ P \approx_{\mathrm{l}} Q\}$$

is an early link bisimilarity. The two forms of bisimilarity only differ in the input clause. Suppose $P \xrightarrow{a(x)} P'$. We want to prove that for every b there exists Q' such that $Q \stackrel{a(x)}{\Longrightarrow} Q'$ and $P'\{^b\!/\!_x\} \ \mathcal{S} \ Q'\{^b\!/\!_x\}$. The case $b = x$ is trivial, so we suppose $b \neq x$. Since $P \approx_{\mathrm{l}} Q$, there exists Q' such that $Q \stackrel{a(x)}{\Longrightarrow} Q'$ and $P' \approx_{\mathrm{l}} Q'$. Notice that, since P' and Q' are Lπ-processes, x does not appear free in P' and Q' in input subject position. By Lemma 3.4.3, Lemma 3.3.8 and Lemma 3.3.12 link bisimilarity is an equivalence relation and it is preserved by all operators of the calculus. So, $(\boldsymbol{\nu} x)(P' \mid x \triangleright b) \approx_{\mathrm{l}} (\boldsymbol{\nu} x)(Q' \mid x \triangleright b)$ for every name b. By Lemma 3.3.3 and Lemma 3.4.3 it holds that $P'\{^b\!/\!_x\} \approx_{\mathrm{l}} (\boldsymbol{\nu} x)(P' \mid x \triangleright b)$ and $Q'\{^b\!/\!_x\} \approx_{\mathrm{l}} (\boldsymbol{\nu} x)(Q' \mid x \triangleright b)$. By transitivity we have that $P'\{^b\!/\!_x\} \approx_{\mathrm{l}} Q'\{^b\!/\!_x\}$ and therefore $P'\{^b\!/\!_x\} \ \mathcal{S} \ Q'\{^b\!/\!_x\}$. $\Box$

4.2 The replication theorems

The *replication theorems* [51] express useful distributivity properties of private replicated processes. The assertions of the theorems can be read thus: A passive resource that is shared among a certain number of clients can be made private to each of them.

Theorem 4.2.1 (Standard replication theorems) *Assume that name a occurs free in processes P, Q and R only in output subject position. Then:*

1. $(\boldsymbol{\nu} a)\big(!a(x).\, R \mid P \mid Q\big) \approx (\boldsymbol{\nu} x)(!a(x).\, R \mid P) \mid (\boldsymbol{\nu} a)(!a(x).\, R \mid Q)$.
2. $(\boldsymbol{\nu} a)(!a(x).\, R \mid !P) \approx \, !(\boldsymbol{\nu} a)(!a(x).\, R \mid P)$.

The side condition in the theorems prevents the restricted name a from being exported. As a consequence, the theorem cannot be used in situations where the set of clients of the resource $a(x).\, R$ may change dynamically. To see why this side condition is necessary, take:

$$\begin{array}{lcl} P_1 & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} a)(!a(x).\, R \mid \overline{b}a \mid Q) \\ P_2 & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} a)(!a(x).\, R \mid \overline{b}a) \mid (\boldsymbol{\nu} a)(!a(x).\, R \mid Q) \end{array}$$

These processes are in general not equivalent in π-calculus. Intuitively, the environment external to P_1 can receive a along b and then use it in input position to interfere with an attempt by Q to activate a copy of R. This is not possible in P_2, where Q has its own private access to R. The difference between P_1 and P_2 can be observed in a context that receives a and then uses it in input; in this way, the context may steal messages that were supposed to reach the resource.

Pierce and Sangiorgi [67] have shown that the side condition can be relaxed using the type system with input/output capabilities, and requiring that the processes R, P and Q only possess the output capability on a. The same result is proved in [78] by previously translating processes (by means of an encoding very close to our $[\![\cdot]\!]$) and then proving that the image are bisimilar. In both cases the stronger replication theorems are shown valid with respect to (typed) barbed congruence by proving a few barbed bisimilarities. Here we propose easier proofs of stronger replication theorems without using typed bisimulations. While [67, 78] works on the π-calculus our results only apply to $\mathrm{L}\pi$.

Theorem 4.2.2 (stronger replication theorems) *Let P, Q, and R be processes in $\mathrm{L}\pi$ not containing name a in input position. Then:*

1. $(\boldsymbol{\nu}a)\big(!a(x).\,R \mid P \mid Q\big) \cong_{\mathrm{L}\pi} (\boldsymbol{\nu}x)(!a(x).\,R \mid P) \mid (\boldsymbol{\nu}a)(!a(x).\,R \mid Q)$;
2. $(\boldsymbol{\nu}a)(!a(x).\,R \mid !P) \cong_{\mathrm{L}\pi} !(\boldsymbol{\nu}a)(!a(x).\,R \mid P)$.

Proof: We code up, via $[\![\cdot]\!]$, both members of each equation. By definition of $[\![\cdot]\!]$ name a may appear free in the translated terms only in output subject position. Therefore, the translated terms are under the hypotheses of the Theorem 4.2.1 and we can assert that the images (of each equation) are synchronous ground bisimilar. Since $\approx$ implies $\asymp_{\mathrm{a}}$, by theorem 3.3.13 we can conclude. $\Box$

The replication theorems have been proved in [51, 67] with respect to a *strong* form of bisimilarity which is sensitive to τ-actions. This strong form of bisimilarity implies $\approx$. On the other hand, Theorem 4.2.2 is proved with respect to a weak version of barbed congruence. Our proof cannot be adapted to strong barbed congruence because we rely on the encoding $[\![\cdot]\!]$ which does not preserve the number of τ-actions performed by a process.

4.3 The delayed input

In an asynchronous calculus message emission is non-blocking. Milner, Parrow, Victor and others have advocated also non-blocking message reception (which is among the motivations for the Update and the Fusion calculi [64, 65] and, implicitly for the Chi calculus [28]). Such a *delayed input prefix*, written $a(x)P$, should allow the continuation P to evolve underneath the input guard, except for observable actions along x. The delayed input replaces *temporal precedencies*, imposed by plain input, with *causal dependencies*. This appears, for instance, in Abramsky's representation of Linear Logic proofs as π-calculus processes [2, 7]. Non-blocking message reception has been considered by Bellin and Scott [7], Boudol [14], Fu [28], Parrow and Victor [65], Yoshida [92] and van Breugel [85]. Bellin and Scott give a reduction semantics for a version of π-calculus, proposed by Milner, where both message emission and message reception are non-blocking [52]; van Breugel defines a labelled transition system for such calculus and proves a correspondence with the Bellin and Scott's reduction semantics.

$$\texttt{d-in:}\ \frac{}{a(b)P \xrightarrow{a(b)} P} \qquad \texttt{s-com:}\ \frac{P \xrightarrow{\overline{a}c} P'}{a(b)P \xrightarrow{\tau} (\boldsymbol{\nu} b)(P'\{c/b\})}$$

$$\texttt{p-in:}\ \frac{P \xrightarrow{\mu} P' \quad b \notin \mathrm{n}(\mu) \quad a \notin \mathrm{bn}(\mu)}{a(b)P \xrightarrow{\mu} a(b)P'} \qquad \texttt{s-cls:}\ \frac{P \xrightarrow{\beta_c \overline{a}c} P' \quad b \notin \mathrm{n}(\beta_c \overline{a}c)}{a(b)P \xrightarrow{\tau} \beta_c(P'\{c/b\})}$$

$$\texttt{o-}\boldsymbol{\nu}\texttt{:}\ \frac{P \xrightarrow{\mu} P' \quad [\mu = \overline{a}c \vee \mu = c(b)\overline{a}b] \quad a \neq c}{(\boldsymbol{\nu} c)P \xrightarrow{(\boldsymbol{\nu} c)\mu} P'} \qquad \texttt{o-in:}\ \frac{P \xrightarrow{\overline{a}b} P' \qquad b \neq a}{c(b)P \xrightarrow{c(b)\overline{a}b} P'}$$

$$\texttt{cls:}\ \frac{P \xrightarrow{\beta_c \overline{a}c} P' \quad Q \xrightarrow{a(b)} Q' \quad \mathrm{bn}(\beta_c) \cap \mathrm{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\tau} \beta_c(P' \mid Q'\{c/b\})}$$

Table 4.1: Inference rules for delayed input and restriction.

Let $\mathrm{DL}\pi$ be the calculus obtained by adding the delayed input construct $a(b)P$ to the grammar of $\mathrm{L}\pi$ (with the same constraint as plain input that b may not appear free in P in input position). Actions are extended as follows:

$$\mu ::= \tau \mid a(b) \mid \overline{a}b \mid \beta_b \overline{a}b$$

where

$$\beta_b ::= (\boldsymbol{\nu} b) \mid c(b) \mid (\boldsymbol{\nu} c)c(b) \ \text{ for some } c.$$

β_b represents the binding part of bound output actions. We extend free and bound names as follows: $\mathrm{fn}((\boldsymbol{\nu} b)\overline{a}b) = \{a\}$, $\mathrm{bn}((\boldsymbol{\nu} b)\overline{a}b) = \{b\}$, $\mathrm{fn}(c(b)\overline{a}b) = \{c, a\}$, $\mathrm{bn}(c(b)\overline{a}b) = \{b\}$, $\mathrm{fn}((\boldsymbol{\nu} c)c(b)\overline{a}b) = \{a\}$, $\mathrm{bn}((\boldsymbol{\nu} c)c(b)\overline{a}b) = \{c, b\}$. In Table 4.1, we give the transition rules of $\mathrm{DL}\pi$. More precisely, we enrich the rules in Table 2.2 as follows: (i) rule `close` is split into rules `cls` and `s-cls`; (ii) rule `open` is split into rules `o-in` and `o-`$\boldsymbol{\nu}$; (iii) the rules `d-inp`, `s-com`, and `p-inp` are similar to the rules `inp`, `com`, and `par`, but involve the delayed input. Our labelled transition system has two main differences with respect to that of van Breugel's [85]: (i) actions have a simpler syntax, because only the output capability of name may be transmitted; (ii) a restriction $(\boldsymbol{\nu} b)$ is added in rule `s-com` to model *self communications*, as in $a(b)(\overline{a}b \mid P) \xrightarrow{\tau} (\boldsymbol{\nu} b)P$.

We prove that the delayed input is a derived operator in $\mathrm{L}\pi$. In Table 4.2 we give an encoding $\{\!|\cdot|\!\}$ of $\mathrm{DL}\pi$ into $\mathrm{L}\pi$ and prove that it is fully-abstract with respect to barbed congruence. The encoding $\{\!|\cdot|\!\}$ is an homomorphism on all operators except delayed input. (A similar encoding, but with the double link $b \triangleright b' \mid b' \triangleright b$ in place of $b \triangleright b'$, has been suggested by Yoshida in [92] for the asynchronous π-calculus.)

$$\{\!|\, P_1 \mid P_2 \,|\!\} \stackrel{\text{def}}{=} \{\!|\, P_1 \,|\!\} \mid \{\!|\, P_2 \,|\!\} \qquad \{\!|\, (\boldsymbol{\nu} a)P \,|\!\} \stackrel{\text{def}}{=} (\boldsymbol{\nu} a)\{\!|\, P \,|\!\} \qquad \{\!|\, \overline{a}b \,|\!\} \stackrel{\text{def}}{=} \overline{a}b$$

$$\{\!|\, a(b).\, P \,|\!\} \stackrel{\text{def}}{=} a(b).\, \{\!|\, P \,|\!\} \qquad \{\!|\, !a(b).\, P \,|\!\} \stackrel{\text{def}}{=} !a(b).\, \{\!|\, P \,|\!\} \qquad \{\!|\, \mathbf{0} \,|\!\} \stackrel{\text{def}}{=} \mathbf{0}$$

$$\{\!|\, a(b)P \,|\!\} \stackrel{\text{def}}{=} (\boldsymbol{\nu} b)(a(b').\, b \rhd b' \mid \{\!|\, P \,|\!\})$$

Table 4.2: The encoding $\{\!|\cdot|\!\}$

In order to prove the full abstraction of $\{\!|\cdot|\!\}$, we first prove an adequacy result with respect to barbed bisimulation. The proof of this adequacy result requires an operational correspondence between processes P and $\{\!|\, P \,|\!\}$, up to some notion of expansion. As we will argue in Remark 4.3.7, such an operational correspondence is not easy to prove. Thus, for convenience, we define an auxiliary encoding $\{\![\cdot]\!\}$ as the composition of the encodings $\{\!|\cdot|\!\}$ and $[\![\cdot]\!]$ (of Section 3.3). Precisely, if P is a $\mathrm{DL}\pi$-process

$$\{\![P]\!\} \stackrel{\text{def}}{=} [\![\{\!|\, P \,|\!\}]\!].$$

We prove that the encoding $\{\![\cdot]\!\}$ enjoys an operational correspondence up to $\succcurlyeq_{\mathrm{a}}$, where $\preccurlyeq_{\mathrm{a}}$ denotes the expansion variant of $\precsim_{\mathrm{a}}$ (Definition 3.3.7) in the same way as $\lesssim_{\mathrm{a}}$ denotes the expansion variant of $\approx_{\mathrm{a}}$ (see Definition 2.5.5). This operational correspondence, together with Theorem 3.3.4, allows us to prove the soundness of $\{\!|\cdot|\!\}$. Then, by exploiting the inclusion $\mathrm{L}\pi \subset \mathrm{DL}\pi$ we prove the completeness of $\{\!|\cdot|\!\}$.

For proving the operational correspondence of $\{\![\cdot]\!\}$ we need to know that $\preccurlyeq_{\mathrm{a}}$ is a congruence in $\mathcal{L}\pi$ (simply adapt Lemma 3.3.12). We also need the following technical lemma.

Lemma 4.3.1 *Given an $\mathrm{L}\pi$-process P and a name a it holds that*

$$(\boldsymbol{\nu} a)(a \to a \mid [\![P]\!]) \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} a)[\![P]\!].$$

Proof: Because $a \to a \succcurlyeq_{\mathrm{a}} \mathbf{0}$ and $\succcurlyeq_{\mathrm{a}}$ is a congruence. $\square$

Lemma 4.3.2 (Operational correspondence of $\{\![\cdot]\!\}$) *Let P be a process in $\mathrm{DL}\pi$.*

1. *Suppose that $P \xrightarrow{\mu} P'$. Then we have:*
   (a) *if $\mu = a(b)$ then $\{\![P]\!\} \xrightarrow{a(b')} \gtrsim \{\![P']\!\}\{b'/b\}$ and $b' \notin \mathrm{fn}(P)$;*
   (b) *if $\mu = \overline{a}b$ then $\{\![P]\!\} \xrightarrow{(\boldsymbol{\nu} c)\overline{a}c} \succcurlyeq_{\mathrm{a}} (c \to b \mid \{\![P']\!\})$, with $c \notin \mathrm{fn}(P)$;*
   (c) *if $\mu = (\boldsymbol{\nu} b)\overline{a}b$ then $\{\![P]\!\} \xrightarrow{(\boldsymbol{\nu} c)\overline{a}c} \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} b)(c \to b \mid \{\![P']\!\})$, with $c \notin \mathrm{fn}(P)$;*

(d) *if* $\mu = d(b)\overline{a}b$ *then* $\{\!\![P]\!\!\} \xrightarrow{(\boldsymbol{\nu}c)\overline{a}c} \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu}b)(d(b').\, b \to b' \mid c \to b \mid \{\!\![P']\!\!\})$ *with* $\{b', c\} \cap \mathrm{fn}(P) = \emptyset$;

(e) *if* $\mu = (\boldsymbol{\nu}d)d(b)\overline{a}b$ *then* $\{\!\![P]\!\!\} \xrightarrow{(\boldsymbol{\nu}c)\overline{a}c} \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu}b)(\boldsymbol{\nu}d)(d(b').\, b \to b' \mid c \to b \mid \{\!\![P']\!\!\})$, *with* $\{b', c\} \notin fn(P)$;

(f) *if* $\mu = \tau$ *then* $\{\!\![P]\!\!\} \xrightarrow{\tau} \succcurlyeq_{\mathrm{a}} \{\!\![P']\!\!\}$.

2. *Suppose that* $\{\!\![P]\!\!\} \xrightarrow{\mu} P_1$. *Then there exists* $P' \in \mathrm{DL}\pi$ *such that:*

(a) *if* $\mu = a(b')$ *then* $P \xrightarrow{a(b)} P'$ *and* $P_1 \succcurlyeq_{\mathrm{a}} \{\!\![P']\!\!\}\{^{b'}\!/b\}$;

(b) *if* $\mu = (\boldsymbol{\nu}c)\overline{a}c$ *then:*

i. *either* $P \xrightarrow{\overline{a}b} P'$ *and* $P_1 \succcurlyeq_{\mathrm{a}} (c \to b \mid \{\!\![P']\!\!\})$, *with* $c \notin \mathrm{fn}(P)$;

ii. *or* $P \xrightarrow{(\boldsymbol{\nu}b)\overline{a}b} P'$ *and* $P_1 \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu}b)(c \to b \mid \{\!\![P']\!\!\})$, *with* $c \notin \mathrm{fn}(P)$;

iii. *or* $P \xrightarrow{d(b)\overline{a}b} P'$ *and* $P_1 \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu}b)(d(b').\, b \to b' \mid c \to b \mid \{\!\![P']\!\!\})$, *with* $\{b', c\} \cap \mathrm{fn}(P) = \emptyset$;

iv. *or* $P \xrightarrow{(\boldsymbol{\nu}d)d(b)\overline{a}b} P'$ *and* $P_1 \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu}b)(\boldsymbol{\nu}d)(d(b').\, b \to b' \mid c \to b \mid \{\!\![P']\!\!\})$, *with* $\{b', c\} \cap \mathrm{fn}(P)$;

(c) *if* $\mu = \tau$ *then* $P \xrightarrow{\tau} P'$ *with* $P_1 \succcurlyeq_{\mathrm{a}} \{\!\![P']\!\!\}$.

Proof: By transition induction. The proof relies on Proposition 3.2.3(2), Lemma 3.3.3, and Lemma 4.3.1. Details can be found in Appendix A.2. □

From Lemma 4.3.2 we can derive a weak operational correspondence.

Lemma 4.3.3

1. *If* $P \Longrightarrow P'$ *then* $\{\!\![P]\!\!\} \Longrightarrow \succcurlyeq_{\mathrm{a}} \{\!\![P']\!\!\}$.
2. *If* $\{\!\![P]\!\!\} \Longrightarrow P_1$ *then there is* P' *such that* $P \Longrightarrow P'$ *and* $P_1 \succcurlyeq_{\mathrm{a}} \{\!\![P']\!\!\}$.
3. $P \Downarrow_a$ *iff* $\{\!\![P]\!\!\} \Downarrow_a$.

Proof: Parts 1 and 2 are proven by induction on the number of τ-moves by exploiting Lemma 4.3.2. Part 3 is a consequence of parts 1 and 2, and Lemma 4.3.2. □

Lemma 4.3.3 allows us to prove that the encoding $\{\!\![\cdot]\!\!\}$ is adequate with respect to barbed bisimulation.

Lemma 4.3.4 *Let* P *and* Q *be two processes in* $\mathrm{DL}\pi$. *Then:*

$$P \stackrel{\bullet}{\approx} Q \ \text{ iff } \ \{\!\![P]\!\!\} \stackrel{\bullet}{\approx} \{\!\![Q]\!\!\}.$$

Proof: In the implication from left to right, we use Lemma 4.3.3 to prove that the relation $\mathcal{R} = \{(\{\!\![P]\!\!\}, \{\!\![Q]\!\!\}) : P \stackrel{\bullet}{\approx} Q\}$ is a barbed bisimulation up-to $\succcurlyeq_{\mathrm{a}}$. In the implication from right to left we use Lemma 4.3.3 and the fact that $\precsim_{\mathrm{a}} \stackrel{\bullet}{\approx} \succcurlyeq_{\mathrm{a}} \subset \stackrel{\bullet}{\approx}$ to prove that the relation $\mathcal{R} = \{(P, Q) : \{\!\![P]\!\!\} \stackrel{\bullet}{\approx} \{\!\![Q]\!\!\}\}$ is a barbed bisimulation. □

Lemma 4.3.4 allows us to prove that the encoding $\{\!|\cdot|\!\}$ is adequate with respect to barbed bisimulation and therefore sound with respect to barbed congruence.

Lemma 4.3.5 *Let P and Q be two* DLπ*-processes. Then:*

$$P \stackrel{\bullet}{\approx} Q \text{ iff } \{\!| P |\!\} \stackrel{\bullet}{\approx} \{\!| Q |\!\}.$$

Proof: By the definition of $\{\![\![\cdot]\!]\!\}$, Lemma 4.3.4, and Theorem 3.3.4. □

Notice that if $P \in \mathrm{L}\pi$ then $\{\!| P |\!\} = P$. This will allow us to prove the completeness of $\{\!|\cdot|\!\}$.

Theorem 4.3.6 (Full abstraction of $\{\!|\cdot|\!\}$) *Let P and Q be two processes in* DLπ. *Then:*

$$P \cong_{\mathrm{DL}\pi} Q \text{ iff } \{\!| P |\!\} \cong_{\mathrm{L}\pi} \{\!| Q |\!\}.$$

Proof: The soundness follows from Lemma 4.3.5 and the compositionality of $\{\!|\cdot|\!\}$. As regards the completeness, we want to prove that for every Lπ-context $C[\cdot]$ it holds that $C[\{\!| P |\!\}] \stackrel{\bullet}{\approx} C[\{\!| Q |\!\}]$. Let $C[\cdot]$ be a such a context. By the compositionality of $\{\!|\cdot|\!\}$ and the fact that $\{\!|\cdot|\!\}$ does not affect Lπ-processes, it follows that $\{\!| C[P] |\!\} = C[\{\!| P |\!\}]$ and $\{\!| C[Q] |\!\} = C[\{\!| Q |\!\}]$. Since an L$\pi$-context is also a DL$\pi$-context, by hypothesis we have $C[P] \stackrel{\bullet}{\approx} C[Q]$. By Lemma 4.3.5 this implies $\{\!| C[P] |\!\} \stackrel{\bullet}{\approx} \{\!| C[Q] |\!\}$. So, we conclude that $C[\{\!| P |\!\}] \stackrel{\bullet}{\approx} C[\{\!| Q |\!\}]$. □

Remark 4.3.7 *In order to clarify the usefulness of the auxiliary encoding $\{\![\![\cdot]\!]\!\}$, notice that if one wanted to prove the operational correspondence of $\{\!|\cdot|\!\}$ (instead of $\{\![\![\cdot]\!]\!\}$), then Proposition 3.2.3(2) and Lemma 4.3.1 would not be necessary anymore. By contrast, an expansion variant of Proposition 4.1.1 would be necessary. The proof of a such a result would require an expansion variant of Theorem 3.3.13 (or Corollary 3.3.14), which, as already pointed out in Remark 3.3.15, does not hold.*

Using $\{\!|\cdot|\!\}$ and the theory of Lπ we can prove laws for delayed input like:

$$\begin{aligned}
a(b)(P \mid Q) &= (a(b)P) \mid Q && \text{if } b \notin \mathrm{fn}(Q) && (4.1)\\
a(b)c(d)P &= c(d)a(b)P && \text{if } c \neq b \text{ and } d \neq a && (4.2)\\
(\boldsymbol{\nu} a)(a(x)(\overline{a}x \mid P)) &= (\boldsymbol{\nu} x)P && \text{if } a \notin \mathrm{fn}(P) && (4.3)
\end{aligned}$$

Laws 4.1 and 4.2 are similar to structural rules for restriction. Similar laws have been proposed in [14]. Law 4.3 transforms a delayed input binder into a restriction binder (it might be interesting to examine delayed input from within action calculi [54]; for instance, law 4.3 is reminiscent of the definition of restriction in reflexive action calculi [55]).

4.4 Encodings of the λ-calculus

In this example, we use polyadicity, which is straightforward to accommodate in the theory of Lπ developed in Chapter 3; we remind that we write $\overline{a}\langle b_1 \dots b_n\rangle$ for polyadic outputs and $a(x_1, \dots x_n).\, P$ for polyadic inputs. Below, we give Milner's encoding of call-by-name λ-calculus into π-calculus (more precisely, the variant in [63]).

$$\begin{aligned}
(\!|\ \lambda x.\, M\ |\!)_p &\stackrel{\text{def}}{=} (\boldsymbol{\nu} v)(\overline{p}\langle v\rangle \mid v(x,q).\, (\!|\ M\ |\!)_q) \\
(\!|\ x\ |\!)_p &\stackrel{\text{def}}{=} \overline{x}\langle p\rangle \\
(\!|\ MN\ |\!)_p &\stackrel{\text{def}}{=} (\boldsymbol{\nu} q)\Big((\!|\ M\ |\!)_q \mid q(v).\, (\boldsymbol{\nu} x)(\overline{v}\langle x,p\rangle \mid !x(r).\, (\!|\ N\ |\!)_r)\Big)
\end{aligned}$$

This is also an encoding into (polyadic) Lπ. By applying Proposition 4.1.1, we can prove the following optimisation of the definition of application in the case when the argument is a variable (a tail-call-like optimisation):

$$(\!|\ My\ |\!)_p \stackrel{\text{def}}{=} (\boldsymbol{\nu} q)\Big((\!|\ M\ |\!)_q \mid q(v).\, \overline{v}\langle y,p\rangle\Big)$$

We can also exploit the delayed input operator, that is a derived operator in Lπ, to get an encoding of the *strong* call-by-name strategy, where reductions can also occur underneath an abstraction (i.e., the `Xi` rule, saying that if $M \longrightarrow M'$ then $\lambda x.\, M \longrightarrow \lambda x.\, M'$, is allowed). For this, we have to relax, in the translation of $\lambda x.\, M$, the sequentiality imposed by the input prefix $v(x,q)$ that guards the body $(\!|\ M\ |\!)_q$ of the function. Precisely, we have to replace this input with a delayed input:

$$(\!|\ \lambda x.\, M\ |\!)_p \stackrel{\text{def}}{=} (\boldsymbol{\nu} v)(\overline{p}\langle v\rangle \mid v(x,q)(\!|\ M\ |\!)_q) \tag{4.4}$$

Using the above encoding of delayed input, we get:

$$(\!|\ \lambda x.\, M\ |\!)_p \stackrel{\text{def}}{=} (\boldsymbol{\nu} v,x,q)\Big(\overline{p}\langle v\rangle \mid v(y,r).\, (x \triangleright y \mid q \triangleright r) \mid (\!|\ M\ |\!)_q\Big)$$

Results of operational correspondence and validity of β-reduction, similar to those in [53, 73] for the call-by-name λ-calculus, can be easily proved for this encoding. (The modelling of strong reductions is a major motivation behind Fusion and χ; indeed both calculi allow us to encode strong call-by-name λ-calculus [65, 28].)

4.5 Some properties of the Join calculus

We apply the theory of Lπ to prove some laws in the Join calculus [24], a calculus for distributed and concurrent programming (c.f. Section 2.5.3).

In order to compare the expressivity of Join and π-calculus, Fournet and Gonthier give the encoding in Table 4.3 of the Join calculus into the π-calculus. This encoding, as

$$\langle\!| \, a\langle b\rangle \, |\!\rangle \overset{\text{def}}{=} \overline{a}b \qquad \langle\!| \, P \mid Q \, |\!\rangle \overset{\text{def}}{=} \langle\!| \, P \, |\!\rangle \mid \langle\!| \, Q \, |\!\rangle$$

$$\langle\!| \, \mathtt{def}\ a\langle x\rangle \mid b\langle y\rangle = P_1 \ \mathtt{in}\ P_2 \, |\!\rangle \overset{\text{def}}{=} (\boldsymbol{\nu} a, b)(!a(x).\, b(y).\, \langle\!| \, P_1 \, |\!\rangle \mid \langle\!| \, P_2 \, |\!\rangle)$$

Table 4.3: Mapping of the Join calculus into $\mathrm{L}\pi$

---

an encoding of Join into π_{a} or π-calculus, is not fully-abstract. To obtain full abstraction, Fournet and Gonthier have to add a layer of "firewalls" to the encoding. We conjecture that the above encoding is fully-abstract with respect to barbed congruence as an encoding of Join into $\mathrm{L}\pi$ (a similar conjecture is made by Fournet and Gonthier [24]). It is easy to prove soundness, and this suffices for using the encoding and the theory of $\mathrm{L}\pi$ for proving properties of Join processes.

Theorem 4.5.1 (soundness of $\langle\!| \cdot |\!\rangle$) *Let P and Q be two processes of core Join. Then:*

$$\langle\!| \, P \, |\!\rangle \cong_{\mathrm{L}\pi} \langle\!| \, Q \, |\!\rangle \ \text{ implies } \ P \cong_{\mathrm{J}} Q$$

where $\cong_{\mathrm{J}}$ is barbed congruence in core Join.

Proof: The proof follows from the compositionality of the encoding and the operational correspondence between a Join process P and its encoding $\langle\!| \, P \, |\!\rangle$. Such an operational correspondence has already been proved in [26]. $\square$

Using this theorem and the theory of $\mathrm{L}\pi$ we can prove the following laws for the Join calculus:

$$\begin{aligned}
\mathtt{def}\ a\langle x\rangle = R\ \mathtt{in}\ P \mid Q &\cong_{\mathrm{J}} (\mathtt{def}\ a\langle x\rangle = R\ \mathtt{in}\ P) \mid (\mathtt{def}\ a\langle x\rangle = R\ \mathtt{in}\ Q) && (4.5)\\
\mathtt{def}\ a\langle x\rangle = b\langle x\rangle\ \mathtt{in}\ P &\cong_{\mathrm{J}} P\{^b\!/\!_a\} \ \text{ if } \ a \neq b && (4.6)\\
\mathtt{def}\ a\langle x\rangle = P\ \mathtt{in}\ (Q \mid a\langle b\rangle) &\cong_{\mathrm{J}} \mathtt{def}\ a\langle x\rangle = P\ \mathtt{in}\ (Q \mid P\{^b\!/\!_x\}) && (4.7)\\
\mathtt{def}\ a\langle x\rangle = P\ \mathtt{in}\ C[a\langle b\rangle] &\cong_{\mathrm{J}} \mathtt{def}\ a\langle x\rangle = P\ \mathtt{in}\ C[P\{^b\!/\!_x\}] && (4.8)
\end{aligned}$$

if context $C[\cdot]$ does not capture name a.

Law 4.5 is the Join calculus version of the replication theorem for parallel composition; it is proved by applying Theorem 4.2.2(1). Law 4.6 is the Join calculus version of Proposition 4.1.1. Law 4.7 shows a sort of insensitiveness to τ-actions; it is proved in three steps: (i) we apply Theorem 4.2.2(1), (ii) we use the property $(\boldsymbol{\nu} a)(!a(x).\, P \mid \overline{a}b) \approx (\boldsymbol{\nu} a)(!a(x).\, P \mid P\{^b\!/\!_x\})$, (iii) we apply again Theorem 4.2.2(1). Law 4.8 is a generalisation of Law 4.7 and it reminds us of *inline expansion*, an optimisation technique for functional languages which replaces a function call with a copy of the function body. The proof of Law 4.8 is similar to that of Law 4.7 where all replication theorems (also w.r.t. restriction and input prefix [51]), and not only Theorem 4.2.2(1), are exploited. Note that none of these laws can be proved by using the encoding $\langle\!|\cdot|\!\rangle$ and the theory of $\pi_{\rm a}$ or π-calculus. Indeed, if the local name a is exported, the encodings of the processes in the laws can be distinguished both in $\pi_{\rm a}$ and π-calculus. In [11], a labelled bisimilarity for the Join is introduced in which the labels of matching transitions must be syntactically the same. As a consequence, this labelled bisimilarity cannot be used to prove the laws above.

4.6 External versus internal mobility

In [10], Boreale gives an encoding of asynchronous π-calculus into πI. Boreale's encoding is composed by two encodings. The former maps the asynchronous π-calculus into Lπ; the latter (essentially the encoding $[\![\cdot]\!]$ of Section 3.3) maps Lπ into πI. Boreale shows that the whole encoding is adequate with respect to barbed bisimulation by proving the adequacy of the two encodings separately. This result is not quite satisfactory because barbed bisimulation is an extremely coarse relation, too coarse to be considered, per se, as an important behavioural equivalence (for instance, it is not even preserved by parallel composition). Boreale leaves as an open problem whether the encoding is fully-abstract for some finer behavioural equivalence. Here, we show that the encoding of Lπ into πI (that is $[\![\cdot]\!]$) is not fully-abstract. As a counterexample, take

$$P = a(x).\, \overline{a}x \;\text{ and }\; Q = \mathbf{0};$$

then, by applying Theorem 3.3.13, we can show that $P \cong_{\mathrm{L}\pi} Q$. Since $[\![P]\!] \not\approx [\![Q]\!]$, and in πI the relation $\approx$ coincides with barbed congruence (on image-finite processes [77]), it holds that $[\![P]\!] \not\cong_{\pi\mathrm{I}} [\![Q]\!]$. This is not surprising because the source language is asynchronous while the target language is synchronous. However, also if we considered as target language the asynchronous variant of πI, that is $\pi\mathrm{I}_{\rm a}$ (see Section 2.5.2), the encoding would not be fully-abstract. As a counterexample, take the same processes P and Q above. Let

$$R \stackrel{\text{def}}{=} (\boldsymbol{\nu} p)(\overline{a}p \mid a(y).\, (y(z).\, (\boldsymbol{\nu} q)\overline{z}q \mid (\boldsymbol{\nu} r)(\overline{y}r \mid r(x).\, (\boldsymbol{\nu} u)\overline{b}u)))$$

be a $\pi\mathrm{I}_{\rm a}$-process. Since $[\![P]\!] = a(x).\, (\boldsymbol{\nu} d)(\overline{a}d \mid d \to x)$, the process $[\![P]\!] \mid R$ would evolves after four τ-steps into the process A where:

$$A \stackrel{\text{def}}{=} (\boldsymbol{\nu} pdr)(d \to p \mid (\boldsymbol{\nu} h)(\overline{p}h \mid h \to r) \mid d(z).\, (\boldsymbol{\nu} q)\overline{z}q \mid r(x).\, (\boldsymbol{\nu} u)\overline{b}u).$$

It holds that $A \not\Downarrow_b$, while, for all processes B such that $[\![Q]\!] \mid R \Longrightarrow B$, it holds that $B \Downarrow_b$. We conclude that $[\![P]\!] \not\cong_{\pi\mathrm{I}_{\rm a}} [\![Q]\!]$.

In the remainder of this section we prove that the encoding $[\![\cdot]\!]$ *is* fully-abstract when choosing as target calculus the asynchronous variant πI where only output capability of names may be transmitted. We call *Local* πI, in short LπI, the calculus denoted by the following grammar:

$$P ::= \mathbf{0} \;\Big|\; a(b).\, P \;\Big|\; (\boldsymbol{\nu} b)(\overline{a}b \mid P) \;\Big|\; P \mid P \;\Big|\; (\boldsymbol{\nu} a)P \;\Big|\; A\langle a_1, \ldots, a_n\rangle$$

where in processes of the form $a(b).\,P$ and $!a(b).\,P$ name b may not occur free in P in input position. In LπI instead of replication we have recursive definitions. This because, when only internal mobility is allowed, recursion is strictly more expressive than replication [77]. Constant definitions are defined as usual with the further constraint that in a constant definition $A \stackrel{\text{def}}{=} (\tilde{x})P$ names $\tilde{x}$ may not appear in input subject position in P.

The lemma below will be crucial for proving the desired full abstraction result. We recall that $\approx_1$ denotes the link bisimilarity of Definition 3.4.1.

Lemma 4.6.1 *Let P be a process in* Lπ*. Then $P \approx_1 [\![P]\!]$.*

Proof: We prove that the relation

$$\mathcal{S} = \{(P, [\![P]\!]) : P \in \mathrm{L}\pi\}$$

is a link bisimulation up to expansion. Let us consider the possible actions of P (we reason in a similar manner when dealing with the possible actions of $[\![P]\!]$).

1. If $P \xrightarrow{\tau} P'$, by Lemma 3.3.2, there exists P_1 such that $[\![P]\!] \xrightarrow{\tau} P_1$ and $P_1 \gtrsim [\![P']\!]$. So, $P' \gtrsim P' \,\mathcal{S}\, [\![P']\!] \lesssim P_1$.
2. If $P \xrightarrow{a(b)} P'$ we reason as in the previous case.
3. If $P \xrightarrow{\overline{a}b} P'$, by Lemma 3.3.2, P_1 exists such that $[\![P]\!] \xrightarrow{\overline{a}(c)} P_1$ and $P_1 \gtrsim (c \to b \mid [\![P']\!])$, with $c \notin \mathrm{fn}(P')$. Since $\gtrsim$ is a congruence, it holds that

   $$(\boldsymbol{\nu}c)(p \rhd c \mid P_1) \gtrsim (\boldsymbol{\nu}c)(p \rhd c \mid c \to b \mid [\![P']\!])$$

   for p fresh. By Proposition 3.2.3(3) it holds that

   $$p \rhd b \mid P' \;\mathcal{S}\; [\![p \rhd b \mid P']\!] = p \to b \mid [\![P']\!] \lesssim (\boldsymbol{\nu}c)(p \rhd c \mid P_1).$$

4. If $P \xrightarrow{\overline{a}(b)} P'$, by Lemma 3.3.2, P_1 exists s.t. $[\![P]\!] \xrightarrow{\overline{a}(c)} P_1 \gtrsim (\boldsymbol{\nu}b)(c \to b \mid [\![P']\!])$ with $c \notin \mathrm{fn}(P')$. By α-conversion there exists P_2 such that

   $$[\![P]\!] \xrightarrow{\overline{a}(b)} P_2 \gtrsim (\boldsymbol{\nu}c)(b \to c \mid [\![P']\!]\{c/b\}).$$

   This implies that

   $$(\boldsymbol{\nu}b)(p \rhd b \mid P_2) \gtrsim (\boldsymbol{\nu}b)(p \rhd b \mid (\boldsymbol{\nu}c)(b \to c \mid [\![P']\!]\{c/b\})).$$

   By Proposition 3.2.3(3) we have

   $$(\boldsymbol{\nu}b)(p \rhd b \mid (\boldsymbol{\nu}c)(b \to c \mid [\![P']\!]\{c/b\})) \gtrsim (\boldsymbol{\nu}c)(p \to c \mid [\![P']\!]\{c/b\}) = (\boldsymbol{\nu}b)(p \to b \mid [\![P']\!]).$$

   We conclude by observing that

   $$(\boldsymbol{\nu}b)(p \rhd b \mid P') \;\mathcal{S}\; (\boldsymbol{\nu}b)(p \to b \mid [\![P']\!]) \lesssim (\boldsymbol{\nu}b)(p \rhd b \mid P_2).$$

□

Theorem 4.6.2 (Full abstraction of $[\![\cdot]\!]$) *Let P and Q be two processes in $\mathrm{L}\pi$. Then:*

$$P \cong_{\mathrm{L}\pi} Q \text{ iff } [\![P]\!] \cong_{\mathrm{L}\pi\mathrm{I}} [\![Q]\!].$$

Proof: The implication from right to left follows from Theorem 3.3.4 and the compositionality of $[\![\cdot]\!]$. As regards the implication from left to right, by Lemma 4.6.1 and transitivity of $\cong_{\mathrm{L}\pi}$, it follows that $[\![P]\!] \cong_{\mathrm{L}\pi} [\![Q]\!]$. Up to bisimilarity [82], it holds that $\mathrm{L}\pi\mathrm{I} \subset \mathrm{L}\pi$ and therefore $[\![P]\!] \cong_{\mathrm{L}\pi\mathrm{I}} [\![Q]\!]$. □

4.7 Operational soundness of CPS axioms

In his PhD thesis, Thielecke studies the target calculi of Continuation Passing Style transforms [84]. He introduces a CPS-calculus similar to the intermediate language of Appel's compiler [5]. The CPS-calculus is very simple and low-level: only variables may be passed as arguments, moreover application is like a jump, with variables as argument. The terms of the (recursive) CPS-calculus are given by the following grammar:

$$M ::= a\langle b\rangle \;\Big|\; M\{a\langle b\rangle \Leftarrow M\}$$

The intended meaning is that $a\langle b\rangle$ is a jump to the continuation a with actual parameter b, while $M\{a\langle b\rangle = N\}$ binds the continuation with body N and formal parameter b to a in M. The calculus is recursive because in $M\{a\langle b\rangle \Leftarrow N\}$ the term N may refer to itself under a. For simplicity we consider a monadic variant of the CPS-calculus. The results of this sections can be straightforwardly extended to the polyadic CPS-calculus.

The set of free variables $\mathrm{fv}(M)$ of a CPS term M is defined as follows.

- $\mathrm{fv}(a\langle b\rangle) \stackrel{\text{def}}{=} \{a, b\}$
- $\mathrm{fv}(M\{a\langle b\rangle \Leftarrow N\}) \stackrel{\text{def}}{=} (\mathrm{fv}(M) \setminus \{a\}) \cup (\mathrm{fv}(N) \setminus \{b\})$

The *axiomatic semantics* for the CPS-calculus is defined as the congruence induced by the following four axioms:

(DISTR) $L\{a\langle b\rangle \Leftarrow M\}\{c\langle d\rangle \Leftarrow N\} = L\{c\langle d\rangle \Leftarrow N\}\{a\langle b\rangle \Leftarrow M\{c\langle d\rangle \Leftarrow N\}\}$
such that $a \neq c$ and $a, b \notin \mathrm{fv}(N)$
(GC) $a\langle b\rangle\{c\langle d\rangle \Leftarrow N\} = a\langle b\rangle,\; c \notin \mathrm{fv}(a\langle b\rangle)$
(JMP) $a\langle b\rangle\{a\langle c\rangle \Leftarrow N\} = N\{^b\!/\!_c\}\{a\langle c\rangle \Leftarrow N\}$
(ETA) $M\{a\langle b\rangle \Leftarrow c\langle b\rangle\} = M\{^c\!/\!_a\}$, $a \neq c$.

The (JMP) law is in some sense what drives the computation. By contrast, (GC) and (DISTR) are "structural" laws like those given for $\mathrm{L}\pi$. Most of the axioms above appear in [5]. We write $\mathrm{CPS} \vdash M{=}N$ to denote that the equality $M = N$ can be derived by the above axiomatic semantics.

Remark 4.7.1 *The CPS term $M_1\{a\langle b\rangle \Leftarrow M_2\}$ reminds us the construct* `def` $a\langle x\rangle = M_2$ `in` M_1 *of the Join calculus. Actually, the CPS-calculus can be seen as a confluent subset of the Join calculus*

In the remainder of this section we will prove that the axiomatic semantics defined above is sound with respect to barbed congruence. To our knowledge this is the first proof of this result. Our proof strongly relies on the theory of Lπ.

We give an operational semantics for the CPS-calculus (which is a slight variant of the operational semantics given by Thielecke). It is easy to see that every CPS-term M is in the form $a\langle b\rangle\{a_1\langle b_1\rangle \Leftarrow M_1\}\dots\{a_n\langle b_n\rangle \Leftarrow M_n\}$ for some $n \geq 0$. This allows us to model the behaviour of CPS-terms by means of just one (global) reduction rule:

$$a_i\langle b\rangle\{a_1\langle b_1\rangle \Leftarrow M_1\}\dots\{a_i\langle b_i\rangle \Leftarrow M_i\}\dots\{a_n\langle b_n\rangle \Leftarrow M_n\}$$
$$\rightsquigarrow$$
$$M_i\{{}^{b}\!/\!{}_{b_i}\}\{a_1\langle b_1\rangle \Leftarrow M_1\}\dots\{a_i\langle b_i\rangle \Leftarrow M_i\}\dots\{a_n\langle b_n\rangle \Leftarrow M_1\}$$

where $1 \leq i \leq n$ and $a_j \notin \mathrm{fv}(M_i)$ for $1 \leq j < i$. The rule above is actually, a contextual variant of the (JMP) axiom.

In the CPS-calculus the notion of observability is represented by the "external" jump that a term may perform after some internal jumps. For instance, in a jump of the form $a\langle b\rangle$, we can observe (the occurrence of a jump to) a. More generally, a free variable in the leftmost position can be observed.

Definition 4.7.2 (Observability in CPS-calculus) *Let M be a term of the CPS-calculus and a a name, we write $M \downarrow_a$ if there are names $b, a_1, b_1, \dots, a_n, b_n$, for some integer $n \geq 0$ with $a \neq a_i$ for every $1 \leq i \leq n$, such that $M = a\langle b\rangle\{a_1\langle b_1\rangle \Leftarrow M_1\}\dots\{a_n\langle b_n\rangle \Leftarrow M_n\}$. We write $M \Downarrow_a$ if there exists a CPS-term N such that $M \rightsquigarrow^* N \downarrow_a$, where $\rightsquigarrow^*$ denotes the reflexive and transitive closure of $\rightsquigarrow$.*

We denote with $\cong_{\text{CPS}}$ the barbed congruence on CPS terms. We prove the soundness of the axiomatics semantics with respect to $\cong_{\text{CPS}}$ by exploiting a straightforward encoding of the CPS-calculus into π-calculus already appeared in [84].

- $(\!|\, a\langle b\rangle \,|\!) \stackrel{\text{def}}{=} \overline{a}b$
- $(\!|\, M\{a\langle b\rangle \Leftarrow N\} \,|\!) \stackrel{\text{def}}{=} (\boldsymbol{\nu} a)((\!|\, M \,|\!) \mid !a(b).\,(\!|\, N \,|\!))$.

There is a straightforward operational correspondence between a CPS-term M and its encoding $(\!|\, M \,|\!)$.

Lemma 4.7.3 (Operational correspondence of $(\!|\cdot|\!)$) *Let M be a CPS-term. Then:*

1. *If $M \rightsquigarrow N$ then $(\!|\, M \,|\!) \xrightarrow{\tau} \equiv (\!|\, N \,|\!)$.*
2. *If $(\!|\, M \,|\!) \xrightarrow{\tau} P$ then there is a CPS-term N s.t. $M \rightsquigarrow N$ and $P \equiv (\!|\, N \,|\!)$.*
3. *$M \downarrow_a$ iff $(\!|\, M \,|\!) \downarrow_a$.*
4. *If $M \rightsquigarrow^* N$ then $(\!|\, M \,|\!) \Longrightarrow \equiv (\!|\, N \,|\!)$.*

5. *If* $(\!|\, M \,|\!) \Longrightarrow P$ *then there is a CPS-term* N *s.t.* $M \leadsto^* N$ *and* $P \equiv (\!|\, N \,|\!)$.
6. $M \Downarrow_a$ *iff* $(\!|\, M \,|\!) \Downarrow_a$.

Proof: The proofs of parts 1, 2, and 3 are easy. Parts 4 and 5 are proved by induction on the number of silent moves. Part 6 is a consequence of previous parts. □

Lemma 4.7.4 *Let M and N be two CPS-terms. Then:*

$$(\!|\, M \,|\!) \cong_{\mathrm{L}\pi} (\!|\, N \,|\!) \;\text{ implies }\; M \cong_{\mathrm{CPS}} N.$$

Proof: It follows from the operational correspondence in Lemma 4.7.3 and the compositionality of $(\!|\cdot|\!)$. □

Lemma 4.7.5 *Let M and N be two CPS-terms. Then:*

$$\mathrm{CPS} \vdash M{=}N \;\text{ implies }\; (\!|\, M \,|\!) \cong_{\mathrm{L}\pi} (\!|\, N \,|\!).$$

Proof: Since $\cong_{\mathrm{L}\pi}$ is a congruence, it suffices to prove the soundness of the four axioms (DISTR), (GC), (JMP), and (ETA). More precisely, we prove that if $M = N$ is obtained by applying one of these axioms then $(\!|\, M \,|\!) \cong_{\mathrm{L}\pi} (\!|\, N \,|\!)$. Let us consider the four possible cases.

