





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Numéro National de Thèse : 2022ENSL0017

THESE

en vue de l'obtention du grade de Docteur, délivré par

l'ECOLE NORMALE SUPERIEURE DE LYON

en cotutelle avec

Università di Bologna

Ecole Doctorale N° 512

Ecole Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 27/09/2022, par :

Enguerrand PREBET

Typed Behavioural Equivalences in the Pi-Calculus

Équivalences comportementales typées dans le pi-calcul

Devant le jury composé de :

KOUTAVAS, Vasileios	Professeur, Trinity College	Rapporteur
MURAWSKI, Andrzej	Professeur, University of Oxford	Rapporteur
KÖNIG, Barbara	Professeure, Universität Duisburg-Essen	Examinatrice
YOSHIDA, Nobuko	Professeure, Imperial College London	Examinatrice
HIRSCHKOFF, Daniel	Maître de conférences-HDR, ENS de Lyon	Directeur de thèse
SANGIORGI, Davide	Professeur, Università di Bologna	Co-tuteur de thèse

Abstract

In this thesis, I study the notion of program equivalence, i.e. proving that two programs can be used interchangeably without altering the overall observable behaviour. This definition is highly dependent on the contexts in which these programs can be used; does the context have exceptions, parallelism, etc... So proofs also need to be adapted according to the expressiveness of those contexts. This thesis presents the pi-calculus – a concurrent programming language – under various typing constraints. Types allow us to impose different disciplines like forcing a sequential execution, or ensuring linearity, meaning an object can be used once. In each case, the bisimulation, a standard proof technique for the pi-calculus, needs to be adapted accordingly to obtain a suitable equivalence. We then test how using the modified bisimulations can be used to reason about a language with higher-order functions and references, which once translated into the pi-calculus satisfies the typing constraints.

Résumé

Dans cette thèse, j'étudie la notion d'équivalences entre programmes, c'est-à-dire prouver que deux programmes peuvent être utilisé de façon identique sans modifier le comportement global. Cette définition dépend beaucoup du contexte dans lequel ces programmes sont exécutés ; le contexte a-t-il accès à des exceptions, du parallélisme, etc... Ainsi, les preuves doivent être adaptées pour tenir compte des différents niveaux d'expressivité du contexte. Cette thèse s'intéresse au pi-calcul – un langage de programmation concurrent – sous différentes contraintes de typage. Ces types nous permettent de spécifier différentes disciplines comme imposer une exécution séquentielle du programme, ou encore assurer un comportement linéaire, c'est-à-dire que les objets ne peuvent être utilisés qu'une seule fois. Dans chaque cas, la bisimulation, une technique de preuve standard pour le pi-calcul, doit être adapté en conséquence afin d'obtenir une équivalence satisfaisante. Nous testons ensuite comment ces bisimulations modifiées peuvent être mise en pratique pour raisonner sur un langage avec des fonctions d'ordre supérieur et des références, langage qui une fois traduit dans le pi-calcul vérifie les contraintes de typage.

Résumé en français

Dans cette thèse, nous nous intéressons aux équivalences comportementales entre programmes. Cette notion s'appuie sur l'idée que deux programmes doivent être considérés comme égaux s'ils se comportent de façon similaire. Ces programmes peuvent correspondre à différents objets : fonctions, processus, etc. La principale équivalence comportementale étudiée est l'équivalence contextuelle [36], où des programmes équivalents peuvent être utilisés de manière interchangeable peu importe le contexte dans lequel ils sont utilisés, et cela sans altérer le résultat final. Lorsque l'on remplace un programme par un autre plus efficace, avoir une façon fiable de montrer que ces deux programmes sont bel et bien équivalents permet de s'assurer que l'optimisation n'a pas introduit de nouveaux bogues.