1. (DISTR). By applying the encoding $(\!|\cdot|\!)$ to the distributive law we get an instance of the replication theorems seen in Section 4.2. So, we prove this case by simply exploiting Theorem 4.2.2 and Milner's replication theorem for restriction and input prefix [51].
2. (GC). This is an easy case of garbage collection which can be proved with standard techniques.
3. (JMP). It follows from the equation

$$(\boldsymbol{\nu} a)(\overline{a}b \mid !a(c).\,P) \approx (\boldsymbol{\nu} a)(P\{^b\!/\!_c\} \mid !a(c).\,P).$$

   The proof of this equation is easy.
4. (ETA). This is an application of Proposition 4.1.1.

□

Note that in the previous proof, laws (DISTR) and (ETA) cannot be proved in π or π_{a}.

Theorem 4.7.6 (Soundness of the axiomatic semantics) *Let M and N be two CPS-terms. Then:*

$$\mathrm{CPS} \vdash M{=}N \;\text{ implies }\; M \cong_{\mathrm{CPS}} N.$$

Proof: By Lemmas 4.7.4 and 4.7.5. □

The completeness of the above axiomatics semantics is left as an open problem.

Chapter 5

Encoding polyadicity in Local π

As usual for name-passing calculi, we have conducted the theoretical developments on Lπ on a monadic setting in which exactly one name is exchanged during each communication. On the other hand, most applications in name-passing calculi use polyadic communications, i.e., communications involving tuples of names (see for instance Section 4.4). So, an interesting issue is to investigate whether monadic and polyadic name-passing calculi have the same expressive power.

In this chapter we show that, under the locality hypothesis on channels, monadic and polyadic π-calculi have the same expressive power. More precisely, we give an encoding $\langle\!\langle\!\llbracket \cdot \rrbracket\!\rangle\!\rangle$ of polyadic Lπ into monadic Lπ, and we prove it *fully-abstract* with respect to *synchronous barbed congruence* (Definition 2.3.3).

We also show how the idea of our encoding can be adapted to name-passing calculi with *non-binding input prefix*, such as the *Fusion calculus* [65].

5.1 The polyadic Local π

Polyadic Lπ, in short L$\widetilde{\pi}$, is the polyadic variant of Lπ where $a(\widetilde{x}).\,P$ denotes polyadic input, and $\overline{a}\langle\widetilde{b}\rangle$ polyadic output. The locality hypothesis on channels is straightforwardly adapted to the polyadic case: in processes $a(\widetilde{x}).\,P$ and $!a(\widetilde{x}).\,P$ names in $\widetilde{x}$ are all distinct and may not occur free in P in input position.

As abbreviations we write (i) $\prod_n P_n$ for the process $P_1 \mid \ldots \mid P_n$, (ii) $(\boldsymbol{\nu}\widetilde{a})P$ for the process $(\boldsymbol{\nu}a_1)\ldots(\boldsymbol{\nu}a_n)P$, and (iii) $\overline{a}b$ for $\overline{a}\langle b\rangle$. The *labelled transition system* is the usual one for polyadic π-calculus. Transitions are of the form $P \xrightarrow{\mu} P'$, where *action* μ can be: τ (interaction); $a(\widetilde{b})$ (input); $(\boldsymbol{\nu}\widetilde{c})\overline{a}\langle\widetilde{b}\rangle$ (output) where $\widetilde{c} \subseteq \widetilde{b}$ and $\overline{a}(\widetilde{b})$ is an abbreviation for $(\boldsymbol{\nu}\widetilde{b})\overline{a}\langle\widetilde{b}\rangle$.

We assume Milner's *sorting system*, under which all processes are well-sorted [51]. Names are partitioned into a collection of *sorts*. A *sorting function* Γ is defined which maps sorts onto sequences of sorts. If a sort $\mathtt{S}$ is mapped onto a sequence of sorts $\widetilde{\mathtt{T}}$, this means that channels in $\mathtt{S}$ can only carry tuples in $\widetilde{\mathtt{T}}$. A sorting system is necessary to prevent arity mismatching in communications, like in $\overline{a}\langle b, c\rangle \mid a(x).\, P$. Substitutions must map names onto names of the same sort.

As already pointed out in Section 3.1, the theory of $\mathrm{L}\pi$ can be generalised to the polyadic case. For instance, the encoding $[\![\cdot]\!]$ of free outputs in terms of bound outputs of Section 3.3, can be generalised to the polyadic case by exploiting the informations given by the sorting. More precisely, let Γ be a sorting function, a a channel, and $\widetilde{b}$ a tuple of channels such that $\overline{a}\langle\widetilde{b}\rangle$ respects the sorting Γ. Furthermore, let $\mathtt{S}{=}\Gamma(a)$, $n{=}|\Gamma(\mathtt{S})|$, $\Gamma(\mathtt{S}){=}\mathtt{T}_1 \ldots \mathtt{T}_n$, and $m_i{=}|\Gamma(\mathtt{T}_i)|$, for $1{\le}i{\le}n$. Then, the encoding of the polyadic output $\overline{a}\langle\widetilde{b}\rangle$ is recursively defined as follows:

$$[\![\overline{a}\langle\widetilde{b}\rangle]\!]_\Gamma \stackrel{\text{def}}{=} ((\boldsymbol{\nu} c_1 \ldots c_n))\,(\overline{a}\langle\widetilde{c}\rangle \mid \prod_{1\le i\le n} !c_i(y_1, \ldots, y_{m_i}).\, [\![\overline{b_i}\langle y_1, \ldots, y_{m_i}\rangle]\!]_\Gamma)$$

Remark 5.1.1 *The recursive process $[\![\overline{a}\langle\widetilde{b}\rangle]\!]_\Gamma$ can be represented in $\mathrm{L}\widetilde{\pi}$ (in terms of replication) up to bisimilarity. The encoding is a generalisation of the encoding showed in Section 3.2 for $\mathrm{L}\pi$.*

Given two tuples of names $\widetilde{b} = (b_1, \ldots, b_n)$ and $\widetilde{c} = (c_1, \ldots, c_n)$, where $\Gamma(b_i) = \Gamma(c_i)$ for $1{\le}i{\le}n$, we denote with $\widetilde{c} \triangleright \widetilde{b}$ the process $\prod_{1\le i\le n} !c_j(\widetilde{x}).\, \overline{b_j}\langle\widetilde{x}\rangle$. We also denote with $\widetilde{c} \to \widetilde{b}$ the process $[\![\widetilde{c} \triangleright \widetilde{b}]\!]_\Gamma$.

All the results concerning the encoding $[\![\cdot]\!]$ of Section 3.3 can be straightforwardly generalised to $[\![\cdot]\!]_\Gamma$.

Theorem 5.1.2 (Boreale [10]) *Let P and Q be two well-sorted processes in $\mathrm{L}\widetilde{\pi}$ with sorting Γ. Then:*

$$P \stackrel{\bullet}{\approx} Q \ \text{ iff } \ [\![P]\!]_\Gamma \stackrel{\bullet}{\approx} [\![Q]\!]_\Gamma.$$

In this section we focus on synchronous barbed congruence. Below, we give a polyadic synchronous variant of Theorem 3.5.2.

Theorem 5.1.3 *Let P and Q be two well-sorted $\mathrm{L}\widetilde{\pi}$-processes with sorting Γ. Then:*

1. $P \cong^{\mathrm{s}}_{\mathrm{L}\widetilde{\pi}} Q$ implies $[\![P]\!]_\Gamma \approx [\![Q]\!]_\Gamma$, *for P and Q image-finite processes;*
2. $[\![P]\!]_\Gamma \approx [\![Q]\!]_\Gamma$ implies $P \cong^{\mathrm{s}}_{\mathrm{L}\widetilde{\pi}} Q$.

5.2 Encoding polyadicity

We present the encoding of polyadic $\mathrm{L}\pi$ into monadic $\mathrm{L}\pi$ by comparison with Milner's encoding of polyadic synchronous π-calculus into monadic synchronous π-calculus [51]. Milner's idea is quite simple: The communication of a tuple $\widetilde{b}$ is emulated by means of the communication of a fresh channel w along which all names b_i are transmitted sequentially. More precisely, Milner gives an encoding $\{\!|\cdot|\!\}$ from polyadic processes to monadic ones which is an homomorphism on all operators except input and output for which we have:

- $\{\!|\, a(x_1, \ldots, x_n).\, P \,|\!\} \stackrel{\text{def}}{=} a(w).\, \mathrm{INP}\langle w, x_1, \ldots, x_n\rangle.\, \{\!|\, P \,|\!\}$
- $\{\!|\, \overline{a}\langle b_1, \ldots, b_n\rangle.\, Q \,|\!\} \stackrel{\text{def}}{=} (\boldsymbol{\nu} w)(\overline{a}w.\, \mathrm{OUT}\langle w, b_1, \ldots, b_n\rangle.\, \{\!|\, Q \,|\!\})$

where

- $\mathrm{INP}\langle w, x_1, \ldots, x_n\rangle \stackrel{\text{def}}{=} w(x_1).\, w(x_2) \ldots w(x_n)$
- $\mathrm{OUT}\langle w, b_1, \ldots, b_n\rangle \stackrel{\text{def}}{=} \overline{w}b_1.\, \overline{w}b_2 \ldots \overline{w}b_n$

and w is a fresh name, i.e., it is not free in the translated processes. Intuitively, $\mathrm{INP}\langle w, \widetilde{x}\rangle$ and $\mathrm{OUT}\langle w, \widetilde{b}\rangle$ model a protocol which takes care of instantiating each variable x_i with the correspondent name b_i by using a fresh channel w. Since w is private to $\mathrm{INP}\langle w, \widetilde{x}\rangle$ and $\mathrm{OUT}\langle w, \widetilde{b}\rangle$, no interferences are possible. It is easy to show that there is an *operational correspondence* between a polyadic process P and its translation $\{\!|\, P \,|\!\}$: (i) if $P \xrightarrow{\tau} P'$ then $\{\!|\, P \,|\!\} \xrightarrow{\tau} \gtrsim \{\!|\, P' \,|\!\}$ and (ii) if $\{\!|\, P \,|\!\} \xrightarrow{\tau} P_1$ then there exists P' such that $P \xrightarrow{\tau} P'$ and $P_1 \gtrsim \{\!|\, P' \,|\!\}$ where $\lesssim$ is the *synchronous expansion* (see Definition 2.3.9). From the operational correspondence one can derive the *soundness* of the encoding when considering *barbed congruence* as the behavioural equivalence in both source and target languages. Unfortunately, Milner's encoding is not complete and therefore it is not *fully-abstract*. As a counterexample, take $R = a(\widetilde{x}).\, a(\widetilde{y}).\, \mathbf{0}$ and $S = a(\widetilde{x}).\, \mathbf{0} \mid a(\widetilde{y}).\, \mathbf{0}$; then R and S are barbed congruent but their encodings are not: $\{\!|\, S \,|\!\}$ may perform two consecutive inputs along a while in $\{\!|\, R \,|\!\}$ the input protocol $\mathrm{INP}\langle w, \widetilde{x}\rangle$ blocks the second input along a. In synchronous π-calculus, a similar counterexample can be given by using outputs instead of inputs. These counterexamples essentially say that Milner's encoding is not fully-abstract because the protocols $\mathrm{INP}\langle w, \widetilde{x}\rangle$ and $\mathrm{OUT}\langle w, \widetilde{b}\rangle$ prevent the continuations $\{\!|\, P \,|\!\}$ and $\{\!|\, Q \,|\!\}$ from evolving. Thus, one might think of adapting, somehow, Milner's encoding so that the protocols $\mathrm{INP}\langle w, \widetilde{x}\rangle$ and $\mathrm{OUT}\langle w, \widetilde{b}\rangle$ (or a variant of them) are in parallel with the continuations and not in sequence. In (full) π-calculus, such an adaptation is not possible because of the binding nature of the input prefix. By contrast, in $\mathrm{L}\pi$ we can avoid the obstacle by using (a synchronous polyadic variant of) Proposition 4.1.1.

Lemma 5.2.1 *Let Γ be a sorting and $\widetilde{a}$ and $\widetilde{b}$ two tuples of distinct names such that: (i) $\widetilde{a}$ and $\widetilde{b}$ have the same arity; (ii) $\Gamma(a_j) = \Gamma(b_j)$, for all j; and (iii) $\widetilde{a} \cap \widetilde{b} = \emptyset$. Let P be a well-sorted $\mathrm{L}\widetilde{\pi}$-process with sorting Γ. If all names in $\widetilde{a}$ do not appear free in input position in P, then it holds that:*

$$(\boldsymbol{\nu}\widetilde{a})(\widetilde{a} \triangleright \widetilde{b} \mid P) \cong^{\mathrm{s}}_{\mathrm{L}\widetilde{\pi}} P\{\widetilde{b}/\widetilde{a}\}.$$

In the following we show how Lemma 5.2.1 can be used to define an encoding $\langle\!|\cdot|\!\rangle$ of polyadic $\mathrm{L}\pi$-processes into monadic ones. For simplicity, we restrict ourselves to processes transmitting pairs of names. The general case, when tuples of arbitrary size are transmitted, can be derived straightforwardly. The encoding $\langle\!|\cdot|\!\rangle$ is an homomorphism on all operators except input and output, for which we have:

- $\langle\!| a(\widetilde{x}).\, P |\!\rangle \stackrel{\text{def}}{=} a(w).\,(\boldsymbol{\nu}\widetilde{x})(\mathrm{INP}\langle w, \widetilde{x}\rangle \mid \langle\!| P |\!\rangle)$
- $\langle\!| \overline{a}\langle\widetilde{b}\rangle |\!\rangle \quad \stackrel{\text{def}}{=} (\boldsymbol{\nu}w)(\overline{a}w \mid \mathrm{OUT}\langle w, \widetilde{b}\rangle)$

where $w \notin \mathrm{fn}(\langle\!| P |\!\rangle)$, and supposing $\widetilde{x} = (x_1, x_2)$, $\widetilde{y} = (y_1, y_2)$, $\widetilde{b} = (b_1, b_2)$, and

$$\widetilde{x} \triangleright \widetilde{y} = !x_1(z).\,\overline{y_1}z \mid !x_2(z).\,\overline{y_2}z$$

we define:

- $\mathrm{INP}\langle w, \widetilde{x}\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu}c_1c_3)(\overline{w}c_1 \mid c_1(c_2).\,(\overline{c_2}c_3 \mid c_3(y_1).\,c_1(y_2).\,\widetilde{x} \triangleright \widetilde{y}))$
- $\mathrm{OUT}\langle w, \widetilde{b}\rangle \stackrel{\text{def}}{=} w(c_1).\,(\boldsymbol{\nu}c_2)(\overline{c_1}c_2 \mid c_2(c_3).\,(\overline{c_3}b_1 \mid \overline{c_1}b_2))$.

Like Milner's encoding, $\langle\!| \cdot |\!\rangle$ is based on the exchange of a private channel w used by $\mathrm{INP}\langle w, \widetilde{x}\rangle$ and $\mathrm{OUT}\langle w, \widetilde{b}\rangle$ for transmitting names b_i. Unlike Milner's encoding, in $\langle\!| \cdot |\!\rangle$ the transmission of names b_i produces n forwarders $x_i \triangleright b_i$ in parallel. More precisely, by Lemma 5.2.1, it holds that:

$$\langle\!| \overline{a}\langle\widetilde{b}\rangle \mid a(\widetilde{x}).\,P |\!\rangle \xrightarrow{\tau} \gtrsim (\boldsymbol{\nu}\widetilde{x})(\widetilde{x} \triangleright \widetilde{b} \mid \langle\!| P |\!\rangle) \cong^{\mathrm{s}}_{\mathrm{L}\pi} \langle\!| P |\!\rangle\{\widetilde{b}/\widetilde{x}\} \equiv \langle\!| P\{\widetilde{b}/\widetilde{x}\} |\!\rangle.$$

The encoding $\langle\!| \cdot |\!\rangle$ can be proved sound with respect to barbed congruence. Unfortunately, in this form, the encoding is not fully-abstract yet because it is not complete. As a counterexample, take the processes

$$R = \overline{a}\langle\widetilde{b}\rangle \;\text{ and }\; S = (\boldsymbol{\nu}\widetilde{d})(\overline{a}\langle\widetilde{d}\rangle \mid \widetilde{d} \triangleright \widetilde{b})$$

with $\widetilde{b}{=}(b_1, b_2)$ and $\widetilde{d}{=}(d_1, d_2)$. By Lemma 5.2.1 we have $R \cong^{\mathrm{s}}_{\mathrm{L}\widetilde{\pi}} S$; while $\langle\!| R |\!\rangle \not\cong^{\mathrm{s}}_{\mathrm{L}\pi} \langle\!| S |\!\rangle$. Indeed, let $C[\cdot] = [\cdot] \mid T$ with

$$T \stackrel{\text{def}}{=} a(w).\,(\boldsymbol{\nu}c_1c_3h)(\overline{w}c_1 \mid c_1(c_2).\,(\overline{c_2}c_3 \mid c_3(y_1).\,c_1(y_2).\,(\overline{y_1}h \mid h(x).\,\overline{m})))$$

then $C[\langle\!| R |\!\rangle] \not\Downarrow_m$ whereas $C[\langle\!| S |\!\rangle] \Downarrow_m$. However, a similar counterexample would not be possible starting from two processes R and S related by $\approx$. This observation allows us to give an amended variant of $\langle\!| \cdot |\!\rangle$. By Theorem 5.1.3 we know that the encoding $[\![\cdot]\!]_\Gamma$ maps barbed congruent processes into bisimilar processes; that is, on image-finite processes:

$$P \cong^{\mathrm{s}}_{\mathrm{L}\widetilde{\pi}} Q \text{ iff } [\![P]\!]_\Gamma \approx [\![Q]\!]_\Gamma.$$

So, we can refine the encoding $\langle\!| \cdot |\!\rangle$ by simply combining $\langle\!| \cdot |\!\rangle$ with $[\![\cdot]\!]_\Gamma$. More precisely, we define an encoding $\langle\!\langle\cdot\rangle\!\rangle_\Gamma$ of $\mathrm{L}\widetilde{\pi}$ into $\mathrm{L}\pi$ as the composition of $[\![\cdot]\!]_\Gamma$ and $\langle\!| \cdot |\!\rangle$, thus if P is an $\mathrm{L}\widetilde{\pi}$-process

$$\langle\!\langle P\rangle\!\rangle_\Gamma \stackrel{\text{def}}{=} \langle\!| [\![P]\!]_\Gamma |\!\rangle.$$

5.3 Full abstraction of the encoding

We prove that, on image-finite and well-sorted processes, $\langle\!\langle\!\llbracket \cdot \rrbracket\!\rangle\!\rangle_\Gamma$ is fully abstract with respect to synchronous barbed congruence. To this end, we study the encoding $\llbracket \langle\!| \cdot |\!\rangle \rrbracket_\Gamma$ obtained by inverting the order of the application of the encodings $\llbracket \cdot \rrbracket_\Gamma$ and $\langle\!| \cdot |\!\rangle$. We first prove an operational correspondence, up to expansion, between processes P and $\llbracket \langle\!| P |\!\rangle \rrbracket_\Gamma$. This will allows us to prove the soundness of $\langle\!\langle\!\llbracket \cdot \rrbracket\!\rangle\!\rangle_\Gamma$. Afterwards, we derive the completeness of $\langle\!\langle\!\llbracket \cdot \rrbracket\!\rangle\!\rangle_\Gamma$ from a completeness result for $\llbracket \langle\!| \cdot |\!\rangle \rrbracket_\Gamma$.

Lemma 5.3.1 allows us to prove the operational correspondence between processes P and $\llbracket \langle\!| P |\!\rangle \rrbracket_\Gamma$.

Lemma 5.3.1 (Boreale [10])

1. *Given a sorting Γ and three tuples of names $\widetilde{a}$, $\widetilde{b}$, $\widetilde{c}$ such that: (i) $\widetilde{a}$, $\widetilde{b}$, and $\widetilde{c}$ have the same arity; (ii) $\Gamma(a_j) = \Gamma(b_j) = \Gamma(c_j)$ for all j; and (iii) $(\widetilde{a} \cup \widetilde{c}) \cap \widetilde{b} = \emptyset$. Then $(\boldsymbol{\nu}\widetilde{b})(\widetilde{a} \to \widetilde{b} \mid \widetilde{b} \to \widetilde{c}) \gtrsim \widetilde{a} \to \widetilde{c}$.*
2. *Let Γ be a sorting and $\widetilde{a}$ and $\widetilde{b}$ two tuples of names such that (i) $\widetilde{a}$ and $\widetilde{b}$ have the same arity; (ii) $\Gamma(a_j) = \Gamma(b_j)$, for all j; and (iii) $\widetilde{a} \cap \widetilde{b} = \emptyset$. Let P be a well-sorted $\mathrm{L}\widetilde{\pi}$-process with sorting Γ. If all names in $\widetilde{a}$ do not appear free in input position in P, then it holds that*

$$(\boldsymbol{\nu}\widetilde{a})(\widetilde{a} \to \widetilde{b} \mid \llbracket P \rrbracket_\Gamma) \gtrsim \llbracket P \rrbracket_\Gamma\{\widetilde{b}/\widetilde{a}\}.$$

Lemma 5.3.2 (Operational correspondence) *Given a well-sorted process P in $\mathrm{L}\widetilde{\pi}$ with sorting Γ. Then:*

1. *Suppose that $P \xrightarrow{\alpha} P'$. Then we have:*
   (a) *if $\alpha = a(\widetilde{x})$ then $\llbracket \langle\!| P |\!\rangle \rrbracket_\Gamma \xrightarrow{a(w)} \gtrsim (\boldsymbol{\nu}\widetilde{x})(\llbracket \mathrm{INP}\langle w, \widetilde{x}\rangle \rrbracket_\Gamma \mid \llbracket \langle\!| P' |\!\rangle \rrbracket_\Gamma)$;*
   (b) *if $\alpha = (\boldsymbol{\nu}\widetilde{c})\overline{a}\langle\widetilde{b}\rangle$, with $\widetilde{c}$ possibly empty, then*
   $\llbracket \langle\!| P |\!\rangle \rrbracket_\Gamma \xrightarrow{\overline{a}(p)} \gtrsim (\boldsymbol{\nu}\widetilde{c})((\boldsymbol{\nu}w)(p \to w \mid \llbracket \mathrm{OUT}\langle w, \widetilde{b}\rangle \rrbracket_\Gamma) \mid \llbracket \langle\!| P' |\!\rangle \rrbracket_\Gamma)$, *with $p \notin \mathrm{fn}(P')$;*
   (c) *if $\alpha = \tau$ then $\llbracket \langle\!| P |\!\rangle \rrbracket_\Gamma \xrightarrow{\tau} \gtrsim \llbracket \langle\!| P' |\!\rangle \rrbracket_\Gamma$.*
2. *Suppose that $\llbracket \langle\!| P |\!\rangle \rrbracket_\Gamma \xrightarrow{\alpha} P_1$. Then there exists $P' \in \mathrm{L}\widetilde{\pi}$ such that:*
   (a) *if $\alpha = a(w)$ then $P \xrightarrow{a(\widetilde{x})} P'$, for some $\widetilde{x}$, with $P_1 \gtrsim (\boldsymbol{\nu}\widetilde{x})(\llbracket \mathrm{INP}\langle w, \widetilde{x}\rangle \rrbracket_\Gamma \mid \llbracket \langle\!| P' |\!\rangle \rrbracket_\Gamma)$;*
   (b) *if $\alpha = \overline{a}(p)$ then $P \xrightarrow{(\boldsymbol{\nu}\widetilde{c})\overline{a}\langle\widetilde{b}\rangle} P'$, with $\widetilde{c}$ possibly empty, $p \notin \mathrm{fn}(P')$ and*
   $P_1 \gtrsim (\boldsymbol{\nu}\widetilde{c})((\boldsymbol{\nu}w)(p \to w \mid \llbracket \mathrm{OUT}\langle w, \widetilde{b}\rangle \rrbracket_\Gamma) \mid \llbracket \langle\!| P' |\!\rangle \rrbracket_\Gamma)$;
   (c) *if $\alpha = \tau$ then $P \xrightarrow{\tau} P'$ with $P_1 \gtrsim \llbracket \langle\!| P' |\!\rangle \rrbracket_\Gamma$.*

Proof: By transition induction. The only subtle points arise in parts 1(c) and 2(c) where also Lemma 5.3.1 is used. Details can be found in Appendix A.3.

Remark 5.3.3 *The lemma above is not true when considering ill-sorted processes. For instance, if* $P = \overline{a}\langle b, c\rangle \mid a(x).\,Q$ *then* $[\![\langle\!| P |\!\rangle]\!]_\Gamma \xrightarrow{\tau}$ *while* $P \stackrel{\tau}{\not\longrightarrow}$.

From Lemma 5.3.2 we can derive a weak operational correspondence.

Lemma 5.3.4 *Let P be a well-sorted process in* $\mathrm{L}\widetilde{\pi}$ *with sorting* Γ. *Then:*

1. *If* $P \Longrightarrow P'$ *then* $[\![\langle\!| P |\!\rangle]\!]_\Gamma \Longrightarrow\gtrsim [\![\langle\!| P' |\!\rangle]\!]_\Gamma$.
2. *If* $[\![\langle\!| P |\!\rangle]\!]_\Gamma \Longrightarrow P_1$ *then there is* P' *s.t.* $P \Longrightarrow P'$ *and* $P_1 \gtrsim [\![\langle\!| P' |\!\rangle]\!]_\Gamma$.
3. $P \Downarrow_a$ *iff* $[\![\langle\!| P |\!\rangle]\!]_\Gamma \Downarrow_a$.

Proof: Parts 1 and 2 are proven by induction on the number of τ-moves by exploiting Lemma 5.3.2. Part 3 follows from parts 1 and 2, and Lemma 5.3.2. Lemmas 5.3.2 and 5.3.4 allow us to prove the following two lemmas which will be used to prove the soundness of $\langle\!\![\![\cdot]\!]\!\!\rangle_\Gamma$.

Lemma 5.3.5 *Let P and Q be two well-sorted* $\mathrm{L}\widetilde{\pi}$*-processes with sorting* Γ. *Then:*

$$[\![\langle\!| P |\!\rangle]\!]_\Gamma \stackrel{\bullet}{\approx} [\![\langle\!| Q |\!\rangle]\!]_\Gamma \;\text{ implies }\; P \stackrel{\bullet}{\approx} Q.$$

Proof: We use Lemmas 5.3.2 and 5.3.4 and the fact that $\lesssim \stackrel{\bullet}{\approx} \gtrsim \;\subset\; \stackrel{\bullet}{\approx}$ to prove that the relation

$$\mathcal{R} = \{(P, Q) : [\![\langle\!| P |\!\rangle]\!]_\Gamma \stackrel{\bullet}{\approx} [\![\langle\!| Q |\!\rangle]\!]_\Gamma\}$$

is a barbed bisimulation.

Lemma 5.3.6 *Let P and Q be two well-sorted* $\mathrm{L}\widetilde{\pi}$*-processes with sorting* Γ. *Then:*

$$\langle\!\![\![P]\!]\!\!\rangle_\Gamma \stackrel{\bullet}{\approx} \langle\!\![\![Q]\!]\!\!\rangle_\Gamma \;\text{ implies }\; P \stackrel{\bullet}{\approx} Q.$$

Proof: Since $\langle\!\![\![\cdot]\!]\!\!\rangle_\Gamma \stackrel{\text{def}}{=} \langle\!| [\![\cdot]\!]_\Gamma |\!\rangle$, by Theorem 5.1.2, $[\![\langle\!| [\![P]\!]_\Gamma |\!\rangle]\!]_\Gamma \stackrel{\bullet}{\approx} [\![\langle\!| [\![Q]\!]_\Gamma |\!\rangle]\!]_\Gamma$. By Lemma 5.3.5, we have $[\![P]\!]_\Gamma \stackrel{\bullet}{\approx} [\![Q]\!]_\Gamma$. By Theorem 5.1.2, we have $P \stackrel{\bullet}{\approx} Q$. The following lemma will allow us to prove the completeness of $\langle\!\![\![\cdot]\!]\!\!\rangle_\Gamma$.

Lemma 5.3.7 *Let P and Q be two well-sorted* $\mathrm{L}\widetilde{\pi}$*-processes with sorting* Γ. *Then:*

$$P \approx Q \;\text{ implies }\; [\![\langle\!| P |\!\rangle]\!]_\Gamma \approx [\![\langle\!| Q |\!\rangle]\!]_\Gamma.$$

Proof: We prove that $\mathcal{S} = \{([\![\langle\!| P |\!\rangle]\!]_\Gamma, [\![\langle\!| Q |\!\rangle]\!]_\Gamma) : P \approx Q\}$ is a bisimulation up to context and up to $\gtrsim$. Details can be found in Appendix A.3.

Finally, we prove that, on image-finite and well-sorted processes, the encoding $\langle\!\![\![\cdot]\!]\!\!\rangle_\Gamma$ is fully-abstract with respect to barbed congruence.

Theorem 5.3.8 (Full abstraction of $\langle\!\![\![\cdot]\!]\!\!\rangle_\Gamma$) *Let P and Q be two image-finite and well-sorted processes in* $\mathrm{L}\widetilde{\pi}$ *with sorting* Γ. *Then:*

$$P \cong^{\mathrm{s}}_{\mathrm{L}\widetilde{\pi}} Q \;\text{ iff }\; \langle\!\![\![P]\!]\!\!\rangle_\Gamma \cong^{\mathrm{s}}_{\mathrm{L}\pi} \langle\!\![\![Q]\!]\!\!\rangle_\Gamma.$$

Proof: The soundness follows from the compositionality of both $[\![\cdot]\!]_\Gamma$ and $\langle\!| \cdot |\!\rangle$, and from Lemma 5.3.6. As for completeness, by Theorem 5.1.3(1) $[\![P]\!]_\Gamma \approx [\![Q]\!]_\Gamma$. By Lemma 5.3.7, we have $[\![\langle\!| [\![P]\!]_\Gamma |\!\rangle]\!]_\Gamma \approx [\![\langle\!| [\![Q]\!]_\Gamma |\!\rangle]\!]_\Gamma$. By the monadic variant of Theorem 5.1.3(2) it follows that $\langle\!| [\![P]\!]_\Gamma |\!\rangle \cong^{\mathrm{s}}_{\mathrm{L}\pi} \langle\!| [\![Q]\!]_\Gamma |\!\rangle$, i.e., $\langle\!\![\![P]\!]\!\!\rangle_\Gamma \cong^{\mathrm{s}}_{\mathrm{L}\pi} \langle\!\![\![Q]\!]\!\!\rangle_\Gamma$.

5.4 Encoding polyadicity in Fusion calculus

In Section 5.2, we said that Milner's encoding is not fully-abstract because the protocols $\mathrm{INP}\langle w, \widetilde{x}\rangle$ and $\mathrm{OUT}\langle w, \widetilde{b}\rangle$ prevent the continuations $\{\!|\ P\ |\!\}$ and $\{\!|\ Q\ |\!\}$ from evolving. More precisely, the real problem is the binding nature of the input prefix. Indeed, we can easily change the encoding of $\overline{a}\langle\widetilde{b}\rangle. P$ by putting the protocol $\mathrm{OUT}\langle w, \widetilde{b}\rangle$ in parallel with the continuation but we cannot do the same with the encoding of input prefix.

By contrast, in Fusion calculus (see Section 2.5.4) the input prefix is non-binding. So, in Fusion we can adapt Milner's encoding by simply putting the protocols $\mathrm{INP}\langle w, \widetilde{x}\rangle$ and $\mathrm{OUT}\langle w, \widetilde{b}\rangle$ in parallel with the continuations, as showed below:

$$\begin{aligned}(\!|\, a\langle x_1, \ldots, x_n\rangle. P\,|\!) &\stackrel{\text{def}}{=} (w)aw.\,(wx_1.\,wx_2.\ldots wx_n \mid (\!|\, P\,|\!))\\ (\!|\, \overline{a}\langle b_1, \ldots, b_n\rangle. Q\,|\!) &\stackrel{\text{def}}{=} (w)\overline{a}w.\,(\overline{w}b_1.\,\overline{w}b_2.\ldots \overline{w}b_n \mid (\!|\, Q\,|\!)).\end{aligned}$$

Note that the continuations $(\!|\, P\,|\!)$ and $(\!|\, Q\,|\!)$ may evolve without waiting for the n communications along the private channel w.

Unfortunately, the encoding $(\!|\cdot|\!)$ is not sound with respect to either barbed congruence or *hyperequivalence* (a labelled bisimilarity introduced by Parrow and Victor in [65]). This because, in some sense, the encoding breaks the preemptive power of fusions processes[1]. More precisely, if we take

$$R = \{a = b\} \mid \{c = d\} \text{ and } S = \{a = b\}.\{c = d\}^2$$

then R and S are not equivalent while their translations are.

In Fusion calculus, communications between two processes may arise only when carrying restricted names, that is, names with a delimited scope. So, in Fusion, it makes sense to have a definition of barbed congruence where contexts always bind the names of the tested processes.

Definition 5.4.1 (Synchronous closed barbed congruence) *Processes P and Q are* synchronous closed barbed congruent *in $\mathcal{L}$, if for all contexts $C[\cdot]$ in $\mathcal{L}$, such that $\mathrm{fn}(P) \cap \mathrm{fn}(C[P]) = \mathrm{fn}(Q) \cap \mathrm{fn}(C[Q]) = \emptyset$, it holds that $C[P] \stackrel{\bullet}{\approx}^{\mathrm{s}} C[Q]$.*

In closed barbed congruence, as well as in *testing equivalences* [21], the contexts signal success by emitting along names that do not appear in the tested processes.

In π-calculus and CCS, closed barbed congruence coincides with barbed congruence. Indeed, one can prove that closed barbed congruence coincides with the closure under substitutions of early bisimilarity, by adapting the proofs in [72]; since the closure under substitutions of early bisimilarity is known to coincide with barbed congruence, the two forms of barbed congruence coincide. By contrast, in Fusion calculus, closed barbed

[1] We recall that fusion processes are defined as $\{\widetilde{a} = \widetilde{b}\} \stackrel{\text{def}}{=} (u)(\overline{u}\langle\widetilde{a}\rangle. \mathbf{0} \mid u\langle\widetilde{b}\rangle. \mathbf{0})$.

[2] A formal definition for S is $(uz)(ua.\, zc \mid \overline{u}b.\, \overline{z}d)$

congruence is strictly larger than barbed congruence. As an example, consider the two processes:

$$P = \overline{a}\langle b\rangle \mid \{b = c\} \quad \text{and} \quad Q = \overline{a}\langle c\rangle \mid \{b = c\}.$$

Processes P and Q are closed barbed congruent (the intuition is that the two processes may emit along a two names which should be considered equal) but not barbed congruent because the context $[\cdot] \mid (x)a\langle x\rangle.\,\overline{x}$ is able to distinguish them.

We believe that the encoding $\llparenthesis \cdot \rrparenthesis$ is fully-abstract with respect to *closed barbed congruence*. Our conjecture is due to the fact that closed barbed congruence is insensitive to fusion prefixing, that is, it considers fusion actions like silent moves. As a consequence, the counterexample above is not valid anymore because processes $\{a = b\} \mid \{c = d\}$ and $\{a = b\}.\,\{c = d\}$ are closed barbed congruent. Unfortunately, we cannot prove the full abstraction of $\llparenthesis \cdot \rrparenthesis$ with the same proof techniques of Section 5.3 because we do not know yet a labelled characterisation of closed barbed congruence in Fusion.

5.5 Discussion

We have presented an encoding $\langle\!\langle[\![\cdot]\!]\rangle\!\rangle_\Gamma$ of polyadic Lπ into monadic Lπ inspired by Milner's encoding. We have proved that, on image-finite and well-sorted processes, $\langle\!\langle[\![\cdot]\!]\rangle\!\rangle_\Gamma$ is fully-abstract with respect to synchronous barbed congruence. This result confirms that Lπ is a very expressive subcalculus of asynchronous π. We conjecture that the encoding $\langle\!\langle[\![\cdot]\!]\rangle\!\rangle_\Gamma$ is *divergence-free*, that is, it does not introduce infinite internal computations.

Other possible candidates for a fully-abstract encoding of polyadic Lπ into monadic Lπ are $\langle\!| \cdot |\!\rangle$ and $[\![\langle\!| \cdot |\!\rangle]\!]_\Gamma$. Unfortunately, none of them is fully-abstract with respect to barbed congruence or bisimilarity.

In Section 5.2 we already showed that $\langle\!| \cdot |\!\rangle$ is not fully-abstract with respect to barbed congruence. The encoding $\langle\!| \cdot |\!\rangle$ is not fully-abstract with respect to bisimilarity either. As a counterexample, take the processes :

$$P = (\boldsymbol{\nu} a)(\overline{a}\langle\widetilde{b}\rangle \mid a(\widetilde{x}).\,\overline{c}\langle\widetilde{x}\rangle) \;\text{ and }\; Q = \overline{c}\langle\widetilde{b}\rangle.$$

It holds that, $P \approx Q$; whereas $\langle\!| P |\!\rangle \approx (\boldsymbol{\nu}\widetilde{x})(\widetilde{x} \triangleright \widetilde{b} \mid \langle\!| \overline{c}\langle\widetilde{x}\rangle |\!\rangle)$, and $\langle\!| Q |\!\rangle = \langle\!| \overline{c}\langle\widetilde{b}\rangle |\!\rangle$, and thus $\langle\!| P |\!\rangle \not\approx \langle\!| Q |\!\rangle$.

By Lemma 5.3.7 the encoding $[\![\langle\!| \cdot |\!\rangle]\!]_\Gamma$ is complete with respect to $\approx$. Since the encoding $[\![\langle\!| \cdot |\!\rangle]\!]_\Gamma$ enjoys an operational correspondence up to expansion (Lemma 5.3.2), one may hope that $[\![\langle\!| \cdot |\!\rangle]\!]_\Gamma$ is sound with respect to bisimilarity and therefore fully-abstract. Unfortunately, $[\![\langle\!| \cdot |\!\rangle]\!]_\Gamma$ is not sound with respect to bisimilarity. As a counterexample, take:

$$P = (\boldsymbol{\nu}\widetilde{c})\overline{a}\langle\widetilde{c}\rangle \;\text{ and }\; Q = (\boldsymbol{\nu}\widetilde{c})(\overline{a}\langle\widetilde{c}\rangle \mid \overline{c_1}\langle b\rangle)$$

with $\widetilde{c} = (c_1, c_2)$. Then, $P \cong^{\mathrm{s}}_{\mathrm{L}\widetilde{\pi}} Q$ and $\langle\!| P |\!\rangle \cong^{\mathrm{s}}_{\mathrm{L}\pi} \langle\!| Q |\!\rangle$; by Theorem 5.1.3 it holds that $[\![\langle\!| P |\!\rangle]\!]_\Gamma \approx [\![\langle\!| Q |\!\rangle]\!]_\Gamma$, but $P \not\approx Q$. The encoding $[\![\langle\!| \cdot |\!\rangle]\!]_\Gamma$ is not fully-abstract with respect to barbed congruence either. As a counterexample, take:

$$P = \overline{a}\langle\widetilde{b}\rangle \ \text{ and } \ Q = (\boldsymbol{\nu}\widetilde{d})(\overline{a}\langle\widetilde{d}\rangle \mid \widetilde{d} \triangleright \widetilde{b})$$

with $\widetilde{b} = (b_1, b_2)$ and $\widetilde{d} = (d_1, d_2)$. Then, as already shown in Section 5.2, $P \cong^{\mathrm{s}}_{\mathrm{L}\widetilde{\pi}} Q$ and $\langle\!|\, P \,|\!\rangle \ncong^{\mathrm{s}}_{\mathrm{L}\pi} \langle\!|\, Q \,|\!\rangle$; since the encoding $[\![\cdot]\!]_\Gamma$ is sound with respect to barbed congruence (by Theorem 5.1.2 and the compositionality of $[\![\cdot]\!]_\Gamma$) we have that $[\![\langle\!|\, P \,|\!\rangle]\!]_\Gamma \ncong^{\mathrm{s}}_{\mathrm{L}\pi} [\![\langle\!|\, Q \,|\!\rangle]\!]_\Gamma$.

Note that the encoding $\langle\!\langle\cdot\rangle\!\rangle_\Gamma$ (as well as Milner's encoding) does not preserve well-sortedness.

Note also that we have used synchronous barbed congruence instead of asynchronous barbed congruence. Actually, the encoding $\langle\!\langle\cdot\rangle\!\rangle_\Gamma$ is sound but not complete with respect to asynchronous barbed congruence. As counterexample, consider the processes $P = a(\widetilde{x}).\,\overline{a}\langle\widetilde{x}\rangle$ and $Q = \mathbf{0}$; then P and Q are asynchronous barbed congruent, while $\langle\!\langle P\rangle\!\rangle_\Gamma$ and $\langle\!\langle Q\rangle\!\rangle_\Gamma$ are not. On the other hand, although law $a(\widetilde{x}).\,\overline{a}\langle\widetilde{x}\rangle = \mathbf{0}$ is sometimes useful, it introduces divergences. For instance, from $a(\widetilde{x}).\,\overline{a}\langle\widetilde{x}\rangle = \mathbf{0}$ we can derive the equality $!a(\widetilde{x}).\,\overline{a}\langle\widetilde{x}\rangle \mid \overline{a}b = \overline{a}b$ between a divergent and a non-divergent process. We conjecture that $\langle\!\langle\cdot\rangle\!\rangle_\Gamma$ is fully-abstract with respect to a variant of asynchronous barbed congruence which is sensitive to divergence, along the lines of [89].

Chapter 6

From Local π to asynchronous π

In $\mathrm{L}\pi$, the recipient of a name may only use it in output actions. In Sections 3.4, we showed that such an "inversion of polarity" can be represented by making appropriate use of the "directional" link process $!a(x).\,\overline{b}x$.

In π_{a}, the recipient of a name may use it in both input and output actions. In this chapter, we show that the theory of $\mathrm{L}\pi$ can be adapted to π_{a}, by essentially replacing the link process with the "bidirectional" *equator* process $!a(x).\,\overline{b}x \mid !b(x).\,\overline{a}x$ [37]. The theory of π_{a}, based on equators, presents a few crucial differences when compared with theory of $\mathrm{L}\pi$, based on links (see Remarks 6.1.3, 6.1.9, and 6.1.13).

As suggested by their name, equators induce a sort of equivalence relation over names and are very close to *fusions* of names, as defined in the *Fusion calculus* [65]. In Section 6.2, we give an encoding of asynchronous Fusion calculus into the asynchronous π-calculus. Roughly, the encoding models fusion of names in terms of equators. We prove the encoding fully-abstract with respect to asynchronous closed barbed congruence, an asynchronous variant of the closed barbed congruence introduced in Section 5.4.

Most of the proof techniques in this chapter are inspired by those in [10] and those in Sections 3.3 and 3.4. So, when possible, we just sketch the proofs pointing out the differences with the existing proofs.

6.1 Asynchronous π and equators

Equators, defined as

$$\mathcal{E}(a,b) \stackrel{\mathrm{def}}{=} !a(x).\,\overline{b}x \mid !b(x).\,\overline{a}x$$

have been first studied by Honda and Yoshida in [37].

The following result shows that equators induce a sort of equivalence relation over names.

Lemma 6.1.1

1. $(\boldsymbol{\nu} a)\mathcal{E}(a,b) \cong_{\pi a} \mathcal{E}(b,b) \cong_{\pi a} \mathbf{0}$
2. $\mathcal{E}(a,b) \cong_{\pi a} \mathcal{E}(b,a)$
3. $(\boldsymbol{\nu} c)(\mathcal{E}(a,c) \mid \mathcal{E}(c,b)) \cong_{\pi a} \mathcal{E}(a,b)$.

Proof: Part 1 is proved by exhibiting the appropriate asynchronous ground bisimulations. Parts 2 and 3 are proved by exhibiting the appropriate synchronous ground bisimulations. Since both synchronous and asynchronous bisimilarity imply $\cong_{\pi a}$ we can conclude. □

Equators play a central role in the asynchronous π-calculus without matching. Indeed, they permit equating output processes emitting different names along the same channel. This property appears clear in the following two laws:

$$\overline{a}b \mid \mathcal{E}(b,c) \;\cong_{\pi a}\; \overline{a}c \mid \mathcal{E}(b,c) \tag{6.1}$$

$$(\boldsymbol{\nu} c)(\overline{a}c \mid \mathcal{E}(c,b)) \;\cong_{\pi a}\; \overline{a}b. \tag{6.2}$$

Laws 6.1 and 6.2 have already been proved in [37] with respect to the *maximum sound theory*, a behavioural equality which coincides with barbed congruence [25]. The two laws above show that, in absence of matching, a labelled characterisation of barbed congruence must be able to match output actions with the same subject part but different object part. Somehow, it is as if the two names b and c in object position were considered as "the same" name.

Definition 6.1.2 (Synonymous bisimilarity) A symmetric relation $\mathcal{S}$ on π_a-terms is a *synonymous bisimulation* if whenever $P\ \mathcal{S}\ Q$ the following holds:

1. If $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \Longrightarrow Q'$ and $P'\ \mathcal{S}\ Q'$.
2. If $P \xrightarrow{a(w)} P'$, with $w \notin \mathrm{fn}(Q)$, then there exists Q' such that
   (a) either $Q \stackrel{a(w)}{\Longrightarrow} Q'$ and $P'\ \mathcal{S}\ Q'$
   (b) or $Q \Longrightarrow Q'$ and $P'\ \mathcal{S}\ (Q' \mid \overline{a}w)$
3. If $P \xrightarrow{\overline{a}b} P'$ then there exists Q' such that
   (a) either $Q \stackrel{\overline{a}c}{\Longrightarrow} Q'$ and for some $w \notin \mathrm{fn}(P \mid Q)$ it holds that $(\mathcal{E}(w,b) \mid P')\ \mathcal{S}\ (\mathcal{E}(w,c) \mid Q')$
   (b) or $Q \stackrel{\overline{a}(w)}{\Longrightarrow} Q'$, with $w \notin \mathrm{fn}(P)$, and $(\mathcal{E}(w,b) \mid P')\ \mathcal{S}\ Q'$.
4. If $P \xrightarrow{\overline{a}(w)} P'$, with $w \notin \mathrm{fn}(Q)$, then there exists Q' such that
   (a) either $Q \stackrel{\overline{a}(w)}{\Longrightarrow} Q'$ and $P'\ \mathcal{S}\ Q'$
   (b) or $Q \stackrel{\overline{a}b}{\Longrightarrow} Q'$, for some b, and $P'\ \mathcal{S}\ (\mathcal{E}(w,b) \mid Q')$.

P and Q are synonymous bisimilar, written $P \approx_\varepsilon Q$, if $P \ \mathcal{S}\ Q$ for some synonymous bisimulation $\mathcal{S}$.

Remark 6.1.3 *The differences between synonymous bisimilarity and link bisimilarity (Definition 3.4.1) are essentially two: (i) in Definition 6.1.2 we have equators instead of links ; (ii) in Definition 6.1.2, when matching bound output actions, we do not introduce any additional construct (link or equator) in the derivatives.*

6.1.1 A characterisation of barbed congruence in $\pi_{\rm a}$

In this section we prove that $\approx_\varepsilon$ characterises barbed congruence in $\pi_{\rm a}$.

We prove that $\approx_\varepsilon$ is contained in barbed congruence by exploiting a compositional encoding $[\![\cdot]\!]_\varepsilon$ which is an homomorphism on all operators except output. The encoding for outputs is the following:

$$[\![\overline{a}b]\!]_\varepsilon \stackrel{\text{def}}{=} (\boldsymbol{\nu} c)(\overline{a}c \mid \overrightarrow{\mathcal{E}}(b,c))$$

where

$$\overrightarrow{\mathcal{E}}(b,c) \stackrel{\text{def}}{=} !b(x).\,[\![\overline{c}x]\!]_\varepsilon \mid !c(x).\,[\![\overline{b}x]\!]_\varepsilon .$$

The encoding $[\![\cdot]\!]_\varepsilon$ is clearly inspired by the encoding $[\![\cdot]\!]$ in Section 3.3. The process $\overrightarrow{\mathcal{E}}(b,c)$ is a recursive variant of $\mathcal{E}(a,b)$ and it holds that $[\![\mathcal{E}(a,b)]\!]_\varepsilon = \overrightarrow{\mathcal{E}}(a,b)$.

Remark 6.1.4 *Like the recursive link $a \to b$, the recursive equator $\overrightarrow{\mathcal{E}}(a,b)$ is not in $\pi_{\rm a}$. However, there exists a process P in $\pi_{\rm a}$ which is early bisimilar to $\overrightarrow{\mathcal{E}}(a,b)$.*

In order to prove that $\approx_\varepsilon$ implies barbed congruence, we use a proof technique similar to that exploited in Sections 3.3 and 3.4:

- we denote with $\mathcal{A}\pi$ the image of $[\![\cdot]\!]_\varepsilon$, that is, the calculus obtained from $\pi_{\rm a}$ by replacing in the syntax the output particle $\overline{a}b$ with $[\![\overline{a}b]\!]_\varepsilon$;
- we write $\asymp_{\rm A}$ to denote the variant of asynchronous ground bisimulation in which, in the asynchronous clause for input, in the requirement for the derivatives, the output particle $\overline{a}b$ is replaced by its translation $[\![\overline{a}b]\!]_\varepsilon$;
- we write $\preccurlyeq_{\rm A}$ to denote the *expansion* variant of $\asymp_{\rm A}$.

The following lemma says that recursive equator $\overrightarrow{\mathcal{E}}(a,b)$ has properties similar to those of $\mathcal{E}(a,b)$.

Lemma 6.1.5

1. $(\boldsymbol{\nu} a)\overrightarrow{\mathcal{E}}(a,b) \succcurlyeq_{\rm A} \overrightarrow{\mathcal{E}}(b,b) \succcurlyeq_{\rm A} \mathbf{0}$
2. $\overrightarrow{\mathcal{E}}(a,b) \approx \overrightarrow{\mathcal{E}}(b,a)$

3. $(\boldsymbol{\nu} c)(\overrightarrow{\mathcal{E}}(a,c) \mid \overrightarrow{\mathcal{E}}(c,b)) \gtrsim \overrightarrow{\mathcal{E}}(a,b)$.

Proof: By exhibiting the appropriate expansions and/or bisimulations. The proof of part 3 is similar to that of Proposition 3.2.3(2). □

Notice that parts 2 and 3 of Lemma 6.1.5 hold for the stronger relations $\approx$ and $\gtrsim$, respectively.

Lemma 6.1.6 *The relation $\asymp_{\mathrm{A}}$ is a congruence in $\mathcal{A}\pi$.*

Proof: The proof is similar to that of Lemma 3.3.12 for the relation $\asymp_{\mathrm{a}}$. □

In the following we show that there is an operational correspondence, up to $\succcurlyeq_{\mathrm{A}}$, between processes P and $[\![P]\!]_{\mathcal{E}}$. The lemma below is crucial to prove such a operational correspondence. Notice that Lemma 6.1.7 is the $[\![\cdot]\!]_{\mathcal{E}}$ counterpart of Lemma 3.3.3.

Lemma 6.1.7 *If $P \in \pi_{\mathrm{a}}$ then $(\boldsymbol{\nu} a)(\overrightarrow{\mathcal{E}}(a,b) \mid [\![P]\!]_{\mathcal{E}}) \succcurlyeq_{\mathrm{A}} [\![P]\!]_{\mathcal{E}}\{b/a\}$.*

Proof: We show that the relation

$$\mathcal{S} = \{((\boldsymbol{\nu} a)(\overrightarrow{\mathcal{E}}(a,b) \mid P)\,,\, P\{b/a\}) : P \in \mathcal{A}\pi\}$$

is a $\succcurlyeq_{\mathrm{A}}$ relation up to $\succcurlyeq_{\mathrm{A}}$. The proof is easy because processes in $\mathcal{A}\pi$ never perform free output actions. □

Lemma 6.1.8 (Operational correspondence of $[\![\cdot]\!]_{\mathcal{E}}$) *Let P be a process in π_{a}.*

1. *Suppose that $P \xrightarrow{\mu} P'$. Then:*
   (a) *if $\mu = a(c)$ then $[\![P]\!]_{\mathcal{E}} \xrightarrow{a(c)} \succcurlyeq_{\mathrm{A}} [\![P']\!]_{\mathcal{E}}$;*
   (b) *if $\mu = \overline{a}b$ then $[\![P]\!]_{\mathcal{E}} \xrightarrow{\overline{a}(c)} \succcurlyeq_{\mathrm{A}} (\overrightarrow{\mathcal{E}}(c,b) \mid [\![P']\!]_{\mathcal{E}})$, with $c \notin \mathrm{fn}(P')$;*
   (c) *if $\mu = \overline{a}(b)$ then $[\![P]\!]_{\mathcal{E}} \xrightarrow{\overline{a}(c)} \succcurlyeq_{\mathrm{A}} (\boldsymbol{\nu} b)(\overrightarrow{\mathcal{E}}(c,b) \mid [\![P']\!]_{\mathcal{E}})$, with $c \notin \mathrm{fn}(P')$;*
   (d) *if $\mu = \tau$ then $[\![P]\!]_{\mathcal{E}} \xrightarrow{\tau} \succcurlyeq_{\mathrm{A}} [\![P']\!]_{\mathcal{E}}$.*
2. *Suppose that $[\![P]\!]_{\mathcal{E}} \xrightarrow{\mu} P_1$. Then there exists $P' \in \pi_{\mathrm{a}}$ such that:*
   (a) *if $\mu = a(c)$ then*
      i. *either $P \xrightarrow{a(c)} P'$ and $P_1 \succcurlyeq_{\mathrm{A}} [\![P']\!]_{\mathcal{E}}$*
      ii. *or $P \xrightarrow{\hat{\tau}} P'$ and $P_1 \succcurlyeq_{\mathrm{A}} [\![P']\!]_{\mathcal{E}} \mid [\![\overline{a}c]\!]_{\mathcal{E}}$;*
   (b) *if $\mu = \overline{a}(c)$ then:*
      i. *either $P \xrightarrow{\overline{a}b} P'$ and $P_1 \succcurlyeq_{\mathrm{A}} (\overrightarrow{\mathcal{E}}(c,b) \mid [\![P']\!]_{\mathcal{E}})$, with $c \notin \mathrm{fn}(P')$,*
      ii. *or $P \xrightarrow{\overline{a}(b)} P'$ and $P_1 \succcurlyeq_{\mathrm{A}} (\boldsymbol{\nu} b)(\overrightarrow{\mathcal{E}}(c,b) \mid [\![P']\!]_{\mathcal{E}})$, with $c \notin \mathrm{fn}(P')$;*
   (c) *if $\mu = \tau$ then $P \xrightarrow{\tau} P'$ and $P_1 \succcurlyeq_{\mathrm{A}} [\![P']\!]_{\mathcal{E}}$.*

Proof: By transition induction. The proof is similar to that of Lemma 3.3.2, whose proof is in [10]. As in [10], the only interesting points arise in the proofs of parts 1d and 2c, where also Lemma 6.1.7 is used. $\square$

Remark 6.1.9 *There are two main differences between Lemma 6.1.8 and Lemma 3.3.2: (i) the operational correspondence is up to $\succcurlyeq_{\mathrm{A}}$ and not up to $\gtrsim$ (ii) the asynchronous clause 2(a)ii does not exist in Lemma 3.3.2.*

The following lemmas is a weak variant of the above operational correspondence.

Lemma 6.1.10

1. *If $P \Longrightarrow P'$ then $[\![P]\!]_{\mathcal{E}} \Longrightarrow \succcurlyeq_{\mathrm{A}} [\![P']\!]_{\mathcal{E}}$.*
2. *If $[\![P]\!]_{\mathcal{E}} \Longrightarrow P_1$ then there is P' such that $P \Longrightarrow P'$ and $P_1 \succcurlyeq_{\mathrm{A}} [\![P']\!]_{\mathcal{E}}$.*
3. *$P \Downarrow_a$ iff $[\![P]\!]_{\mathcal{E}} \Downarrow_a$.*

Proof: Parts 1 and 2 are proven by induction on the number of τ-moves by exploiting Lemma 6.1.8. Part 3 is a consequence of parts 1 and 2, and Lemma 6.1.8. $\square$

Lemma 6.1.11 *Let P and Q be two π_{a}-processes. Then:*

$$[\![P]\!]_{\mathcal{E}} \stackrel{\bullet}{\approx} [\![Q]\!]_{\mathcal{E}} \text{ implies } P \stackrel{\bullet}{\approx} Q.$$

Proof: We use Lemmas 6.1.8 and 6.1.10, and the fact that $\preccurlyeq_{\mathrm{A}} \stackrel{\bullet}{\approx} \succcurlyeq_{\mathrm{A}} \subset \stackrel{\bullet}{\approx}$ to prove that the relation $\mathcal{R} = \{(P, Q) : [\![P]\!]_{\mathcal{E}} \stackrel{\bullet}{\approx} [\![Q]\!]_{\mathcal{E}}\}$ is a barbed bisimulation. $\square$

The following two lemmas are technical results. They are useful in the proof of Lemma 6.1.15.

Lemma 6.1.12 *Let P be a π_{a}-process. Then $(\boldsymbol{\nu} c)(\mathcal{E}(c, b) \mid P) \approx_{\mathcal{E}} P\{^b\!/\!_c\}$.*

Proof: By showing that the relation

$$\mathcal{S}=\{((\boldsymbol{\nu} c)(\mathcal{E}(c,b) \mid P)\,,\,P\{^b\!/\!_c\}) : P\in\pi_{\mathrm{a}}\} \cup \{(\mathcal{E}(c,b) \mid P\,,\,\mathcal{E}(c,b) \mid P\{^b\!/\!_c\}) : P\in\pi_{\mathrm{a}}\}$$

is a synonymous bisimulation. $\square$

Remark 6.1.13 *Notice the similarities between Lemma 6.1.12 and Proposition 4.1.1. Notice also that in Lemma 6.1.12 there is no constraint on the restricted name. This is actually the reason why synonymous bisimilarity and link bisimilarity work differently on bound output actions (see Remark 6.1.3).*

Lemma 6.1.14 *Let P and Q be two π_{a}-processes.*

1. *If $P \approx_{\mathcal{E}} Q$ then, for each $R \in \pi_{\mathrm{a}}$ s.t. $\mathrm{fn}(R) \cap \mathrm{fn}(P, Q)=\emptyset$, it holds that $P \mid R \approx_{\mathcal{E}} Q \mid R$.*
2. *If $P \approx_{\mathcal{E}} Q$ and $c \notin \mathrm{fn}(P, Q)$, then $P \mid \overline{a}c \approx_{\mathcal{E}} Q \mid \overline{a}c$.*

Proof: In the proof of part 1 it suffices to show that the relation

$$\mathcal{S} = \{(P \mid R\,,\ Q \mid R) : \mathrm{fn}(R) \cap \mathrm{fn}(P, Q) = \emptyset\}$$

is a synonymous bisimulation. In the proof of part 2 it suffices to show that the relation

$$\mathcal{S} = \{(P \mid \overline{a}c\,,\ Q \mid \overline{a}c) : P \approx_{\mathcal{E}} Q \ \text{ and } \ c \notin \mathrm{fn}(P, Q)\} \cup \approx_{\mathcal{E}}$$

is a synonymous bisimulation. In the proof of part 2, part 1 is used when matching the two free output actions $\overline{a}c$. □

The following result is the $[\![\cdot]\!]_{\mathcal{E}}$ counterpart of Lemma 3.4.3.

Lemma 6.1.15 *Let P and Q be two π_{a}-processes. Then:*

$$P \approx_{\mathcal{E}} Q \ \text{ implies } \ [\![P]\!]_{\mathcal{E}} \asymp_{\mathrm{A}} [\![Q]\!]_{\mathcal{E}}.$$

Proof: By applying Lemmas 6.1.8, 6.1.10, 6.1.12, and 6.1.14, we show that the relation

$$\mathcal{S} = \{([\![P]\!]_{\mathcal{E}}\,,\ [\![Q]\!]_{\mathcal{E}}) : P \approx_{\mathcal{E}} Q\}$$

is a $\asymp_{\mathrm{A}}$-bisimulation up-to $\succcurlyeq_{\mathrm{A}}$. The proof is analogous to that of Lemma 3.4.3 except for a few subtles due to the differences pointed out in Remarks 6.1.3(ii) and 6.1.9(ii). These differences are taken into account by Lemma 6.1.12 and Lemma 6.1.14, respectively. □

Theorem 6.1.16 (Soundness and completeness of $\approx_{\mathcal{E}}$) *Let P and Q be two π_{a}-processes. Then:*

1. *$P \approx_{\mathcal{E}} Q$ implies $P \cong_{\pi_{\mathrm{a}}} Q$;*
2. *$P \cong_{\pi_{\mathrm{a}}}$ implies $P \approx_{\mathcal{E}} Q$, for P and Q image-finite.*

Proof: As regards part 1, by Lemma 6.1.15, we have $[\![P]\!]_{\mathcal{E}} \asymp_{\mathrm{A}} [\![Q]\!]_{\mathcal{E}}$. By Lemma 6.1.6 we have $[\![P]\!]_{\mathcal{E}} \cong_{\mathcal{A}\pi} [\![Q]\!]_{\mathcal{E}}$. By Lemma 6.1.11 and the compositionality of $[\![\cdot]\!]_{\mathcal{E}}$ we have $P \cong_{\pi_{\mathrm{a}}} Q$. As regards part 2, the proof is similar to that of Lemma 3.5.1. Due to the differences between synonymous bisimilarity and link bisimilarity (see Remark 6.1.3) we also need Lemma 6.1.12. □

6.1.2 Internal mobility in asynchronous π

In this section we extend the result of Section 4.6 to asynchronous π. Briefly, we show that the encoding $[\![\cdot]\!]_{\mathcal{E}}$ is fully-abstract with respect to barbed congruence. The encoding $[\![\cdot]\!]_{\mathcal{E}}$ goes from π_{a} to $\pi\mathrm{I}_{\mathrm{a}}$, the asynchronous variant of Sangiorgi's $\pi\mathrm{I}$ [77] (c.f. Section 2.5.2)

The lemma below will be crucial for proving the desired full abstraction result.

Lemma 6.1.17 *Let P be a process in π_{a}. Then:*

$$P \approx_{\mathcal{E}} [\![P]\!]_{\mathcal{E}}.$$

Proof: The proof is similar to that of Lemma 4.6.1 for $[\![\cdot]\!]$. □

Theorem 6.1.18 (Full abstraction of $[\![\cdot]\!]_{\mathcal{E}}$) *Let P and Q be two processes in π_a. Then:*

$$P \cong_{\pi_\mathrm{a}} Q \text{ iff } [\![P]\!]_{\mathcal{E}} \cong_{\pi\mathrm{I_a}} [\![Q]\!]_{\mathcal{E}}.$$

Proof: The soundness follows from Lemma 5.3.5 and the compositionality of $[\![\cdot]\!]_{\mathcal{E}}$. As to the completeness, by Lemma 6.1.17 and Theorem 6.1.16 we have $[\![P]\!]_{\mathcal{E}} \cong_{\pi_\mathrm{a}} [\![Q]\!]_{\mathcal{E}}$. Up to bisimilarity, $\pi\mathrm{I_a} \subset \pi_\mathrm{a}$ and therefore $[\![P]\!]_{\mathcal{E}} \cong_{\pi\mathrm{I_a}} [\![Q]\!]_{\mathcal{E}}$. □

The theorem above shows that in asynchronous π-calculus external mobility can be expressed in terms of internal mobility. Note that the encoding introduces divergence, so by no means it should be regarded as an implementation.

6.2 Fusions of names versus equators

In Fusion calculus [65], fusion of names model equivalence relations over names. An obvious encoding from π into Chi, which can be straightforwardly adapted to Fusion calculus, has been suggested in [28]. This encoding justifies the claim that the Fusion calculus inherit all π-calculus' expressive power. It remains to understand whether Fusion is more expressive than π-calculus. In this respect, in [64, 65] Parrow and Victor give a couples of examples, which have no direct counterpart in π-calculus. The examples are essentially the following:

1. In Fusion calculus, the delayed input $a(x)P$ (see Section 4.3) can be easily defined as:

   $$a(x)P \stackrel{\text{def}}{=} (x)(ax \mid P).$$

   As already discussed in Section 4.3, there is no a direct way to encode the delayed input in the (full) π-calculus.
2. The Fusion calculus allows the same name to be received twice. This implies the "fusion" of the received names, as shown by the following example:

   $$(b)((x)axx \mid \overline{a}bc \mid \overline{b}h) \longrightarrow \overline{c}h.$$

   Also in this case there is no a direct counterpart in polyadic π-calculus.

In this chapter we define a fully-abstract encoding of asynchronous monadic Fusion calculus into asynchronous monadic π-calculus. Roughly, the encoding maps fusions of names in terms of equators. Our encoding is proved fully-abstract with respect to *asynchronous closed barbed congruence*, the asynchronous variant of closed barbed congruence defined in Section 5.4. The encoding can be straightforwardly generalised to deal with polyadic processes. So, we prove that asynchronous Fusion calculus is not more expressive than asynchronous π-calculus. Finally, by using our encoding, we give the π-calculus counterpart of the two examples discussed above.

$$(\texttt{comm1}) : (x)(P \,|\, ax.\, Q \,|\, \overline{a}y) \longrightarrow (x)((P \,|\, Q)\{y/x\})$$

$$(\texttt{comm2}) : (y)(P \,|\, ax.\, Q \,|\, \overline{a}y) \longrightarrow (y)((P \,|\, Q)\{x/y\}).$$

Table 6.1: Main reduction rules of the asynchronous monadic Fusion calculus

6.2.1 The asynchronous monadic fusion calculus

In this section we define an asynchronous monadic variant of the Fusion calculus, in short $\mathcal{F}_{\mathrm{a}}$, without matching nor summation operators. The grammar of $\mathcal{F}_{\mathrm{a}}$ has operators of inaction, *unbound input prefix*, asynchronous output, parallel composition, restriction and replication:

$$P ::= \mathbf{0} \;\Big|\; ax.\, P \;\Big|\; \overline{a}b \;\Big|\; P \mid P \;\Big|\; (a)P \;\Big|\; !P$$

Following [86, 28] we give the *reduction semantics* of $\mathcal{F}_{\mathrm{a}}$. As in π-calculus, we have a notion of structural congruence and a reduction relation similar to that of π except for the rule of communication which is replaced by the rules `comm1` and `comm2` in Table 6.1 (we recall that the generalization to the polyadic case is slightly more complex).
In rules `comm1` and `comm2`, the restrictions (x) and (y), in the derivatives, make sense only when $x = y$, otherwise, up-to structural congruence, they disappear.

For avoiding heavy notations we use symbols $\longrightarrow$, $\Longrightarrow$, $\downarrow_a$, $\Downarrow_a$, $\dot{\approx}$ both for π_{a} and $\mathcal{F}_{\mathrm{a}}$ processes.

In Section 5.4, we have introduced the notion of closed barbed congruence as an appropriated behavioural equality for the Fusion calculus. Since we are considering asynchronous Fusion calculus, we give here the asynchronous variant of closed barbed congruence.

Definition 6.2.1 (Asynchronous closed barbed congruence) *Two processes P and Q in a set of processes $\mathcal{L}$ are* asynchronous closed barbed congruent, *written $\cong^{\mathrm{c}}_{\mathcal{L}}$, if for all contexts $C[\cdot]$ such that:*

1. $\mathrm{fn}(P) \cap \mathrm{fn}(C[P]) = \mathrm{fn}(Q) \cap \mathrm{fn}(C[Q]) = \emptyset$, *and*
2. *all free names in $C[\cdot]$ appear only in output subject position;*

it holds that $C[P] \dot{\approx} C[Q]$.

We add the clause 2 because, as already pointed out, only output actions are observable in asynchronous calculi.

$\langle\!\lvert P_1 \mid P_2 \rvert\!\rangle \stackrel{\text{def}}{=} \langle\!\lvert P_1 \rvert\!\rangle \mid \langle\!\lvert P_2 \rvert\!\rangle$	$\langle\!\lvert (a)P \rvert\!\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu} a)\langle\!\lvert P \rvert\!\rangle$
$\langle\!\lvert !P \rvert\!\rangle \stackrel{\text{def}}{=} !\langle\!\lvert P \rvert\!\rangle$	$\langle\!\lvert \overline{a}b \rvert\!\rangle \stackrel{\text{def}}{=} \overline{a}b$
$\langle\!\lvert ax.P \rvert\!\rangle \stackrel{\text{def}}{=} a(z).\,(\mathcal{E}(z,x) \mid \langle\!\lvert P \rvert\!\rangle) \quad z \notin \text{fn}(P)$	$\langle\!\lvert \mathbf{0} \rvert\!\rangle \stackrel{\text{def}}{=} \mathbf{0}$

Table 6.2: The encoding of fusions

6.2.2 The encoding

The encoding of $\mathcal{F}_\text{a}$ into π_a, denoted by $\langle\!\lvert \cdot \rvert\!\rangle$, is formally defined in Table 6.2. Lemma 6.2.2 gives some trivial (but useful) properties of $\langle\!\lvert \cdot \rvert\!\rangle$.

Lemma 6.2.2 *Let P be a process in $\mathcal{F}_\text{a}$. Then:*

1. $\text{fn}(P) = \text{fn}(\langle\!\lvert P \rvert\!\rangle)$
2. $\langle\!\lvert P\{y/x\} \rvert\!\rangle = \langle\!\lvert P \rvert\!\rangle\{y/x\}$
3. $P \equiv Q$ *implies* $\langle\!\lvert P \rvert\!\rangle \equiv \langle\!\lvert Q \rvert\!\rangle$.

Proof: Parts 1 and 2 are proved by structural induction. Part 3 is proved by case analysis on the rules defining structural congruence. □

We rewrite Lemma 6.1.1(1) and Lemma 6.1.12 in terms of $\lesssim_\varepsilon$, where $\lesssim_\varepsilon$ denotes the expansion variant of $\approx_\varepsilon$.

Lemma 6.2.3

1. $\mathcal{E}(a,a) \gtrsim_\varepsilon \mathbf{0}$;
2. $(\boldsymbol{\nu} b)(\mathcal{E}(a,b) \mid \mathcal{E}(b,c)) \gtrsim_\varepsilon \mathcal{E}(a,c)$;
3. $(\boldsymbol{\nu} a)(P \mid \mathcal{E}(a,b)) \gtrsim_\varepsilon P\{b/a\}$.

Lemma 6.2.4 (Operational correspondence) *Let P be a process in $\mathcal{F}_\text{a}$ in which all free names appear only in output subject position. Then:*

1. *If $P \longrightarrow P'$ then there is $P_1 \in \pi_\text{a}$ such that $\langle\!\lvert P \rvert\!\rangle \longrightarrow P_1 \gtrsim_\varepsilon \langle\!\lvert P' \rvert\!\rangle$.*
2. *If $\langle\!\lvert P \rvert\!\rangle \longrightarrow P_1$ then there is $P' \in \mathcal{F}_\text{a}$ s.t. $P \longrightarrow P'$ and $P_1 \gtrsim_\varepsilon \langle\!\lvert P' \rvert\!\rangle$.*
3. $P \downarrow_a$ iff $\langle\!\lvert P \rvert\!\rangle \downarrow_a$.

Proof:

1. The proof is by transition induction. The only non-trivial case is when the last rule applied for deriving $P \longrightarrow P'$ is either `comm1` or `comm2`. We recall that all free names in P appear only in output subject position. There are three cases:

(a) $P \equiv (\widetilde{w})(y)(x)(Q \mid ax.\,R \mid \overline{a}y)$ where $x \neq y$, $a \in \widetilde{w}$ and by rule `comm1` $P' \equiv (\widetilde{w})(y)((Q \mid R)\{y/x\})$. Then by Lemma 6.2.2(3):

$$\begin{aligned}
\langle\!| P |\!\rangle \equiv\ & (\boldsymbol{\nu}\widetilde{w})(\boldsymbol{\nu}y)(\boldsymbol{\nu}x)(\langle\!| Q |\!\rangle \mid a(z).\,(\mathcal{E}(z,x) \mid \langle\!| R |\!\rangle) \mid \overline{a}y) \\
\longrightarrow\ & (\boldsymbol{\nu}\widetilde{w})(\boldsymbol{\nu}y)(\boldsymbol{\nu}x)(\langle\!| Q |\!\rangle \mid \mathcal{E}(y,x) \mid \langle\!| R |\!\rangle) \\
\gtrsim_{\varepsilon}\ & (\boldsymbol{\nu}\widetilde{w})(\boldsymbol{\nu}y)((\langle\!| Q |\!\rangle \mid \langle\!| R |\!\rangle)\{y/x\}) \qquad \text{(by Lemma 6.2.3(3))} \\
\equiv\ & \langle\!| P' |\!\rangle. \qquad \text{(by Lemmas 6.2.2(2) and 6.2.2(3))}
\end{aligned}$$

(b) $P \equiv (\widetilde{w})(x)(y)(Q \mid ax.\,R \mid \overline{a}y)$ where $x \neq y$, $a \in \widetilde{w}$ and by rule `comm2` $P' \equiv (\widetilde{w})(x)((Q \mid R)\{x/y\})$. This case is similar to the previous one.

(c) $P \equiv (\widetilde{w})(x)(Q \mid ax.\,R \mid \overline{a}x)$ where $a \in \widetilde{w}$ and by rule `comm1` (or `comm2`) $P' \equiv (\widetilde{w})(x)(Q \mid R)$. Then by Lemma 6.2.2(3)

$$\begin{aligned}
\langle\!| P |\!\rangle \equiv\ & (\boldsymbol{\nu}\widetilde{w})(\boldsymbol{\nu}x)(\langle\!| Q |\!\rangle \mid a(z).\,(\mathcal{E}(z,x) \mid \langle\!| R |\!\rangle) \mid \overline{a}x) \\
\longrightarrow\ & (\boldsymbol{\nu}\widetilde{w})(\boldsymbol{\nu}x)(\langle\!| Q |\!\rangle \mid \mathcal{E}(x,x) \mid \langle\!| R |\!\rangle) \\
\gtrsim_{\varepsilon}\ & (\boldsymbol{\nu}\widetilde{w})(\boldsymbol{\nu}x)(\langle\!| Q |\!\rangle \mid \langle\!| R |\!\rangle) \qquad \text{(Lemma 6.2.3(1))} \\
\equiv\ & \langle\!| P' |\!\rangle.
\end{aligned}$$

2. The proof is by transition induction of the rule applied for deriving $\langle\!| P |\!\rangle \longrightarrow P_1$. The proof is similar to that of part 1. Notice that we need the hypothesis on the free names appearing in P to ensure that there exists P' such that $P \longrightarrow P'$.
3. It can be easily proved by structural induction.

□

We need a weak version of the above operational correspondence.

Lemma 6.2.5 *Let P be a process in $\mathcal{F}_{\mathrm{a}}$ in which all free names appear only in output subject position. Then:*

1. *If $P \Longrightarrow P'$ then there is $P_1 \in \pi_{\mathrm{a}}$ such that $\langle\!| P |\!\rangle \Longrightarrow P_1 \gtrsim_{\varepsilon} \langle\!| P' |\!\rangle$.*
2. *If $\langle\!| P |\!\rangle \Longrightarrow P_1$ then there is $P' \in \mathcal{F}_{\mathrm{a}}$ s.t. $P \Longrightarrow P'$ and $P_1 \gtrsim_{\varepsilon} \langle\!| P' |\!\rangle$.*
3. *$P \Downarrow_a$ iff $\langle\!| P |\!\rangle \Downarrow_a$.*

Lemma 6.2.6 *Let P and Q be two processes in $\mathcal{F}_{\mathrm{a}}$ in which all free names appear only in output subject position. Then:*

$$P \stackrel{\bullet}{\approx} Q \ \text{ iff } \ \langle\!| P |\!\rangle \stackrel{\bullet}{\approx} \langle\!| Q |\!\rangle.$$

Proof: In the implication from left to right, we use Lemma 6.2.5 to prove that the relation $\mathcal{R} = \{(\langle\!| P |\!\rangle, \langle\!| Q |\!\rangle) : P \stackrel{\bullet}{\approx} Q\}$ is a barbed bisimulation up to $\gtrsim_{\varepsilon}$. In the implication from right to left, we use Lemma 6.2.5 and the fact that $\lesssim_{\varepsilon} \stackrel{\bullet}{\approx} \gtrsim_{\varepsilon} \subset \stackrel{\bullet}{\approx}$ to prove that the relation $\mathcal{R} = \{(P, Q) : \langle\!| P |\!\rangle \stackrel{\bullet}{\approx} \langle\!| Q |\!\rangle\}$ is a barbed bisimulation. □

The following lemma is crucial for proving the *completeness* of our encoding.

Lemma 6.2.7 *If $P \in \pi_{\mathrm{a}}$ then there exists $\widehat{P} \in \mathcal{F}_{\mathrm{a}}$ such that $\langle\!| \widehat{P} |\!\rangle \gtrsim_{\varepsilon} P$.*

Proof: By structural induction. The most interesting case is when $P \equiv a(x).\, P_1$. We choose $\widehat{P} \stackrel{\text{def}}{=} (x)ax.\, \widehat{P_1}$ where, by inductive hypothesis, $\langle\!|\, \widehat{P_1} \,|\!\rangle \gtrsim_\varepsilon P_1$. Then:

$$\begin{array}{lll} \langle\!|\, \widehat{P} \,|\!\rangle & = (\boldsymbol{\nu} x)(a(z).\,(\mathcal{E}(z,x) \mid \langle\!|\, \widehat{P_1} \,|\!\rangle)) & \text{(with } z \notin \mathrm{fn}(\widehat{P_1})\text{)} \\ & \equiv (\boldsymbol{\nu} z)(a(x).\,(\mathcal{E}(z,x) \mid \langle\!|\, \widehat{P_1} \,|\!\rangle\{z/x\})) & \text{(alpha-conversion)} \\ & \gtrsim_\varepsilon a(x).\,(\boldsymbol{\nu} z)(\mathcal{E}(z,x) \mid \langle\!|\, \widehat{P_1} \,|\!\rangle\{z/x\}) & \text{(easy law for } \gtrsim_\varepsilon\text{)} \\ & \gtrsim_\varepsilon a(x).\, \langle\!|\, \widehat{P_1} \,|\!\rangle & \text{(Lemma 6.2.3(3))} \\ & \gtrsim_\varepsilon a(x).\, P_1 & \text{(inductive hypothesis)} \\ & \equiv P. & \end{array}$$

□

Theorem 6.2.8 (Full-abstraction of $\langle\!|\cdot|\!\rangle$) *Let P and Q be two processes in $\mathcal{F}_\mathrm{a}$. Then:*

$$P \cong^{\mathrm{c}}_{\mathcal{F}_\mathrm{a}} Q \ \text{ iff } \ \langle\!|\, P \,|\!\rangle \cong^{\mathrm{c}}_{\pi_\mathrm{a}} \langle\!|\, Q \,|\!\rangle.$$

Proof: The soundness part follows from Lemma 6.2.6 and compositionality of $\langle\!|\cdot|\!\rangle$. As to the completeness part, let $C[\cdot]$ be a closing context for $\langle\!|\, P \,|\!\rangle$ and $\langle\!|\, Q \,|\!\rangle$ in π_a. We want to prove that $C[\langle\!|\, P \,|\!\rangle] \stackrel{\bullet}{\approx} C[\langle\!|\, Q \,|\!\rangle]$. By Lemma 6.2.7 and the compositionality of $\langle\!|\cdot|\!\rangle$ there exists a closing context $\widehat{C}[\cdot]$ for processes P and Q such that for every process R it holds that $\langle\!|\, \widehat{C}[R] \,|\!\rangle \gtrsim_\varepsilon C[\langle\!|\, R \,|\!\rangle]$. By hypothesis, $\widehat{C}[P] \stackrel{\bullet}{\approx} \widehat{C}[Q]$. By Lemma 6.2.6 we have $\langle\!|\, \widehat{C}[P] \,|\!\rangle \stackrel{\bullet}{\approx} \langle\!|\, \widehat{C}[Q] \,|\!\rangle$ and therefore $C[\langle\!|\, P \,|\!\rangle] \lesssim_\varepsilon \langle\!|\, \widehat{C}[P] \,|\!\rangle \stackrel{\bullet}{\approx} \langle\!|\, \widehat{C}[Q] \,|\!\rangle \gtrsim_\varepsilon C[\langle\!|\, Q \,|\!\rangle]$. Since $\lesssim_\varepsilon \stackrel{\bullet}{\approx} \gtrsim_\varepsilon \ \subset \ \stackrel{\bullet}{\approx}$ we have the thesis. □

Like the encoding $[\![\cdot]\!]_\varepsilon$ in Section 6.1.2, the encoding $\langle\!|\cdot|\!\rangle$ is based on equators and therefore introduces divergence. Moreover, the result above does not hold when adding to the calculus synchronous outputs or matching. Both operators would destroy the key result in Lemma 6.2.3(3).

6.2.3 Two applications of the encoding

We show here an application of the expressive power of equators. We apply the encoding $\langle\!|\cdot|\!\rangle$ to give the π-calculus counterpart of the two examples in Fusion discussed Section 6.2.

By applying $\langle\!|\cdot|\!\rangle$ to $(x)(ax \mid P)$ we obtain the following definition for the *delayed input* in π_a:

$$a(x)P \stackrel{\text{def}}{=} (\boldsymbol{\nu} x)(a(z).\, \mathcal{E}(z,x) \mid P) \qquad \text{where } \ z \notin \mathrm{fn}(P)$$

A similar encoding has already been suggested in [92]. Notice the similarity with the encoding for delayed input given in Section 4.3.

The "fusion" of different names in

$$(b)((x)axx \mid \overline{a}bc \mid \overline{b}h) \longrightarrow \overline{c}h$$

can be expressed in polyadic π_a by using the straightforward extension of our encoding to polyadic Fusion:

$$
\begin{array}{rcll}
\langle\!|\, (b)((x)axx \mid \overline{a}bc \mid \overline{b}h) \,|\!\rangle & = & (\boldsymbol{\nu} b)((\boldsymbol{\nu} x)a(z,y).\,(\mathcal{E}(z,x) \mid \mathcal{E}(y,x)) \mid \overline{a}bc \mid \overline{b}h) & \\
& \longrightarrow & (\boldsymbol{\nu} b)((\boldsymbol{\nu} x)(\mathcal{E}(b,x) \mid \mathcal{E}(c,x)) \mid \overline{b}h) & \\
& \gtrsim_{\varepsilon} & (\boldsymbol{\nu} b)(\mathcal{E}(b,c) \mid \overline{b}h) & \text{(by Lemma 6.2.3(2))} \\
& \gtrsim_{\varepsilon} & \overline{c}h & \text{(by Lemma 6.2.3(3)).}
\end{array}
$$

Chapter 7

Migrating objects as mobile processes

In this chapter we study *object migration* as modelled in Cardelli's distributed object based programming language Obliq [18].

In Section 7.1, we introduce the calculus $\mathrm{L}\pi^+$, a typed extension of $\mathrm{L}\pi$. In section 7.2, we introduce *Øjeblik*, a distribution-free abstraction of Obliq. In Øjeblik, object migration is replaced by its abstraction *object surrogation*. In Section 7.3, we explain how object aliases behave in Obliq/Øjeblik. In Section 7.4, we give a translational semantics of Øjeblik into $\mathrm{L}\pi^+$ according to the original informal semantics of Obliq [18]. In Section 7.5, we define a program equivalence for Øjeblik based on may-convergence of π-processes. In Section 7.6, the correctness of object surrogation is formalised as an equation called the *safety equation*. In Section 7.7, we show that the semantics given in Section 7.4 does not respect the safety equation. Similar counterexamples apply to object migration in Obliq, as we have verified using the Obliq interpreter [17]. In Section 7.8, we propose an amended variant of the semantics given in Section 7.4. In Sections 7.9, we give a few properties of the translational semantics exploiting the $\mathrm{L}\pi^+$ constraint on the output capability. In Section 7.10, we characterise a wide class of Øjeblik programs for which we want to prove the safety of object surrogation. In Section 7.11, we give a crucial result on the amended semantics, which represents the essence of the safety of object surrogation. Finally, in Section 7.12, we prove that the amended semantics satisfies the safety equation for the class of Øjeblik programs characterised in Section 7.10.

7.1 The calculus $\mathrm{L}\pi^+$

7.1.1 Terms and types

In Table 7.1, we give a typed version of polyadic $\mathrm{L}\pi$ with: (i) labelled values ℓ_v, called *variants* [75], with case analysis; (ii) tuple values $\langle\, v_1 \ldots v_n \,\rangle$, with pattern matching, (iii) constants k, called *keys*, with equality; (iv) a `wrong` construct to model run time errors.

We introduce a few syntactic categories: the set $\mathbf{X}$ of *variables* includes the set $\mathbf{N}$ of *names* (constants and variables) consisting of the two disjoint sets $\mathbf{C}$ of *channels* and $\mathbf{K}$

Channels: $c \in \mathbf{C}$	*Values*	
Keys: $k \in \mathbf{K}$	$v ::= x$	variable
Names: $\in \mathbf{N}$	$\mid \ell\_v$	variant
$n ::= c \mid k$	$\mid \langle v_1 .. v_n \rangle$	tuple
Auxiliary: $u \in \mathbf{U}$	*Types*	
Variables: $\in \mathbf{X}$	$T ::= \mathbf{C}(T)$	channel type
$x ::= n \mid u$	$\mid \mathbf{K}$	key type
	$\mid [\,\ell_1{:}T_1\,; \ldots ; \ell_m{:}T_m\,]$	variant type
Labels $\in \mathbf{L}$	$\mid \langle\, T_1 \,..\, T_m \,\rangle$	tuple type
$\ell, \ell_1, \ell_2, \ldots$	$\mid X$	type variable
	$\mid \mu X . T$	recursive type

Processes	
$P ::= \mathbf{0}$	nil process
$\mid c(x).P$	single *input*
$\mid \overline{c}v$	output
$\mid P_1 \mid P_2$	parallel
$\mid (\boldsymbol{\nu} n{:}T)\, P$	restriction
$\mid \mathbf{!}\, c(x).P$	replicated *input*
$\mid \mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P\ \mathsf{else}\ Q$	key testing
$\mid \mathsf{case}\ v\ \mathsf{of}\ \ell_1\_(x_1) \triangleright P_1\,; \ldots ; \ell_m\_(x_m) \triangleright P_m$	variant *destructor*
$\mid \mathsf{let}\ (\,x_1 \,..\, x_m\,) = v\ \mathsf{in}\ P$	tuple *destructor*
$\mid \mathsf{wrong}$	run time error

The *locality constraint* requires that in (*single* and *replicated*) *inputs* and in (*variant* and *tuple*) *destructors* the bound names $x, x_1, \ldots, x_m$ must not be used in free input position within the respective scope $P, P_1, \ldots, P_m$.

Table 7.1: The calculus $\mathrm{L}\pi^+$

of *keys*. The auxiliary variables in the set **U** are variables for complex values. **L** is the set of *labels*. In addition to the metavariables mentioned in the grammar, we let s, p, q, r, m, t range over channels, y over variables, w over values, Q over processes, and i, j, d, h, m over tuple, variant, or other indices. We abbreviate $\ell\_\langle\rangle$ and $\ell\_()$ as ℓ, as well as $\overline{q}\langle\rangle$ and $q().P$ as $\overline{q}$ and $q.P$, respectively, while $\widetilde{v}$ denotes a sequence $v_1 \mathinner{..} v_m$.

Restriction, both inputs, and both destructors are *binders* for the names $x, x_1, \dots, x_m$ in the respective scopes $P, P_1, \dots, P_m$. We assume the usual definitions of free and bound occurrences of names, based on these binders; the inductively defined functions $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$ denote those of process P. Similarly, $\mathrm{fc}(P)$ and $\mathrm{bc}(P)$ denote the free and bound channels of process P. Moreover, $\mathrm{n}(P)\!=\mathrm{fn}(P) \cup \mathrm{bn}(P)$ and $\mathrm{c}(P)\!=\!\mathrm{fc}(P) \cup \mathrm{bc}(P)$. *Substitutions*, ranged over by σ, are type-preserving functions from variables to values (types are introduced below). For an expression e, $e\sigma$ is the result of applying σ to e, with the usual renaming to avoid captures. *Relabellings*, ranged over by ρ, permit replacing a label ℓ with another label ℓ'. We denote such a relabelling with $[^{\ell'}/_{\ell}]$. The application of a relabelling to a term is defined thus:

- $(\ell\_v)[^{\ell'}/_{\ell}] := \ell'\_v[^{\ell'}/_{\ell}]$,
- $(\ell''\_v)[^{\ell'}/_{\ell}] := \ell''\_v[^{\ell'}/_{\ell}]$ if $\ell''\neq\ell$.
- $x\rho := x$,
- $\mathsf{wrong}\rho := \mathsf{wrong}$,
- $((\boldsymbol{\nu} n{:}T)\, P)\rho := (\boldsymbol{\nu} n{:}T\rho)\, P\rho$,
- the application $(\mathsf{case}\ v\ \mathsf{of}\ \ell_1\_(x_1){\triangleright}P_1\,;\dots;\ell_n\_(x_n){\triangleright}P_n)\rho$ give rises to the process $\mathsf{case}\ v\rho\ \mathsf{of}\ \ell_1\_(x_1){\triangleright}(P_1\rho)\,;\dots;\ell_n\_(x_n){\triangleright}(P_n\rho)$.

For the remaining (value and process) constructors, relabellings act as simple homomorphism. Substitution and relabelling have the highest operator precedence, parallel composition the lowest.

To rearrange processes we use the following notion of *structural equivalence* that is extended to deal with if-, case-, and let-constructs.

Definition 7.1.1 Structural equivalence, *written* $\equiv$, *is the smallest relation preserved by parallel composition and restriction, which satisfies the axioms below:*

- $P \equiv Q$, *if* P *is* α*-convertible to* Q;
- $P \mid \mathbf{0} \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$;
- $(\boldsymbol{\nu} n{:}T)\, \mathbf{0} \equiv \mathbf{0}$, $(\boldsymbol{\nu} n_1{:}T_1)\,(\boldsymbol{\nu} n_2{:}T_2)\, P \equiv (\boldsymbol{\nu} n_2{:}T_2)\,(\boldsymbol{\nu} n_1{:}T_1)\, P$, *if* $n_1 \neq n_2$;
- $(\boldsymbol{\nu} n{:}T)\,(P \mid Q) \equiv P \mid (\boldsymbol{\nu} n{:}T)\, Q$, *if* $n \notin \mathrm{fn}(P)$;
- $\mathsf{if}\ [k_1{=}k_1]\ \mathsf{then}\ P\ \mathsf{else}\ Q \ \equiv\ P$;
- $\mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P\ \mathsf{else}\ Q \ \equiv\ Q$, *if* $k_1{\neq}k$;
- $\mathsf{case}\ \ell_j\_v_j\ \mathsf{of}\ \ell_1\_(x_1){\triangleright}P_1\,;\dots;\ell_j\_(x_j){\triangleright}P_j\,;\dots;\ell_m\_(x_m){\triangleright}P_m \equiv P_j\{^{v_j}/_{x_j}\}$;
- $\mathsf{case}\ v\ \mathsf{of}\ \ell_1\_(x_1){\triangleright}P_1\,;\dots;\ell_m\_(x_m){\triangleright}P_m \equiv \mathsf{wrong}$, *if* $v{\neq}\ell_j\_v_j$ *for some* $j{\in}\ 1 \mathinner{..} m$ *and any value* v_j;
- $\mathsf{let}\ (\, x_1 \mathinner{..} x_m\,) = \langle\, v_1 \mathinner{..} v_m\,\rangle\ \mathsf{in}\ P \equiv P\{^{\widetilde{v}}/_{\widetilde{x}}\}$;
- $\mathsf{let}\ (\, x_1 \mathinner{..} x_m\,) = v\ \mathsf{in}\ P \equiv \mathsf{wrong}$, *if* $v{\neq}\langle\, v_1 \mathinner{..} v_m\,\rangle$ *for some values* $v_1 \mathinner{..} v_m$.

In Table 7.2 we give the *typing rules* for values and processes. Abusing the notation for sets of names and the corresponding types, we use $\mathbf{K}$ and $\mathbf{C}$ also as type constructors, where channel types are parameterised over the type of value they carry. For variants and tuples we use standard notations (c.f. [75]). In a recursive type $\mu X.T$, occurrences of variable X in type T must be guarded, i.e., underneath variant, tuple, or channel constructors. We often omit the type annotation of restriction, when it is clear from the context or not important for the discussion. A *type environment* Γ is a finite mapping from variables to types. A *typing judgement* $\Gamma \vdash P$ asserts that process P is well-typed in Γ, and $\Gamma \vdash v{:}T$ that value v has type T in Γ. We say that a type environment Γ is *closed* if all names mentioned in Γ are of type channel $\mathbf{C}(T)$ or of type key $\mathbf{K}$. We only consider *closed terms*, i.e. terms which are well-typed in some closed typing Γ.

As expected, the typing in Table 7.2 satisfies all basic fundamental properties of type environments such as: *weakening*, *contraction*, *substitution*, and *narrowing*.

7.1.2 Operational and behavioural semantics

Labelled transitions are of the form $P \xrightarrow{\mu} P'$, where *action* μ is: τ (interaction), cv (free input), $(\boldsymbol{\nu}\widetilde{n}{:}\widetilde{T})\,\overline{c}v$ (output at c of value v containing private names $\widetilde{n}$ of type $\widetilde{T}$, which we often omit), where c is the *subject* and v the *object*. The functions $\mathrm{fn}(\cdot)$, $\mathrm{bn}(\cdot)$, $\mathrm{n}(\cdot)$, $\mathrm{fc}(\cdot)$, $\mathrm{bc}(\cdot)$, and $\mathrm{c}(\cdot)$ are extended to actions as usual. Table 7.3 presents the transition rules for our π-calculus in an *early* style; the symmetric rules of (COM$_1$) and (PAR$_1$) are omitted. Relation $\Rightarrow$ is the reflexive-transitive closure of $\xrightarrow{\tau}$; $\stackrel{\mu}{\Longrightarrow}$ denotes $\Rightarrow \xrightarrow{\mu} \Rightarrow$; $\stackrel{\hat{\mu}}{\Longrightarrow}$ denotes $\stackrel{\mu}{\Longrightarrow}$ if $\mu \neq \tau$, and $\Rightarrow$ if $\mu = \tau$. For any relation $\mathcal{R}$ on processes, $\xrightarrow{\tau}_{\mathcal{R}}$ denotes $\xrightarrow{\tau}\mathcal{R}$, and $\Rightarrow_{\mathcal{R}}$ the reflexive-transitive closure of $\xrightarrow{\tau}_{\mathcal{R}}$.

The typing in Table 7.2 is preserved under silent moves.

Theorem 7.1.2 (Subject Reduction) *Let Γ be a closed type environment.*

1. *If $\Gamma \vdash P$ then $P \not\equiv (\boldsymbol{\nu}\widetilde{n})\,(P' \mid \mathsf{wrong})$.*
2. *If $\Gamma \vdash P$ and $P \Rightarrow Q$, then $\Gamma \vdash Q$.*

The proof of the above result is standard (see for instance [75, 42]).

As regards behavioural equivalences, we focus on *typed barbed relations*. Notice that in a *typed* scenario, only well-typed context are considered when working with contextual equivalences. We also need the standard notion of *extension* on type environments and generalise it to *closed extension*.

Definition 7.1.3 *Let Γ and Δ be two type environments. We say that Γ* extends *Δ if $\mathtt{dom}(\Delta) \subseteq \mathtt{dom}(\Gamma)$ and $\Gamma \vdash n{:}\Delta(n)$ for all names n on which Δ is defined. We say that Γ is a* closed extension *of Δ if Γ is closed and extends Δ.*

$$(\text{T-BAS})\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

$$(\text{T-REC1})\ \frac{\Gamma \vdash v{:}T\{{}^{\mu X.T}\!/\!_X\}}{\Gamma \vdash v{:}\mu X.\, T} \qquad (\text{T-REC2})\ \frac{\Gamma \vdash v{:}\mu X.\, T}{\Gamma \vdash v{:}T\{{}^{\mu X.T}\!/\!_X\}}$$

$$(\text{T-VAR})\ \frac{\Gamma \vdash v{:}T}{\Gamma \vdash \ell\_v : [\,\dots ; \ell{:}T\,; \dots\,]} \qquad (\text{T-TUP})\ \frac{\Gamma \vdash v_i{:}T_i \quad \forall i \in 1\,..\,m}{\Gamma \vdash \langle\, v_1\,..\,v_m\,\rangle : \langle\, T_1\,..\,T_m\,\rangle}$$

$$(\text{T-RES1})\ \frac{\Gamma, c{:}\mathbf{C}(T) \vdash P}{\Gamma \vdash (\boldsymbol{\nu} c{:}\mathbf{C}(T))\, P} \qquad (\text{T-RES2})\ \frac{\Gamma, k{:}\mathbf{K} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} k{:}\mathbf{K})\, P}$$

$$(\text{T-PAR})\ \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \mid P_2} \qquad (\text{T-REP})\ \frac{\Gamma \vdash P}{\Gamma \vdash \mathbf{!}\, P}$$

$$(\text{T-INP})\ \frac{\Gamma \vdash c{:}\mathbf{C}(T) \quad \Gamma, x{:}T \vdash P}{\Gamma \vdash c(x).P} \qquad (\text{T-OUT})\ \frac{\Gamma \vdash c{:}\mathbf{C}(T) \quad \Gamma \vdash v{:}T}{\Gamma \vdash \overline{c}v}$$

$$(\text{T-NIL})\ \frac{-}{\Gamma \vdash \mathbf{0}} \qquad (\text{T-IF})\ \frac{\Gamma \vdash k, k_1 : \mathbf{K} \qquad \Gamma \vdash P, Q}{\Gamma \vdash \mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

$$(\text{T-LET})\ \frac{\Gamma \vdash v : \langle\, T_1\,..\,T_m\,\rangle \qquad \Gamma, x_1{:}T_1, \dots, x_m{:}T_m \vdash P}{\Gamma \vdash \mathsf{let}\ (\, x_1\,..\,x_m\,) = v\ \mathsf{in}\ P}$$

$$(\text{T-CASE})\ \frac{\Gamma \vdash v : [\,\ell_1{:}T_1\,; \dots ; \ell_m{:}T_m\,] \qquad \Gamma, x_i{:}T_i \vdash P_i \quad \forall i \in 1\,..\,m}{\Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ \ell_1\_(x_1) \triangleright P_1\,; \dots ; \ell_m\_(x_m) \triangleright P_m}$$

Table 7.2: Typing for values and processes

$$(\textsc{Inp})\ \frac{-}{c(x).P \xrightarrow{cv} P\{{}^{v}\!/_{x}\}} \qquad (\textsc{Rep})\ \frac{-}{!\,c(x).P \xrightarrow{cv} P\{{}^{v}\!/_{x}\} \mid !\,c(x).P}$$

$$(\textsc{Out})\ \frac{-}{\overline{c}v \xrightarrow{\overline{c}v} \mathbf{0}} \qquad (\textsc{Open})\ \frac{P \xrightarrow{(\boldsymbol{\nu}\widetilde{q}:\widetilde{T})\,\overline{c}v} P' \quad n \in \mathrm{n}(v) \setminus \{\widetilde{q}, c\}}{(\boldsymbol{\nu}n{:}T)\ P \xrightarrow{(\boldsymbol{\nu}n:T,\widetilde{q}:\widetilde{T})\,\overline{c}v} P'}$$

$$(\textsc{Com}_1)\ \frac{P_1 \xrightarrow{(\boldsymbol{\nu}\widetilde{q}:\widetilde{T})\,\overline{c}v} P_1' \qquad P_2 \xrightarrow{cv} P_2' \qquad \widetilde{q} \cap \mathrm{fn}(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{q}{:}\widetilde{T})\ (P_1' \mid P_2')}$$

$$(\textsc{Par}_1)\ \frac{P_1 \xrightarrow{\mu} P_1' \qquad \mathrm{bn}(\mu) \cap \mathrm{fn}(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2}$$

$$(\textsc{Res})\ \frac{P \xrightarrow{\mu} P' \qquad n \notin \mathrm{n}(\mu)}{(\boldsymbol{\nu}n{:}T)\ P \xrightarrow{\mu} (\boldsymbol{\nu}n{:}T)\ P'}$$

$$(\textsc{Test-1})\ \frac{P \xrightarrow{\mu} P'}{\textsf{if } [k{=}k] \textsf{ then } P \textsf{ else } Q \xrightarrow{\mu} P'}$$

$$(\textsc{Test-2})\ \frac{Q \xrightarrow{\mu} Q' \qquad k_1 \neq k}{\textsf{if } [k{=}k_1] \textsf{ then } P \textsf{ else } Q \xrightarrow{\mu} Q'}$$

$$(\textsc{Case})\ \frac{P_j\{{}^{v}\!/_{x_j}\} \xrightarrow{\mu} Q \qquad j \in 1\,..\,m}{\textsf{case } \ell_j\_v \textsf{ of } \ell_1\_(x_1) \triangleright P_1\,; \ldots ; \ell_m\_(x_m) \triangleright P_m \xrightarrow{\mu} Q}$$

$$(\textsc{Let})\ \frac{P\{{}^{v_1\,..\,v_m}\!/_{x_1\,..\,x_m}\} \xrightarrow{\mu} Q}{\textsf{let } (\,x_1\,..\,x_m\,) = \langle\, v_1\,..\,v_m \,\rangle \textsf{ in } P \xrightarrow{\mu} Q}$$

Table 7.3: Labelled Transition System for $\mathrm{L}\pi^+$.

Definition 7.1.4 *Let Γ and Δ be two type environments. We say that $C[\cdot]$ is a (Δ/Γ)-context if $\Delta \vdash C[\cdot]$ is a valid type judgement when the hole $[\cdot]$ is considered as a process and the following typing rule for $[\cdot]$ is added:*

$$(\text{T-HOLE}) \ \frac{\Theta \ \text{extends} \ \Gamma}{\Theta \vdash [\cdot]}$$

(in the rule, Γ is one of the given type environments and Θ is a metavariable over type environments)

We recall that $\dot{\approx}$ denotes (asynchronous) barbed bisimulation (c.f. Definition 2.5.2).

Definition 7.1.5 (Typed barbed relations) *Let Γ be a typing, and P and Q two processes such that $\Gamma \vdash P, Q$. We say that P and Q are* barbed Γ-equivalent, *written $P \simeq_\Gamma Q$, if for each closed type environment Δ and static (Δ/Γ)-context $C[\cdot]$, we have $C[P] \dot{\approx} C[Q]$. We say that P and Q are* barbed Γ-congruent, *written $P \cong_\Gamma Q$, if for each closed type environment Δ and (Δ/Γ)-context $C[\cdot]$, we have $C[P] \dot{\approx} C[Q]$.*

Labelled characterisations for typed barbed relations must take into accounts types. A *typed relation* is a set of triples $(\Delta; P; Q)$ where Δ is a closed typing and $\Delta \vdash P, Q$. We adapt the definition of asynchronous bisimilarity to Lπ^+-processes.

Definition 7.1.6 (Typed asynchronous bisimilarity) Typed asynchronous bisimilarity, *is the largest typed relation $\mathcal{S}$ such that $(\Delta; P; Q) \in \mathcal{S}$ implies:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' s.t. $Q \Rightarrow Q'$ and $(\Delta; P'; Q') \in \mathcal{S}$.*
2. *If $P \xrightarrow{(\boldsymbol{\nu}\widetilde{n}:\widetilde{T})\,\overline{c}v} P'$, with $\widetilde{n} \cap \mathrm{fn}(Q) = \emptyset$, then there exists Q' such that $Q \xRightarrow{(\boldsymbol{\nu}\widetilde{n}:\widetilde{T})\,\overline{c}v} Q'$ and $((\Delta, \widetilde{n}:\widetilde{T}); P'; Q') \in \mathcal{S}$.*
3. *If*
   - *(a) Γ is a closed extension of Δ,*
   - *(b) $\Gamma \vdash c{:}\mathbf{C}(T)$ and $\Gamma \vdash v{:}T$,*
   - *(c) $P \xrightarrow{cv} P'$, with $\mathrm{fc}(v) \cap \mathrm{fc}(P \mid Q) = \emptyset$,*

   then there exists Q' such that:
   - *(a) either $Q \xRightarrow{cv} Q'$ and $(\Gamma; P'; Q') \in \mathcal{S}$,*
   - *(b) or $Q \Rightarrow Q'$ and $(\Gamma; P'; (\,Q' \mid \overline{c}v\,)) \in \mathcal{S}$.*

Let Γ be a closed typing with $\Gamma \vdash P, Q$. We say that P and Q are typed asynchronously bisimilar at Γ, *written $P \approx_\Gamma Q$, if $(\Gamma; P; Q)$ is in the typed asynchronous bisimilarity.*

The bisimulation above is early on keys and ground on channels. We already said that in asynchronous calculi without name testing, ground and early bisimilarity coincide [74]. Since we only have testing on keys, it makes sense to have the simpler ground clause on channels. As a generalisation of Amadio, Castellani, and Sangiorgi result [4], it holds that, on well-typed Lπ^+-processes, typed asynchronous bisimilarity implies barbed Γ-equivalence.

Later on, we will work with processes containing channels which can be used by the environment only in output. We model this constraint as follows:

Definition 7.1.7 (Barbed $\Gamma;\mathcal{C}$-relations) *Let $\mathcal{C} \subseteq \mathbf{C}$.* Barbed $\mathcal{C}$-bisimilarity, *written $\dot{\approx}_{\mathcal{C}}$, is the largest symmetric relation on processes, such that $P \dot{\approx}_{\mathcal{C}} Q$ implies:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \Rightarrow Q'$ and $P' \dot{\approx}_{\mathcal{C}} Q'$*
2. *If $P{\downarrow_c}$, with $c \notin \mathcal{C}$, then $Q{\Downarrow_c}$.*

Let Γ be a typing, and P and Q two processes such that $\Gamma \vdash P, Q$. We say that P and Q are barbed $\Gamma;\mathcal{C}$-equivalent, *written $P \simeq_{\Gamma;\mathcal{C}} Q$, if for each closed type environment Δ and static (Δ/Γ)-context $C[\cdot]$ not containing names in $\mathcal{C}$ in input position, we have $C[P] \dot{\approx}_{\mathcal{C}} C[Q]$. We say that P and Q are* barbed $\Gamma;\mathcal{C}$-congruent, *written $P \cong_{\Gamma;\mathcal{C}} Q$, if for each closed type environment Δ and (Δ/Γ)-context $C[\cdot]$ not containing names in $\mathcal{C}$ in input position, we have $C[P] \dot{\approx}_{\mathcal{C}} C[Q]$.*

Roughly, $\mathcal{C}$ denotes the set of channels which cannot be used in input by the environment. In Definition 7.1.7, when $\mathcal{C}=\emptyset$, we get the standard definitions of typed barbed bisimilarity. If $\mathcal{C}=\{s\}$, as abbreviations, we write $\cong_{\Gamma;s}$ for $\cong_{\Gamma;\mathcal{C}}$ and $\simeq_{\Gamma;s}$ for $\simeq_{\Gamma;\mathcal{C}}$. Due to the restriction on the contexts, it holds that $\overline{s}v \cong_{\Gamma;s} \mathbf{0}$ and, by asynchrony, $s(u).\mathbf{0} \cong_{\Gamma;s} \mathbf{0}$.

A labelled characterisation of barbed $\Gamma;\mathcal{C}$-equivalence is the following:

Definition 7.1.8 (Typed asynchronous $\mathcal{C}$-bisimilarity) Typed asynchronous $\mathcal{C}$-bisimilarity *is the largest typed relation $\mathcal{S}$ such that $(\Delta; P; Q) \in \mathcal{S}$ implies:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' s.t. $Q \Rightarrow Q'$ and $(\Delta; P'; Q') \in \mathcal{S}$.*
2. *If $P \xrightarrow{(\boldsymbol{\nu}\widetilde{n}:\widetilde{T})\,\overline{c}v} P'$, with $c \notin \mathcal{C}$ and $\widetilde{n} \cap \mathrm{fn}(Q) = \emptyset$, then there exists Q' such that $Q \xRightarrow{(\boldsymbol{\nu}\widetilde{n}:\widetilde{T})\,\overline{c}v} Q'$ and $((\Delta, \widetilde{n}:\widetilde{T}); P'; Q') \in \mathcal{S}$.*
3. *If*
   - *(a) Γ is a closed extension of Δ,*
   - *(b) $\Gamma \vdash c{:}\mathbf{C}(T)$ and $\Gamma \vdash v{:}T$,*
   - *(c) $P \xrightarrow{cv} P'$, with $\mathrm{fc}(v) \cap \mathrm{fn}(P \mid Q) = \emptyset$,*

   then there exists Q' such that:

   - *(a) either $Q \xRightarrow{cv} Q'$ and $(\Gamma; P'; Q') \in \mathcal{S}$,*
   - *(b) or $Q \Rightarrow Q'$ and $(\Gamma; P'; (\, Q' \mid \overline{c}v\,)) \in \mathcal{S}$.*

Let Γ be a closed typing with $\Gamma \vdash P, Q$. We say that P and Q are typed asynchronously $\mathcal{C}$-bisimilar at Γ, *written $P \approx_{\Gamma;\mathcal{C}} Q$, if $(\Gamma; P; Q)$ is in the typed asynchronous $\mathcal{C}$-bisimilarity.*

When $\mathcal{C}=\{s\}$, for some channel s, we abbreviate $\approx_{\Gamma;\mathcal{C}}$ with $\approx_{\Gamma;s}$.

Theorem 7.1.9 *Let Γ be a type environment, $\mathcal{C}$ a set of channels, and P and Q two processes such that $\Gamma \vdash P, Q$. Then, $P \approx_{\Gamma;\mathcal{C}} Q$ implies $P \simeq_{\Gamma;\mathcal{C}} Q$.*

Proof: Since $\mathrm{L}\pi^+$ is an asynchronous calculus without testing on channels, $\approx_{\Gamma;\mathcal{C}}$ coincides with its variant where the requirement $\mathrm{fc}(v) \cap \mathrm{fn}(P \mid Q) = \emptyset$, in the input clause, is omitted. The proof that this (early) variant is preserved by parallel composition and restriction is standard (parallel composition require some care because the processes in parallel must not contain input along channels in $\mathcal{C}$). So, also $\approx_{\Gamma;\mathcal{C}}$ is preserved by parallel composition and restriction. Since $\approx_{\Gamma;\mathcal{C}}$ implies $\dot{\approx}$, it follows that $\approx_{\Gamma;\mathcal{C}} \subseteq \simeq_{\Gamma;\mathcal{C}}$. $\square$

It is easy to prove that $\approx_\Gamma$ implies $\approx_{\Gamma;\mathcal{C}}$ and $\simeq_\Gamma$ implies $\simeq_{\Gamma;\mathcal{C}}$.

Finally, Lemma 7.1.11 provides the $\mathrm{L}\pi^+$ variant of the $\mathrm{L}\pi$ law given in Lemma 3.3.16. Lemma 7.1.11 will be crucial for proving the main result of this chapter (Theorem 7.12.1). In order to prove Lemma 7.1.11, we need the following technical lemma.

Lemma 7.1.10 *Let Q be a process and p and q two channels. Let Γ be a type environment such that $\Gamma \vdash Q$ and $\Gamma \vdash p, q{:}\mathbf{C}(T)$. If q appears only in output position in Q, then*

$$Q\{p/q\} \;\dot{\approx}\; (\boldsymbol{\nu} q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p).$$

Proof: See the proof in Appendix A.4. $\square$

Lemma 7.1.11 *Let $\Gamma \vdash \overline{p}v$, for some type environment Γ. Let $q \in \mathrm{fc}(v)$ with $\Gamma \vdash q{:}\mathbf{C}(T)$. Let $r \notin \mathrm{c}(v)$ and $w = v\{r/q\}$. Then*

$$\overline{p}v \cong_\Gamma (\boldsymbol{\nu} r{:}\mathbf{C}(T))\,(\overline{p}w \mid r \triangleright q).$$

Proof: We cannot adapt the proof of Lemma 3.3.16 because the encoding $[\![\cdot]\!]$ (of Section 3.3) does not take types into account. A such an adaptation is not completely trivial. So, we prove that for any well-typed context $C[\cdot]$, it holds that:

$$C[\overline{p}v] \;\dot{\approx}\; C[(\boldsymbol{\nu} r{:}\mathbf{C}(T))\,(\overline{p}w \mid r \triangleright q)].$$

The prove is by structural induction on the context $C[\cdot]$. The most interesting case is when $C[\cdot] \equiv [\cdot] \mid R$ for some process R. So, in order to prove that

$$\overline{p}v \mid R \;\dot{\approx}\; (\boldsymbol{\nu} r{:}\mathbf{C}(T))\,(\overline{p}w \mid r \triangleright q) \mid R$$

we show that the relation

$$\mathcal{S} = \{\big(\overline{p}v \mid R\,,\ (\boldsymbol{\nu} r{:}\mathbf{C}(T))\,(\overline{p}w \mid r \triangleright q) \mid R\big)\} \cup \dot{\approx}$$

is a barbed bisimulation up to $\equiv$. The requirements on the barbs are easily satisfied. As for the bisimulation game on silent moves, the only interesting case is when there is a communication along p, that is, when $R \xrightarrow{p(x)} R'$. In this case we get, up to structural equivalence, the pair of processes

$$\big(Q\{q/r\}\,,\ (\boldsymbol{\nu} r{:}\mathbf{C}(T))\,(Q \mid r \triangleright q)\big)$$

where $Q = R'\{w/x\}$. By Lemma 7.1.10 we can conclude. $\square$

a, b	$::=$	$\mathbb{O}$	object
	$\mid$	$a.\mathrm{l}\langle\, a_1 \mathinner{\ldotp\ldotp} a_n \,\rangle$	method invocation
	$\mid$	$a.\mathrm{l}{\Leftarrow}m$	method update
	$\mid$	$a.\mathsf{clone}$	shallow copy
	$\mid$	$a.\mathsf{alias}\langle b\rangle$	object aliasing
	$\mid$	$a.\mathsf{surrogate}$	object surrogation
	$\mid$	$a.\mathsf{ping}$	object ping
	$\mid$	s, x, y, z	variables
	$\mid$	$\mathsf{let}\, x = a \,\mathsf{in}\, b$	local definition
	$\mid$	$\mathsf{fork}\langle a\rangle$	thread creation
	$\mid$	$\mathsf{join}\langle a\rangle$	thread destruction
$\mathbb{O}$	$::=$	$[\mathrm{l}_j{=}m_j]_{j\in J}$	object record
m_j	$::=$	$\varsigma(s_j, \tilde{x}_j)b_j$	method

Table 7.4: Øjeblik Syntax

7.2 Øjeblik: syntax and informal semantics

In this section, we introduce Øjeblik, an abstraction of Obliq designed to study object migration. The set of Øjeblik-expressions is generated by the grammar in Table 7.4, where l ranges over *method labels*. For the sake of simplicity, in Øjeblik we omit ground values (like numbers, booleans, strings, etc.), data operations, and procedures, we restrict field selection to method invocation, we restrict multiple cloning to single cloning, we omit flexibility of object attributes, we replace field aliasing with object aliasing, we omit explicit distribution, and we omit exceptions and advanced synchronisation, so we get a feasible, but still non-trivial language. As in Obliq, computation follows the call-by-value evaluation order. In particular, in the following, whenever we use a term a, we implicitly assume that we have first evaluated a to some actual value, i.e. in most cases to an object reference.

In this section, we introduce the semantics of Øjeblik, according to Cardelli's (informal) semantics of Obliq [18].

Objects An object record $[\mathrm{l}_j{=}m_j]_{j\in J}$ is a finite collection of updatable named methods $\mathrm{l}_j{=}m_j$, for pairwise distinct labels l_j. In a method $\varsigma(s, \tilde{x})b$, the letter ς denotes a binder for the self variable s and argument variables $\tilde{x}$ within the body b. Moreover, every object in Øjeblik comes equipped with special methods for *cloning*, *aliasing*, *surrogation*, and *ping*, which cannot be overwritten by the *update* operation.

Method invocation $a.\mathrm{l}\langle\, \tilde{c} \,\rangle$ with field l of the object a containing the method $\varsigma(s, \tilde{x})b$ results in the body b with the self variable s replaced by (a reference to) the enclosing object a, and the formal parameters $\tilde{x}$ replaced by (references to) the actual parameters $\tilde{c}$

of the invocation.

Method update $a.\mathrm{l}{\Leftarrow}m$ overwrites the current content of the named field l in object a with method m and returns a reference to the modified object.

The operation $a.\mathsf{clone}$ creates a clone a' of a and returns a reference to a'.

The operation $a.\mathsf{alias}\langle b\rangle$ replaces object a with an alias to b, written $a{\gg}b$, regardless of whether a is already an alias or still an object record; if b itself is an alias, e.g. $b{\gg}c$, then we consequently and naturally create an *alias chain* $a{\gg}b{\gg}c$. From the computational point of view, requests arriving at a after the operation $a.\mathsf{alias}\langle b\rangle$ should be forwarded to b. The operation $a.\mathsf{alias}\langle b\rangle$ returns a reference to b.

The operation $a.\mathsf{surrogate}$ represents our abstraction of migration: by calling it, object a is turned into a proxy for a copy of itself. Surrogation is implemented by providing a uniform method $\mathsf{surrogate}{=}\varsigma(s)s.\mathsf{alias}\langle s.\mathsf{clone}\rangle$. It returns a reference to the just created clone. Behaving like standard methods, surrogation is forwarded by aliased objects. This is necessary to correctly mimic migration: an object should be surrogatable more than once, so double-surrogation $a.\mathsf{surrogate}; a.\mathsf{surrogate}$ (where ; denotes sequential composition, as defined below) should be equivalent to $a.\mathsf{surrogate}.\mathsf{surrogate}$. Without forwarding, the surrogation of an already surrogated object would mistakenly surrogate the proxy.

The operation $a.\mathsf{ping}$ is implemented by providing a uniform method: $\mathsf{ping}{=}\varsigma(s)s$. Thus, $a.\mathsf{ping}$ returns the "identity" of the object o resulting from the evaluation of a; note that, due to aliasing and forwarding, this could be the "identity" of the current endpoint of an alias chain potentially starting at object o.

Scoping Apart from the binding of variables in method bodies, Øjeblik also offers explicit scope declarations. An expression $\mathsf{let}\,x = a\,\mathsf{in}\,b$ first evaluates a, binding the result to x, and then evaluates b within the scope of the new binding. We use the standard inductive definition $\mathrm{fv}(a)$ to denote the free variables of term a with respect to our two forms of binding. Øjeblik only admits non-recursive expressions $\mathsf{let}\,x = a\,\mathsf{in}\,b$, i.e., with $x \notin \mathrm{fv}(a)$. Then, $a; b$ denotes $\mathsf{let}\,x = a\,\mathsf{in}\,b$, where $x \notin \mathrm{fv}(b)$.

Concurrency While objects represent persistent stateful structural entities, computational activity takes place within *threads*. In addition to the main thread that is initially started up with the execution of a term, new separated threads can be created by the fork command. The term $\mathsf{fork}\langle a\rangle$ returns a new thread identifier to denote the thread evaluating a. The result of a fork'ed computation is grabbed by the join command. If a evaluates to a thread identifier, then $\mathsf{join}\langle a\rangle$ potentially blocks until that thread finishes and returns the thread's result, or blocks forever, if a join on thread a was already performed earlier.

Self-Infliction The *current method* of a thread is the last method invoked in it that has not yet completed. The *current self* of a thread is the self of its current method. An Øjeblik operation is *self-inflicted*, also called *internal*, if it addresses the current self; an operation is *external* if it is not self-inflicted. However, self-inflicted operations can be invoked from within methods not only literally on the self variable s, but also indirectly by an expression that evaluates to the object itself. For instance, in

$$\mathsf{let}\, x = [\, \mathrm{l}{=}\varsigma(s,z)z.\mathsf{clone}\,]\ \mathsf{in}\ x.\mathrm{l}\langle x\rangle$$

the call $z.\mathsf{clone}$ is self-inflicted.

Serialisation In concurrent object-based settings, the invariant that at most one thread at a time may be active within an object is often called *serialisation*. The simplest way to ensure serialisation is to associate with an object a *mutex* that is locked when a thread enters the object and released when the thread exits the object. However, this approach is too restrictive, for instance, it prevents recursion. Based on the notion of *thread*, so-called *reentrant* mutexes, as in Java, can be used to allow an operation to re-enter an object under the assumption that this operation belongs to the same thread as the operation that is currently active in the object. In Obliq, however, the more cautious idea of *self-serialisation* requires, based on the above notion of self-infliction, that the mutex is always acquired for external operations, but never for self-inflicted ones. Note that this concept allows a method to recursively call its siblings through self, but it excludes the kind of inter-object mutual recursion, where a method in an object a calls a method in another object b, which then tries to 'call back' another method in a.

Protection Based on self-infliction, objects are *protected* against external modifications in a natural way: updates, cloning, and aliasing, are only allowed if these operations are self-inflicted.

In Obliq, object migration is supposed to be correct only for both protected and serialised objects. So, since we are interested in proving the correctness of object migration, all objects in (our abstraction) Øjeblik are both protected and serialised.

Executive Summary It is instructive to classify the operations on Øjeblik objects, as we do in the following table, according to the (i) *protection conditions* (subject to self-infliction or not), and the (ii) *node of action*, that is, the node where the operation is finally carried out (either locally, at the initially called node, or at the endpoint of the alias chain starting at the called node). All of these properties are unambiguously stated in the informal semantics of Obliq [18]:

operation	*protection condition?*	*node of action*
cloning, aliasing	only self-inflicted	local
update	only self-inflicted	endpoint
invocation, surrogation, ping	unconstrained	endpoint

Note the hybrid role of update requests, which are carried out at the endpoint but at the same they are subject to local protection.

7.3 Towards a formal semantics for Øjeblik

Although, at first sight, the informal semantics of Øjeblik/Obliq is reasonably clear, its formalisation requires some care. For instance, some crucial facts about alias chains merit to be spelled out explicitly.

7.3.1 On the stability of alias chains

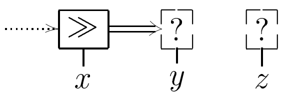
As a matter of fact, according to the operations' character with respect to self-infliction and the intended node of action, a node x in an alias chain can be *unstable*, which means that if it currently points to node y, it may later on point to a different node z. In order to clarify this phenomenon, we distinguish two cases based on the notion of a *task*, which is the run-time entity that is created by method invocation within a single object. A thread may then actually be seen as a stack of tasks connected via invocations. Now, a node can be *active*, in which case it contains running *tasks*, or not. The punchline of this subsection is then that an *alias node* can not become stable before it has terminated its current tasks.

Below, we introduce pictures where we use *single/double* boxes to denote *inactive/active* nodes, respectively, and *single/double* arrows to denote *unstable/stable* aliases, respectively. Furthermore, *dashed* boxes and *dotted* arrows denote unspecified respective entities.

Inactive Nodes: No Tasks By definition, the only way to receive a self-inflicted request is to have already at least one local task running. In other words, if there is no local task, then each incoming request is doomed to be external. Now, let us focus on the example term:

$$\begin{array}{l} \mathsf{let}\ z = [\, \mathrm{l}{=}\text{“}\mathit{bar}\text{”}\,]\ \mathsf{in} \\ \quad \mathsf{let}\ y = [\, \mathrm{l}{=}\text{“}\mathit{foo}\text{”}\,]\ \mathsf{in} \\ \quad\quad \mathsf{let}\ x = [\, \mathrm{l}{=}\varsigma(s, w)s.\mathsf{alias}\langle w \rangle\,]\ \mathsf{in}\ x.\mathrm{l}\langle\, y \,\rangle; x.\mathrm{l}\langle\, z \,\rangle \end{array}$$

after it carried out the invocation $x.\mathrm{l}\langle\, y \,\rangle$, that is, when the object referred to by x has turned itself into an alias for y and then terminated its activity. We depict the situation as follows

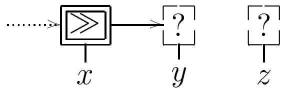


where, in general, the node x may itself be referred to by other aliases, while y and z may be either an alias or an object record. In fact, the alias $x{\gg}y$ is *stable* in the very sense: no re-aliasing operation on x to another node will ever possibly take place since it could only be carried out in a self-inflicted way by one of its own methods, but any request to such a method potentially starting such a self-inflicted operation, e.g. by calling $x.\mathrm{l}\langle\, z\,\rangle$, is itself forwarded to y such that it can never take place in x.

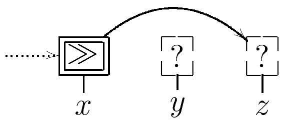
Active nodes: at least one task As an example, let us first consider the term

$$\begin{array}{l}\mathsf{let}\; z = [\,\mathrm{l}{=}\textit{``bar''}\,]\;\mathsf{in}\\ \quad \mathsf{let}\; y = [\,\mathrm{l}{=}\textit{``foo''}\,]\;\mathsf{in}\\ \quad\quad \mathsf{let}\; x = [\,\mathrm{l}{=}\varsigma(s,w)s.\mathsf{alias}\langle w\rangle;\,\underline{\textit{``bla''}}\,]\;\mathsf{in}\; x.\mathrm{l}\langle\, y\,\rangle\end{array}$$

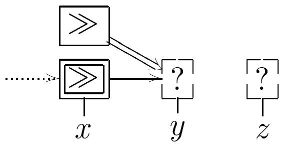
just after object x has accepted the request for method l and turned itself into an alias for y. Since x continues to operate on itself, according to *"bla"* in method l, x is an *active* alias node:



The alias $x{\gg}y$ is marked as *unstable* since *"bla"* may contain further self-inflicted requests, e.g., to perform a re-aliasing or a cloning. Thus, if *"bla"* calls $s.\mathsf{alias}\langle z\rangle$ or $s.\mathsf{clone}$, we get



or



and such changes may continue as long some current task in x is active. Here, the re-aliased x remains active, thus unstable, until all current tasks in x, in our example according to *"bla"*, have terminated. Note that the cloning of an active unstable alias $x{\gg}y$ provides a new inactive stable alias $x'{\gg}y$, because only the state of x is copied, not its tasks.

Generalising the above example, we may consider the case where several tasks of the current thread are running in an alias or an object. However, by the definition of synchronous method invocation, only one of them may be active—namely the one on top of the thread's call-stack, while the others must be blocked. Now, note that it is the active task or any of its ancestors in the call-stack who turned the current node into an alias (in the example it is method l); otherwise, the node would be stable and the current tasks would not exist, but have been created in one of the successors of the stable alias node.

7.3.2 Cyclic alias chains

Obliq does not prevent the programmer from (either consciously or accidentally) introducing, via substitution, self-aliases or alias chains with cycles. Consider the following example:

$$\mathsf{let}\ x = [\,\mathrm{k}{=}\mathrm{id}\,,\ \mathrm{l}{=}\varsigma(s,z)s.\mathsf{alias}\langle z\rangle\,]\ \mathsf{in}\ x.\mathrm{l}\langle x\rangle; x.\mathrm{k}.$$

By calling $x.\mathrm{l}\langle x\rangle$, the aliasing operation $x.\mathsf{alias}\langle x\rangle$ is carried out giving rise to the cyclic alias chain $x \gg x$. As a consequence, the following external method call $x.\mathrm{k}$ will give rise to a diverging computation.

7.3.3 On forwarding requests along the alias chain

It turns out that details concerning alias nodes in Obliq are not explicitly spelled out in [18]. Below, we describe the behaviour of alias nodes in Obliq by addressing four crucial questions:

1. *Which* requests are forwarded and which requests fail in an alias node?
2. *What* is the current self when forwarding requests?
3. *When* does the forwarding take place?
4. *Who* is in charge of sending the result of a forwarded external request?

Which? Cardelli's semantics keeps *protection* in alias nodes, so cloning, aliasing, and update requests fail, if external. Method invocations are always forwarded to the end-point. Self-inflicted cloning, self-inflicted aliasing, and self-inflicted update requests are forwarded to the end-point.

According to the above requirements, in Øjeblik, for uniformity, ping and sur requests are forwarded to the end point as method invocations. Aliasing operations are performed at the alias node if self-inflicted, and fail otherwise. Moreover, for simplicity, we discard failing requests.

What? Let a be an alias node forwarding requests to b, that is, $a \gg b$. Let c be a third object invoking a method of a. Then, when serving the request, the alias a becomes the new current self. The self-inflicted case is trivial because would mean $a = c$.

When? Here it is crucial to recall the fact that alias nodes are unstable as long as they are active (Section 7.3.1). Should external requests that are to be forwarded wait until an alias node becomes stable, or should they be forwarded immediately? Cardelli's semantics keeps *serialisation* in alias nodes. This means that, in order to be forwarded by an alias node, external requests have to wait until the alias node becomes stable while self-inflicted requests can be forwarded immediately.

Who? Let $a \gg b$. Let c be a third object invoking a method of a. When serving the request the alias a becomes active and it is in charge of sending c the result of the request forwarded to b. In doing that, since active, a is locked until the forwarded request has successfully signalled termination from b.

7.4 A translational semantics for Øjeblik

In this section we give a *translational semantics* of Øjeblik into $\mathrm{L}\pi^+$ according to Obliq's informal semantics [18]. In addition to the syntax of $\mathrm{L}\pi^+$ we use standard abbreviations for:

- *polyadic input*

$$a(x_1 \mathinner{..} x_n).P \stackrel{\text{def}}{=} a(y).\mathsf{let}\,(\,x_1 \mathinner{..} x_n\,) = y \;\mathsf{in}\; P;$$

- *polyadic case destructor*

$$\ell\_(x_1 \mathinner{..} x_n){:}P \stackrel{\text{def}}{=} \ell\_(x){:}\mathsf{let}\,(\,x_1 \mathinner{..} x_n\,) = x \;\mathsf{in}\; P,$$

  where $x \notin \mathrm{fn}(P)$;
- *parameterised recursive definitions*, which can be faithfully represented in terms of replication [51];
- *structured key testing*,

$$\mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k{=}k_2]\ \ \mathsf{then}\ P_2\ \mathsf{else}\ P_3$$

  as a macro for

$$\mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{else}\ \mathsf{if}\ [k{=}k_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3.$$

The semantics, as presented in Tables 7.5, 7.6, and 7.7, is a mapping parameterised on two names: in a term $[\![\, a \,]\!]_p^k$, the channel p is used to return the term's result, while the key k represents the term's current self, which is required to deal with self-infliction.

In all phases of the translation, whenever we create $\boldsymbol{\nu}$- or input-bindings, we assume that there are no name-clashes. The essence of the semantics is to set up processes representing objects that serve clients' requests. Different requests for operating on objects are distinguished by corresponding labels cln, ali, upd_j, inv_j, png, and sur. We explain the semantics by showing how requests are generated by clients, and then how they are served by objects. Scoping and concurrency are explained along the way.

Clients. In Table 7.5, the current self k of encoded terms is 'used' as the current self of the evaluation of the first subterm in left-to-right evaluation order. All the translations in Table 7.5 follow a common scheme. For example, in the translation of a method invocation $[\![\, a.\mathrm{l_j}\langle\, a_1 \mathinner{..} a_n \,\rangle \,]\!]_p^k$, the subterms $a, a_1 \mathinner{..} a_n$ have to be evaluated one after the other: the individual evaluations use private return channels $q, q_1 \mathinner{..} q_n$, which are subsequently asked for the respective results $y, x_1 \mathinner{..} x_n$, but also for the respective new current self $k, k_1 \mathinner{..} k_n$ to be used by the next evaluation. After the last subterm a_n has returned its result, the accumulated information is used to send a suitable request with label inv_j on self-channel y of object a, also carrying the overall result channel p and the latest current self k_n. Thus, the responsibility to signal a result on p is passed on to the respective object waiting at y.

$$
\begin{array}{rcl}
[\![\, a.\mathsf{clone} \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\Big(\, [\![\, a \,]\!]_q^k \mid q(y,k') \,.\, \overline{y}\langle \mathsf{cln\_}p, k' \rangle \,\Big) \\
[\![\, a.\mathsf{alias}\langle b \rangle \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q_x q_y)\Big(\, [\![\, a \,]\!]_{q_y}^k \mid q_y(y,k_y).([\![\, b \,]\!]_{q_x}^{k_y} \mid q_x(x,k_x) \,.\, \overline{y}\langle \mathsf{ali\_}\langle x,p\rangle, k_x \rangle) \,\Big) \\
[\![\, a.\mathrm{l}_j \Leftarrow \varsigma(s,\tilde{x})b \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\Big([\![\, a \,]\!]_q^k \mid q(y,k').(\boldsymbol{\nu} t)\big(\, !\, t(s,\widetilde{x},r,k).[\![\, b \,]\!]_r^k \mid \overline{y}\langle \mathsf{upd}_j\mathsf{\_}\langle t,p\rangle, k' \rangle\big)\Big) \\
[\![\, a.\mathrm{l}_j\langle\, a_1 \ldots a_n \,\rangle \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q q_1 \cdots q_n)\Big(\, [\![\, a \,]\!]_q^k \mid q(y,k_0).([\![\, a_1 \,]\!]_{q_1}^{k_0} \mid q_1(x_1,k_1).([\![\, a_2 \,]\!]_{q_2}^{k_1} \mid \cdots \\
 & & \qquad q_n(x_n,k_n).\overline{y}\langle \mathsf{inv}_j\mathsf{\_}\langle x_1 \ldots x_n, p\rangle, k_n \rangle \cdots))\Big) \\
[\![\, a.\mathsf{surrogate} \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\big(\, [\![\, a \,]\!]_q^k \mid q(y,k') \,.\, \overline{y}\langle \mathsf{sur\_}p, k' \rangle \,\big) \\
[\![\, a.\mathsf{ping} \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\big(\, [\![\, a \,]\!]_q^k \mid q(y,k') \,.\, \overline{y}\langle \mathsf{png\_}p, k' \rangle \,\big) \\
\hline
[\![\, \mathsf{let}\; x = a \;\mathsf{in}\; b \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\big(\, [\![\, a \,]\!]_q^k \mid q(x,k') \,.\, [\![\, b \,]\!]_p^{k'} \,\big) \\
[\![\, x \,]\!]_p^k & \stackrel{\text{def}}{=} & \overline{p}\langle x, k \rangle \\
\hline
[\![\, \mathsf{fork}(a) \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q t)\big(\, (\boldsymbol{\nu} \;) \mid k' \mid ([\![\, a \,]\!]_q^{k'})\overline{p}\langle t,k\rangle q(x,k').t(r,k'').\overline{r}\langle x,k'' \rangle \,\big) \\
[\![\, \mathsf{join}(b) \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\big(\, [\![\, b \,]\!]_q^k \mid q(t,k') \,.\, \overline{t}\langle p, k' \rangle \,\big)
\end{array}
$$

Table 7.5: Translational semantics of Øjeblik — Clients, Scoping, Concurrency

Scoping. The semantics of let is analogous to [42] and represents the core of the call-by-value evaluation order in that first a is evaluated, and then b possibly using the value of a. Here, in addition, the evaluation of a passes on the current self k' to be used afterwards.

Concurrency. To fork a thread means to create a new activity running in parallel with the current one(s), which is done using the parallel operator. Upon thread creation, a fresh key is created to become the forked thread's current self. Sometimes, we use $[\![\, a \,]\!]_q^{\boldsymbol{\nu}}$ to abbreviate $(\boldsymbol{\nu} k)\,([\![\, a \,]\!]_q^k)$. The term $[\![\, \mathsf{fork}\langle a \rangle \,]\!]_p^k$ immediately returns on p a private name t, which can be used to retrieve the value of a from the forked thread. Therefore, $[\![\, \mathsf{join}\langle b \rangle \,]\!]_p^k$ sends its own result channel p, together with its latest current self k', along the value of b.

Objects. The semantics $[\![\, \mathbb{O} \,]\!]_p^k$ of an object $\mathbb{O} := [\mathrm{l}_j{=}\varsigma(s_j,\tilde{x}_j)b_j]_{j\in J}$, as shown in Table 7.6 (again similar to [42]), consists of a message that returns the object's reference s together with the current self k on channel p, a composition of replicated processes that give access to the method bodies $[\![\, b_j \,]\!]_r^{k'}$, and a new object process $\mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t} \,\rangle$ that connects invocations on s from the outside to the method bodies, which are invoked by the trigger names $\widetilde{t}$.

Inside $\mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t} \,\rangle$, several private names are needed: *mutexes* $\widetilde{m} := m_{\mathrm{e}}, m_{\mathrm{i}}$ are used for serialisation; the (*internal*) key k_{i} is used to detect self-infliction; the (*external*) key k_{e} is used to implement serialisation in a concurrent environment (see later on). The behaviour of (serialised) objects and aliases is represented by the object manager OM and alias manager AM, respectively, which both, for each request arriving along their reference s, first check for self-infliction and then, by simple case analysis, for the kind

$$[\![\mathbb{O}]\!]_p^k \stackrel{\text{def}}{=} (\boldsymbol{\nu} s\,\widetilde{t})\Big(\overline{p}\langle s,k\rangle \mid \mathsf{newO}_{\mathbb{O}}\langle\, s,\widetilde{t}\,\rangle \mid \prod_{j\in J} \boldsymbol{!}\,t_j(s_j,\widetilde{x}_j,r,k').[\![\, b_j \,]\!]_r^{k'} \Big)$$

$$\begin{array}{rcl} \mathsf{newO}_{\mathbb{O}}\langle\, s,\widetilde{t}\,\rangle & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} m_{\mathrm{e}} m_{\mathrm{i}} k_{\mathrm{e}} k_{\mathrm{i}})\Big(\overline{m_{\mathrm{e}}} \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},\widetilde{k},\widetilde{t}\,\rangle \Big) \\ & & \text{with } \widetilde{m} := m_{\mathrm{e}}, m_{\mathrm{i}} \text{ and } \widetilde{k} := k_{\mathrm{e}}, k_{\mathrm{i}} \end{array}$$

$$\begin{array}{l}
\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},\widetilde{k},\widetilde{t}\,\rangle \stackrel{\text{def}}{=} s(l,k).(\boldsymbol{\nu} k^*)\Big(\\
\quad \mathsf{if}\ [k{=}k_{\mathrm{i}}]\ \mathsf{then} \\
\qquad \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln_}(r) \triangleright \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid (\boldsymbol{\nu} s^*)\big(\overline{r}\langle s^*,k^*\rangle \mid \mathsf{newO}_{\mathbb{O}}\langle\, s^*,\widetilde{t}\,\rangle \big)\ ; \\
\qquad\qquad \mathsf{ali_}(s_{\mathrm{a}},r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid \overline{r}\langle s_{\mathrm{a}},k^*\rangle\ ; \\
\qquad\qquad \mathsf{upd}_{j}\mathsf{_}(t',r) \triangleright \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,t_1\,..\,t_{j-1},t',t_{j+1}\,..\,t_n\,\rangle \mid \overline{r}\langle s,k^*\rangle\ ; \\
\qquad\qquad \mathsf{inv}_{j}\mathsf{_}(\widetilde{x},r) \triangleright \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid \overline{t_j}\langle s,\widetilde{x},r,k^*\rangle\ ; \\
\qquad\qquad \mathsf{sur_}(r) \triangleright \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid [\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]_r^{k^*}\ ; \\
\qquad\qquad \mathsf{png_}(r) \triangleright \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid [\![\, s \,]\!]_r^{k^*} \\
\quad \mathsf{elif}\ [k{=}k_{\mathrm{e}}]\ \mathsf{then} \\
\qquad \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln_}(r) \triangleright m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ ; \\
\qquad\qquad \mathsf{ali_}(s_{\mathrm{a}},r) \triangleright m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ ; \\
\qquad\qquad \mathsf{upd}_{j}\mathsf{_}(t',r) \triangleright m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ ; \\
\qquad\qquad \mathsf{inv}_{j}\mathsf{_}(\widetilde{x},r) \triangleright \mathsf{CM}[\,\overline{t_j}\langle s,\widetilde{x},r^*,k^*\rangle\,]\ ; \\
\qquad\qquad \mathsf{sur_}(r) \triangleright \mathsf{CM}[\,[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]_{r^*}^{k^*}\,]\ ; \\
\qquad\qquad \mathsf{png_}(r) \triangleright \mathsf{CM}[\,[\![\, s \,]\!]_{r^*}^{k^*}\,] \\
\quad \mathsf{else}\ \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},\widetilde{t}\,\rangle \mid m_{\mathrm{e}}.\big(\overline{s}\langle l,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k \big)\Big)
\end{array}$$

$$\mathsf{CM}[\cdot] \stackrel{\text{def}}{=} (\boldsymbol{\nu} r^*)\big([\cdot] \mid r^*(y,k').m_{\mathrm{i}}(k'').(\overline{r}\langle y,k''\rangle \mid \overline{m_{\mathrm{e}}}) \big)$$

Table 7.6: Translational semantics of Øjeblik — Object manager

$$\mathsf{newA}_{\mathbb{O}}\langle\, s, s_{\mathrm{a}} \,\rangle \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu} m_{\mathrm{e}} m_{\mathrm{i}} k_{\mathrm{e}} k_{\mathrm{i}}) \Big(\, \overline{m_{\mathrm{e}}} \mid \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, s_{\mathrm{a}} \,\rangle \Big)$$

$$\text{with } \widetilde{m} := m_{\mathrm{e}}, m_{\mathrm{i}} \text{ and } \widetilde{k} := k_{\mathrm{e}}, k_{\mathrm{i}}$$

$$
\begin{array}{l}
\mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, s_{\mathrm{a}} \,\rangle \stackrel{\mathrm{def}}{=} s(l,k).(\boldsymbol{\nu} k^*) \Big(\\
\quad \mathsf{if}\ [k{=}k_{\mathrm{i}}]\ \mathsf{then} \\
\qquad \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln}\_(r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}} \,\rangle \mid (\boldsymbol{\nu} s^*) \big(\, \overline{r}\langle s^*, k^* \rangle \mid \mathsf{newA}_{\mathbb{O}}\langle\, s^*, s_{\mathrm{a}} \,\rangle \big) \ ; \\
\qquad\qquad \mathsf{ali}\_(s'_{\mathrm{a}}, r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s'_{\mathrm{a}} \,\rangle \mid \overline{r}\langle s'_{\mathrm{a}}, k^* \rangle \ ; \\
\qquad\qquad \mathsf{upd}_j\_(t', r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}} \,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k^* \rangle \ ; \\
\qquad\qquad \mathsf{inv}_j\_(\widetilde{x}, r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}} \,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k^* \rangle \ ; \\
\qquad\qquad \mathsf{sur}\_(r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}} \,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k^* \rangle \ ; \\
\qquad\qquad \mathsf{png}\_(r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}} \,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k^* \rangle \\
\quad \mathsf{elif}\ [k{=}k_{\mathrm{e}}]\ \mathsf{then} \\
\qquad \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}} \,\rangle \;\Big|\; \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln}\_(r) \triangleright m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}} \ ; \\
\qquad\qquad\qquad\qquad \mathsf{ali}\_(s_{\mathrm{a}}, r) \triangleright m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}} \ ; \\
\qquad\qquad\qquad\qquad \mathsf{upd}_j\_(t', r) \triangleright m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}} \ ; \\
\qquad\qquad\qquad\qquad \mathsf{inv}_j\_(\widetilde{x}, r) \triangleright \mathsf{CM}[\, \overline{s_{\mathrm{a}}}\langle \mathsf{inv}_j\_\langle \widetilde{x}, r^* \rangle, k^* \rangle \,] \ ; \\
\qquad\qquad\qquad\qquad \mathsf{sur}\_(r) \triangleright \mathsf{CM}[\, \overline{s_{\mathrm{a}}}\langle \mathsf{sur}\_\langle r^* \rangle, k^* \rangle \,] \ ; \\
\qquad\qquad\qquad\qquad \mathsf{png}\_(r) \triangleright \mathsf{CM}[\, \overline{s_{\mathrm{a}}}\langle \mathsf{png}\_\langle r^* \rangle, k^* \rangle \,] \\
\quad \mathsf{else}\ \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, s_{\mathrm{a}} \,\rangle \mid m_{\mathrm{e}}.\big(\, \overline{s}\langle l, k_{\mathrm{e}} \rangle \mid \overline{m_{\mathrm{i}}} k \, \big) \Big)
\end{array}
$$

Table 7.7: Translational semantics of Øjeblik — Alias manager

of the request. Below, we explain in detail the behaviour of object and alias managers.

Serving self-inflicted requests $[k{=}k_{\mathrm{i}}]$

Object Managers (OM). For each field, the manager may activate appropriate instances of method bodies (case inv_j: the method body bound to l_j along trigger name t_j) and administer updates (case upd_j: install a new trigger name t'). Cloning (case cln) restarts the current object manager in parallel with a new object, which uses the same method bodies $\widetilde{t}$, but is accessible through a fresh reference s^*. Aliasing (case ali) starts an appropriate alias manager AM instead of re-starting the previous object manager OM. Surrogation and ping (cases sur and png) are modelled according to their uniform method definitions.

Alias Managers (AM). Cloning and alias requests are allowed and behave analogous to the respective clauses in object managers, but restarting AM instead of OM. Invocation, surrogation, and ping requests are forwarded to the aliasing target s_{a} replacing the current self with the (new) current self of the alias manager. External update requests are discarded while self-inflicted ones are forwarded. Notice that self-inflicted update requests will fail in the following node of the alias chain because, in that node, they will be external.

Nonces (k^*). To guarantee the receptiveness of objects, managers OM and AM always have to be restarted with some possibly changed state. However, serialisation requires that at any moment, only one request shall be *active* in an object. According to our semantics, program contexts will never give rise to several competing self-inflicted requests, but, when reasoning within arbitrary $\mathrm{L}\pi^+$ contexts, as we do in Section 7.11, their existence must be taken into account. Therefore, we add another layer of protection to increase the robustness of serialisation: each time a request enters a manager, a fresh key k^* is created to be used in the restarted manager; this key must subsequently be used as the current self for all activities enabled by the current request. Thus, the consumption of one competing pending requests renders the other competitors external.

Pre-processing $[k_{\mathrm{i}}{\neq}k{\neq}k_{\mathrm{e}}]$

As mentioned earlier, mutual exclusion within an object is implemented by mutexes, so, upon creation of a new object newO, the fresh mutex channel m_{e} is initialised. According to serialisation, the intended continuation behaviour of an incoming external requests is blocked on m_{e}, once it enters a manager. The manager itself is immediately restarted and remains receptive. Arbitrarily many requests can be blocked this way and compete for the mutex m_{e} once it becomes available. A successfully unblocked request is resent to the same manager, but now carrying the key k_{e}, which allows the manager to detect that the request has grabbed the mutex. We call *pre-processing* the procedure of intermediate blocking of requests and their subsequent reemission with key k_{e} instead.

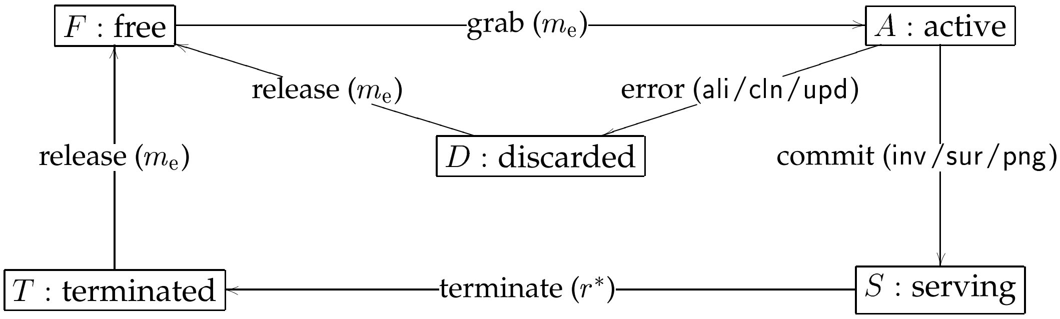


Figure 7.1: Object Manager Serving External Requests

Alongside with the pre-processed request, its former current self k is stored on the (internal) mutex m_i for recovery after termination. This recovery is actually necessary since the original current self k is possibly required for use later on by the sender of the request. Note that pre-processing also properly takes care of the fact that competing requests may change the state of an object, and even turn it into an alias, so pre-processed requests should not be bound too early to some specific object manager behaviour. By only resending a request once it has grabbed the mutex it will be handled by the current manager, not by the manager in the state of the moment when the request originally entered the object.

Nonces (k^*). Pre-processing must not reinitialise the key k_i of the restarted manager: a currently self-inflicted operation interleaved by pre-processing might be hindered to proceed, because it could unintendedly become external.

Serving external requests $[k{=}k_e]$

In order to clarify the behaviour of an object manager serving an external request, Figure 7.1 shows the five relevant "states", starting from a *free* manager that becomes *active* by serving some pending request which has grabbed the serialisation-lock m_e. Then, if the request is protection-critical it is *discarded*, otherwise the manager *commits* to it and *serves* it until explicit *termination*. In both cases, the manager becomes free again by releasing the lock. Self-inflicted requests can only be served in "state" S (note that, when serving self-inflicted aliasing requests, for simplicity, Figure 7.1 does not represent the fact that the object manager becomes an alias manager). In Subsection 7.9, we explain Figure 7.1 in more detail; for now, it just offers an informal device to guide the explanations of the semantics of object managers.

Object Managers (OM). Cloning, aliasing, or update, are critical operations. Once a respective pre-processed request is consumed, the manager evolves from "state" A into "state" D: the request and its former current self k, stored on channel m_i, will be simply discarded by consuming $\overline{m_i}k$ and releasing $\overline{m_e}$.

Invocation, surrogation, and ping are non-critical operations. Once a respective pre-processed request is consumed, the manager evolves from "state" A into "state" S implying that no other external request shall be served (apart from pre-processing) until the current one has terminated. In order to be notified of that event, we employ a *call manager* protocol, represented by the context $\mathsf{CM}[\cdot]$: instead of delegating to some other process the responsibility of returning a result on r, a fresh return channel r^* is created to be used within $[\cdot]$ in place of r, such that the result will first appear on r^*. Until this event, other external requests remain blocked, while internal request may well be served. After this event, the manager evolves from "state" S into "state" T, where the former current self can be grabbed from m_{i}, the result y be forwarded to the intended result channel r (along with the former current self), and the mutex m_{e} be released. Note that externally triggered method bodies $[\![\, b_j \,]\!]$, and also surrogation and ping bodies $[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]$ and $[\![\, s \,]\!]$, are all run in the context of the nonce k^*, which is now the internal key of the OM, so their further calls to s will be self-inflicted. This is essential for surrogation, since cloning and aliasing are only allowed internally.

Alias Managers (AM). As pointed out in Section 7.3.3, cloning, aliasing, and update requests fail. Invocation, surrogation, and ping requests are forwarded, and the alias manager is locked until a forwarded request has successfully signalled termination from its point of action. This is implemented by using the call manager $\mathsf{CM}[\cdot]$. The pre-processing of requests, presumably superfluous in alias managers, is necessary also there, because there may be pending pre-processed requests that have come to existence when s was connected to an OM.

7.5 Behavioural equivalence for Øjeblik

A standard way to define *program equivalences* is to compare the convergence behaviour of programs within arbitrary program contexts. This equivalence is usually referred to as *observational congruence* [59].

Since we have given a translational semantics for Øjeblik, our program equivalence is based on the encoding $[\![\, \cdot \,]\!]_p^k$. Roughly, the semantics $[\![\, a \,]\!]_p^k$, of an Øjeblik term a is a $\mathrm{L}\pi^+$-process which returns the result on name p as soon as it knows it. An Øjeblik term *converges*, if its semantics is a (well-typed) process which *may* report its result on the name p.

Definition 7.5.1 (Convergence) *Given an Øjeblik term a, we write $a\Downarrow$ if $\Gamma \vdash [\![\, a \,]\!]_p^k$ for some Γ, and $[\![\, a \,]\!]_p^k \Downarrow_p$.*

We require well-typedness for the semantics because the target of the translation is a typed calculus. In this way we implicitly discard Øjeblik terms containing structural run-time errors (e.g. invocations of un-existing methods). At the same time, types allows us to have easier proofs (see Theorem 7.11.1). Alternatively, a more complex approach, would be to equip Øjeblik with types. The translational semantics should be accompanied with a translation of the corresponding types. We have chosen the former approach for simplicity.

$$
\begin{array}{rlll}
C[\cdot] & ::= & [\cdot] & \mid [\ \mathrm{l}_k{=}\varsigma(s,\widetilde{x})C[\cdot]\ ,\ \mathrm{l}_{j\neq k}{=}m_{j\neq k}\ ]_{j\in J} \\
& \mid & C[\cdot].\mathrm{l}\langle\,\tilde{a}\,\rangle & \mid a.\mathrm{l}\langle\,\tilde{a}, C[\cdot], \tilde{a}\,\rangle \\
& \mid & C[\cdot].\mathrm{l}{\Leftarrow}m & \mid a.\mathrm{l}{\Leftarrow}\varsigma(s,\widetilde{x})C[\cdot] \\
& \mid & C[\cdot].\mathsf{alias}\langle b\rangle & \mid a.\mathsf{alias}\langle C[\cdot]\rangle \\
& \mid & C[\cdot].\mathsf{clone} & \\
& \mid & C[\cdot].\mathsf{surrogate} & \mid C[\cdot].\mathsf{ping} \\
& \mid & \mathsf{let}\, x = C[\cdot]\, \mathsf{in}\, b & \mid \mathsf{let}\, x = a\, \mathsf{in}\, C[\cdot] \\
& \mid & \mathsf{fork}\langle C[\cdot]\rangle & \mid \mathsf{join}\langle C[\cdot]\rangle
\end{array}
$$

Table 7.8: Øjeblik contexts

The program equivalence for Øjeblik terms is a contextual equivalence. We recall that according to Table 7.8, an *Øjeblik context* $C[\cdot]$ has a single hole $[\cdot]$ that may be filled with an Øjeblik term.

Definition 7.5.2 (Behavioural equivalence) *Two Øjeblik terms a and b are* behaviourally equivalent, *written $a \doteq b$, if*

$$C[a] \Downarrow \quad \text{iff} \quad C[b] \Downarrow$$

for all Øjeblik contexts $C[\cdot]$.

7.6 Safety of surrogation as an equation

As already discussed, object migration/surrogation should be transparent to object clients. In other words *an object should behave the same before and after surrogation*, in all possible contexts. The following equation is a first attempt to model this property:

$$a \doteq a.\mathsf{surrogate} \tag{7.1}$$

The simplest case of Equation 7.1 is when a is an object $\mathbb{O}$. In this case the surrogation is surely safe, because (1) the process of surrogation is carried out correctly since, due to serialisation, only the surrogation thread can interact with the object $\mathbb{O}$, i.e., there cannot be any interference with another thread or activity, and (2) every interaction with $\mathbb{O}$ is mimicked identically by $\mathbb{O}.\mathsf{surrogate}$, which suffices since after surrogation nobody has access to the previous $\mathbb{O}$.

In the general case, however, neither of the two above arguments holds. The reason is because of possible copying of a reference to the former object such that, after surrogation, requests can still be directed to that reference. Observing that $a \doteq \mathsf{let}\, x = a\, \mathsf{in}\, x$ (in all contexts, the let just adds one unconditional step after reducing a) and that the notion of equivalence takes all Øjeblik contexts into account, Equation 7.1 can be reduced to the problem of surrogation on variables:

$$x \doteq x.\mathsf{surrogate} \tag{7.2}$$

However, there is an inherent problem with Equation 7.2, which is exhibited by the following context that creates a self-alias via method call:

$$C[\cdot] := \mathsf{let}\ x = [\ \mathrm{l}{=}\varsigma(s)s.\mathsf{alias}\langle s\rangle\ ]\ \mathsf{in}\ x.\mathrm{l};\ [\cdot] \tag{7.3}$$

It holds that $C[x]\Downarrow$, whereas $C[x.\mathsf{surrogate}]\not\Downarrow$. Indeed, in $C[x]\Downarrow$ the evaluation of x returns immediately, while in $C[x.\mathsf{surrogate}]\not\Downarrow$, the request $x.\mathsf{surrogate}$ is never served because it travels into a loop along the self alias chain $x \gg x$. The problem in Equation 7.2 is that we do not check whether the "object before surrogation" is actually reachable. This can be easily done as follows

$$x.\mathsf{ping} \doteq x.\mathsf{surrogate} \tag{7.4}$$

The equation 7.4 detects cyclic chains by means of the ping-request which travels to the endpoint of the alias chain possibly starting at x. For the above context, $C[x.\mathsf{ping}]\not\Downarrow$.

In the remainder of this thesis, Equation 7.4 will be referred to as the *safety equation*.

7.7 Unsafe surrogations

In this section we present a number of counterexamples showing that the safety equation for surrogation is not valid. Similar counterexamples apply to object migration in Obliq, as we have tested using the Obliq interpreter [17]. The counterexamples concern both external and self-inflicted surrogation and consist in applying some context $C[\cdot]$ to $x.\mathsf{ping}$ and $x.\mathsf{surrogate}$, showing that the terms $C[x.\mathsf{ping}]$ and $C[x.\mathsf{surrogate}]$ do not converge/diverge in the same manner.

As an abbreviation, we use the method definitions $\mathrm{l}{=}\mathrm{id}$ and $\mathrm{k}{=}\Omega$ to denote $\mathrm{l}{=}\varsigma(s)s$ and $\mathrm{k}{=}\varsigma(s)s.\mathrm{k}$, respectively, where, $[\,\ldots,\mathrm{l}{=}\mathrm{id},\ldots\,].\mathrm{l}\Downarrow$ and $[\,\ldots,\mathrm{k}{=}\Omega,\ldots\,].\mathrm{k}\not\Downarrow$.

7.7.1 Countering self-inflicted surrogations

A particular class of examples is represented by objects that perform surrogation in a self-inflicted manner. This means that after the surrogation, as long as the current method is still active, self-inflicted operations can be performed by the object itself. We classify three different counterexamples depending on whether they exhibit problems with *protection*, *serialisation* or *re-aliasing*.

Protection problems In the following example a self-inflicted surrogation is followed by a protected operation addressed to the proxy created by surrogation. Consider the context:

$$C[\cdot] := [\ \mathrm{k}{=}\varsigma(s)\mathsf{let}\ y = [\cdot]\ \mathsf{in}\ s.\mathrm{k}{\Leftarrow}\ \mathrm{id}\,].\mathrm{k} \tag{7.5}$$

The execution of $C[s.\mathsf{ping}]$ yields the following computation:

1. Method k is externally invoked.
2. Method k executes a self-inflicted ping request $s.\mathsf{ping}$ which returns s.
3. Finally, a self-inflicted update request $s.\mathrm{k}{\Leftarrow}\,\mathrm{id}$ is performed.
4. As consequence, $C[s.\mathsf{ping}]{\Downarrow}$.

On the other hand, the execution of $C[s.\mathsf{surrogate}]$ yields a different result. Here are the steps of its computation:

1. Method k is externally invoked.
2. Method k invokes a self-inflicted surrogation request $s.\mathsf{surrogate}$ which returns a clone s' and yields an alias chain $s{\gg}s'$.
3. The update request $s.\mathrm{k}{\Leftarrow}\,\mathrm{id}$ is invoked in a self-inflicted manner. Since s is an alias node, the request is forwarded to s'.
4. Since s is still the current self, the forwarded update request is external in s' and, *by protection*, fails.
5. As a consequence, $C[s.\mathsf{surrogate}]{\not\Downarrow}$.

Similar counterexamples can be given using other protected operations, such as cloning and aliasing operations, and/or addressing the protected operation to the surrogated object instead of the proxy. For instance, consider the context:

$$C[\cdot] := [\,\mathrm{k}{=}\varsigma(s)\mathsf{let}\; y = [\cdot]\;\mathsf{in}\; y.\mathsf{clone}\,].\mathrm{k} \tag{7.6}$$

The execution of $C[s.\mathsf{ping}]$ yields the following computation:

1. Method k is externally invoked.
2. Method k invokes a self-inflicted ping request and then a self-inflicted clone request. Both requests are served, and therefore $C[s.\mathsf{ping}]{\Downarrow}$.

The execution of $C[s.\mathsf{surrogate}]$ yields a different result. Here are the steps of its computation:

1. Method k is externally invoked.
2. Method k invokes a self-inflicted surrogation request $s.\mathsf{surrogate}$ which returns a clone s' and yields an alias chain $s{\gg}s'$.
3. The request $s.\mathsf{clone}$ is invoked in a self-inflicted manner.
4. Since s is a proxy, the clone request is forwarded to s'.
5. Since s is the current self, the forwarded clone request is external in s' and, *by protection*, fails.
6. As a consequence, $C[s.\mathsf{surrogate}]{\not\Downarrow}$.

Serialisation problems Here is an example of unsafe self-inflicted surrogation due to serialisation. Consider the context:

$$C[\cdot] := [\,\mathrm{k}{=}\varsigma(s,z)z.\mathrm{l}\,,\mathrm{l}{=}\mathrm{id}\,,\mathrm{m}{=}\varsigma(s)[\cdot].\mathrm{k}\langle s\rangle\,].\mathrm{m} \tag{7.7}$$

The computation of $C[s.\mathsf{ping}]$ is the following:

1. Method m is invoked externally.
2. Method m invokes a self-inflicted ping request $s.\mathsf{ping}$ which returns s.
3. Method m invokes the method call $s.\mathrm{k}\langle s\rangle$ in a self-inflicted manner.
4. Method k invokes a self-inflicted call $s.\mathrm{l}$.
5. Since method l is defined as the identity, the computation returns s.
6. As a consequence, $C[s.\mathsf{ping}]\Downarrow$.

The computation of $C[s.\mathsf{surrogate}]$ is the following:

1. Method m is invoked externally.
2. Method m invokes a self-inflicted surrogation request $s.\mathsf{surrogate}$ which returns a clone s' and yields an alias chain $s{\gg}s'$.
3. Method m (of s) invokes the method call $s'.\mathrm{k}\langle s\rangle$ which is served by s' as external.
4. Method k (of s') invokes an external method call $s.\mathrm{l}$ (indeed, the current self is now s').
5. But s is blocked because its method m has not returned yet. So, *by serialisation*, the request $s.\mathrm{l}$ blocks.
6. As a consequence, $C[s.\mathsf{surrogate}]\not\Downarrow$.

Re-aliasing problems The next example is intended to exhibit the effect of re-aliasing. We recall that re-aliasing is possible only in a self-inflicted manner (c.f. Section 7.3.1). Consider the context:

$$\begin{aligned} C[\cdot] := \mathsf{let}\ x = {} & [\,\mathrm{l}{=}\Omega\,,\mathrm{k}{=}\Omega\,]\ \mathsf{in} \\ & [\,\mathrm{l}{=}\mathrm{id}\,,\mathrm{k}{=}\varsigma(s)\mathsf{let}\ y = [\cdot]\ \mathsf{in}\ s.\mathsf{alias}\langle x\rangle; y.\mathrm{l}\,].\mathrm{k} \end{aligned} \tag{7.8}$$

The execution of $C[s.\mathsf{ping}]$ gives rise to the following computation:

1. x becomes a reference to $[\,\mathrm{l}{=}\Omega\,,\mathrm{k}{=}\Omega\,]$.
2. The invocation

$$[\,\mathrm{l}{=}\mathrm{id}\,,\mathrm{k}{=}\varsigma(s)\mathsf{let}\ y = [\cdot]\ \mathsf{in}\ s.\mathsf{alias}\langle x\rangle; y.\mathrm{l}\,].\mathrm{k}$$

   implies a self-inflicted ping request $s.\mathsf{ping}$, which returns s, and y becomes a reference to s.
3. The alias operation $s.\mathsf{alias}\langle x\rangle$ is performed in a self-inflicted manner yielding an alias chain $s{\gg}x$.
4. Since $s{\gg}x$ and y is a reference to s, the self-inflicted call $y.\mathrm{l}$ is forwarded to x.

5. The forwarded method call $x.\mathrm{l}$ diverges by definition.
6. As a consequence, $C[s.\mathsf{ping}]\not\Downarrow$

The execution of $C[s.\mathsf{surrogate}]$ gives rise to the following computation:

1. x becomes a reference to $[\,\mathrm{l}{=}\Omega\,,\mathrm{k}{=}\Omega\,]$.
2. The invocation

$$[\,\mathrm{l}{=}\mathrm{id}\,,\mathrm{k}{=}\varsigma(s)\mathsf{let}\,y=[\cdot]\,\mathsf{in}\,s.\mathsf{alias}\langle x\rangle;y.\mathrm{l}\,].\mathrm{k}$$

   implies a self-inflicted surrogation request $s.\mathsf{surrogate}$ which returns a clone s' and yields an alias chain $s{\gg}s'$, and y becomes a reference to s'.
3. The alias operation $s.\mathsf{alias}\langle x\rangle$ is performed in a self-inflicted manner changing the alias chain as $s{\gg}x$.
4. Since y is a reference to s' and s is the current self, the method call $s'.\mathrm{l}$ is externally invoked.
5. Since method l is defined as the identity, the computation returns s'.
6. As a consequence, $C[s.\mathsf{surrogate}]\Downarrow$.

All of these examples of "observable self-surrogation" can be interpreted as *programming errors*. In each case, the current method containing the surrogation request has the complete knowledge of the object and its surrogated counterpart, but nevertheless operates on either of the two in an obviously "stupid" and avoidable manner. In those cases, it is not really a fault of the semantics, but rather a fault of the programmer.

By contrast, unsafe external surrogations represent a much more serious problem. In that case, "external" object clients can distinguish whether an object has moved or not.

7.7.2 Countering external surrogations

In this section, we give two counterexamples showing that external surrogations (resp. migration) may be unsafe in Øjeblik (resp. in Obliq). These two examples exhibit problems with *protection* and *serialisation*.

Protection problems Consider the context:

$$C[\cdot] := \mathsf{let}\ x = \mathbb{O}\ \mathsf{in}\ \mathsf{let}\ y = [\cdot]\ \mathsf{in}\ y.\mathrm{k}\langle x\rangle \tag{7.9}$$

with $\mathbb{O}$ instantiated as:

$$\mathbb{O} := [\,\mathrm{k}{=}\varsigma(s,z)z.\mathsf{clone}\,].$$

The execution of $C[x.\mathsf{ping}]$ yields the following computation:

1. x becomes a reference to $\mathbb{O}$.

2. The external request $x.\mathsf{ping}$ is served and returns x.
3. The external method call $x.\mathrm{k}\langle x\rangle$ is invoked.
4. Method k invokes a self-inflicted clone request $x.\mathsf{clone}$ which yields a clone x'.
5. Method k completes and, as a consequence, $C[x.\mathsf{ping}]\Downarrow$.

The execution of $C[x.\mathsf{surrogate}]$ yields the following computation:

1. x becomes a reference to $\mathbb{O}$.
2. The external surrogation $x.\mathsf{surrogate}$ returns a clone x' and yields an alias chain $x \gg x'$; y becomes a reference to x'.
3. The external method call $x'.\mathrm{k}\langle x\rangle$ is invoked.
4. Method k (of x') invokes a clone request $x.\mathsf{clone}$.
5. Since the current self is x', the request $x.\mathsf{clone}$ is external and, *by protection*, fails.

Serialisation problems Consider the context:

$$C[\cdot] := \mathsf{let}\ x = \mathbb{O}\ \mathsf{in}\ [\cdot].\mathrm{l}\langle x\rangle \tag{7.10}$$

with $\mathbb{O}$ instantiated as:

$$\mathbb{O} := [\,\mathrm{k}{=}\mathrm{id}\,, \mathrm{l}{=}\varsigma(s,z)z.\mathrm{k}\,].$$

The execution of $C[x.\mathsf{ping}]$ yields the following computation:

1. x becomes a reference to $\mathbb{O}$.
2. The request $x.\mathsf{ping}$ is served as external and returns x.
3. The method call $x.\mathrm{l}\langle x\rangle$ is served as external.
4. Method l invokes a self-inflicted method call $x.\mathrm{k}$.
5. Since method k is defined as the identity function, the computation returns x.
6. As a consequence, $C[x.\mathsf{ping}]\Downarrow$.

On the other hand, the execution of $C[x.\mathsf{surrogate}]$ yields a different result. Here are the steps of its computation:

1. x becomes a reference to $\mathbb{O}$.
2. The request $x.\mathsf{surrogate}$ is served as external returning a clone x' and yielding an alias chain $x \gg x'$.
3. The method call $x'.\mathrm{l}\langle x\rangle$ is served as external.
4. Method l (of x') invokes an external method call $x.\mathrm{k}$.
5. Since x is a proxy for x', the method call is forwarded to x', and x becomes the new current self.
6. Since x is now the current self, the forwarded request $x'.\mathrm{k}$ is external in x'. But x' is locked because its method l has not returned yet. So, *by serialisation*, the request $x'.\mathrm{k}$ blocks.

7. As a consequence, $C[x.\mathsf{surrogate}] \not\Downarrow$.

Re-aliasing does not represent a problem for external surrogations because aliasing is a protected operation which can be served only if self-inflicted. So, summarising, external surrogations are unsafe essentially because alias nodes, like non-aliased ones, are still protected and serialised.

As regards serialisation, notice that in the computation of $C[x.\mathsf{surrogate}]$ of Counterexample 7.10, the key-point is when the alias node forwards the request, and the *current self changes* (see Item 5). This happens because alias nodes, like non-aliased ones, are active (c.f. 7.3.1), and therefore locked until previously forwarded requests have successfully signalled termination from their point of action. This is somehow in contradiction with the intuition that an alias node should be nothing more than a forwarder.

7.8 The amended semantics

In this section, we give a variant of the translational semantics seen in Section 7.4, where alias nodes forwards external requests. Our semantics provides a milder form of both protection and serialisation. In Section 7.12 we will prove that, in this new semantics, external surrogations are safe (c.f. Theorem 7.12.1).

To better explain our design choices we recall the four questions, concerning alias nodes, addressed in Section 7.3.3.

1. *What* is the current self when forwarding requests?
2. *Who* is in charge of sending the result of a forwarded external request?
3. *When* does the forwarding take place?
4. *Which* requests are forwarded and which requests fail in an alias node?

Our semantics will behave as follows:

What? Let a be an alias node forwarding requests to b, that is, $a \gg b$. Let c be a third object invoking a method of a. Then, when serving the (external) request, the alias a simply forwards the request to b, and *c is still the current self*. Roughly speaking, it is as if c invokes directly a method of b. The self-inflicted case is trivial because would be $a = c$.

Who? As above, let $a \gg b$ and c be a third object invoking a method of a. Since alias nodes simply forward requests, *the transmission of the result of the request is delegated to b*. As a consequence, once the request has been forwarded, the mutex on the alias node a can be released.

When? Incoming external requests do not have to wait anymore until previously forwarded external requests have successfully signalled termination from their point of action. The mutex on the alias node is released when forwarding.

$$
\begin{array}{l}
\mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, s_{\mathrm{a}}\,\rangle \stackrel{\mathrm{def}}{=} s(l,k).(\boldsymbol{\nu} k^*)\Big(\\
\quad \mathsf{if}\ [k{=}k_{\mathrm{i}}]\ \mathsf{then}\\
\qquad \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln}\_(r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid (\boldsymbol{\nu} s^*)\left(\,\overline{r}\langle s^*, k^*\rangle \mid \mathsf{newA}_{\mathbb{O}}\langle\, s^*, s_{\mathrm{a}}\,\rangle\,\right)\ ;\\
\qquad\qquad \mathsf{ali}\_(s'_{\mathrm{a}}, r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s'_{\mathrm{a}}\,\rangle \mid \overline{r}\langle s'_{\mathrm{a}}, k^*\rangle\ ;\\
\qquad\qquad \mathsf{upd}_j\_(t', r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle\ ;\\
\qquad\qquad \mathsf{inv}_j\_(\widetilde{x}, r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle\ ;\\
\qquad\qquad \mathsf{sur}\_(r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle\ ;\\
\qquad\qquad \mathsf{png}\_(r) \triangleright \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle\\
\quad \mathsf{elif}\ [k{=}k_{\mathrm{e}}]\ \mathsf{then}\ \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid m_{\mathrm{i}}(k).\left(\,\overline{s_{\mathrm{a}}}\langle l, k\rangle \mid \overline{m_{\mathrm{e}}}\,\right)\\
\quad \mathsf{else}\qquad\qquad \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, s_{\mathrm{a}}\,\rangle \mid m_{\mathrm{e}}.\left(\,\overline{s}\langle l, k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k\,\right)\Big)
\end{array}
$$

Table 7.9: Øjeblik forwarding semantics — Alias manager

Which? Protected external requests are supposed to fail only when addressed to non-aliased nodes.

- Method invocations (as well as pings and surrogations) are always forwarded to the endpoint.
- Self-inflicted cloning and self-inflicted aliasing are performed at the alias node; external cloning and external aliasing are forwarded because they can reach a node in the alias chain where they are self-inflicted and therefore executable.
- Self-inflicted update requests are forwarded. External update requests are forwarded because they may reach a (non-aliased) object that serves them.

The requirements above can be implemented in the translational semantics by simply changing the alias manager as described in Table 7.9. In the remainder of the chapter, we will prove that if alias mangers behave as described in Table 7.9, then it holds that

$$C[x.\mathsf{ping}]\Downarrow \ \text{iff}\ \ C[x.\mathsf{surrogate}]\Downarrow$$

for precisely those cases where the request $x.\mathsf{surrogate}$ is served as external.

7.9 Properties of object managers

A crucial property in object-oriented languages is the uniqueness of objects. The Lπ constraint on the output capability guarantees this property.

Lemma 7.9.1 (Uniqueness of objects) *Let a be an Øjeblik term. If $[\![\, a \,]\!]_p^k \Rightarrow Z$ with*

$$\text{either}\ \ Z \equiv (\boldsymbol{\nu}\widetilde{z})\,(\, M \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \ldots\,\rangle\,)\ \ \text{or}\ \ Z \equiv (\boldsymbol{\nu}\widetilde{z})\,(\, M \mid \mathsf{AM}_{\mathbb{O}}\langle\, s, \ldots\,\rangle\,)$$

then $s \in \widetilde{z}$ and s does not appear free in input position within M.

Proof: By inspection of the encoding. If a manager is present, it must have been created at some point as described in the encoding, because initially, there is none. Upon creation, its name s is bound. Since we only consider reductions, the name remains bound. Finally, the encoding shows that managers are only restarted if the former incarnation disappears. Since there are never two copies restarted, and only the output capability of channels may be transmitted, the uniqueness of the receptor s is preserved. $\Box$

We now analyse, referring directly to Figure 7.1, how the shape of the context around a particular object manager evolves during computation (c.f. Lemma 7.9.2). Later on, we will need a special case of this result (Lemma 7.9.3) in the proof of Theorem 7.12.1.

Observation 1: Pre-processing does not change the "state" of object managers. At any time, an object/alias manager is ready to receive a request $\overline{s}\langle l,k\rangle$ with $k_{\rm e}{\neq}k{\neq}k_{\rm i}$. The manager is restarted afterwards, but there will be a process $m_{\rm e}.(\,\overline{s}\langle l,k_{\rm e}\rangle \mid \overline{m_{\rm i}}k\,)$ that replaces the consumed request. Let us assume requests $\overline{s}v_j$, with $v_j := \langle\, l_j,k_j\,\rangle$ for $1{\le}j{\le}h$, (and $\widetilde{v}{:=}v_1..v_h$) are pre-processed by the object manager $\mathsf{OM}_{\mathbb{O}}\langle\, s,m_{\rm e},m_{\rm i},k_{\rm e},k_{\rm i},\widetilde{t}\,\rangle$, so $k_{\rm e}{\neq}k_j{\neq}k_{\rm i}$ for all $1{\le}j{\le}h$. Then:

$$\mathsf{PP}_{\mathbb{O}}\langle\, s,m_{\rm e},m_{\rm i},k_{\rm e},\widetilde{v}\,\rangle \stackrel{\rm def}{=} \prod_{1\le j\le h} m_{\rm e}.(\;\overline{s}\langle l_j,k_{\rm e}\rangle \mid \overline{m_{\rm i}}k_j\;)$$

Observation 2: During the evolution of object managers, the internal key $k_{\rm i}$ may be extruded to object clients, whereas $m_{\rm e},m_{\rm i},k_{\rm e}$ may not. Assume that an inv_j-request along s is pre-processed at $\mathsf{OM}_{\mathbb{O}}\langle\, s,m_{\rm e},m_{\rm i},k_{\rm e},k_{\rm i},\widetilde{t}\,\rangle$, gets the mutex $m_{\rm e}$ and re-enters along s with key $k_{\rm e}$. At that point, according to the semantics, a fresh internal key k^* is created and extruded to the corresponding method body. The names $\widetilde{n} := m_{\rm e},m_{\rm i},k_{\rm e}$ are never extruded; they constitute the proper boundary of a manager during computation. Observation 2 will be crucial to explain the evolution of object managers described in Figure 7.1. For simplicity we restrict our analysis to object managers, but a similar argument applies to alias managers.

Lemma 7.9.2 (Object manager evolution) *Let a be an Øjeblik term. If $[\![\, a\,]\!]_p^k \;\Rightarrow\; Z$ and $Z = E[\mathsf{OM}_{\mathbb{O}}\langle\, s,\ldots\rangle]$, then—without α-converting the name s—*

$$Z \equiv \widehat{E}[\;(\boldsymbol{\nu}\widetilde{n})\,(\;M_Z \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{n},k_{\rm i},\widetilde{t}\,\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle\, s,\widetilde{n},\widetilde{v}\,\rangle\;)\;]$$

where $\widetilde{n} := m_{\rm e},m_{\rm i},k_{\rm e}$ and M_Z is either of

"Z"	M_Z
F	$\overline{m_{\rm e}}$
A	$\overline{m_{\rm i}}k \;\mid\; \overline{s}\langle l,k_{\rm e}\rangle$
D	$\overline{m_{\rm i}}k \;\mid\; m_{\rm i}(k).\overline{m_{\rm e}}$
S	$\overline{m_{\rm i}}k \;\mid\; r^*(y,k').m_{\rm i}(k'').(\,\overline{r}\langle y,k''\rangle \;\mid\; \overline{m_{\rm e}}\,)$
T	$\overline{m_{\rm i}}k \;\mid\; m_{\rm i}(k'').(\,\overline{r}\langle y,k''\rangle \;\mid\; \overline{m_{\rm e}}\,).$

with "Z" denoting the "state" of Z as in Figure 7.1.

Proof: By induction on the length of $[\![\, a \,]\!]_p^k \Rightarrow Z$ for some fixed s, where we assume that the predecessor "state" Z' of Z is in one of the five described "states". Details can be found in Appendix A.4 □

Observation 3: An object manager is free, *if its external mutex is available*. In our semantics, a manager is willing to grant access to external requests, if the external mutex m_e occurs unguarded in the term that describes the current "state", so the general shape of a *free object* (and analogously *alias*) *manager* is:

$$\begin{array}{rcl}\mathsf{freeO}_{\mathbb{O}}\langle\, s, k_\mathrm{i}, \widetilde{t}, \widetilde{v}\,\rangle & \stackrel{\mathrm{def}}{=} & (\boldsymbol{\nu}\widetilde{n})\big(\, \overline{m_\mathrm{e}} \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_\mathrm{i}, \widetilde{t}\,\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle\,\big) \\ \mathsf{freeA}_{\mathbb{O}}\langle\, s, k_\mathrm{i}, s_\mathrm{a}, \widetilde{v}\,\rangle & \stackrel{\mathrm{def}}{=} & (\boldsymbol{\nu}\widetilde{n})\big(\, \overline{m_\mathrm{e}} \mid \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_\mathrm{i}, s_\mathrm{a}\,\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle\,\big)\end{array}$$

where the keys mentioned in $\widetilde{v}$ of $\mathsf{PP}_{\mathbb{O}}\langle \ldots \rangle$ neither match k_e nor k_i. Notice that $\mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t}\,\rangle \equiv (\boldsymbol{\nu}k_\mathrm{i})\, \mathsf{freeO}_{\mathbb{O}}\langle\, s, k_\mathrm{i}, \widetilde{t}, \emptyset\,\rangle$, and analogously for $\mathsf{newA}_{\mathbb{O}}\langle \ldots \rangle$.

Observation 4: An object manager is ready to commit, *if it may consume a valid pre-processed request*. The following lemma derives from the ability to commit to a valid external request—visible as the availability of a valid pre-processed request, i.e., a request carrying k_e—the shape of the object manager before and after commitment, including all of its current pre-processed requests.

Lemma 7.9.3 (Committing object manager) *Let a be an Øjeblik term. If $[\![\, a \,]\!]_p^k \Rightarrow Z$ and $Z \equiv E[\, \overline{s}\langle l, k_\mathrm{e}\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_\mathrm{i}, \widetilde{t}\,\rangle\,]$ with $\widetilde{n} = m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}$ and $l \in \{\mathsf{inv}_j\_\langle \widetilde{x}, r\rangle, \mathsf{png}\_r, \mathsf{sur}\_r\}$, then $Z \stackrel{\tau}{\longrightarrow} Z'$ for*

$$\begin{array}{rcl} Z & \equiv & \widehat{E}[\, (\boldsymbol{\nu}\widetilde{n})\big(\overline{m_\mathrm{i}}k \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_\mathrm{i}, \widetilde{t}\,\rangle \mid \overline{s}\langle l, k_\mathrm{e}\rangle\big)\,] \\ Z' & \equiv & \widehat{E}[\, (\boldsymbol{\nu}\widetilde{n})\big(\overline{m_\mathrm{i}}k \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \mid (\boldsymbol{\nu}k^*)(\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \mid \mathsf{CM}[\, X_l\langle\, s\,\rangle_{r^*}^{k^*}\,]\,)\big)\,]\end{array}$$

for some key k, some set $\widetilde{v}$ of pre-processed requests, and $X_l\langle\, s\,\rangle$ denoting the respective continuation behaviour of Table 7.6.

Proof: According to Lemma 7.9.2, the property holds in "state" A which is the only "state" that matches the premise. □

As special cases, for $l \in \{\mathsf{png}\_r, \mathsf{sur}\_r\}$, of *committed* object managers, we define

$$\begin{array}{rcl} F[\cdot] & \stackrel{\mathrm{def}}{=} & (\boldsymbol{\nu}\widetilde{n}k^*)\big(\,\overline{m_\mathrm{i}}k \big| \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, [\cdot]\,\big) \\ \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle & \stackrel{\mathrm{def}}{=} & F[\mathsf{CM}[\,[\![\, s \,]\!]_{r^*}^{k^*}\,]] \\ \mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle & \stackrel{\mathrm{def}}{=} & F[\mathsf{CM}[\,[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]_{r^*}^{k^*}\,]]\end{array}$$

As we will see in Section 7.11, $\mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle$ model the object manager before and after surrogation, respectively.

7.10 On the absence of internal surrogation

In this section, we study how to formalise that a context $C[\cdot]$ will never lead to self-inflicted occurrences of the term $x.\mathsf{surrogate}$, when plugged into the hole.

In our semantics, the computation $[\![\, a \,]\!]_p^k \Rightarrow Z$ of an Øjeblik term a gives rise to a self-inflicted sur-request if $Z \equiv E[\ \overline{s}\langle \mathsf{sur\_}r, k\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle\ ]$, with $k{=}k_{\mathrm{i}}$. Since we must ensure that a sur-request *never* leads to internal surrogation, we must quantify over all derivatives of $[\![\, a \,]\!]_p^k$ and check for self-infliction in each of them.

Note that, starting from the term $[\![\, C[x.\mathsf{surrogate}] \,]\!]_p^k$, we should not be concerned with arbitrary sur-requests that appear at top-level during computation, but only with those that "arise from the request in the hole". However, this property is hard to determine for two different reasons: (1) *All* of the names mentioned in a sur-request may be changed dynamically by instantiation: s (due to forwarding), r (due to a call manager protocol), and k (due to pre-processing). (2) We have to consider arbitrarily many duplications of the request in the case that the hole appears, at the level of Øjeblik terms, within in a method body, which leads to replication in the π-calculus semantics. For both reasons, we need a tool to uniquely identify the various incarnations of the request.

Let $\mathsf{operate} \in \{\mathsf{ping}, \mathsf{surrogate}\}$, and let $\mathsf{op} \in \{\mathsf{png}, \mathsf{sur}\}$ denote the corresponding π-calculus labels (c.f. Table 7.5). We introduce the *additional* Øjeblik labels $\mathsf{operate}^\star \in \{\mathsf{ping}^\star, \mathsf{surrogate}^\star\}$. The intuition is that tagged labels are semantically treated exactly like their untagged counterparts, but can syntactically be distinguished from them. Consequently, we have to adapt the given semantics to take this into account. Table 7.10 presents the required straightforward additions, where we use the tagged π-calculus labels $\mathsf{op}^\star \in \{\mathsf{png}^\star, \mathsf{sur}^\star\}$, respectively: the individual clauses of the tagged semantics, written $[\![\![\, \cdot \,]\!]\!]_p^k$, are just copies of the clauses for the untagged requests.

As a result, both tagged and untagged requests can be sent to object and alias managers; object managers ignore the tagging information of requests and treat $\mathsf{op}^\star$-and op-requests identically, but alias managers preserve the tagging information since they simply forward requests. We also add a tag to all parameterised definitions and abbreviations when considering the tagged semantics, for instance, $\mathsf{OM}^\star$, $\mathsf{AM}^\star$, $\mathsf{pingO}^\star$ and $\mathsf{surO}^\star$ are defined as expected. Notice that the semantics is not affected by including tagging information. As a consequence, all results proved for the untagged semantics are valid for the tagged semantics as well.

Lemma 7.10.1 *Let x be an Øjeblik variable and $C[\cdot]$ an untagged Øjeblik context. Then: $C[x.\mathsf{operate}]\Downarrow$ iff $[\![\![\, C[x.\mathsf{operate}^\star] \,]\!]\!]_p^k \Downarrow_p$.*

Proof: The proof is in two steps:

$$[\![\, C[x.\mathsf{operate}] \,]\!]_p^k \Downarrow_p \ \text{ iff } \ [\![\![\, C[x.\mathsf{operate}] \,]\!]\!]_p^k \Downarrow_p \ \text{ iff } \ [\![\![\, C[x.\mathsf{operate}^\star] \,]\!]\!]_p^k \Downarrow_p$$

The first step compares the convergence behaviour of untagged requests—note that $C[x.\mathsf{operate}]$ is untagged by assumption—with respect to the tagged and the untagged

$$
\begin{aligned}
[\![\, a.\mathsf{surrogate}^\star \,]\!]_p^k &\stackrel{\text{def}}{=} (\boldsymbol{\nu} q)\left([\![\, a \,]\!]_q^k \;\mid\; q(y,i)\,.\,\overline{y}\langle \mathsf{sur}^\star\_p, i\rangle \right)\\
[\![\, a.\mathsf{ping}^\star \,]\!]_p^k &\stackrel{\text{def}}{=} (\boldsymbol{\nu} q)\left([\![\, a \,]\!]_q^k \;\mid\; q(y,i)\,.\,\overline{y}\langle \mathsf{png}^\star\_p, i\rangle \right)
\end{aligned}
$$

$$
\begin{array}{l}
\mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle \stackrel{\text{def}}{=} s(l,k).(\boldsymbol{\nu} k^*)\Big(\\
\text{if } [k{=}k_{\mathrm{i}}] \text{ then}\\
\quad \text{case } l \text{ of} \quad \ldots \quad \triangleright \quad \ldots\\
\qquad\qquad \mathsf{sur}\_(r) \triangleright \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, \widetilde{t}\,\rangle \mid [\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]_r^{k^*} \;;\\
\qquad\qquad \mathsf{png}\_(r) \triangleright \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, \widetilde{t}\,\rangle \mid [\![\, s \,]\!]_r^{k^*} \;;\\
\qquad\qquad \mathsf{sur}^\star\_(r) \triangleright \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, \widetilde{t}\,\rangle \mid [\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]_r^{k^*} \;;\\
\qquad\qquad \mathsf{png}^\star\_(r) \triangleright \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, \widetilde{t}\,\rangle \mid [\![\, s \,]\!]_r^{k^*}\\
\text{elif } [k{=}k_{\mathrm{e}}] \text{ then}\\
\quad \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, \widetilde{t}\,\rangle \;\Big|\; \text{case } l \text{ of} \quad \ldots \quad \triangleright \quad \ldots\\
\qquad\qquad\qquad\qquad \mathsf{sur}\_(r) \triangleright \mathsf{CM}[\, [\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]_{r^*}^{k^*} \,] \;;\\
\qquad\qquad\qquad\qquad \mathsf{png}\_(r) \triangleright \mathsf{CM}[\, [\![\, s \,]\!]_{r^*}^{k^*} \,] \;;\\
\qquad\qquad\qquad\qquad \mathsf{sur}^\star\_(r) \triangleright \mathsf{CM}[\, [\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]_{r^*}^{k^*} \,] \;;\\
\qquad\qquad\qquad\qquad \mathsf{png}^\star\_(r) \triangleright \mathsf{CM}[\, [\![\, s \,]\!]_{r^*}^{k^*} \,]\\
\text{else } \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle \;\Big|\; m_{\mathrm{e}}.\left(\overline{s}\langle l, k_{\mathrm{e}}\rangle \;\mid\; \overline{m_{\mathrm{i}}}k \right)\Big)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, s_{\mathrm{a}}\,\rangle \stackrel{\text{def}}{=} s(l,k).(\boldsymbol{\nu} k^*)\Big(\\
\text{if } [k{=}k_{\mathrm{i}}] \text{ then}\\
\quad \text{case } l \text{ of} \quad \ldots \quad \triangleright \quad \ldots\\
\qquad\qquad \mathsf{sur}\_(r) \triangleright \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle \;;\\
\qquad\qquad \mathsf{png}\_(r) \triangleright \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle \;;\\
\qquad\qquad \mathsf{sur}^\star\_(r) \triangleright \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle \;;\\
\qquad\qquad \mathsf{png}^\star\_(r) \triangleright \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle\\
\text{elif } [k{=}k_{\mathrm{e}}] \text{ then } \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \;\mid\; m_{\mathrm{i}}(k).\left(\overline{s_{\mathrm{a}}}\langle l, k\rangle \;\mid\; \overline{m_{\mathrm{e}}} \right)\\
\text{else} \qquad\quad \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, s_{\mathrm{a}}\,\rangle \;\mid\; m_{\mathrm{e}}.\left(\overline{s}\langle l, k_{\mathrm{e}}\rangle \;\mid\; \overline{m_{\mathrm{i}}}k \right)\Big)
\end{array}
$$

Table 7.10: Translational semantics — Additional tagged clauses

semantics. On untagged requests, the tagged and the untagged semantics behave exactly the same. The second step compares the convergence behaviour of a tagged term and its untagged counterpart with respect to the tagged semantics. By definition, the tagged semantics treats tagged and untagged requests in exactly the same manner. □

Tagging helps us to detect all "requests arising from the hole".

Definition 7.10.2 (External Contexts) *Let x be a variable and $C[\cdot]$ an untagged Øjeblik context. Then, $C[\cdot]$ is called* external *for $x.\mathsf{surrogate}$, if whenever*

$$[\![\, C[x.\mathsf{surrogate}^\star]\,]\!]_p^k \;\Rightarrow_\equiv\; E[\;\overline{s}\langle \mathsf{sur}^\star\_r, k\rangle \mid \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle\;]$$

it holds that $k \neq k_{\mathrm{i}}$.

We replay the definition using ping instead of surrogate. By definition of the semantics, an Øjeblik context $C[\cdot]$ is then external for $x.\mathsf{surrogate}$ if and only if it is external for $x.\mathsf{ping}$. For convenience, by abuse, we simply call $C[\cdot]$ to be *external for x*.

7.11 On committing external surrogations

By Lemma 7.9.3, when committing png and sur requests, we get the following two processes, respectively:

$$\mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle \quad \text{and} \quad \mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle.$$

The processes $\mathsf{pingO}_{\mathbb{O}}\langle\, s, \dots\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, \dots\rangle$ represent (the state of) the object manager before and after external surrogation, respectively. Notice that due to the use of nonces (c.f. page 112) in the implementation of the object and alias managers, in both processes $\mathsf{pingO}_{\mathbb{O}}\langle\, s, \dots\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, \dots\rangle$ the key k^* is fresh and therefore different from any key appearing in process $\mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle$.

In the following we show that processes $\mathsf{pingO}_{\mathbb{O}}\langle\, s, \dots\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, \dots\rangle$ are related by typed barbed equivalence $\simeq_{\Gamma;s}$ (Definition 7.1.7).

Theorem 7.11.1 *Let $\Gamma \vdash \mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle$ and $\Gamma \vdash \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle$ for some type environment Γ. Then:*

$$\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle \simeq_{\Gamma;s} \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle.$$

The proof of Theorem 7.11.1 requires several strong lemmas. The proofs of the latter can be found in Appendix A.4. In all the lemmas below the well-typedness requirement is necessary to ensure that (i) the environment sends along the channel s only values of the right type, (ii) the environment never uses channel s in input.

Lemma 7.11.2 proves that surrogation results in an alias pointing to a clone of the old object. The proof relies on the nonces (c.f. page 112) used in the implementation of object and alias managers, which control the interference with the environment.

Lemma 7.11.2 *If Γ is a suitable type environment for the processes below, then:*

$$\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle \approx_{\Gamma;s} (\boldsymbol{\nu} s^*)\big((\boldsymbol{\nu} k_{\mathrm{i}})\, \mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \widetilde{v}\,\rangle \;\big|\; \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \;\big|\; \overline{r}\langle s^*, k\rangle\big).$$

Lemma 7.11.3 proves that the alias object manager appearing in Lemma 7.11.2 behaves as a forwarder. This will allow us to apply the theory of $\mathrm{L}\pi$.

Lemma 7.11.3 *Let $\widetilde{v} := v_1..v_n$, and $v_j{:=}\langle\, l_j, k_j\,\rangle$ for $1{\leq}j{\leq}n$. If Γ is a suitable type environment for the processes below, then:*

$$(\boldsymbol{\nu} k_{\mathrm{i}})\, \mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \widetilde{v}\,\rangle \approx_{\Gamma;s} s \rhd s^* \Big|\prod_{1\leq j\leq n} \overline{s^*} v_j.$$

Notice that without the well-typedness hypothesis, after having received a wrong value along s, the two processes above would have a different behaviour.

Lemma 7.11.4 uses the algebraic law of $\mathrm{L}\pi^+$ of Lemma 7.1.11. Note that the proof of Lemma 7.11.4 is not a trivial application of Lemma 7.1.11. Indeed, unlike Proposition 4.1.1, the restriction is on the target name of the link and not on the source.

Lemma 7.11.4 *Let P be a process and s a channel such that $s \notin \mathrm{fc}(P)$. If Γ is a suitable type environment for the processes below, then:*

$$(\boldsymbol{\nu} s^*)\,\big(\, s \rhd s^* \mid P\,\big) \simeq_{\Gamma;s} P\{{}^{s}\!/\!{}_{s^*}\}.$$

Lemma 7.11.5 proves that pre-processing external requests does not preclude other requests.

Lemma 7.11.5 *Let $\widetilde{v} := v_1..v_n$ with $v_j{:=}\langle\, l_j, k_j\,\rangle$ and $k_j{\neq}k_{\mathrm{i}}$ for $1{\leq}j{\leq}n$. If Γ is a suitable type environment for the processes below, then:*

$$\prod_{1\leq j\leq n} \overline{s} v_j \;\Big|\; \mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t}\,\rangle \approx_{\Gamma;s} (\boldsymbol{\nu} k_{\mathrm{i}})\, \mathsf{freeO}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, \widetilde{t}, \widetilde{v}\,\rangle.$$

Lemma 7.11.6 is a technical lemma involving two confluent reductions.

Lemma 7.11.6 *Let $\widetilde{v} := v_1..v_n$ with $v_j{:=}\langle\, l_j, k_j\,\rangle$ and $k_j{\neq}k_{\mathrm{i}}$ for $1{\leq}j{\leq}n$. If Γ is a suitable type environment for the processes below, then:*

$$\overline{r}\langle s, k\rangle \;\big|\; (\boldsymbol{\nu} k_{\mathrm{i}})\, \mathsf{freeO}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, \widetilde{t}, \widetilde{v}\,\rangle \approx_{\Gamma} \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle.$$

Proof of Theorem 7.11.1

Proof: We recall that relations $\approx_\Gamma$ and $\approx_{\Gamma;s}$ imply $\simeq_{\Gamma;s}$. By subsequently applying Lemmas 7.11.2, 7.11.3, 7.11.4, 7.11.5, and 7.11.6 we have:

$$\begin{aligned}
&\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \widetilde{v}\,\rangle \\
&\quad\simeq_{\Gamma;s}\ (\boldsymbol{\nu} s^*)\left((\boldsymbol{\nu} k_{\mathrm{i}})\,(\mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \widetilde{v}\,\rangle)\ \mid\ \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle\ \mid\ \overline{r}\langle s^*, k\rangle\right) \\
&\quad\simeq_{\Gamma;s}\ (\boldsymbol{\nu} s^*)\left(\, s \rhd s^*\ \mid\ \prod_{1\le j\le n} \overline{s^*} v_j\ \mid\ \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle\ \mid\ \overline{r}\langle s^*, k\rangle\,\right) \\
&\quad\simeq_{\Gamma;s}\ \prod_{1\le j\le n} \overline{s} v_j\ \mid\ \mathsf{newO}_{\mathbb{O}}\langle\, s, \tilde{t}\,\rangle\ \mid\ \overline{r}\langle s, k\rangle \\
&\quad\simeq_{\Gamma;s}\ (\boldsymbol{\nu}\widetilde{m}\widetilde{k})\left(\overline{m_{\mathrm{e}}}\ \mid\ \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, \tilde{t}\,\rangle\ \mid\ \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle\right)\ \mid\ \overline{r}\langle s, k\rangle \\
&\quad\simeq_{\Gamma;s}\ \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle.
\end{aligned}$$

□

7.12 External Surrogation is Safe

In this section we give the main result of the chapter, that is, we prove the equation

$$x.\mathsf{ping} \doteq x.\mathsf{surrogate}$$

for all those contexts $C[\cdot]$ which are external for x (Definition 7.10.2).

Theorem 7.12.1 (Safety) *Let x be an object variable and $C[\cdot]$ an (untagged) context in Øjeblik. If $C[\cdot]$ is external for x, then*

$$C[x.\mathsf{ping}]\Downarrow \ \text{ iff }\ C[x.\mathsf{surrogate}]\Downarrow.$$

Proof: By Lemma 7.10.1 our proof obligation is equivalent to:

$$[\![\, C[x.\mathsf{ping}^\star]\,]\!]^k_p \Downarrow_p \ \text{ iff }\ [\![\, C[x.\mathsf{surrogate}^\star]\,]\!]^k_p \Downarrow_p.$$

This allows us to make use of the assumption on context $C[\cdot]$.
Since the semantics $[\![\,\cdot\,]\!]^k_p$ is compositional, there is an $\mathrm{L}\pi^+$ context $D[\cdot]$ and names y, j, q, such that $[\![\, C[x.\mathsf{operate}^\star]\,]\!]^k_p = D[\overline{y}\langle \mathsf{op}^\star\_q, j\rangle]$, where $D[\cdot]$ itself does not contain any message carrying a tagged request and $\Gamma \vdash D[\overline{y}\langle \mathsf{op}^\star\_q, j\rangle]$, for some typing Γ. We prove that

$$D[\overline{y}\langle \mathsf{png}^\star\_q, j\rangle]\Downarrow_p \ \text{ iff }\ D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle]\Downarrow_p$$

and concentrate on the implication from right to left. The converse is analogous.
Assume that $D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] \Downarrow_p$. If $D[N]\Downarrow_p$ for every process N, then this is also the case for $N = \overline{y}\langle \mathsf{png}^\star\_q, j\rangle$; otherwise, the $\mathsf{sur}^\star$-request must contribute to the barb. Therefore, we assume $D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] \ \Rightarrow\ P \downarrow_p$ and show that there is a corresponding sequence $D[\overline{y}\langle \mathsf{png}^\star\_q, j\rangle] \ \Rightarrow_{\simeq_\Gamma}\ Q \downarrow_p$ where $Q = P[{}^{\mathsf{png}^\star}\!/_{\mathsf{sur}^\star}]$. Since typed barbed equivalence $\simeq_\Gamma$ and relabelling preserve convergence, this suffices.

According to the discussion in Section 7.9, a reduction step due to an external request is *committing*, if it represents the consumption of a pre-processed request by an object manager. Now, we combine this knowledge with the fact that we have to concentrate on surrogation requests arising from the hole within the reduction sequence $D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] \;\Rightarrow\; P\downarrow_p$ and call *significant* ($\rightarrow_\mathrm{s}$) precisely those steps that exhibit the commitment to a $\mathsf{sur}^\star$-request. All the other steps can be considered *insignificant* because—as we show during the proof—they can be mimicked in a straightforward way by the $\mathsf{png}^\star$-ed counterpart.

Whenever $P \xrightarrow{\tau} P'$, we know that either

1. $P \equiv (\boldsymbol{\nu}\widetilde{z})\,(\,\overline{w}v \mid w(x).R \mid M\,)$ and $P' \equiv (\boldsymbol{\nu}\widetilde{z})\,(\,R\{^v\!/\!_x\} \mid M\,)$, or
2. $P \equiv (\boldsymbol{\nu}\widetilde{z})\,(\,\overline{w}v \mid \mathbf{!}\,w(x).R \mid M\,)$ and $P' \equiv (\boldsymbol{\nu}\widetilde{z})\,(\,R\{^v\!/\!_x\} \mid \mathbf{!}\,w(x).R \mid M\,)$.

A silent move $P \xrightarrow{\tau} P'$ (decomposed as above) is called

significant if case 1 applies where $\overline{w}v = \overline{s}\langle \mathsf{sur}^\star\_q, k_\mathrm{e}\rangle$ and $w(x).R = \mathsf{OM}\langle\, s, \widetilde{m}, k_\mathrm{e}, k_\mathrm{i}, \widetilde{t}\,\rangle$. We denote these $P \;\rightarrow_\mathrm{s}\; P'$.

insignificant if either

- case 2 applies, or
- case 1 applies where v does not carry a $\mathsf{sur}^\star$-request, or
- case 1 applies where $\overline{w}v = \overline{s}\langle \mathsf{sur}^\star\_q, j\rangle$ and $w(x).R = \mathsf{AM}\langle\, s, \widetilde{m}, k_\mathrm{e}, k_\mathrm{i}, \widetilde{t}\,\rangle$, or
- case 1 applies where $\overline{w}v = \overline{s}\langle \mathsf{sur}^\star\_q, j\rangle$ and $w(x).R = \mathsf{OM}\langle\, s, \widetilde{m}, k_\mathrm{e}, k_\mathrm{i}, \widetilde{t}\,\rangle$ with $k_\mathrm{i} \neq j \neq k_\mathrm{e}$.

We denote this as $P \;\rightarrow_\mathrm{i}\; P'$.

The missing case of $\overline{w}v = \overline{s}\langle \mathsf{sur}^\star\_q, k\rangle$ and $w(x).R = \mathsf{OM}\langle\, s, \widetilde{m}, k_\mathrm{e}, k_\mathrm{i}, \widetilde{t}\,\rangle$ with $k = k_\mathrm{i}$ is excluded by the assumption that $C[\cdot]$ is external for x (c.f. Definition 7.10.2). Note that starting with a $\mathsf{sur}^\star$-request in the hole, we will never encounter $\mathsf{png}^\star$-requests during the computation, and vice versa.

Now, we apply the classification of reduction steps to the given reduction sequence $D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] \;\Rightarrow\; P\downarrow_p$, assuming that it contains $d > 0$ significant steps (if $d = 0$, then $D[N]\Downarrow_p$ for all processes N):

$$
\begin{array}{l}
D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] = \\[4pt]
\qquad \begin{array}{lllllllllll}
P_{1,1} & \rightarrow_\mathrm{i} & P_{1,2} & \rightarrow_\mathrm{i} & \cdots & \rightarrow_\mathrm{i} & P_{1,n_1} & \rightarrow_\mathrm{s} & P_1 & = & P_{2,1} \\
P_{2,1} & \rightarrow_\mathrm{i} & P_{2,2} & \rightarrow_\mathrm{i} & \cdots & \rightarrow_\mathrm{i} & P_{2,n_2} & \rightarrow_\mathrm{s} & P_2 & = & P_{3,1} \\
\vdots & & \vdots & & & & \vdots & & \vdots & & \\
P_{d,1} & \rightarrow_\mathrm{i} & P_{d,2} & \rightarrow_\mathrm{i} & \cdots & \rightarrow_\mathrm{i} & P_{d,n_d} & \rightarrow_\mathrm{s} & P_d & = & P_{d+1,1} \\
P_{d+1,1} & \rightarrow_\mathrm{i} & P_{d+1,2} & \rightarrow_\mathrm{i} & \cdots & \rightarrow_\mathrm{i} & P_{d+1,n_{d+1}} & = & P\downarrow_p & &
\end{array}
\end{array}
$$

By (the tagged counterpart of) Lemma 7.9.3 it holds that:

$$P_h \;\;\equiv\;\; (\boldsymbol{\nu}\widetilde{z}_h)\,\big(\,M_h \mid \mathsf{surO}^\star_{\mathbb{O}}\langle\, s_h, q_h, k_h, \widetilde{t}_h, \widetilde{v}_h\,\rangle\,\big)$$

for some $\widetilde{z}_h$ and M_h. Now, we simulate the previous reduction sequence, which uses sur*-requests, but now using png*-requests and proceeding up to structural equivalence *and* barbed equivalence.

$$
\begin{array}{l}
D[\overline{y}\langle \mathsf{png}^\star\_q, j\rangle] = \\
\begin{array}{lllllllllllllll}
Q_{1,1} & \to_{\mathrm{i}} & Q_{1,2} & \to_{\mathrm{i}} & \cdots & \to_{\mathrm{i}} & Q_{1,n_1} & \to_{\mathrm{s}} & Q_1 & \simeq_\Gamma & \widehat{Q}_1 & \equiv & Q_{2,1} \\
Q_{2,1} & \to_{\mathrm{i}} & Q_{2,2} & \to_{\mathrm{i}} & \cdots & \to_{\mathrm{i}} & Q_{2,n_2} & \to_{\mathrm{s}} & Q_2 & \simeq_\Gamma & \widehat{Q}_2 & \equiv & Q_{3,1} \\
\vdots & & \vdots & & & & \vdots & & \vdots & & \vdots & & \vdots \\
Q_{d,1} & \to_{\mathrm{i}} & Q_{d,2} & \to_{\mathrm{i}} & \cdots & \to_{\mathrm{i}} & Q_{d,n_d} & \to_{\mathrm{s}} & Q_d & \simeq_\Gamma & \widehat{Q}_d & \equiv & Q_{d+1,1} \\
Q_{d+1,1} & \to_{\mathrm{i}} & Q_{d+1,2} & \to_{\mathrm{i}} & \cdots & \to_{\mathrm{i}} & Q_{d+1,n_{d+1}} & \stackrel{\mathrm{def}}{=} & Q{\downarrow_p} & & & &
\end{array}
\end{array}
$$

where:

$$Q_{h,g} \quad \stackrel{\mathrm{def}}{=} \quad P_{h,g}[{}^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}]$$

The insignificant reduction steps $\to_{\mathrm{i}}$ exist because of the Lemma 7.12.2. The significant reduction steps $Q_{h,n_h} \to_{\mathrm{s}} Q_h$ are analogous to their counterparts $P_{h,n_h} \to_{\mathrm{s}} P_h$. Precisely, by (the tagged counterpart of) Lemma 7.9.3, they give rise (up to structural equivalence) to a pingO* instead of a surO*, that is:

$$Q_h \;\equiv\; (\boldsymbol{\nu}\widetilde{z}_h)\;\big(\;M_h \mid \mathsf{pingO}^\star_{\mathbb{O}}\langle\, s_h, q_h, j_h, \widetilde{t}_h, \widetilde{v}_h\,\rangle\;\big)[{}^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}].$$

The processes $\widehat{Q}_h$ are defined as follows:

$$\widehat{Q}_h \;\stackrel{\mathrm{def}}{=}\; (\boldsymbol{\nu}\widetilde{z}_h)\;\big(\;M_h \mid \mathsf{surO}^\star_{\mathbb{O}}\langle\, s_h, q_h, j_h, \widetilde{t}_h, \widetilde{v}_h\,\rangle\;\big)[{}^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}]$$

The relations $Q_h \simeq_\Gamma \widehat{Q}_h$ hold by application of (the tagged counterparts of) Theorem 7.11.1 and Lemma 7.9.1, and since $\simeq_\Gamma$ is preserved by relabelling $[{}^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}]$. The relations $\widehat{Q}_h \equiv Q_{h+1,1}$ hold since

$$\widehat{Q}_h \equiv P_h[{}^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}] = P_{h+1,1}[{}^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}] \stackrel{\mathrm{def}}{=} Q_{h+1,1}.$$

Lemma 7.12.2 *Let a be an Øjeblikterm possibly containing a tagged request. If $[\![\, a \,]\!]^k_p \Rightarrow R \to_{\mathrm{i}} R'$, then $R[{}^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}] \to_{\mathrm{i}} R'[{}^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}]$ and $R[{}^{\mathsf{sur}^\star}/_{\mathsf{png}^\star}] \to_{\mathrm{i}} R'[{}^{\mathsf{sur}^\star}/_{\mathsf{png}^\star}]$.*

Proof: By case analysis on the four different shapes of insignificant steps. In each of them, the relabelling distributes over the components of R, which allows us afterwards to derive the corresponding reduction step. □

This concludes the proof of Theorem 7.12.1. □

Chapter 8

Conclusions

In this thesis we have introduced, Local π, in short Lπ, an asynchronous variant of Milner's π-calculus designed to model concurrent and distributed programming languages. We have developed the foundational theory of Lπ, and given a certain number of applications. Below, we summarise the main achievements of the thesis.

Local π-calculus We have given a number of algebraic laws which are valid in Local π but not in synchronous or asynchronous π-calculus. We have given two labelled characterisations of barbed congruence in Lπ. Technical differences with the well-known bisimilarities for the π-calculus or the asynchronous π-calculus are:

1. Our bisimilarities are congruence relations and therefore do not have to be closed under substitutions.
2. Our bisimilarities are in ground style (without universal quantification on the received names).
3. Our bisimilarities characterise barbed congruence in absence of matching construct for testing equality between names.

The bisimilarities proposed in this thesis can be enhanced by means of up to proof techniques.

We have also given a fully-abstract encoding of polyadic Local π into monadic Local π. Results showing that monadic calculi are as expressive as their polyadic extensions are useful in theoretical developments, since it is technically simpler to work with a monadic calculus.

The three fundamental ingredients of the Local π-calculus are: *asynchrony*, absence of *name-testing*, and the constraint on the *output capability*. However, the theory of Lπ can be applied also to calculi where the usage of some names goes beyond the syntax of Lπ. For instance, there could be a distinct set of synchronous names, or names that can be tested for identity (see Chapter 7). A type system could be used to distinguish between "Lπ names" and the other names, and the theory of Lπ can then be applied to the former.

The following applications of the theory of Lπ are presented in the thesis:

1. An alternative way to express name substitution in terms of the other operators of the calculus.
2. The proof that the *delayed input* (a form of non-blocking input prefixing) is a derivable construct in Lπ.
3. Stronger forms of Milner's *replication theorems*.
4. An optimisation of the encoding of *call-by-name λ-calculus*.
5. An encoding of *strong call-by-name λ-calculus*.
6. Some algebraic laws for the Join-calculus.
7. Some non-full abstraction and full abstraction results for Boreale's encoding of *external mobility* (when global names are communicated) in terms of *internal mobility* (when only private names are communicated).
8. The proof that Thielecke's axiomatic semantics of his *Continuation Passing Style calculus* is operationally sound.
9. Finally, a major application of the theory of Lπ is the study of the correctness of object migration in Obliq, as discussed below.

Equating names We have showed that the theory of Local π can be adapted to the asynchronous π-calculus, with some minor changes. As an application, a fully-abstract encoding of asynchronous Fusion calculus into asynchronous π-calculus is presented. Roughly, fusions of names are encoded in terms of Honda's equators.

Mobile objects as mobile processes We have given a formal translational semantics of (an appropriate abstraction of) Obliq into (an appropriate typed extension of) Lπ. In Obliq, objects are not physically moved during migration, rather a clone of the object is created at the target site and the original (local) object is turned into an alias for the (remote) clone. We have shown, by means of a series of counterexamples, that object migration in Obliq is not transparent to the object clients when considering the original semantics proposed by Cardelli. Roughly, this happens because alias nodes do not really behave like forwarders. Although the counterexamples have been presented in the source language Obliq, it is important to stress that we have found them by using the Lπ translational semantics. Consequenltly, we have proposed an amended semantics of Obliq and we have proved that this semantics guarantees transparency of object migration for a wide class of Obliq programs. To our knowledge, our proofs on object migration in Obliq are among the most complex applications of the theory of the π-calculus. These proofs would seem to be an interesting challenge for a theorem prover formalisation of the π-calculus as in [31, 71].

Appendix A

Proofs

A.1 Proofs of Chapter 3

A.1.1 Proof of Lemma 3.3.12

Proof:
1. $(\boldsymbol{\nu} a)P \asymp_{\mathrm{a}} (\boldsymbol{\nu} a)Q$.
It suffices to show that the relation

$$\mathcal{S}= \{((\boldsymbol{\nu} a)P\,,\ (\boldsymbol{\nu} a)Q) :\ P, Q \in \mathcal{L}\pi \text{ and } P \asymp_{\mathrm{a}} Q\}$$

is a $\asymp_{\mathrm{a}}$-bisimulation up-to structural congruence [1]. The proof is easy because the output actions performed by an $\mathcal{L}\pi$-process are always bound. We work up to structural congruence when dealing with the asynchronous clause for input.

2. $\forall\, R$ it holds that $P \mid R \asymp_{\mathrm{a}} Q \mid R$.
We prove that the relation

$$\mathcal{S} = \{((\boldsymbol{\nu}\tilde{a})(P \mid R), (\boldsymbol{\nu}\tilde{a})(Q \mid R)) :\ P, Q, R \in \mathcal{L}\pi \text{ and } P \asymp_{\mathrm{a}} Q\}$$

is a $\asymp_{\mathrm{a}}$-bisimulation up to $\gtrsim$ and $\approx$ [2]. Let us consider the possible actions of $(\boldsymbol{\nu}\tilde{a})(P \mid R)$:

1. If $(\boldsymbol{\nu}\tilde{a})(P \mid R) \xrightarrow{\bar{b}(c)} A$, with c fresh and $b \notin \tilde{a}$, then there are two cases:
   (a) either $R \xrightarrow{\bar{b}(c)} R'$, for some R', and $(\boldsymbol{\nu}\tilde{a})(Q \mid R)$ can easily match this action;
   (b) or $P \xrightarrow{\bar{b}(c)} P'$ for some P'. This means $A = (\boldsymbol{\nu}\tilde{a})(P' \mid R)$. Since $P \asymp_{\mathrm{a}} Q$ there exist Q' such that $Q \xRightarrow{\bar{b}(c)} Q'$ and $P' \asymp_{\mathrm{a}} Q'$. So, $(\boldsymbol{\nu}\tilde{a})(Q \mid R) \xRightarrow{\bar{b}(c)} (\boldsymbol{\nu}\tilde{a})(Q' \mid R)$ and $((\boldsymbol{\nu}\tilde{a})(P' \mid R), (\boldsymbol{\nu}\tilde{a})(Q' \mid R)) \in \mathcal{S}$.

[1] It is a special case of the proof technique of Definition 3.3.9
[2] Actually, we only need to work up to $\approx$ on the right.

2. If $(\boldsymbol{\nu}\tilde{a})(P \mid R) \xrightarrow{b(c)} A$, with c fresh and $b \notin \tilde{a}$, then there are two cases:
   (a) either $R \xrightarrow{b(c)} R'$, for some R', and $(\boldsymbol{\nu}\tilde{a})(Q \mid R)$ can easily match this action;
   (b) or $P \xrightarrow{b(c)} P'$ and, since $P \asymp_{\mathrm{a}} Q$, there are two possible cases:
      i. either $Q \xRightarrow{b(c)} Q'$, with $P' \asymp_{\mathrm{a}} Q'$, and we can easily conclude;
      ii. or $Q \Longrightarrow Q'$, with $P' \asymp_{\mathrm{a}} Q' \mid [\![\bar{b}c]\!]$. This means that
      $(\boldsymbol{\nu}\tilde{a})(P \mid R) \xrightarrow{b(c)} (\boldsymbol{\nu}\tilde{a})(P' \mid R)$ and
      $(\boldsymbol{\nu}\tilde{a})(Q \mid R) \Longrightarrow (\boldsymbol{\nu}\tilde{a})(Q' \mid R)$ with
      $(\boldsymbol{\nu}\tilde{a})(P' \mid R)\, \mathcal{S}\, (\boldsymbol{\nu}\tilde{a})(Q' \mid [\![\bar{b}c]\!] \mid R) \equiv (\boldsymbol{\nu}\tilde{a})(Q' \mid R) \mid [\![\bar{b}c]\!]$.

3. If $(\boldsymbol{\nu}\tilde{a})(P \mid R) \xrightarrow{\tau} A$, there are four cases:
   (a) $P \xrightarrow{\tau} P'$. This case is easy.
   (b) $R \xrightarrow{\tau} R'$. This case is easy.
   (c) $P \xrightarrow{\bar{b}(c)} P'$, $R \xrightarrow{b(c)} R'$ and $(\boldsymbol{\nu}\tilde{a})(P \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{a}c)(P' \mid R')$. Since $P \asymp_{\mathrm{a}} Q$ there exists Q' such that $Q \xRightarrow{\bar{b}(c)} Q'$ and $P' \asymp_{\mathrm{a}} Q'$. This means that $(\boldsymbol{\nu}\tilde{a})(Q \mid R) \Longrightarrow (\boldsymbol{\nu}\tilde{a}c)(Q' \mid R')$ and
   $((\boldsymbol{\nu}\tilde{a}c)(P' \mid R'), (\boldsymbol{\nu}\tilde{a}c)(Q' \mid R')) \in \mathcal{S}$.
   (d) $P \xrightarrow{b(c)} P'$, $R \xrightarrow{\bar{b}(c)} R'$ and $(\boldsymbol{\nu}\tilde{a})(P \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{a}c)(P' \mid R')$. Since $P \asymp_{\mathrm{a}} Q$ there are two cases:
      i. either there exists Q' such that $Q \xRightarrow{b(c)} Q'$, with $P' \asymp_{\mathrm{a}} Q'$ and we can easily conclude;
      ii. or there exists Q' such that $Q \Longrightarrow Q'$ and $P' \asymp_{\mathrm{a}} Q' \mid [\![\bar{b}c]\!]$. In this case, by Lemma 3.3.11 (part 2), $R \approx (\boldsymbol{\nu}c)([\![\bar{b}c]\!] \mid R')$ and therefore $(\boldsymbol{\nu}\tilde{a})(P \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{a}c)(P' \mid R')$, $(\boldsymbol{\nu}\tilde{a})(Q \mid R) \Longrightarrow (\boldsymbol{\nu}\tilde{a})(Q' \mid R)$ with $(\boldsymbol{\nu}\tilde{a}c)(P' \mid R')\, \mathcal{S}\, (\boldsymbol{\nu}\tilde{a}c)(Q' \mid [\![\bar{b}c]\!] \mid R') \approx (\boldsymbol{\nu}\tilde{a})(Q' \mid R)$.

3. $\boldsymbol{a(x).\, P \asymp_{\mathrm{a}} a(x).\, Q}$.
It suffices to show that the relation

$$\mathcal{S} = \{(a(x).\, P\,, a(x).\, Q) :\ P, Q \in \mathcal{L}\pi \text{ and } P \asymp_{\mathrm{a}} Q\}$$

is a $\asymp_{\mathrm{a}}$-bisimulation. Since $\asymp_{\mathrm{a}}$ is a ground bisimilarity, (i.e., in the input clause we assume to receive only fresh names) the proof relies on the fact that $\asymp_{\mathrm{a}}$ is preserved by injective substitutions.

4. $\boldsymbol{!a(x).\, P \asymp_{\mathrm{a}} !a(x).\, Q}$.
We prove that the relation $\mathcal{S}$ defined below

$$\{(P_1 \mid !a(x).\, Q_1, P_2 \mid !a(x).\, Q_2) : P_1, P_2, Q_1, Q_2 \in \mathcal{L}\pi\ ,\ P_1 \asymp_{\mathrm{a}} P_2\ ,\ Q_1 \asymp_{\mathrm{a}} Q_2\}$$

is a $\precsim_{\mathrm{a}}$-bisimulation up to structural congruence [3]. If $P_1 \mid !a(x).\,Q_1 \xrightarrow{\mu} R$ for some action μ and some process R then there are three possible cases:

1. $P_1 \xrightarrow{\mu} P_1'$ and $R = P_1' \mid !a(x).\,Q_1$. Then, since $P_1 \precsim_{\mathrm{a}} P_2$, we can easily conclude.
2. $\alpha = a(w)$ and $P_1 \mid !a(x).\,Q_1 \xrightarrow{a(w)} P_1 \mid Q_1\{w/x\} \mid !a(x).\,Q_1$. This implies that on the other side $P_2 \mid !a(x).\,Q_2 \xrightarrow{a(w)} P_2 \mid Q_2\{w/x\} \mid !a(x).\,Q_2$. Since $Q_1 \precsim_{\mathrm{a}} Q_2$, and $\precsim_{\mathrm{a}}$ is preserved by injective substitutions, we have $Q_1\{w/x\} \precsim_{\mathrm{a}} Q_2\{w/x\}$. By Lemma 3.3.12 (part 2) we have $P_1 \mid Q_1\{w/x\} \precsim_{\mathrm{a}} P_2 \mid Q_1\{w/x\} \precsim_{\mathrm{a}} P_2 \mid Q_2\{w/x\}$. By Lemma 3.3.8 (part 3) we conclude that $P_1 \mid Q_1\{w/x\} \mid !a(x).\,Q_1 \;\mathcal{S}\; P_2 \mid Q_2\{w/x\} \mid !a(x).\,Q_2$.
3. $P_1 \xrightarrow{\overline{a}(c)} P_1'$ and $P_1 \mid !a(x).\,Q_1 \xrightarrow{\tau} (\boldsymbol{\nu} c)(P_1' \mid Q_1\{c/x\}) \mid !a(x).\,Q_1$. Since $P_1 \precsim_{\mathrm{a}} P_2$, there exists P_2' such that $P_2 \xRightarrow{\overline{a}(c)} P_2'$ and $P_1' \precsim_{\mathrm{a}} P_2'$. This means that $P_2 \mid !a(x).\,Q_2 \Longrightarrow (\boldsymbol{\nu} c)(P_2' \mid Q_2\{c/x\}) \mid !a(x).\,Q_2$. We reason as in the previous case, and by applying Lemmas 3.3.12(2), 3.3.8(3), and 3.3.12(1) we conclude.

□

A.1.2 Proof of Lemma 3.4.3

We already pointed out in Lemma 3.3.2 that there exists an operational correspondence on strong transitions between processes P and $[\![P]\!]$. Boreale also proved a weak operational correspondence between processes P and $[\![P]\!]$.

Lemma A.1.1 (Boreale [10]) *Let P be an $\mathrm{L}\pi$-process.*

1. *Suppose that $P \xRightarrow{\alpha} P'$. Then we have:*
   (a) *if $\alpha = a(c)$ then $[\![P]\!] \xRightarrow{a(c)} \gtrsim [\![P']\!]$;*
   (b) *if $\alpha = \overline{a}b$ then $[\![P]\!] \xRightarrow{\overline{a}(p)} \gtrsim (p \to b \mid [\![P']\!])$, with $p \notin \mathrm{fn}(P')$;*
   (c) *if $\alpha = \overline{a}(b)$ then $[\![P]\!] \xRightarrow{\overline{a}(p)} \gtrsim (\boldsymbol{\nu} c)(p \to c \mid [\![P']\!])$, with $p \notin \mathrm{fn}(P')$;*
   (d) *if $\alpha = \tau$ then $[\![P]\!] \xRightarrow{\tau} \gtrsim [\![P']\!]$.*
2. *Suppose that $[\![P]\!] \xRightarrow{\alpha} P_1$. Then there exists $P' \in \mathrm{L}\pi$ such that:*
   (a) *if $\alpha = a(c)$ then $P \xRightarrow{a(c)} P'$, with $P_1 \gtrsim [\![P']\!]$;*
   (b) *if $\alpha = \overline{a}(p)$ then:*
      i. *either $P \xRightarrow{\overline{a}b} P'$, with $p \notin \mathrm{fn}(P')$ and $P_1 \gtrsim (p \to b \mid [\![P']\!])$,*
      ii. *or $P \xRightarrow{\overline{a}(c)} P'$, with $p \notin \mathrm{fn}(P')$ and $P_1 \gtrsim (\boldsymbol{\nu} c)(p \to c \mid [\![P']\!])$;*
   (c) *if $\alpha = \tau$ then $P \xRightarrow{\tau} P'$ with $P_1 \gtrsim [\![P']\!]$.*

[3]It is a special case of the proof technique of Definition 3.3.9

This result will be used to prove Lemma 3.4.3

Proof:
$\Longrightarrow$
We show that the relation

$$\mathcal{S} = \{(\llbracket P \rrbracket, \llbracket Q \rrbracket) \mid P, Q \in \mathrm{L}\pi \text{ and } P \approx_{\mathrm{l}} Q\}$$

is a $\asymp_{\mathrm{a}}$-bisimulation up-to $\gtrsim$. Let us consider the three possible actions for $\llbracket P \rrbracket$:

1. If $\llbracket P \rrbracket \xrightarrow{\tau} P_1$, by lemma 3.3.2, there exists P' such that $P \xrightarrow{\tau} P'$ and $P_1 \gtrsim \llbracket P' \rrbracket$. Since $P \approx_{\mathrm{l}} Q$, there exists Q' such that $Q \Longrightarrow Q'$ and also $P' \approx_{\mathrm{l}} Q'$. By Lemma A.1.1, there exists Q_1 such that $\llbracket Q \rrbracket \stackrel{\tau}{\Longrightarrow} Q_1 \gtrsim \llbracket Q' \rrbracket$. We conclude by reasoning up to $\gtrsim$.
2. If $\llbracket P \rrbracket \xrightarrow{a(c)} P_1$, by Lemma 3.3.2, there exists P' such that $P \xrightarrow{a(c)} P'$ and $P_1 \gtrsim \llbracket P' \rrbracket$. Since $P \approx_{\mathrm{l}} Q$ there are two possibilities:
   (a) There exists Q' such that $Q \stackrel{a(c)}{\Longrightarrow} Q'$ and $P' \approx_{\mathrm{l}} Q'$. Then, by Lemma A.1.1, there exists Q_1 such that $\llbracket Q \rrbracket \stackrel{a(c)}{\Longrightarrow} Q_1 \gtrsim \llbracket Q' \rrbracket$. We conclude by reasoning up to $\gtrsim$.
   (b) There exists Q' such that $Q \Longrightarrow Q'$ and $P' \approx_{\mathrm{l}} Q' \mid \overline{a}c$. Then, by Lemma A.1.1, there exists Q_1 such that $\llbracket Q \rrbracket \Longrightarrow Q_1 \gtrsim \llbracket Q' \rrbracket$. Notice that $Q_1 \gtrsim \llbracket Q' \rrbracket$ implies $Q_1 \mid \llbracket \overline{a}c \rrbracket \gtrsim \llbracket Q' \mid \overline{a}c \rrbracket$. So, we conclude by reasoning up to $\gtrsim$.
3. If $\llbracket P \rrbracket \xrightarrow{\overline{a}(p)} P_1$, assuming p fresh, by Lemma 3.3.2, there are two possible cases:
   (a) $P \xrightarrow{\overline{a}b} P'$ and $P_1 \gtrsim (p \to b \mid \llbracket P' \rrbracket) = \llbracket p \triangleright b \mid P' \rrbracket$. Since $P \approx_{\mathrm{l}} Q$, there are still two cases:
      i. There exists a process Q' such that $Q \stackrel{\overline{a}d}{\Longrightarrow} Q'$ and $(p \triangleright b \mid P') \approx_{\mathrm{l}} (p \triangleright d \mid Q')$. By Lemma A.1.1 there exists Q_1 such that $\llbracket Q \rrbracket \stackrel{\overline{a}(p)}{\Longrightarrow} Q_1 \gtrsim (p \to d \mid \llbracket Q' \rrbracket) = \llbracket p \triangleright d \mid Q' \rrbracket$. So, $\llbracket P \rrbracket \xrightarrow{\overline{a}(p)} P_1 \gtrsim \llbracket p \triangleright b \mid P' \rrbracket$, and $\llbracket Q \rrbracket \stackrel{\overline{a}(p)}{\Longrightarrow} Q_1 \gtrsim \llbracket p \triangleright d \mid Q' \rrbracket$, and we can conclude since $(\llbracket p \triangleright b \mid P' \rrbracket, \llbracket p \triangleright d \mid Q' \rrbracket) \in \mathcal{S}$.
      ii. There exists a process Q' such that $Q \stackrel{\overline{a}(c)}{\Longrightarrow} Q'$ and $(p \triangleright b \mid P') \approx_{\mathrm{l}} (\boldsymbol{\nu} c)(p \triangleright c \mid Q')$. By Lemma A.1.1 there exists Q_1 such that $\llbracket Q \rrbracket \stackrel{\overline{a}(p)}{\Longrightarrow} Q_1 \gtrsim (\boldsymbol{\nu} c)(p \to c \mid \llbracket Q' \rrbracket)$. So, $\llbracket P \rrbracket \xrightarrow{\overline{a}(p)} P_1 \gtrsim \llbracket p \triangleright b \mid P' \rrbracket$ and $\llbracket Q \rrbracket \stackrel{\overline{a}(p)}{\Longrightarrow} Q_1 \gtrsim \llbracket (\boldsymbol{\nu} c)(p \triangleright c \mid Q') \rrbracket$, and we can conclude since $(\llbracket p \triangleright b \mid P' \rrbracket, \llbracket (\boldsymbol{\nu} c)(p \triangleright c \mid Q') \rrbracket) \in \mathcal{S}$.
   (b) $P \xrightarrow{\overline{a}(c)} P'$ and $P_1 \gtrsim (\boldsymbol{\nu} c)(p \to c \mid \llbracket P' \rrbracket) = \llbracket (\boldsymbol{\nu} c)(p \triangleright c \mid P') \rrbracket$. Since $P \approx_{\mathrm{l}} Q$, there are two possibilities:
      i. There exists Q' such that $Q \stackrel{\overline{a}b}{\Longrightarrow} Q'$ and $(\boldsymbol{\nu} c)(p \triangleright c \mid P') \approx_{\mathrm{l}} (p \triangleright b \mid Q')$. By Lemma A.1.1, there exists a process Q_1 such that $\llbracket Q \rrbracket \stackrel{\overline{a}(p)}{\Longrightarrow} Q_1 \gtrsim (p \to b \mid Q')$. So, $\llbracket P \rrbracket \xrightarrow{\overline{a}(p)} P_1 \gtrsim \llbracket (\boldsymbol{\nu} c)(p \triangleright c \mid P') \rrbracket$ and $\llbracket Q \rrbracket \stackrel{\overline{a}(p)}{\Longrightarrow} Q_1 \gtrsim \llbracket p \triangleright b \mid Q' \rrbracket$, and we can conclude since $(\llbracket (\boldsymbol{\nu} c)(p \triangleright c \mid P') \rrbracket, \llbracket p \triangleright b \mid Q' \rrbracket) \in \mathcal{S}$.

ii. There exists Q' such that $Q \stackrel{\overline{a}(c)}{\Longrightarrow} Q'$ and $(\boldsymbol{\nu}c)(p \triangleright c \mid P') \approx_1 (\boldsymbol{\nu}c)(p \triangleright c \mid Q')$. By Lemma A.1.1, there exists a process Q_1 such that $[\![Q]\!] \stackrel{\overline{a}(p)}{\Longrightarrow} Q_1 \gtrsim (\boldsymbol{\nu}c)(p \to c \mid [\![Q']\!])$. So, $[\![P]\!] \xrightarrow{\overline{a}(p)} P_1 \gtrsim [\![(\boldsymbol{\nu}c)(p \triangleright c \mid P')]\!]$ and $[\![Q]\!] \stackrel{\overline{a}(p)}{\Longrightarrow} Q_1 \gtrsim [\![(\boldsymbol{\nu}c)(p \triangleright c \mid Q')]\!]$, and we can conclude since $([\![(\boldsymbol{\nu}c)(p \triangleright c \mid P')]\!], [\![(\boldsymbol{\nu}c)(p \triangleright c \mid Q')]\!]) \in \mathcal{S}$.

$\Longleftarrow$

We show that the relation

$$\mathcal{S} = \{(P, Q) \mid P, Q \in \mathrm{L}\pi \ \text{ and } \ [\![P]\!] \asymp_{\mathrm{a}} [\![Q]\!]\}$$

is a link bisimulation. Let us consider the four possible actions.

1. If $P \xrightarrow{\tau} P'$, by Lemma 3.3.2, there exists P_1 such that $[\![P]\!] \xrightarrow{\tau} P_1 \gtrsim [\![P']\!]$. Since $[\![P]\!] \asymp_{\mathrm{a}} [\![Q]\!]$, there exists Q_1 such that $[\![Q]\!] \stackrel{\tau}{\Longrightarrow} Q_1$ and $P_1 \asymp_{\mathrm{a}} Q_1$. By Lemma A.1.1, there exists a process Q' such that $Q \stackrel{\tau}{\Longrightarrow} Q'$ and $Q_1 \gtrsim [\![Q']\!]$. So, we have $[\![P']\!] \lesssim P_1 \asymp_{\mathrm{a}} Q_1 \gtrsim [\![Q']\!]$. Since, on $\mathcal{L}\pi$-processes, $\lesssim \asymp_{\mathrm{a}} \gtrsim \subset \asymp_{\mathrm{a}}$ we conclude that $(P', Q') \in \mathcal{S}$.
2. If $P \xrightarrow{a(c)} P'$, by Lemma 3.3.2, there exists P_1 such that $[\![P]\!] \xrightarrow{a(c)} P_1 \gtrsim [\![P']\!]$. Since $[\![P]\!] \asymp_{\mathrm{a}} [\![Q]\!]$ we have two cases:
   (a) There exists a processes Q_1 such that $[\![Q]\!] \stackrel{a(c)}{\Longrightarrow} Q_1$ and $P_1 \asymp_{\mathrm{a}} Q_1$. By Lemma A.1.1 there exists a process Q' such that $Q \stackrel{a(c)}{\Longrightarrow} Q'$ and $Q_1 \gtrsim [\![Q']\!]$. So, we have $[\![P']\!] \lesssim P_1 \asymp_{\mathrm{a}} Q_1 \gtrsim [\![Q']\!]$. Since, on $\mathcal{L}\pi$-processes, $\lesssim \asymp_{\mathrm{a}} \gtrsim \subset \asymp_{\mathrm{a}}$ we conclude that $(P', Q') \in \mathcal{S}$.
   (b) There exists Q_1 such that $[\![Q]\!] \Longrightarrow Q_1$ and $P_1 \asymp_{\mathrm{a}} Q_1 \mid [\![\overline{a}c]\!]$. By Lemma A.1.1 there exists a process Q' such that $Q \Longrightarrow Q'$ and $Q_1 \gtrsim [\![Q']\!]$. Notice that $Q_1 \mid [\![\overline{a}c]\!] \gtrsim [\![Q' \mid \overline{a}c]\!]$. So, we have

   $$[\![P']\!] \lesssim P_1 \asymp_{\mathrm{a}} Q_1 \mid [\![\overline{a}c]\!] \gtrsim [\![Q' \mid \overline{a}c]\!].$$

   Since, on $\mathcal{L}\pi$-processes, $\lesssim \asymp_{\mathrm{a}} \gtrsim \subset \asymp_{\mathrm{a}}$, we conclude that $(P'\,,\, Q' \mid \overline{a}c) \in \mathcal{S}$.
3. If $P \xrightarrow{\overline{a}b} P'$, by Lemma 3.3.2, there exists P_1 such that $[\![P]\!] \xrightarrow{\overline{a}(p)} P_1 \gtrsim p \to b \mid [\![P']\!] = [\![p \triangleright b \mid P']\!]$. Since $[\![P]\!] \asymp_{\mathrm{a}} [\![Q]\!]$ there exists Q_1 such that $[\![Q]\!] \stackrel{\overline{a}(p)}{\Longrightarrow} Q_1$ and $P_1 \asymp_{\mathrm{a}} Q_1$. By Lemma A.1.1, we have two possibilities:
   (a) There exists Q' such that $Q \stackrel{\overline{a}d}{\Longrightarrow} Q'$ and $Q_1 \gtrsim p \to d \mid [\![Q']\!] = [\![p \triangleright d \mid Q']\!]$. So, we have $[\![p \triangleright b \mid P']\!] \lesssim P_1 \asymp_{\mathrm{a}} Q_1 \gtrsim [\![p \triangleright d \mid Q']\!]$. Since, on $\mathcal{L}\pi$-processes, it holds that $\lesssim \asymp_{\mathrm{a}} \gtrsim \subset \asymp_{\mathrm{a}}$, we can conclude that $((p \triangleright b \mid P')\,,\, (p \triangleright d \mid Q')) \in \mathcal{S}$.
   (b) There exists Q' such that $Q \stackrel{\overline{a}(c)}{\Longrightarrow} Q'$ and $Q_1 \gtrsim (\boldsymbol{\nu}c)(p \to c \mid [\![Q']\!]) = [\![(\boldsymbol{\nu}c)(p \to c \mid Q')]\!]$. So, we have $[\![p \triangleright b \mid P']\!] \lesssim P_1 \asymp_{\mathrm{a}} Q_1 \gtrsim [\![(\boldsymbol{\nu}c)(p \triangleright c \mid Q')]\!]$. Since, on $\mathcal{L}\pi$-processes, $\lesssim \asymp_{\mathrm{a}} \gtrsim \subset \asymp_{\mathrm{a}}$ we conclude that $((p \triangleright b \mid P')\,,\, (\boldsymbol{\nu}c)(p \triangleright c \mid Q')) \in \mathcal{S}$.

4. $P \xrightarrow{\overline{a}(c)} P'$, by Lemma 3.3.2 there exists P_1 such that $[\![P]\!] \xrightarrow{\overline{a}(p)} P_1 \gtrsim (\boldsymbol{\nu} c)(p \to c \mid [\![P']\!]) = [\![(\boldsymbol{\nu} c)(p \triangleright c \mid P')]\!]$. Since $[\![P]\!] \precsim_{\mathrm{a}} [\![Q]\!]$ there exists Q_1 such that $[\![Q]\!] \stackrel{\overline{a}(p)}{\Longrightarrow} Q_1$ and $P_1 \precsim_{\mathrm{a}} Q_1$. By Lemma A.1.1 we have two possibilities:

   (a) There exists Q' such that $Q \stackrel{\overline{a}b}{\Longrightarrow} Q'$ and $Q_1 \gtrsim p \to b \mid [\![Q']\!] = [\![p \triangleright b \mid Q']\!]$. So, we have $[\![(\boldsymbol{\nu} c)(p \triangleright c \mid P')]\!] \lesssim P_1 \precsim_{\mathrm{a}} Q_1 \gtrsim [\![p \triangleright b \mid Q']\!]$. Since, on $\mathcal{L}\pi$-processes, $\lesssim \precsim_{\mathrm{a}} \gtrsim \subset \precsim_{\mathrm{a}}$ we can conclude that $((\boldsymbol{\nu} c)(p \triangleright c \mid P')\,,\,(p \triangleright b \mid Q')) \in \mathcal{S}$.

   (b) There exists Q' such that $Q \stackrel{\overline{a}(c)}{\Longrightarrow} Q'$ and $Q_1 \gtrsim (\boldsymbol{\nu} c)(p \to c \mid [\![Q']\!])$. So, we have $[\![(\boldsymbol{\nu} c)(p \triangleright c \mid P')]\!] \lesssim P_1 \precsim_{\mathrm{a}} Q_1 \gtrsim [\![(\boldsymbol{\nu} c)(p \triangleright c \mid Q')]\!]$. Since, on $\mathcal{L}\pi$-processes, $\lesssim \precsim_{\mathrm{a}} \gtrsim \subset \precsim_{\mathrm{a}}$ we can conclude that $((\boldsymbol{\nu} c)(p \triangleright c \mid P')\,,\,((\boldsymbol{\nu} c)(p \triangleright c \mid Q')) \in \mathcal{S}$.

□

A.1.3 Complement to the Proof of Lemma 3.5.1

Proof: The proof is by induction on n. The case $n = 0$ is trivial because $\simeq_1^0 = \mathrm{L}\pi \times \mathrm{L}\pi$. If $n > 0$, by induction, we suppose that,

$$(\boldsymbol{\nu}\mathcal{L}')(P \mid R(n, L)) \doteq (\boldsymbol{\nu}\mathcal{L}')(Q \mid R(n, L)) \text{ and } P \stackrel{\mu}{\Longrightarrow} P'.$$

We proceed by case analysis on the action μ to show that Q can match the action μ.

1. $\mu = \tau$. Then:

$$(\boldsymbol{\nu}\mathcal{L}')(P \mid R(n, \mathcal{L}, \mathcal{M})) \stackrel{\tau}{\Longrightarrow} (\boldsymbol{\nu}\mathcal{L}')(P \mid (\overline{c_n^\tau} \oplus R(n-1, \mathcal{L}, \mathcal{M})))$$

   To match this reduction up to barbed bisimulation we have to have:

$$(\boldsymbol{\nu}\mathcal{L}')(Q \mid R(n, \mathcal{L}, \mathcal{M})) \stackrel{\tau}{\Longrightarrow} (\boldsymbol{\nu}\mathcal{L}')(Q_1 \mid (\overline{c_n^\tau} \oplus R(n-1, \mathcal{L}, \mathcal{M})))$$

   We make a further reduction on the lhs:

$$(\boldsymbol{\nu}\mathcal{L}')(P \mid (\overline{c_n^\tau} \oplus R(n-1, \mathcal{L}, \mathcal{M}))) \stackrel{\tau}{\Longrightarrow} (\boldsymbol{\nu}\mathcal{L}')(P' \mid R(n-1, \mathcal{L}, \mathcal{M}))$$

   Again this has to be matched by (note that we cannot run the process $R(n-1, \mathcal{L}, \mathcal{M})$ without losing a commitment $\overline{b_n}$ or $\overline{b'_n}$):

$$(\boldsymbol{\nu}\mathcal{L}')(Q_1 \mid (\overline{c_n^\tau} \oplus R(n-1, \mathcal{L}, \mathcal{M}))) \stackrel{\tau}{\Longrightarrow} (\boldsymbol{\nu}\mathcal{L}')(Q' \mid R(n-1, \mathcal{L}, \mathcal{M}))$$

   We observe that $Q \stackrel{\tau}{\Longrightarrow} Q_1 \stackrel{\tau}{\Longrightarrow} Q'$ and we conclude by applying the inductive hypothesis.

2. $\mu = \overline{a}b$. We may suppose $b \in L$. Then
   $(\boldsymbol{\nu}\mathcal{L}')(P \mid R(n, \mathcal{L}, \mathcal{M})) \stackrel{\tau}{\Longrightarrow}$
   $(\boldsymbol{\nu}\mathcal{L}')(P \mid \overline{c_n^a} \oplus a(x).\,(a' \triangleright x \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\})))$

This has to be matched by: $(\boldsymbol{\nu}\mathcal{L}')(Q \mid R(n, \mathcal{L}, \mathcal{M})) \stackrel{\tau}{\Longrightarrow}$
$(\boldsymbol{\nu}\mathcal{L}')(Q_1 \mid \overline{c_n^a} \oplus a(x).\, (a' \triangleright x \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\})))$

We take a further step on the lhs:

$(\boldsymbol{\nu}\mathcal{L}')(P \mid \overline{c_n^a} \oplus a(x).\, (a' \triangleright x \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\},))) \stackrel{\tau}{\Longrightarrow}$
$(\boldsymbol{\nu}\mathcal{L}')(P' \mid a' \triangleright b \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\}))$

This can be matched:

(a) either by
$(\boldsymbol{\nu}\mathcal{L}')(Q_1 \mid \overline{c_n^a} \oplus a(x).\, (a' \triangleright x \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\},))) \stackrel{\tau}{\Longrightarrow}$
$(\boldsymbol{\nu}\mathcal{L}')(Q' \mid a' \triangleright d \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\}))$.

This means that $Q \stackrel{\tau}{\Longrightarrow} Q_1 \stackrel{\overline{a}d}{\Longrightarrow} Q'$. By inductive hypothesis, and reasoning up to structural congruence, we can conclude that $a' \triangleright b \mid P' \simeq_1^{n-1} a' \triangleright d \mid Q'$.

(b) or by
$(\boldsymbol{\nu}\mathcal{L}')(Q_1 \mid \overline{c_n^a} \oplus a(x).\, (a' \triangleright x \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\}))) \stackrel{\tau}{\Longrightarrow}$
$(\boldsymbol{\nu}\mathcal{L}')((\boldsymbol{\nu}c)(Q' \mid a' \triangleright c) \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\}))$.

This means that that $Q \stackrel{\tau}{\Longrightarrow} Q_1 \stackrel{\overline{a}(c)}{\Longrightarrow} Q'$. By inductive hypothesis, and reasoning up to structural congruence, we can conclude that $a' \triangleright b \mid P' \simeq_1^{n-1} (\boldsymbol{\nu}c)(a' \triangleright c \mid Q')$.

3. $\mu = \overline{a}(c)$. We may suppose $c \notin \mathrm{fn}(Q)$. Then

$(\boldsymbol{\nu}\mathcal{L}')(P \mid R(n, \mathcal{L}, \mathcal{M})) \stackrel{\tau}{\Longrightarrow}$
$(\boldsymbol{\nu}\mathcal{L}')(P \mid \overline{c_n^a} \oplus a(x).\, (a' \triangleright x \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\})))$

This has to be matched by:
$(\boldsymbol{\nu}\mathcal{L}')(Q \mid R(n, \mathcal{L}, \mathcal{M})) \stackrel{\tau}{\Longrightarrow}$
$(\boldsymbol{\nu}\mathcal{L}')(Q_1 \mid \overline{c_n^a} \oplus a(x).\, (a' \triangleright x \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\})))$

We take a further step on the lhs:

$(\boldsymbol{\nu}\mathcal{L}')(P \mid \overline{c_n^a} \oplus a(x).\, (a' \triangleright x \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\}))) \stackrel{\tau}{\Longrightarrow}$
$(\boldsymbol{\nu}\mathcal{L}')((\boldsymbol{\nu}c)(P' \mid a' \triangleright c) \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\}))$

This can be matched:

(a) either by

$(\boldsymbol{\nu}\mathcal{L}')(Q_1 \mid \overline{c_n^a} \oplus a(x).\,(a' \triangleright x \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\}))) \stackrel{\tau}{\Longrightarrow}$
$(\boldsymbol{\nu}\mathcal{L}')(Q' \mid a' \triangleright b \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\}))$

This means that $Q \stackrel{\tau}{\Longrightarrow} Q_1 \stackrel{\overline{a}b}{\Longrightarrow} Q'$. By inductive hypothesis, and reasoning up to structural congruence, we can conclude that $(\boldsymbol{\nu}c)(a' \triangleright c \mid P') \simeq_1^{n-1} a' \triangleright b \mid Q'$.

(b) or by
$(\boldsymbol{\nu}\mathcal{L}')(Q_1 \mid \overline{c_n^a} \oplus a(x).\,(a' \triangleright x \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\}))) \stackrel{\tau}{\Longrightarrow}$
$(\boldsymbol{\nu}\mathcal{L}')((\boldsymbol{\nu}d)(Q' \mid a' \triangleright d) \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M} \cup \{a'\}))$
This means that $Q \stackrel{\tau}{\Longrightarrow} Q_1 \stackrel{\overline{a}(d)}{\Longrightarrow} Q'$. Since $c \notin \mathrm{fn}(Q)$ we also have that $Q \stackrel{\tau}{\Longrightarrow} Q_1$ and $Q_1 \stackrel{\overline{a}(c)}{\Longrightarrow} Q'\{c/d\}$. By inductive hypothesis, and reasoning up to structural congruence, we can conclude that $(\boldsymbol{\nu}c)(a' \triangleright c \mid P' \simeq_1^{n-1} (\boldsymbol{\nu}c)(a' \triangleright c \mid Q')$.

4. $\mu = aa'$. We may suppose a' is the first element in Ch$''\backslash$L (otherwise we rename and use injective substitution). Then
$(\boldsymbol{\nu}\mathcal{L}')(P \mid R(n, \mathcal{L}, \mathcal{M})) \stackrel{\tau}{\Longrightarrow}$
$(\boldsymbol{\nu}\mathcal{L}')(P \mid (\overline{c_n^{\overline{a}}} \oplus ((\boldsymbol{\nu}a')(\overline{a}a' \mid R(n-1, \mathcal{L} \cup \{a', \mathcal{M}\})))))$

This has to be matched by: $(\boldsymbol{\nu}\mathcal{L}')(Q \mid R(n, \mathcal{L}, \mathcal{M})) \stackrel{\tau}{\Longrightarrow}$
$(\boldsymbol{\nu}\mathcal{L}')(Q_1 \mid (\overline{c_n^{\overline{a}}} \oplus ((\boldsymbol{\nu}a')(\overline{a}a' \mid R(n-1, \mathcal{L} \cup \{a', \mathcal{M}\})))))$

We make a further reduction on the lhs:

$(\boldsymbol{\nu}\mathcal{L}')(P \mid (\overline{c_n^{\overline{a}}} \oplus ((\boldsymbol{\nu}a')(\overline{a}a' \mid R(n-1, \mathcal{L} \cup \{a', \mathcal{M}\})))))$
$\stackrel{\tau}{\Longrightarrow} (\boldsymbol{\nu}\mathcal{L}'a')(P' \mid R(n-1, \mathcal{L} \cup \{a', \mathcal{M}\}))$

This is matched by:

$$(\boldsymbol{\nu}\mathcal{L}')(Q_1 \mid (\overline{c_n^{\overline{a}}} \oplus ((\boldsymbol{\nu}a')(\overline{a}a' \mid R(n-1, \mathcal{L} \cup \{a', \mathcal{M}\}))))) \stackrel{\tau}{\Longrightarrow} Q''$$

We have two possibilities:

(a) $Q_1 \stackrel{\tau}{\Longrightarrow} Q'$ and $Q'' \equiv (\boldsymbol{\nu}\mathcal{L}'a')(Q' \mid \overline{a}a' \mid R(n-1, \mathcal{L} \cup \{a'\}, \mathcal{M}))$.

Then $Q \stackrel{\tau}{\Longrightarrow} Q_1 \stackrel{\tau}{\Longrightarrow} Q'$ and $P' \simeq_1^{n-1} Q' \mid \overline{a}a'$ by inductive hypothesis and up to structural congruence.

(b) $Q_1 \stackrel{aa'}{\Longrightarrow} Q'$ and $Q'' \equiv (\boldsymbol{\nu}\mathcal{L}'a')(Q' \mid R(n-1, \mathcal{L} \cup \{a', \mathcal{M}\}))$. Then $Q \stackrel{\tau}{\Longrightarrow} Q_1 \stackrel{aa'}{\Longrightarrow} Q'$ and $P' \simeq_1^{n-1} Q'$ by inductive hypothesis and up to structural congruence.

□

A.1.4 Proof of Theorem 3.6.4

Lemma A.1.2 *Let P and Q be two $\mathrm{L}\pi$-processes and p a name such that $p \notin \mathrm{fn}(P \mid Q)$. Then $(p \triangleright b \mid P) \approx_1 (p \triangleright b \mid Q)$ implies $P \approx_1 Q$.*

Proof: We show that the relation

$$\mathcal{S} = \{(P,Q) : P,Q \in \mathrm{L}\pi \text{ and } p \notin \mathrm{fn}(P \mid Q) \text{ and } (p \triangleright b \mid P) \approx_{\mathrm{l}} (p \triangleright b \mid Q)\}$$

is a link bisimulation up to structural congruence. The proof is straightforward and it is based on the fact that p is a fresh name and therefore there cannot be any interaction between the link $p \triangleright b$ and the processes P and Q. □

Lemma A.1.3 *Let P and Q be two $\mathrm{L}\pi$-processes and b a name such that $b \notin \mathrm{fn}(P \mid Q)$. Then $P \approx_{\mathrm{lut}} Q$ implies $P \mid \overline{a}b \approx_{\mathrm{l}} Q \mid \overline{a}b$.*

Proof: We show that the relation

$$\mathcal{S} = \{(P \mid \overline{a}b, Q \mid \overline{a}b) : \ P,Q \in \mathrm{L}\pi, \ b \notin \mathrm{fn}(P \mid Q), \ P \approx_{\mathrm{lut}} Q\} \cup \approx_{\mathrm{lut}}$$

is a link bisimulation up to link up to $\equiv$. It is enough to consider two cases:

1. If $P \mid \overline{a}b \xrightarrow{\alpha} P' \mid \overline{a}b$, that is, $P \xrightarrow{\alpha} P'$. The proof works smoothly.
2. If $P \mid \overline{a}b \xrightarrow{\alpha} P''$, that is, we consume the output particle $\overline{a}b$, then there are two possibilities:
   (a) $P \mid \overline{a}b \xrightarrow{\overline{a}b} P$. In this case the proof works smoothly.
   (b) $P \mid \overline{a}b \xrightarrow{\tau} P'$. This means that $P \xrightarrow{a(b)} P'$. Since $P \approx_{\mathrm{lut}} Q$ we have two cases.
      i. $Q \stackrel{a(b)}{\Longrightarrow} Q'$ and $P' \approx_{\mathrm{lut}} Q'$. In this case the proof works smoothly.
      ii. $Q \Longrightarrow Q'$ and $P' \approx_{\mathrm{lut}} Q' \mid \overline{a}b$. Then $Q \mid \overline{a}b \Longrightarrow Q' \mid \overline{a}b$ and since $P' \approx_{\mathrm{lut}} Q' \mid \overline{a}b$ we can conclude that $(P', Q' \mid \overline{a}b) \in \mathcal{S}$.

□

The following lemma shows that $\approx_{\mathrm{lut}}$ is preserved by the introduction of links.

Lemma A.1.4 *Let P and Q be two $\mathrm{L}\pi$-processes and p a name such that $p \notin \mathrm{fn}(P \mid Q)$. Then*

1. *$P \approx_{\mathrm{lut}} Q$ implies $(p \triangleright b \mid P) \approx_{\mathrm{lut}} (p \triangleright b \mid Q)$.*
2. *$P \approx_{\mathrm{lut}} Q$ implies $(\boldsymbol{\nu} c)(p \triangleright c \mid P) \approx_{\mathrm{lut}} (\boldsymbol{\nu} c)(p \triangleright c \mid Q)$.*
3. *$P\{p/c\} \approx_{\mathrm{lut}} (\boldsymbol{\nu} c)(p \triangleright c \mid Q)$ implies $(\boldsymbol{\nu} c)(p \triangleright c \mid P) \approx_{\mathrm{lut}} (\boldsymbol{\nu} c)(p \triangleright c \mid Q)$.*
4. *$P\{p/c\} \approx_{\mathrm{lut}} (p \triangleright b \mid Q)$,$c \notin \mathrm{fn}(Q)$, implies $(\boldsymbol{\nu} c)(p \triangleright c \mid P) \approx_{\mathrm{lut}} (p \triangleright b \mid Q)$.*

Proof:
Part 1.
We show that the relation

$$\mathcal{S} = \{(p \triangleright b \mid P \, , \, p \triangleright b \mid Q) : P,Q \in \mathrm{L}\pi, \ P \approx_{\mathrm{lut}} Q, \ p \notin \mathrm{fn}(P \mid Q)\}$$

is a link bisimulation up to link up to $\equiv$. Since $p \notin \mathrm{fn}(P \mid Q)$ there cannot be an interaction between the link and the processes. The most interesting case is when

$p \triangleright b \mid P \xrightarrow{p(w)} p \triangleright b \mid \overline{p}w \mid P$, with w fresh, and $p \triangleright b \mid Q \xrightarrow{p(w)} p \triangleright b \mid \overline{p}w \mid Q$. By Lemma A.1.3 we conclude that $((p \triangleright b \mid \overline{p}w \mid P), (p \triangleright b \mid \overline{p}w \mid Q)) \in \mathcal{S}$.

Part 2.
We show that the relation

$$\mathcal{S} = \{((\boldsymbol{\nu} c)(p \triangleright c \mid P)\,,\,(\boldsymbol{\nu} c)(p \triangleright c \mid Q)) : P \approx_{\mathrm{lut}} Q,\ p \notin \mathrm{fn}(P \mid Q)\}$$

is a link bisimulation up to link up to $\equiv$. We use the following abbreviation: $A = (\boldsymbol{\nu} c)(p \triangleright c \mid P)$ and $B = (\boldsymbol{\nu} c)(p \triangleright c \mid Q)$. Let us consider the possible actions of A.

1. If $A \xrightarrow{\tau} A_1$, since $p \notin \mathrm{fn}(P \mid Q)$, there exists P' such that $P \xrightarrow{\tau} P'$ and $A_1 = (\boldsymbol{\nu} c)(p \triangleright c \mid P')$. Since $P \approx_{\mathrm{lut}} Q$ there exists Q' such that $Q \Longrightarrow Q'$ and $P' \approx_{\mathrm{l}} Q'$. This means that there exists B_1 such that $B \Longrightarrow B_1$, $B_1 = (\boldsymbol{\nu} c)(p \triangleright c \mid Q')$ and $(A_1, B_1) \in \mathcal{S}$.
2. If $A \xrightarrow{a(b)} A_1$, with $a \neq c$ and $b \notin \mathrm{fn}(A \mid B)$. There are two cases:
   (a) $a \neq p$. This means that $P \xrightarrow{a(b)} P'$ and $A_1 = (\boldsymbol{\nu} c)(p \triangleright c \mid P')$. Since $P \approx_{\mathrm{lut}} Q$ there are two possibilities:
      i. $Q \xRightarrow{a(b)} Q'$ and $P' \approx_{\mathrm{lut}} Q'$. In this case $B \xRightarrow{a(b)} B_1$ where $B_1 = (\boldsymbol{\nu} c)(p \triangleright c \mid Q')$ and $(A_1, B_1) \in \mathcal{S}$.
      ii. $Q \Longrightarrow Q'$ and $P' \approx_{\mathrm{lut}} Q' \mid \overline{a}b$. In this case $B \Longrightarrow B_1 = (\boldsymbol{\nu} c)(p \triangleright c \mid Q')$ with $A_1\ \mathcal{S}\ (\boldsymbol{\nu} c)(p \triangleright c \mid (Q' \mid \overline{a}b)) \equiv B_1 \mid \overline{a}b$.

   (b) $a = p$. Since $p \notin \mathrm{fn}(P \mid Q)$, it holds that $A_1 = (\boldsymbol{\nu} c)(p \triangleright c \mid \overline{c}b \mid P)$. So, $B \xRightarrow{a(b)} B_1 = (\boldsymbol{\nu} c)(p \triangleright c \mid \overline{c}b \mid Q)$. Since $P \approx_{\mathrm{lut}} Q$ and $b \notin \mathrm{fn}(P \mid Q)$, by Lemma A.1.3, we have $(P \mid \overline{c}b) \approx_{\mathrm{lut}} (Q \mid \overline{c}b)$. We conclude that $A_1 \equiv (\boldsymbol{\nu} c)(p \triangleright c \mid P \mid \overline{c}b)\ \mathcal{S}\ (\boldsymbol{\nu} c)(p \triangleright c \mid Q \mid \overline{c}b) \equiv B_1$
3. If $A \xrightarrow{\overline{a}b} A_1$. This means that $P \xrightarrow{\overline{a}b} P'$, $\{a, b\} \cap \{c, p\} = \emptyset$ and $A_1 = (\boldsymbol{\nu} c)(p \triangleright c \mid P')$. Since $P \approx_{\mathrm{lut}} Q$, there are three possibilities:
   (a) $Q \xRightarrow{\overline{a}b} Q'$ and $P' \approx_{\mathrm{lut}} Q'$. This implies $B \xRightarrow{\overline{a}b} B_1 = (\boldsymbol{\nu} c)(p \triangleright c \mid Q')$ and $(A_1, B_1) \in \mathcal{S}$.
   (b) $Q \xRightarrow{\overline{a}(d)} Q'$ with $d \notin \mathrm{fn}(P \mid Q)$. We have two possible requirements for the derivatives. Let us consider the most difficult, that is, $(w \triangleright b \mid P') \approx_{\mathrm{lut}} (\boldsymbol{\nu} d)(w \triangleright d \mid Q')$ for some fresh name w. So, $B \xRightarrow{\overline{a}(d)} B_1 = (\boldsymbol{\nu} c)(p \triangleright c \mid Q')$ and $(w \triangleright b \mid A_1) \equiv (\boldsymbol{\nu} c)(p \triangleright c \mid (w \triangleright b \mid P'))\ \mathcal{S}\ (\boldsymbol{\nu} c)(p \triangleright c \mid ((\boldsymbol{\nu} d)(w \triangleright d \mid Q'))) \equiv (\boldsymbol{\nu} d)(w \triangleright d \mid B_1)$.
   (c) $Q \xRightarrow{\overline{a}h} Q'$, with $h \neq b$ and $(w \triangleright b \mid P')\ \mathcal{S}\ (w \triangleright h \mid Q')$ for some fresh name w. We reason as in the previous case.
4. If $A \xrightarrow{\overline{a}(b)} A_1$ the reasoning is similarly as that in case $A \xrightarrow{\overline{a}b} A_1$.

part 3.
The proof follows from Lemma A.1.4(2) and Proposition 3.2.3. Let r be a fresh name, by hypothesis we know that $P\{r/c\} \approx_{\mathrm{lut}} (\boldsymbol{\nu} c)(r \triangleright c \mid Q)$. By Lemma A.1.4(2), we have that

$$(\boldsymbol{\nu} r)(p \triangleright r \mid P\{r/c\}) \approx_{\mathrm{lut}} (\boldsymbol{\nu} r)(p \triangleright r \mid (\boldsymbol{\nu} c)(r \triangleright c \mid Q))$$

for p fresh. By Proposition 3.2.3 we have that

$$(\boldsymbol{\nu} r)(p \triangleright r \mid (\boldsymbol{\nu} c)(r \triangleright c \mid Q)) \equiv (\boldsymbol{\nu} c)((\boldsymbol{\nu} r)(p \triangleright r \mid r \triangleright c) \mid Q) \gtrsim (\boldsymbol{\nu} c)(p \triangleright c \mid Q).$$

Since $(\boldsymbol{\nu} c)(p \triangleright c \mid P) \equiv (\boldsymbol{\nu} r)(p \triangleright r \mid P\{r/c\})$ and $\equiv \approx_{\mathrm{lut}} \equiv \gtrsim \subset \approx_{\mathrm{lut}}$ we can conclude that $(\boldsymbol{\nu} c)(p \triangleright c \mid P) \approx_{\mathrm{lut}} (\boldsymbol{\nu} c)(p \triangleright c \mid Q)$.

Part 4.
As in the previous case the proof can be derived by Lemma A.1.4(2) and Proposition 3.2.3. □

A.2 Proof of Lemma 4.3.2

Proof: The proof is by transition induction. We prove the part 1. The proof of part 2 is similar.

1. $\mu = a(b)$. The only non trivial case is when the last rule applied to get $P \xrightarrow{a(b)} P'$ is `d-in`. By Lemma 3.3.3(1), since $\gtrsim$ implies $\succcurlyeq_{\mathrm{a}}$, it holds that

   $$\{\!|\, a(b)P \,|\!\} = (\boldsymbol{\nu} b)(a(b').\, b \to b' \mid \{\!|\, P \,|\!\})$$

   and

   $$(\boldsymbol{\nu} b)(a(b').\, b \to b' \mid \{\!|\, P \,|\!\}) \xrightarrow{a(b')} (\boldsymbol{\nu} b)(b \to b' \mid \{\!|\, P \,|\!\}) \succcurlyeq_{\mathrm{a}} \{\!|\, P \,|\!\}\{b'/b\}.$$
2. $\mu = \overline{a}b$. Straightforward.
3. $\mu = (\boldsymbol{\nu} b)\overline{a}b$. Straightforward.
4. $\mu = d(b)\overline{a}b$. The only non trivial case is when P' is derived by applying rule `o-in`. The result follows from the definition of the encoding.
5. $\mu = (\boldsymbol{\nu} d)d(b)\overline{a}b$. The only non trivial case is when the last rule applied to get P' is `o-`$\boldsymbol{\nu}$. The proof is easy.
6. $\mu = \tau$. This is the most interesting part. The only non trivial cases are when P' is derived by applying one of the following four rules: `com`, `s-com`, `cls`, and `s-cls`. Let us study in details these four cases.

   (a) Let `com` be the last rule applied for deriving $P \xrightarrow{\tau} P'$.

   $$\texttt{com:}\ \frac{P \xrightarrow{\overline{a}b} P' \qquad Q \xrightarrow{a(x)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{b/x\}}$$

   By induction hypothesis we have:

- $\{\!|\, P \,|\!\} \xrightarrow{(\boldsymbol{\nu} c)\overline{a}c} \succcurlyeq_{\mathrm{a}} c \to b \mid \{\!|\, P' \,|\!\}$
- $\{\!|\, Q \,|\!\} \xrightarrow{a(c)} \succcurlyeq_{\mathrm{a}} \{\!|\, Q' \,|\!\}\{c/x\}$

with $c \notin \mathrm{fn}(P)$. By Lemma 3.3.3 we have:

$$\begin{aligned}\{\!|\, P \mid Q \,|\!\} = \{\!|\, P \,|\!\} \mid \{\!|\, Q \,|\!\} \xrightarrow{\tau} & \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} c)(c \to b \mid \{\!|\, P' \,|\!\} \mid \{\!|\, Q' \,|\!\}\{c/x\}) \\ & \succcurlyeq_{\mathrm{a}} \{\!|\, P' \,|\!\} \mid \{\!|\, Q' \,|\!\}\{b/x\} \\ & = \{\!|\, P' \mid Q'\{b/x\} \,|\!\}.\end{aligned}$$

(b) Let s-com be the last rule applied for deriving $P \xrightarrow{\tau} P'$.

$$\texttt{s-com:}\frac{P \xrightarrow{\overline{a}c} P'}{a(b)P \xrightarrow{\tau} (\boldsymbol{\nu} b)(P'\{c/b\})}.$$

There are two possibilities: either $c \neq b$ or $c = b$.

i. If $c \neq b$ then $(\boldsymbol{\nu} b)(P'\{c/b\}) = P'\{c/b\}$. By induction hypothesis it holds that $\{\!|\, P \,|\!\} \xrightarrow{(\boldsymbol{\nu} d)\overline{a}d} \succcurlyeq_{\mathrm{a}} d \to c \mid \{\!|\, P' \,|\!\}$, with d fresh. Since $\{\!|\, a(b)P \,|\!\} \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu} b)(a(b').\, b \to b' \mid \{\!|\, P \,|\!\})$, by Proposition 3.2.3(2) and Lemma 3.3.3 it holds that:

$$\begin{aligned}\{\!|\, a(b)P \,|\!\} \xrightarrow{\tau} & \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} b)((\boldsymbol{\nu} d)(b \to d \mid d \to c) \mid \{\!|\, P' \,|\!\}) \\ & \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} b)(b \to c \mid \{\!|\, P' \,|\!\}) \\ & \succcurlyeq_{\mathrm{a}} \{\!|\, P' \,|\!\}\{c/b\} \\ & = \{\!|\, P\{c/b\} \,|\!\}.\end{aligned}$$

ii. If $c = b$ then $(\boldsymbol{\nu} b)(P'\{c/b\}) = (\boldsymbol{\nu} b)P'$. By induction hypothesis it holds that $\{\!|\, P \,|\!\} \xrightarrow{(\boldsymbol{\nu} d)\overline{a}d} \succcurlyeq_{\mathrm{a}} d \to c \mid \{\!|\, P' \,|\!\}$, with d fresh. Since $\{\!|\, a(b)P \,|\!\} \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu} b)(a(b').\, b \to b' \mid \{\!|\, P \,|\!\})$, by Proposition 3.2.3(2) and Lemma 4.3.1 it holds that:

$$\begin{aligned}\{\!|\, a(b)P \,|\!\} \xrightarrow{\tau} & \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} b)((\boldsymbol{\nu} d)(b \to d \mid d \to b) \mid \{\!|\, P' \,|\!\}) \\ & \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} b)(b \to b \mid \{\!|\, P' \,|\!\}) \\ & \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} b)\{\!|\, P' \,|\!\}.\end{aligned}$$

(c) Let cls be the last rule applied for deriving $P \xrightarrow{\tau} P'$.

$$\texttt{cls:}\frac{P \xrightarrow{\beta_b \overline{a}b} P' \quad Q \xrightarrow{a(b')} Q' \quad \mathrm{bn}(\beta_b) \cap \mathrm{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\tau} \beta_b(P' \mid Q'\{b/b'\})}$$

We can suppose $\beta_b = d(b)$ for some d. The case when $\beta_b = (\boldsymbol{\nu} d)d(b)$ is similar. By induction hypothesis it holds that:

- $\{\!|\, P \,|\!\} \xrightarrow{(\boldsymbol{\nu} c)\overline{a}c} \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} b)(d(b').\, b \to b' \mid c \to b \mid \{\!|\, P' \,|\!\})$, with $\{b', c\} \cap fn(P) = \emptyset$.
- $\{\!|\, Q \,|\!\} \xrightarrow{a(c)} \succcurlyeq_{\mathrm{a}} \{\!|\, Q' \,|\!\}\{c/b'\}$.

We recall that $\{\!|\, P \mid Q \,|\!\} = \{\!|\, P \,|\!\} \mid \{\!|\, Q \,|\!\}$. Then, by Lemma 3.3.3, we have:

$$\begin{aligned}\{\!|\, P \,|\!\} \mid \{\!|\, Q \,|\!\} \xrightarrow{\tau} & \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} c)(\boldsymbol{\nu} b)(d(b').\, b \to b' \mid c \to b \mid \{\!|\, P' \,|\!\} \mid \{\!|\, Q' \,|\!\}\{c/b'\}) \\ & \equiv (\boldsymbol{\nu} b)(d(b').\, b \to b' \mid (\boldsymbol{\nu} c)(c \to b \mid \{\!|\, P' \,|\!\} \mid \{\!|\, Q' \,|\!\}\{c/b'\}))\end{aligned}$$

$$\succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} b)(d(b').\, b \to b' \mid \{\!|\, P' \,|\!\} \mid \{\!|\, Q' \,|\!\}\{b/b'\})$$
$$= \{\!|\, d(b)(\{\!|\, P' \,|\!\} \mid \{\!|\, Q' \,|\!\}\{b/b'\}) \,|\!\}.$$

(d) Let `s-cls` be the last rule applied for deriving $P \xrightarrow{\tau} P'$.

$$\texttt{s-cls:}\frac{P \xrightarrow{\beta_c \overline{a} c} P' \qquad b \notin \mathrm{n}(\beta_c \overline{a} c)}{a(b)P \xrightarrow{\tau} \beta_c(P'\{c/b\})}$$

We can suppose $\beta_c = d(c)$ for some d. The case when $\beta_c = (\boldsymbol{\nu} d)d(c)$ is similar. By induction hypothesis it holds that:

$$\{\!|\, P \,|\!\} \xrightarrow{(\boldsymbol{\nu} h)\overline{a} h} \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} c)(d(c').\, c \to c' \mid h \to c \mid \{\!|\, P' \,|\!\})$$

with $\{c', h\} \cap fn(P) = \emptyset$.
Since $\{\!|\, a(b)P \,|\!\} \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu} b)(a(b').\, b \to b' \mid \{\!|\, P \,|\!\})$, by Proposition 3.2.3 and Lemma 3.3.3 we have:

$$\begin{aligned}
\{\!|\, a(b)P \,|\!\} \xrightarrow{\tau} & \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} h)(\boldsymbol{\nu} b)(b \to h \mid d(c').\, c \to c' \mid h \to c \mid \{\!|\, P' \,|\!\}) \\
& \equiv (\boldsymbol{\nu} b)((\boldsymbol{\nu} h)(b \to h \mid h \to c) \mid d(c').\, c \to c' \mid \{\!|\, P' \,|\!\}) \\
& \succcurlyeq_{\mathrm{a}} (\boldsymbol{\nu} b)(b \to c \mid d(c').\, c \to c' \mid \{\!|\, P' \,|\!\}) \\
& \succcurlyeq_{\mathrm{a}} d(c').\, c \to c' \mid \{\!|\, P' \,|\!\}\{c/b\} \\
& = \{\!|\, d(c)P'\{c/b\} \,|\!\}.
\end{aligned}$$

□

A.3 Proof of Section 5.3

A.3.1 Proof of Lemma 5.3.2

Proof: Each part of the lemma is proven by transition induction. The only subtle points arise in the proofs of parts 1(d) and 2(c) where also Lemma 5.3.1 is used. As an example, we show part 1(d). The only non trivial case is when the last rule applied for deriving $P \xrightarrow{\tau} P'$ is the communication rule (we suppose for simplicity that the communicated name is free; the case when is restricted can be easily accommodated).

$$\texttt{com:}\frac{P \xrightarrow{\overline{a}\langle \tilde{b} \rangle} P' \qquad Q \xrightarrow{a(\tilde{x})} Q' \qquad |\, \tilde{b} \,| = |\, \tilde{x} \,|}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{\tilde{b}/\tilde{x}\}}$$

By induction hypothesis we have that:

- $[\![\langle\!|\, P \,|\!\rangle]\!]_\Gamma \xrightarrow{\overline{a}(p)} \gtrsim (\boldsymbol{\nu} w)(p \to w \mid [\![\mathrm{OUT}\langle w, \tilde{b} \rangle]\!]) \mid [\![\langle\!|\, P' \,|\!\rangle]\!]_\Gamma$, $w \notin \mathrm{fn}(P')$

- $\llbracket \llparenthesis Q \rrparenthesis \rrbracket_\Gamma \xrightarrow{a(p)} \gtrsim (\boldsymbol{\nu}\tilde{x})(\llbracket \mathrm{INP}\langle p, \tilde{x}\rangle \rrbracket \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma)$ $w \notin \mathrm{fn}(Q')$.

Then using Lemma 5.3.1 we have that:

$$
\begin{aligned}
&\llbracket \llparenthesis P \mid Q \rrparenthesis \rrbracket_\Gamma \xrightarrow{\tau} \\
&\quad \equiv (\boldsymbol{\nu}w\tilde{x})(\llbracket \mathrm{OUT}\langle w, \tilde{b}\rangle \rrbracket \mid (\boldsymbol{\nu}p)(p \to w \mid \llbracket \mathrm{INP}\langle p, \tilde{x}\rangle \rrbracket) \mid \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma) \\
&\quad \gtrsim (\boldsymbol{\nu}w\tilde{x})(\llbracket \mathrm{OUT}\langle w, \tilde{b}\rangle \rrbracket \mid \llbracket \mathrm{INP}\langle w, \tilde{x}\rangle \rrbracket \mid \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma) \\
&\quad \equiv (\boldsymbol{\nu}w\tilde{x})(\, w(c_1).\, (\boldsymbol{\nu}c_2)(\llbracket \overline{c_1}c_2 \rrbracket \mid c_2(c_3).\, (\llbracket \overline{c_3}b_1 \rrbracket \mid \llbracket \overline{c_1}b_2 \rrbracket)) \mid \\
&\qquad\qquad (\boldsymbol{\nu}c_1c_3)((\boldsymbol{\nu}r)(\overline{w}r \mid r \to c_1) \mid c_1(c_2).\, (\llbracket \overline{c_2}c_3 \rrbracket \mid \ldots)) \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,) \\
&\quad \gtrsim (\boldsymbol{\nu}\tilde{x}r)(\, (\boldsymbol{\nu}c_2)(\llbracket \overline{r}c_2 \rrbracket \mid c_2(c_3).\, (\llbracket \overline{c_3}b_1 \rrbracket \mid \llbracket \overline{r}b_2 \rrbracket)) \mid \\
&\qquad\qquad (\boldsymbol{\nu}c_1c_3)(r \to c_1 \mid c_1(c_2).\, (\llbracket \overline{c_2}c_3 \rrbracket \mid c_3(x_1').\, c_1(x_2{}').\, \tilde{x} \to \tilde{x'})) \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,) \\
&\quad \gtrsim (\boldsymbol{\nu}\tilde{x}c_1)(\, (\boldsymbol{\nu}c_2)(\llbracket \overline{c_1}c_2 \rrbracket \mid c_2(c_3).\, (\llbracket \overline{c_3}b_1 \rrbracket \mid \llbracket \overline{c_1}b_2 \rrbracket)) \mid \\
&\qquad\qquad (\boldsymbol{\nu}c_3)(c_1(c_2).\, (\llbracket \overline{c_2}c_3 \rrbracket \mid c_3(x_1').\, c_1(x_2{}').\, \tilde{x} \to \tilde{x'})) \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,) \\
&\quad \equiv (\boldsymbol{\nu}\tilde{x}c_1)(\, (\boldsymbol{\nu}c_2)((\boldsymbol{\nu}r)(\overline{c_1}r \mid r \to c_2) \mid c_2(c_3).\, (\llbracket \overline{c_3}b_1 \rrbracket \mid \llbracket \overline{c_1}b_2 \rrbracket)) \mid \\
&\qquad\qquad (\boldsymbol{\nu}c_3)(c_1(c_2).\, (\llbracket \overline{c_2}c_3 \rrbracket \mid c_3(x_1').\, c_1(x_2{}').\, \tilde{x} \to \tilde{x'})) \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,) \\
&\quad \gtrsim (\boldsymbol{\nu}\tilde{x}c_1r)(\, (\boldsymbol{\nu}c_2)(r \to c_2 \mid c_2(c_3).\, (\llbracket \overline{c_3}b_1 \rrbracket \mid \llbracket \overline{c_1}b_2 \rrbracket)) \mid \\
&\qquad\qquad (\boldsymbol{\nu}c_3)(\llbracket \overline{r}c_3 \rrbracket \mid c_3(x_1').\, c_1(x_2{}').\, \tilde{x} \to \tilde{x'}) \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,) \\
&\quad \gtrsim (\boldsymbol{\nu}\tilde{x}c_1c_2)(\, c_2(c_3).\, (\llbracket \overline{c_3}b_1 \rrbracket \mid \llbracket \overline{c_1}b_2 \rrbracket) \mid \\
&\qquad\qquad (\boldsymbol{\nu}c_3)(\llbracket \overline{c_2}c_3 \rrbracket \mid c_3(x_1').\, c_1(x_2{}').\, \tilde{x} \to \tilde{x'}) \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,) \\
&\quad \equiv (\boldsymbol{\nu}\tilde{x}c_1c_2)(\, c_2(c_3).\, (\llbracket \overline{c_3}b_1 \rrbracket \mid \llbracket \overline{c_1}b_2 \rrbracket) \mid \\
&\qquad\qquad (\boldsymbol{\nu}c_3)((\boldsymbol{\nu}r)(\overline{c_2}r \mid r \to c_3) \mid c_3(x_1').\, c_1(x_2{}').\, \tilde{x} \to \tilde{x'}) \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,) \\
&\quad \gtrsim (\boldsymbol{\nu}\tilde{x}c_1c_3r)(\, r \to c_3 \mid \llbracket \overline{r}b_1 \rrbracket \mid \llbracket \overline{c_1}b_2 \rrbracket \mid \\
&\qquad\qquad c_3(x_1').\, c_1(x_2{}').\, \tilde{x} \to \tilde{x'} \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,) \\
&\quad \gtrsim (\boldsymbol{\nu}\tilde{x}c_1c_3)(\, \llbracket \overline{c_3}b_1 \rrbracket \mid \llbracket \overline{c_1}b_2 \rrbracket \mid \\
&\qquad\qquad c_3(x_1').\, c_1(x_2{}').\, \tilde{x} \to \tilde{x'} \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,) \\
&\quad \equiv (\boldsymbol{\nu}\tilde{x}c_1c_3)(\, (\boldsymbol{\nu}r)(\overline{c_3}r \mid r \to b_1) \mid \llbracket \overline{c_1}b_2 \rrbracket \mid \\
&\qquad\qquad c_3(x_1').\, c_1(x_2{}').\, (x_1 \to x_1' \mid x_2 \to x_2') \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,) \\
&\quad \gtrsim (\boldsymbol{\nu}\tilde{x}c_1r)(\, r \to b_1 \mid \llbracket \overline{c_1}b_2 \rrbracket \mid \\
&\qquad\qquad c_1(x_2{}').\, (x_1 \to r \mid x_2 \to x_2') \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,) \\
&\quad \gtrsim (\boldsymbol{\nu}\tilde{x}c_1)(\, \llbracket \overline{c_1}b_2 \rrbracket \mid \\
&\qquad\qquad c_1(x_2{}').\, (x_1 \to b_1 \mid x_2 \to x_2') \mid \\
&\qquad\qquad \llbracket \llparenthesis P' \rrparenthesis \rrbracket_\Gamma \mid \llbracket \llparenthesis Q' \rrparenthesis \rrbracket_\Gamma\,)
\end{aligned}
$$

$$\begin{aligned}
&\equiv\ (\boldsymbol{\nu}\tilde{x}c_1)(\ (\boldsymbol{\nu}r)(\overline{c_1}r \mid r \to b_2) \mid \\
&\qquad\qquad c_1(x_2{}').\,(x_1 \to b_1 \mid x_2 \to x_2') \mid \\
&\qquad\qquad [\![\langle\!| P' |\!\rangle]\!]_\Gamma \mid [\![\langle\!| Q' |\!\rangle]\!]_\Gamma\ ) \\
&\gtrsim\ (\boldsymbol{\nu}\tilde{x}r)(\ r \to b_2 \mid \\
&\qquad\qquad x_1 \to b_1 \mid x_2 \to r \mid \\
&\qquad\qquad [\![\langle\!| P' |\!\rangle]\!]_\Gamma \mid [\![\langle\!| Q' |\!\rangle]\!]_\Gamma\ ) \\
&\gtrsim\ (\boldsymbol{\nu}\tilde{x})(\ x_1 \to b_1 \mid x_2 \to b_2 \mid \\
&\qquad\qquad [\![\langle\!| P' |\!\rangle]\!]_\Gamma \mid [\![\langle\!| Q' |\!\rangle]\!]_\Gamma\ ) \\
&\equiv\ (\boldsymbol{\nu}\tilde{x})(\tilde{x} \to \tilde{b} \mid [\![\langle\!| P' |\!\rangle]\!]_\Gamma \mid [\![\langle\!| Q' |\!\rangle]\!]_\Gamma) \\
&\gtrsim\ [\![\langle\!| P' \mid Q' |\!\rangle]\!]_\Gamma\{\tilde{b}/\tilde{x}\} \\
&\equiv\ [\![\langle\!| P' \mid Q'\{\tilde{b}/\tilde{x}\} |\!\rangle]\!]_\Gamma\ .
\end{aligned}$$

□

A.3.2 Proof of Lemma 5.3.7

Proof: We show that the relation

$$\mathcal{S} = \{([\![\langle\!| P |\!\rangle]\!]_\Gamma, [\![\langle\!| Q |\!\rangle]\!]_\Gamma) :\ P \approx Q,\ P, Q \in \mathrm{L}\widetilde{\pi}\}$$

is a ground bisimulation up-to context and up-to $\gtrsim$.

1. If $[\![\langle\!| P |\!\rangle]\!]_\Gamma \xrightarrow{\tau} P_1$ then, by Lemma 5.3.2, there exists P' such that $P \xrightarrow{\tau} P'$ and $P_1 \gtrsim [\![\langle\!| P' |\!\rangle]\!]_\Gamma$. Since $P \approx Q$ there exists Q' such that $Q \Longrightarrow Q'$ and $P' \approx Q'$. By Lemmas 5.3.4 there exists Q_1 such that $[\![\langle\!| Q |\!\rangle]\!]_\Gamma \Longrightarrow Q_1$ and $Q_1 \gtrsim [\![\langle\!| Q' |\!\rangle]\!]_\Gamma$. So, we conclude up-to $\gtrsim$.
2. If $[\![\langle\!| P |\!\rangle]\!]_\Gamma \xrightarrow{a(w)} P_1$ then, by Lemma 5.3.2, there exists P' such that, for some $\tilde{x}$, $P \xrightarrow{a(\tilde{x})} P'$ and $P_1 \gtrsim (\boldsymbol{\nu}\tilde{x})([\![\mathrm{INP}\langle w, \tilde{x}\rangle]\!] \mid [\![\langle\!| P' |\!\rangle]\!]_\Gamma)$. Since $P \approx Q$ there exists Q' such that $Q \stackrel{a(\tilde{x})}{\Longrightarrow} Q'$ and $P' \approx Q'$. By Lemmas 5.3.2 and 5.3.4, there exists Q_1 such that $[\![\langle\!| Q |\!\rangle]\!]_\Gamma \stackrel{a(w)}{\Longrightarrow} Q_1$ with $Q_1 \gtrsim (\boldsymbol{\nu}\tilde{x})([\![\mathrm{INP}\langle w, \tilde{x}\rangle]\!] \mid [\![\langle\!| Q' |\!\rangle]\!]_\Gamma)$. So, we conclude up-to context and up-to $\gtrsim$.
3. If $[\![\langle\!| P |\!\rangle]\!]_\Gamma \xrightarrow{\overline{a}(p)} P_1$ then, by Lemma 5.3.2, there exists P' such that either $P \xrightarrow{\overline{a}\langle\tilde{b}\rangle} P'$ and $P_1 \gtrsim (\boldsymbol{\nu}w)(p \to w \mid [\![\mathrm{OUT}\langle w, \tilde{b}\rangle]\!]) \mid [\![\langle\!| P' |\!\rangle]\!]_\Gamma$ or $P \xrightarrow{(\boldsymbol{\nu}\tilde{c})\overline{a}\langle\tilde{b}\rangle} P'$ and

$$P_1 \gtrsim (\boldsymbol{\nu}\tilde{c})((\boldsymbol{\nu}w)(p \to w \mid [\![\mathrm{OUT}\langle w, \tilde{b}\rangle]\!]) \mid [\![\langle\!| P' |\!\rangle]\!]_\Gamma).$$

   (a) In the first case, since $P \approx Q$, there exists Q' such that $Q \stackrel{\overline{a}\langle\tilde{b}\rangle}{\Longrightarrow} Q'$ and $P'\ \mathcal{S}\ Q'$. By Lemmas 5.3.2 and 5.3.4, there exists Q_1 such that $[\![\langle\!| Q |\!\rangle]\!]_\Gamma \stackrel{\overline{a}(p)}{\Longrightarrow} Q_1$ and $Q_1 \gtrsim (\boldsymbol{\nu}w)(p \to w \mid [\![\mathrm{OUT}\langle w, \tilde{b}\rangle]\!]) \mid [\![\langle\!| Q' |\!\rangle]\!]_\Gamma$ and we can conclude up-to context and up-to $\gtrsim$.

(b) In the second case, since $P \approx Q$, there exists Q' s.t. $Q \xRightarrow{(\boldsymbol{\nu}\tilde{c})\overline{a}\langle\tilde{b}\rangle} Q'$ and $P' \mathcal{S} Q'$. By Lemmas 5.3.2 and 5.3.4, there is Q_1 such that $[\![(\!| Q |\!)]\!]_\Gamma \xRightarrow{\overline{a}(p)} Q_1 \gtrsim (\boldsymbol{\nu}\tilde{c})((\boldsymbol{\nu}w)(p \to w \mid [\![\mathrm{OUT}\langle w, \tilde{b}\rangle]\!]) \mid [\![(\!| Q' |\!)]\!]_\Gamma)$. So, we conclude up-to context and up-to $\gtrsim$.

□

A.4 Proofs of Chapter 7

A.4.1 Proof of Lemma 7.1.10

Proof: We show that the relation

$$\mathcal{S} = \{\big(Q\{p/q\}\,,\,(\boldsymbol{\nu}q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p)\big) \;:\; q \text{ in } Q \text{ only in output position}\}$$

is a barbed bisimulation up to structural equivalence.

- Let $Q\{p/q\} \xrightarrow{\tau} Q'\{p/q\}$. There are two cases.
  1. $Q \xrightarrow{\tau} Q'$. This case can be easily treated.
  2. Otherwise, since p and q are channels and they never appear in testing, this means that the τ-action is due to a communication along p. More precisely, Q must contain an occurrence of q in output subject position and an occurrence of p in input position which give rise to the communication. Up to structural equivalence, this implies that

$$(\boldsymbol{\nu}q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p) \xrightarrow{\tau}\xrightarrow{\tau} \equiv (\boldsymbol{\nu}q{:}\mathbf{C}(T))\,(Q' \mid q \triangleright p).$$

     As desired.
- Let $(\boldsymbol{\nu}q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p) \xrightarrow{\tau} R$ for some R. There are two cases.
  1. $R = (\boldsymbol{\nu}q{:}\mathbf{C}(T))\,(Q' \mid q \triangleright p)$ since $Q \xrightarrow{\tau} Q'$. This case can be easily treated.
  2. The τ-action is due to some communication along q between Q and the link $q \triangleright p$. More precisely,

$$(\boldsymbol{\nu}q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p) \equiv (\boldsymbol{\nu}q{:}\mathbf{C}(T))\,((\boldsymbol{\nu}\tilde{z})\,(Q' \mid \overline{q}v) \mid q \triangleright p)$$

     and

$$(\boldsymbol{\nu}q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p) \xrightarrow{\tau} \equiv (\boldsymbol{\nu}q{:}\mathbf{C}(T))\,((\boldsymbol{\nu}\tilde{z})\,(Q' \mid \overline{p}v) \mid q \triangleright p).$$

     The left side can easily mimic the move as follows:

$$Q'\{p/q\} \Longrightarrow \equiv (\boldsymbol{\nu}\tilde{z})\,(Q' \mid \overline{q}v)\{p/q\} = (\boldsymbol{\nu}\tilde{z})\,(Q' \mid \overline{p}v)\{p/q\}.$$

     As desired.

□

A.4.2 Proof of Lemma 7.9.2

Proof: As the base case, we consider Z, where the object manager at s has just been created; all previous steps in the sequence are obviously irrelevant, because the condition of containing $\mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t}\,\rangle$ is not fulfilled. Then

$$Z = C'[\ \mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t}\,\rangle\ ] = C'[\ (\boldsymbol{\nu}\widetilde{n}k_{\mathrm{i}})\,(\ \overline{m_{\mathrm{e}}}\ \mid\ \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_{\mathrm{i}}, \widetilde{t}\,\rangle\ )\ ]$$

Using structural equivalence, we immediately get

$$Z \equiv E[\ (\boldsymbol{\nu}\widetilde{n})\,(\ \overline{m_{\mathrm{e}}}\ \mid\ \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_{\mathrm{i}}, \widetilde{t}\,\rangle\ \mid\ \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \emptyset\,\rangle\ )\ ]$$

for some static context $E[\cdot]$, such that Z corresponds to "state" F. It is important to notice that names in $\widetilde{n}$ will only appear inside the object manager and the pre-processed requests.

"State" F can only evolve into some "state" A; it does so by grabbing the external mutex m_{e} for one of its pre-processed requests in $\widetilde{v}$. The only other possible reduction involving "state" F is pre-processing another request, but such an action does not change the "state"—it only adds to the set of pre-processed requests $\widetilde{v}$. A similar reasoning applies to the other "states", so we simply skip pre-processing.

Thus, by consuming the pre-processed request $\overline{s}\langle l, k\rangle$ and leaving untouched the other pre-processed requests $\widetilde{v}$, we may arrive at some Z of the form:

$$\begin{array}{ll} & E[\ (\boldsymbol{\nu}\widetilde{n})\,(\ \overline{m_{\mathrm{e}}}\ \mid\ \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_{\mathrm{i}}, \widetilde{t}\,\rangle\ \mid\ \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle\ )\ ] \\ \rightarrow_{\equiv} & E[\ (\boldsymbol{\nu}\widetilde{n})\,(\ \underline{\overline{m_{\mathrm{i}}}k \mid \overline{s}\langle l, k_{\mathrm{e}}\rangle}\ \mid\ \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_{\mathrm{i}}, \widetilde{t}\,\rangle\ \mid\ \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v} - \langle\, l, k\,\rangle\,\rangle\ )\ ] \\ \stackrel{\mathrm{def}}{=} & Z \end{array}$$

where Z corresponds to "state" A.

"State" A can only evolve into either "state" D or S, by consuming the request $\overline{s}\langle l, k_{\mathrm{e}}\rangle$:

- "State" A evolves into "state" D if l is one of $\mathsf{ali}\_\langle x, p\rangle$, cln_p, or $\mathsf{upd}_j\_\langle t, p\rangle$, which are disallowed as external request, the object manager is restarted and, up to structural equivalence, we get state D.
- In the remaining cases, that is, when l is one of $\mathsf{inv}_j\_\langle x, p\rangle$, sur_p, or $\mathsf{sur}\_\langle p\rangle$, 'state" A evolves into "state" S. Indeed, a call-manager is started concurrently with the restarted object manager. By using structural equivalence, we can move components that are not in the scope of $\widetilde{n}$ outside this scope, so as to recognize state S.

In "state" S, a png request drives the system into "state" T. In the case of method invocation a reduction along t_j may occur which allows the evaluation of the method body. At this point a number of self-inflicted requests may be served (external requests are blocked because the external mutex m_{e} is no available). This part of the computation will not change the state. Notice that, by hypothesis, since we suppose that Z contain an object manager and non an alias manager, we exclude self-inflicted aliasing operations. When the last self-inflicted request is served, a reply $\overline{r^*}\langle\, o, k\,\rangle$ will appear unguarded.

The confluent reduction along r^* will drive the computation to "state" T. sur requests are treated similarly.
"State" T can only evolve, by reducing along m_{i}, to "state" F.
$\square$

A.4.3 Proof of Lemma 7.11.2

We show that there is a sequence of τ-actions such that:

$$\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle \Rightarrow_{\equiv}$$
$$(\boldsymbol{\nu} s^*)\ \big((\boldsymbol{\nu} k_{\mathrm{i}})\ \mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \tilde{v}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \,\big|\, \overline{r}\langle s^*, k\rangle\big).$$

We prove that $\approx_{\Gamma;s}$ is insensitive to these particular τ-actions. To this end, we supply the two lemmas A.4.1 and A.4.2. We recall that $\mathsf{CM}[\cdot]$ denote the call manager protocol as defined in Table 7.6.

Lemma A.4.1 *Let* $\widetilde{n} := m_{\mathrm{e}}, m_{\mathrm{i}}, k_{\mathrm{e}}$, *and* $\widetilde{v} := v_1 \mathinner{..} v_n$ *with* $v_j := \langle\, l_j, k_j\,\rangle$ *for* $1 \leq j \leq n$, *and*

$$\begin{array}{rcl}
C_1 &:=& \mathsf{CM}[\,(\boldsymbol{\nu} q)\ (\overline{s}\langle \mathsf{cln_}q, k^*\rangle \mid q(x, k').\overline{s}\langle \mathsf{ali_}\langle x, r^*\rangle, k'\rangle)\,] \\
C_2 &:=& \mathsf{CM}[\,(\boldsymbol{\nu} q)\ (\overline{q}\langle s^*, k^*\rangle \mid q(x, k').\overline{s}\langle \mathsf{ali_}\langle x, r^*\rangle, k'\rangle)\,] \\
P\langle\widetilde{v}\rangle &:=& (\boldsymbol{\nu}\widetilde{n}k^*)\ \big(\overline{m_{\mathrm{i}}}k \big|\ \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_1\,\big) \\
&& \text{with } k^* \notin \mathrm{fn}(\widetilde{v}) \\
Q\langle\widetilde{v}\rangle &:=& (\boldsymbol{\nu}\widetilde{n}k^*s^*)\ \big(\overline{m_{\mathrm{i}}}k \big|\ \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_2\,\big) \\
&& \text{with } k^* \notin \mathrm{fn}(\widetilde{v}) \\
\Gamma &\vdash& P\langle\widetilde{v}\rangle, Q\langle\widetilde{v}\rangle \ \text{ for some } \Gamma.
\end{array}$$

Then, $P\langle\widetilde{v}\rangle \approx_{\Gamma;s} Q\langle\widetilde{v}\rangle$.

Proof: For simplictiy, we omit the obligations on types in the coinductive definition of $\approx_{\Gamma;s}$. So, we prove that the relation:

$$\mathcal{S} = \{(P\langle\widetilde{w}\rangle, Q\langle\widetilde{w}\rangle) : \widetilde{w} = w_1 \mathinner{..} w_m \text{ with } w_j := \langle\, l_j, k_j\,\rangle,\ 1 \leq j \leq n\} \cup \mathcal{I}$$

where $\mathcal{I}$ is the identity relation, is a $\approx_{\Gamma;s}$-bisimulation up to $\equiv$.
The only channel which appear free in subject position in $P\langle\widetilde{w}\rangle$ and $Q\langle\widetilde{w}\rangle$ is s. Since both the external key k_{e} and the internal key k^* are restricted in $P\langle\widetilde{w}\rangle$ and $Q\langle\widetilde{w}\rangle$, an by well-typedness, the environment can send requests only of the form $\overline{s}\langle l, k\rangle$ with $k_{\mathrm{e}} \neq k \neq k^*$. The process $P\langle\widetilde{w}\rangle$ can perform only two kinds of actions. Either (i) an input action $s\langle l, k\rangle$ (with $k_{\mathrm{e}} \neq k \neq k^*$), or (ii) a silent move along s involving the self-inflicted cloning request contained in C_1. In case (i), the pre-processing of the request creates the process $m_{\mathrm{e}}.(\overline{s}\langle l, k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k)$ which can be added in $\mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{w}\,\rangle$ obtaining some $\mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{w'}\,\rangle$ with $\widetilde{w'} = \widetilde{w} \cup \langle l, k\rangle$. The process $Q\langle\widetilde{w}\rangle$ can perform the same action and the derivatives are again related by $\mathcal{S}$. In case (ii), the process $Q\langle\widetilde{w}\rangle$ can mimic the τ-action by not

performing any reduction at all. Up to structural equivalence, we get into the identity relation.
The process $Q\langle\widetilde{w}\rangle$ can only perform two kinds of actions. Either (i) a input action $s\langle l,k\rangle$ (with $k_{\rm e} \neq k \neq k^*$), and we reason as above, or (ii) a silent move along the restricted channel q in C_2. In this case $P\langle\widetilde{w}\rangle$ can perform two silent actions, along s and q, getting, up to structural equivalence, into the identity relation.
□

Lemma A.4.2 *Let $\widetilde{n} := m_{\rm e}, m_{\rm i}, k_{\rm e}$, and $\widetilde{v} := v_1 \ldots v_n$ with $v_j := \langle\, l_j, k_j \,\rangle$ for $1{\leq}j{\leq}n$, and*

$$\begin{aligned}
C_3 &:= \mathsf{CM}[\,\overline{s}\langle \mathsf{ali_}\langle s^*, r^*\rangle, k^*\rangle\,] \\
C_4 &:= \mathsf{CM}[\,\overline{r^*}\langle s^*, k^*\rangle\,] \\
P\langle\widetilde{v}\rangle &:= (\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\rm i}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_3\,\big) \\
&\quad\ \text{with } k^* \notin \mathrm{fn}(\widetilde{v}) \\
Q\langle\widetilde{v}\rangle &:= (\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\rm i}}k \,\big|\, \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, s^*\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_4\,\big) \\
&\quad\ \text{with } k^* \notin \mathrm{fn}(\widetilde{v}). \\
\Gamma &\vdash P\langle\widetilde{v}\rangle, Q\langle\widetilde{v}\rangle \ \text{ for some } \Gamma.
\end{aligned}$$

Then, $P\langle\widetilde{v}\rangle \approx_{\Gamma;s} Q\langle\widetilde{v}\rangle$.

Proof: Similar to that of Lemma A.4.1. □

Proof of Lemma 7.11.2

Proof: As said above there is a sequence of τ-actions, such that:

$$\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle \Rightarrow_{\equiv} (\boldsymbol{\nu}s^*)\ \big((\boldsymbol{\nu}k_{\rm i})\,\mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\rm i}, s^*, \tilde{v}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \,\big|\, \overline{r}\langle s^*, k\rangle\big).$$

The above sequence consists of 7 silent steps. These τ-steps are of two kinds: (i) confluent reductions along restricted channels of the form

$$C[(\boldsymbol{\nu}q)\,(\overline{q}\langle\widetilde{v}\rangle \mid q(\widetilde{x}).P)] \xrightarrow{\tau}_{\equiv} C[P\{^{\widetilde{v}}/_{\widetilde{x}}\}]$$

where $q \notin \mathrm{fn}(P)$, let us call these reductions of kind α; (ii) reductions involving self-inflicted requests (induced by the surrogation) of the form

$$C[(\boldsymbol{\nu}k^*)\,(\mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\rm e}, k^*, \tilde{t}\,\rangle \mid \overline{s}\langle \mathsf{op\_}r^*, k^*\rangle)] \xrightarrow{\tau} \ldots$$

let us call these reductions of kind β. It is well-known that $\approx_\Gamma$ (as well as $\approx_{\Gamma;s}$) is insensitive to reductions of kind α. In Lemma A.4.1 and A.4.2 we show that $\approx_{\Gamma;s}$ is insensitive to the reductions of kind β appearing in the sequence mentioned above. This is possible because, in the implementation of object and alias managers, we use nonces (c.f. page 112) in order to guarantee that the self-inflicted key of the object manager is always restricted. In this manner, the environment cannot produce any "malicious"

self-inflicted request which might potentially interfere with the cloning and the aliasing requests.
The first and the second reductions are of kind α and they are due to the process $\mathsf{CM}[\,[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle\,]\!]^{k^*}_{r^*}\,]$ (contained in $\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle$) which, after two τ-steps, reduces to the process

$$\mathsf{CM}[\,(\boldsymbol{\nu} q)\,(\overline{s}\langle \mathsf{cln}\_q, k^*\rangle \mid q(x,i).\overline{s}\langle \mathsf{ali}\_\langle x, r^*\rangle, i\rangle)\,].$$

We abbreviate this process by C_1. The situation is that:

$$\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle \xrightarrow{\tau}\xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{n}k^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_1\,\big)$$

where $k^* \notin \mathrm{fn}(\widetilde{v})$.
The third reduction is of kind β and involves the self-inflicted cloning request in C_1. Let C_2 be the process $\mathsf{CM}[\,(\boldsymbol{\nu} q)\,(\overline{q}\langle s^*, k^*\rangle \mid q(x,i).\overline{s}\langle \mathsf{ali}\_\langle x, r^*\rangle, i\rangle)\,]$, then the process

$$(\boldsymbol{\nu}\widetilde{n}k^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_1\,\big)$$

reduces, up to structural equivalence, to

$$(\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_2\,\big)$$

where $k^* \notin \mathrm{fn}(\widetilde{v})$. By Lemma A.4.1 the relation $\approx_{\Gamma;s}$ is insensitive to this reduction.
The fourth reduction is of kind α and it is due to C_2. So, if we denote with C_3 the process $\mathsf{CM}[\,\overline{s}\langle \mathsf{ali}\_s^*, r^*, k^*\rangle\,]$ the situation is that that $\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle$ evolves in four silent steps, up to structural equivalence, to

$$(\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_3\,\big)$$

where $k^* \notin \mathrm{fn}(\widetilde{v})$.
In the fifth τ-step we reduce the self-inflicted aliasing request contained in C_3. So, let us denote with C_4 the process $\mathsf{CM}[\,\overline{r^*}\langle s^*, k^*\rangle\,]$. It holds that the process

$$(\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_3\,\big)$$

reduces, up to structural equivalence, to

$$(\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, s^*\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_4\,\big)$$

where $k^* \notin \mathrm{fn}(\widetilde{v})$. By Lemma A.4.2 the relation $\approx_{\Gamma;s}$ is insensitive to this reduction.
The sixth and the seventh reductions are of kind α and involve channels r^* and m_{i}, respectively. Up to structural equivalence we get the desired process

$$(\boldsymbol{\nu} s^*)\,\big((\boldsymbol{\nu} k_{\mathrm{i}})\,\mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \tilde{v}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \,\big|\, \overline{r}\langle s^*, k\rangle\big).$$

□

A.4.4 Proof of Lemma 7.11.3

Lemma 7.11.3 proves that the aliased object manager appearing in Lemma 7.11.2 behaves as a forwarder.

As a first step we recall a well-known property of replicated input.

Lemma A.4.3 *Let $C[\cdot]$ be a π-calculus context where channel c does not appear either in input or in output object position. Then*

$$(\boldsymbol{\nu} c)\ \big(\ !\ c(x).P\ \big|\ C[\overline{c}v]\ \big) \approx_\Gamma (\boldsymbol{\nu} c)\ \big(\ !\ c(x).P\ \big|\ C[P\{{}^{v}\!/\!_{x}\}]\ \big)$$

Proof: By applying Milner's replications theorems [51]. □

Proof of Lemma 7.11.3

Proof: The obligations on types guarantee that values received along channel s are of the right type. This allows us to use polyadic input along s. By observing process $(\boldsymbol{\nu} k_{\mathrm{i}})\,\mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, s^*\,\rangle$ we note that, since k_{i} is restricted and never extruded, the aliased object manager will never receive self-inflicted requests. By exhibiting the appropriate bisimulation, we can prove that such a process has the following functional behaviour.

$$\begin{array}{l}(\boldsymbol{\nu} k_{\mathrm{i}})\,(\mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, s^*\,\rangle) \approx_\Gamma\ !\,s(l,k).\mathsf{if}\ [k{=}k_{\mathrm{e}}]\ \mathsf{then}\ m_{\mathrm{i}}(k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}})\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ m_{\mathrm{e}}.(\overline{s}\langle l,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k)\end{array}$$

Since $\approx_\Gamma$ is preserved by parallel composition and restriction, we have that:

$$(\boldsymbol{\nu} k_{\mathrm{i}})\,(\mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \widetilde{v}\,\rangle)$$

$$\approx_\Gamma$$

$$\begin{array}{ll}(\boldsymbol{\nu} \widetilde{m}k_{\mathrm{e}})\,\big(\overline{m_{\mathrm{e}}} \mid & !\,s(l,k).\mathsf{if}\ [k{=}k_{\mathrm{e}}]\ \mathsf{then}\ m_{\mathrm{i}}(k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}})\\ & \qquad\qquad\qquad\quad \mathsf{else}\ \ m_{\mathrm{e}}.(\overline{s}\langle l,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k)\\ \qquad\qquad\ \mid & \prod_{1\le j\le n} m_{\mathrm{e}}.\big(\overline{s}\langle l_j,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k_j\big)\,\big)\end{array}$$

If we assume that the environment cannot use s in input, then all requests on s are captured by the unique replicated input on s. Moreover, the external identity k_{e} is restricted and never extruded to the environment, and therefore only pre-processed requests "knows" k_{e}. Using these informations, up to harmless confluent reductions along m_{i}, we can safely internalise the management of pre-processed requests by introducing a restricted channel s_{e} with the same type as s and s^*. In this manner we can drop the matching on the identity, and the replicated input on s will only take care of serving external requests. Formally, we have the following.

$$\begin{array}{ll}(\boldsymbol{\nu} \widetilde{m}k_{\mathrm{e}})\,\big(\overline{m_{\mathrm{e}}} \mid & !\,s(l,k).\mathsf{if}\ [k{=}k_{\mathrm{e}}]\ \mathsf{then}\ m_{\mathrm{i}}(k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}})\\ & \qquad\qquad\qquad\quad \mathsf{else}\ \ m_{\mathrm{e}}.(\overline{s}\langle l,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k)\\ \qquad\qquad\ \mid & \prod_{1\le j\le n} m_{\mathrm{e}}.\big(\overline{s}\langle l_j,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k_j\big)\,\big)\end{array}$$

$\approx_{\Gamma;s}$ (by exhibiting the appropriate bisimulation)

$$
\begin{aligned}
(\boldsymbol{\nu}\widetilde{m}s_{\mathrm{e}})\,\big(\overline{m_{\mathrm{e}}} \mid\ & \mathbf{!}\, s(l,k).m_{\mathrm{e}}.(\overline{s_{\mathrm{e}}}\langle l,k\rangle \mid \overline{m_{\mathrm{i}}}k) \\
\mid\ & \mathbf{!}\, s_{\mathrm{e}}(l,k).m_{\mathrm{i}}(k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}) \\
\mid\ & \prod_{1\le j\le n} m_{\mathrm{e}}.(\overline{s_{\mathrm{e}}}\langle l_j,k_j\rangle \mid \overline{m_{\mathrm{i}}}k_j)\,\big)
\end{aligned}
$$

$\approx_{\Gamma}$ (reductions on m_{i} are confluent)

$$
\begin{aligned}
(\boldsymbol{\nu}m_{\mathrm{e}}s_{\mathrm{e}})\,\big(\overline{m_{\mathrm{e}}} \mid\ & \mathbf{!}\, s(l,k).m_{\mathrm{e}}.\overline{s_{\mathrm{e}}}\langle l,k\rangle \\
\mid\ & \mathbf{!}\, s_{\mathrm{e}}(l,k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}) \\
\mid\ & \prod_{1\le j\le n} m_{\mathrm{e}}.\overline{s_{\mathrm{e}}}\langle l_j,k_j\rangle\,\big)
\end{aligned}
$$

$\approx_{\Gamma}$ (by Lemma A.4.3)

$$
\begin{aligned}
(\boldsymbol{\nu}m_{\mathrm{e}}s_{\mathrm{e}})\,\big(\overline{m_{\mathrm{e}}} \mid\ & \mathbf{!}\, s(l,k).m_{\mathrm{e}}.\big(\,\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}\,\big) \\
\mid\ & \mathbf{!}\, s_{\mathrm{e}}(l,k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}) \\
\mid\ & \prod_{1\le j\le n} m_{\mathrm{e}}.(\overline{s^*}\langle l_j,k_j\rangle \mid \overline{m_{\mathrm{e}}})\,\big)
\end{aligned}
$$

$\approx_{\Gamma}$ (by garbage collection on s_{e})

$$
\begin{aligned}
(\boldsymbol{\nu}m_{\mathrm{e}})\,\big(\overline{m_{\mathrm{e}}} \mid\ & \mathbf{!}\, s(l,k).m_{\mathrm{e}}.\big(\,\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}\,\big) \\
\mid\ & \prod_{1\le j\le n} m_{\mathrm{e}}.\big(\,\overline{s^*}\langle l_j,k_j\rangle \mid \overline{m_{\mathrm{e}}}\,\big)\,\big)
\end{aligned}
$$

$\approx_{\Gamma}$ (reductions on m_{e} are confluent)

$$
\mathbf{!}\, s(l,k).\overline{s^*}\langle l,k\rangle \;\Big|\; \prod_{1\le j\le n} \overline{s^*}\langle l_j,k_j\rangle
$$

$\stackrel{\mathrm{def}}{=}$ (by definition)

$$
s \triangleright s^* \;\Big|\; \prod_{1\le j\le n} \overline{s^*}v_j
$$

$\square$

A.4.5 Proof of Lemma 7.11.4

This is a rather technical lemma. It is the only place where the theory of Lπ is exploited.

Proof: We apply Lemma 7.1.11 to process P to remove all the occurrences of s^* in output object position. Let's call $\widehat{P}$ the process obtained by applying Lemma 7.1.11 in such a way. Note that we focus only on channel s^*. The other channels are not affected by our transformation. Since Lemma 7.1.11 works with respect to (typed) barbed congruence, it holds that $P \cong_\Gamma \widehat{P}$. This implies

$$
(\boldsymbol{\nu}s^*)\,\big(\,s \triangleright s^* \mid P\,\big) \cong_\Gamma (\boldsymbol{\nu}s^*)\,\big(\,s \triangleright s^* \mid \widehat{P}\,\big) \text{ and } P\{{}^{s}\!/_{s^*}\} \cong_\Gamma \widehat{P}\{{}^{s}\!/_{s^*}\}.
$$

So, we are left with proving that $(\boldsymbol{\nu} s^*)\left(s \triangleright s^* \mid \widehat{P}\right) \simeq_{\Gamma;s} \widehat{P}\{^s/_{s^*}\}$. The proof follows by showing that the relation:

$$\{(\,(\boldsymbol{\nu} s^*)\,(s \triangleright s^* \mid \widehat{P})\,,\,\widehat{P}\{^s/_{s^*}\}\,) : s \notin \mathrm{fn}(\widehat{P}) \text{ and } s^* \text{ not free in obj. pos. in } \widehat{P}\}$$

is a $\approx_{\Gamma;s}$ bisimilarity. The obligations on types guarantee that values received along channel s are of the right type. A part this, we can safely omit the types in the coinductive definition of $\approx_{\Gamma;s}$. We recall that $\approx_{\Gamma;s}$ is ground on channels. This means that we alway suppose to receive fresh channels, in particular, we never receive channels s and s^*.
As regards the left side, the only interesting transition is the input action along s. This action can be emulated by the right side by exploiting the asynchronous clause for input. As regards the right side, we recall that $\approx_{\Gamma;s}$ is not sensitive to output actions along s. Since s^* does not appear free in output object position in $\widehat{P}$, the only interesting action of $\widehat{P}\{^s/_{s^*}\}$ is the input action along s which can be mimicked by the left side up to a τ-action. □

A.4.6 Proof of Lemma 7.11.5

We first prove a more general result asserting that pre-processing of external requests is harmless.

Lemma A.4.4 *Let $\widetilde{v} := v_1 \mathinner{..} v_n$ where $v_j := \langle\, l_j, k_j \,\rangle$ with $k_{\mathrm{e}} \neq k_j \neq k_{\mathrm{i}}$ for $1 \leq j \leq n$. It holds that:*

$$\mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \tilde{t} \,\rangle \;\Big|\; \prod_{1\leq j\leq n} \overline{s} v_j \;\approx_{\Gamma;s}\; \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \tilde{t} \,\rangle \;\Big|\; \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v} \,\rangle.$$

Proof: We prove the result by induction on the number of elements of $\widetilde{v}$.

Case $n = 0$**.** Trivial.

Inductive case. Let

$$\prod_{1\leq j\leq n} \overline{s} v_j \;\stackrel{\mathrm{def}}{=}\; \overline{s} v_1 \;\Big|\; \prod_{2\leq j\leq n} \overline{s} v_j \text{ and}$$
$$\mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v} \,\rangle \;\stackrel{\mathrm{def}}{=}\; m_{\mathrm{e}}.\big(\,\overline{s}\langle l_1, k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}} k_1\,\big) \;\mid\; \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, v_2 \mathinner{..} v_n \,\rangle.$$

By inductive hypothesis it holds that:

$$\mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, \tilde{t} \,\rangle \;\Big|\; \prod_{2\leq j\leq n} \overline{s} v_j \;\approx_{\Gamma;s}\; \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, \tilde{t} \,\rangle \;\Big|\; \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, v_2 \mathinner{..} v_n \,\rangle.$$

Since $\approx_{\Gamma;s}$ is preserved by parallel composition, for proving our result it suffices to show that:

$$\mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, \tilde{t} \,\rangle \;\Big|\; \overline{s} v_1 \;\approx_{\Gamma;s}\; \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, \tilde{t} \,\rangle \;\Big|\; m_{\mathrm{e}}.\big(\,\overline{s}\langle l_1, k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}} k_1\,\big).$$

Let $A \stackrel{\text{def}}{=} \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, \tilde{t}\,\rangle \mid \overline{s}v_1$ and
$B \stackrel{\text{def}}{=} \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, \tilde{t}\,\rangle \mid m_{\mathrm{e}}.\big(\overline{s}\langle l_1, k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k_1\big)$

we prove that the relation:

$$\mathcal{S} = \{\big((\boldsymbol{\nu}\widetilde{z})\,(A \mid R), (\boldsymbol{\nu}\widetilde{z})\,(B \mid R)\big) \;:\; s \notin \widetilde{z} \text{ and } s \text{ not in input in } R\} \cup \mathcal{I}$$

where $\mathcal{I}$ is the identity relation, is a $\approx_{\Gamma;s}$-bisimulation up to structural equivalence. The obligation on types in the coinductive definition of $\approx_{\Gamma;s}$ can be safely omitted. We first show how the right side can emulate the actions performed by the left side and then the vice versa.

From left to right. Let us see the possible actions of $(\boldsymbol{\nu}\widetilde{z})\,(A \mid R)$.

1. If $(\boldsymbol{\nu}\widetilde{z})\,(A \mid R) \xrightarrow{\mu} (\boldsymbol{\nu}\widetilde{y})\,(A \mid R')$ then it is easy.
2. If $(\boldsymbol{\nu}\widetilde{z})\,(A \mid R) \xrightarrow{s\langle l,k\rangle} (\boldsymbol{\nu}\widetilde{z})\,(A' \mid R)$, then there are three possibilities: (i) either $k = k_{\mathrm{i}}$, or (ii) $k = k_{\mathrm{e}}$, or (iii) $k_{\mathrm{i}} \neq k \neq k_{\mathrm{e}}$. In each case the right side can perform an input $s\langle l, k\rangle$ obtaining a process $(\boldsymbol{\nu}\widetilde{z})\,(B' \mid R)$. By inspection of the encoding we have that $(\boldsymbol{\nu}\widetilde{z})\,(A' \mid R) \equiv (\boldsymbol{\nu}\widetilde{y})\,(A'' \mid R')$ and $(\boldsymbol{\nu}\widetilde{z})\,(B' \mid R) \equiv (\boldsymbol{\nu}\widetilde{y})\,(B'' \mid R')$, for some $\widetilde{y}$ and some process R', where A'' (resp. B'') is the same as A (resp. A''), up to renaming k_{i} with a fresh key k^*. Therefore $(\boldsymbol{\nu}\widetilde{y})\,(A'' \mid R')\ \mathcal{S}\ (\boldsymbol{\nu}\widetilde{y})\,(B'' \mid R')$.
3. If $(\boldsymbol{\nu}\widetilde{z})\,(A \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{y})\,(A' \mid R')$, where the τ-action is due to a communication along s between A and R (recall that s can only appear in output in R), then we reason similarly to the previous case.
4. If $(\boldsymbol{\nu}\widetilde{z})\,(A \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{z})\,(A' \mid R)$, where the τ actions is due to a communication along s between the object manager and the external request $\overline{s}v_1$, then, by inspection of the encoding, it holds that $A' \equiv B$. On the right side we can mimic the τ action by performing $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R) \Longrightarrow (\boldsymbol{\nu}\widetilde{z})\,(B \mid R)$. It holds that $(\boldsymbol{\nu}\widetilde{z})\,(A' \mid R) \equiv \mathcal{S}$ $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R)$.

From right to left. Let us see the possible actions of $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R)$.

1. If $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R) \xrightarrow{\mu} (\boldsymbol{\nu}\widetilde{y})\,(B \mid R')$ then it is easy.
2. If $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R) \xrightarrow{s\langle l,k\rangle} (\boldsymbol{\nu}\widetilde{z})\,(B' \mid R)$, then there are three possibilities: (i) either $k = k_{\mathrm{i}}$, or (ii) $k = k_{\mathrm{e}}$, or (iii) $k_{\mathrm{i}} \neq k \neq k_{\mathrm{e}}$. In each case the left side can perform an input $s\langle l, k\rangle$ obtaining a process $(\boldsymbol{\nu}\widetilde{z})\,(A' \mid R)$. By inspection of the encoding we have that $(\boldsymbol{\nu}\widetilde{z})\,(B' \mid R) \equiv (\boldsymbol{\nu}\widetilde{y})\,(B'' \mid R')$ and $(\boldsymbol{\nu}\widetilde{z})\,(A' \mid R) \equiv (\boldsymbol{\nu}\widetilde{y})\,(A'' \mid R')$, for some $\widetilde{y}$ and some process R', where B'' (resp. A'') is the same as B (resp. A), up to renaming k_{i} with a fresh key k^*. Therefore $(\boldsymbol{\nu}\widetilde{y})\,(B'' \mid R')\ \mathcal{S}\ (\boldsymbol{\nu}\widetilde{y})\,(A'' \mid R')$.
3. If $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{y})\,(B' \mid R')$, where the τ-action is due to a communication along s between B and R (recall that s can only appear in output in R), then we reason similarly to the previous case.

4. If $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R) \xrightarrow{m_{\mathrm{e}}} (\boldsymbol{\nu}\widetilde{z})\,(B' \mid R)$ and $B' = \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, \tilde{t}\,\rangle \mid \overline{s}\langle l_1, k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k_1$, then the left side can mimic this action by serving the request $\overline{s}v_1$ and then grabbing the mutex. In practise,
$(\boldsymbol{\nu}\widetilde{z})\,(A \mid R) \xrightarrow{\tau} \xrightarrow{m_{\mathrm{e}}} (\boldsymbol{\nu}\widetilde{z})\,(A' \mid R)$ with $A' \equiv B'$. So, $(\boldsymbol{\nu}\widetilde{z})\,(B' \mid R)\ \mathcal{S}\ (\boldsymbol{\nu}\widetilde{z})\,(A' \mid R)$.

□

Proof of Lemma 7.11.5

Proof: It follows directly from Lemma A.4.4 and the fact that $\approx_{\Gamma;s}$ is preserved by parallel composition and restriction. □

A.4.7 Proof of Lemma 7.11.6

Proof: It holds that:

$$\mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle \xrightarrow{\tau}{}^{2}_{\equiv} (\boldsymbol{\nu}\widetilde{m}\widetilde{k})\,\big(\overline{m_{\mathrm{e}}} \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, \widetilde{t}\,\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle\big) \mid \overline{r}\langle s, k\rangle.$$

Since $\approx_{\Gamma}$ is insensitive to these two silent moves, we can conclude. □

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer Verlag, 1996.

[2] S. Abramsky. Proofs as Processes. *Theoretical Computer Science*, 135(1):5–9, December 1994.

[3] R. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proc. Coordination'97*, volume 1282 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[4] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195:291–324, 1998. Extended abstract in *Proc. CONCUR '96*, LNCS 1119, Springer Verlag.

[5] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[6] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. *Acta Informatica*, 29:737–760, 1992.

[7] G. Bellin and P. Scott. On the π-calculus and Linear Logic. *Theoretical Computer Science*, 135(1):11–65, December 1994.

[8] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60:109–137, 1984.

[9] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96, 1992.

[10] M. Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195:205–226, 1998.

[11] M. Boreale, C. Fournet, and C. Laneve. Bisimulations for the Join Calculus. In *Proc. IFIP Conference PROCOMET'98*, 1997.

[12] M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *13th LICS Conf.* IEEE Computer Society Press, 1998.

[13] G. Boudol. Asynchrony and the π-calculus. Technical Report RR-1702, INRIA-Sophia Antipolis, 1992.

[14] G. Boudol. Some Chemical Abstract Machines. In *Proc. Rex School/Symposium 1993 "A Decade of Concurrency — Reflexions and Perspectives"*, volume 803 of *Lecture Notes in Computer Science*, pages 92–123. Springer Verlag, 1994.

[15] G. Boudol. The pi-calculus in direct style. In *Proc. 24th POPL*. ACM Press, 1997.

[16] N. Busi, R. Gorrieri, and G. Zavattaro. A process algebraic view of linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, 1988.

[17] Luca Cardelli. `obliq-std.exe` — binaries for windows nt. `http://www.luca.demon.co.uk/Obliq/Obliq.html`, 1994.

[18] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995. Short version in *Proc. of POPL '95*.

[19] Ilaria Castellani and Matthew Hennessy. Testing theories for asynchronous languages. In *Proc. FST-TCS'98*, volume 1530 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[20] Silvano Dal-Zilio. Implicit polymorphic type system for the blue calculus. Technical Report RR-3244, Inria, Institut National de Recherche en Informatique et en Automatique, 1997.

[21] R. De Nicola and R. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[22] Paolo Di Blasio and Katheleen Fisher. A concurrent object calculus. In *Proc. CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*, 1996. An extended version appeared as Stanford University Technical Note STAN-CS-TN-96-36, 1996.

[23] U. Engberg and M. Nielsen. A calculus of communicating systems with label-passing. Report DAIMI PB-208, Computer Science Department, University of Aarhus, Denmark, 1986.

[24] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join calculus. In *Proc. 23th POPL*. ACM Press, 1996.

[25] C. Fournet and G. Gonthier. A Hierarchy of Equivalences for Asynchronous Calculi. In *Proc. 25th ICALP*, volume 1443 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[26] Cedric Fournet. *The Join calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, École Polytechnique, 1999.

[27] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implict typing à la ML for the join-calculus. In *Proc. CONCUR 97*, volume 1243 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[28] Y. Fu. A proof theoretical approach to communication. In *24th ICALP*, volume 1256 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[29] Andrew Gordon and Paul Hankin. A concurrent object calculus. In *Proceedings of SOAP'98*, 1998.

[30] M. Hennessy and J. Riely. A typed language for distributed mobile processes. In *Proc. 25th POPL*. ACM Press, 1998.

[31] D. Hirschkoff. A full formalisation of π-calculus theory in the calculus of constructions. In *Proc. TPHOL's '97*, volume 1275 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[32] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[33] K. Honda. Two bisimilarities for the ν-calculus. Technical Report 92-002, Keio University, 1992.

[34] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communications. In *Proc. ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[35] K. Honda and M. Tokoro. A Small Calculus for Concurrent Objects. In *OOPS Messanger*, Association for Computing Machinery. 2(2):50-54, 1991.

[36] K. Honda and N. Yoshida. Replication in Concurrent Combinators. In *Proc. TACS'94*, volume 789 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[37] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.

[38] Kohei Honda. Elementary structures for process theory (1): Sets with renaming. Accepted in the Journal of *Mathematical Structures in Computer Science*, 1999.

[39] Hans Hüttel and Josva Kleist. Objects as mobile processes. Technical report RR-96-38, BRICS - Basic Research in Computer Science, 1996.

[40] C.B. Jones. A π-calculus semantics for an object-based design notation. In *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[41] Eric Jul, Henry Levy, Norman Hutchinson, and Black Andrew. Fine-grained mobility in the emerald system. *ACM Transaction on Computer Systems*, 6(1):109–133, February 1988.

[42] J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *Proc. PROCOMET'98*. North-Holland, 1998.

[43] N. Kobayashi, B.C. Pierce, and D.N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1993. Short version in Proc. POPL'96.

[44] Jean-Jacques Lévy. Some results on the join-calculus. In *Proc. TACS '97*, volume 1281 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[45] M. Merro. On the Expressiveness of Chi, Update and Fusion calculi. In *Proc. EXPRESS'98*, volume 16.2 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[46] M. Merro. On Equators in Asynchronous Name-passing calculi without Matching. In *Proc. EXPRESS'99*, volume 27 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

[47] M Merro. Locality and polyadicity in asynchronous name-passing calculi. In *Proc. FOSSACS2000*, volume 1784 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[48] M. Merro, J. Kleist, and U. Nestmann. Local π at Work: Mobile Objects as Mobile Processes. In *Proc. IFIP TCS2000*, volume 1872 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[49] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proc. ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[50] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[51] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

[52] R. Milner. Action structure for the π-calculus. Technical Report ECS–LFCS–93–264, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1992.

[53] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[54] R.. Milner. Action calculi, or syntactic action structures. In *Proc MFCS'93*, volume 711 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[55] R. Milner. Action calculi V: reflexive molecular forms (with appendix by Ole Jensen). Draft, 1994.

[56] R. Milner. Sorts and types in the π-calculus. Handwritten notes, Edinburgh, December 1990.

[57] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[58] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. ICALP'92*, volume 623 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[59] J. H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

[60] U. Nestmann, H. Hüttel, J. Kleist, and M. Merro. Aliasing Models for Object Migration. In *Proc. EUROPAR'99*, volume 1685 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[61] U. Nestmann, H. Hüttel, J. Kleist, and M. Merro. Aliasing Models for Mobile Objects. *To appear in Journal of Information and Computation*, 2000. The paper is a revised and extended subset of *Migration = Cloning ; Aliasing* which appeared in the *Informal Proceedings of FOOL6*. An extended abstract of this revision entitled *Aliasing Models for Object Migration*, appeared in the *Proceedings of EUROPAR'99*, LNCS 1685, September 1999.

[62] U. Nestmann and B. Pierce. Decoding choice encodings. In *Proc. CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[63] G.K. Ostheimer and A.J.T. Davie. π-calculus characterisations of some practical λ-calculus reductions strategies. Technical Report CS/93/14, St. Andrews, 1993.

[64] J. Parrow and B. Victor. The update calculus. In *Proc. AMAST '97*, volume 1349 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[65] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proc. LICS'98*, IEEE Computer Society Press, 1998.

[66] A. Philippou and D. Walker. On transformations of concurrent object programs. In *Proc. CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[67] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extended abstract in *Proc. LICS '93*, IEEE Computer Society Press.

[68] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press.

[69] G.D. Plotkin. A structural approach to operational semantics. DAIMI-FN-19, Computer Science Department, Aarhus University, 1981.

[70] P. Quaglia and Walker D.. On encoding pπ in mπ. In *Proc. FST & TCS*, volume 1530 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[71] Christine Röckl and Javier Esparza. Proof-checking protocols using bisimulations. In *Proc. CONCUR'99*, volume 1664. Springer Verlag, 1999.

[72] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[73] D. Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, 1994.

[74] D. Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA-Sophia Antipolis, 1995. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press.

[75] D. Sangiorgi. An interpretation of typed objects into typed π-calculus. Technical Report RR-3000, INRIA-Sophia Antipolis, 1996. To appear in Information and Computation.

[76] D. Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 155:39–83, 1996.

[77] D. Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.

[78] D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221:457–493, 1999.

[79] D. Sangiorgi. Typed π-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. *Theory and Practice of Object systems*, 5(1):25–34, 1999.

[80] D. Sangiorgi. Asynchronous process calculi: the first-order and higher-order paradigms (tutorial). To appear in TCS, 2000.

[81] D. Sangiorgi and R. Milner. The problem of "Weak Bisimulation up to". In *Proc. CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[82] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001. To appear.

[83] Carolyn L. Talcott. Obliq semantics notes. Unpublished note. Available from `clt@cs.stanford.edu`, January 1996.

[84] H. Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997. Also available as technical report ECS-LFCS-97-376.

[85] F. van Breugel. A Labelled Transition System for the $\pi\epsilon$-calculus. In *Proc. TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[86] B. Victor and J. Parrow. Concurrent Constrains in the Fusion Calculus. In *Proc. ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[87] Björn Victor. *The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes*. PhD thesis, Dept. of Computer Systems, Uppsala University, Sweden, June 1998. Available from http://www.docs.uu.se/ victor/thesis.shtml.

[88] Jan Vitek. Mobile object systems: Homepage. `http://cuiwww.unige.ch/~ecoopws/`, 1999.

[89] D. Walker. Bisimulation and divergence. *Information and Computation*, 85(2):202–241, 1990.

[90] D. Walker. π-calculus semantics of object-oriented programming languages. In *Proc. TACAS'91*, volume 526 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[91] N. Yoshida. Graph types for monadic mobile processes. In *Proc. FST & TCS*, volume 1180 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[92] N. Yoshida. Minimality and Separation Results on Asynchronous Mobile Processes: representability theorem by concurrent combinators. In *Proc. CONCUR'98*, volume 1466 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.