Cette notion reste malgré tout très générale et il y a beaucoup de libertés quant à la définition de quel comportement peut être considéré comme observable ou non. Prenons par exemple les deux programmes suivants :

$$P_1 \stackrel{\text{def}}{=} x := x + 1; x := x + 1 \qquad\qquad P_2 \stackrel{\text{def}}{=} x := x + 2$$

Le programme P_1 incrémente la valeur de x deux fois, tandis que le second ajoute directement 2 à la valeur de x. À la fin, la valeur stockée est identique. Doit-on pour autant considérer ces deux programmes équivalents ? Comme P_2 effectue moins d'opérations, on peut affirmer que le temps nécessaire pour exécuter ces deux programmes est un critère suffisant pour distinguer ces programmes. Dans notre cas, nous étudions des équivalences dites *faibles*, c'est-à-dire que le temps d'exécution n'est pas pris en compte lorsque l'on compare des programmes. Ce type d'équivalences est pratique si l'on souhaite faire de l'optimisation de programmes. Si l'on prouve que ces deux programmes sont équivalents, il sera alors toujours possible d'utiliser P_2 qui est plus rapide, sans pour autant compromettre la fiabilité du reste du code.

Cependant, même sans regarder la durée d'exécution, il existe des cas où P_1 et P_2 ne doivent pas être considérés comme équivalents. Dans cet exemple, nous n'avons pas précisé quel type de contextes pouvait interagir avec P_1 et P_2. Supposons que nous puissions faire tourner ces deux programmes en parallèle d'un troisième qui affiche la valeur de x. Cet affichage peut donc être effectué à n'importe quel instant de l'exécution. Ainsi, si on considère que x a initialement la valeur 0, alors avec P_2, la valeur 0 ou 2 sera affichée selon le moment où l'affichage a lieu. Au contraire, avec P_1, il est aussi possible que la valeur 1 soit affichée si l'affichage est effectué entre les deux incrémentations.

Notre approche consiste à étudier l'équivalence comportementale dans le π-calcul, un langage où les processus interagissent entre eux via l'envoi et la réception de messages. Il est possible de raisonner avec d'autres langages en utilisant une traduction, ou *encodage*, c'est-à-dire une fonction qui transforme les programmes d'un langage source en programmes d'un langage cible, qui dans notre cas est le π-calcul. Par exemple, les fonctions sont représentées comme des processus qui se comportent comme un serveur. Celui-ci attend la réception d'un message, correspondant à un appel à ladite fonction avec ses arguments, puis exécute le corps de la fonction. En pratique, le message contient, en plus des arguments de la fonction, un nom additionnel qui est utilisé comme adresse de retour, afin de pouvoir renvoyer le résultat une fois calculé. Il est aussi possible de traduire des objets impératifs comme les références. Une référence est représentée par un message dont le contenu correspond à la donnée stockée dans la référence. Ainsi, en recevant ce message puis en en réémettant un nouveau sur le même canal de communication, un processus peut lire le contenu de la référence mais aussi le modifier.

Une traduction est intéressante lorsque les équivalences du langage source et du langage cible sont reliées. On parle d'encodage *correct* quand les programmes traduits (dans le langage cible) sont équivalents seulement s'ils l'étaient aussi dans le langage source. Cette propriété permet de transporter toutes les preuves d'équivalence obtenues dans le langage cible, comme le π-calcul, vers le langage source. La propriété complémentaire, la *complétude*, garantie à l'inverse que toutes les équivalences valables dans le langage source sont également vraies pour les traductions des programmes. Comme le π-calcul est un langage très expressif de part la présence de parallélisme, obtenir la complétude pour un encodage vers le π-calcul est difficile. Pour reprendre l'encodage des références esquissé plus haut, un processus peut très bien recevoir un message sans en réémettre un autre immédiatement, cela empêche des communications futures, chose qui n'est pas possible avec des références. En fait, lorsqu'un encodage est correct, les contextes du langage cible peuvent présenter des comportements qui ne correspondent pas à la traduction de comportements présents dans le langage source.

Dans le π-calcul, les processus peuvent effectuer des actions qui déterminent les processus qu'ils deviendront. Ces actions correspondent à des envois, des réceptions de message ou des calculs internes. Nous nous intéressons à la technique de la bisimulation, une équivalence comportementale qui utilise toutes ces actions afin de distinguer les processus. Intuitivement, si un processus P évolue en un processus P' en effectuant une action μ, alors n'importe quel processus Q équivalent – ou *bisimilaire* – doit aussi être capable d'effectuer l'action μ tout en évoluant vers un processus Q' bisimilaire à P'. Cette équivalence est comparée à une équivalence contextuelle appelée *congruence barbelée*. La seconde est souvent considérée comme une équivalence plus naturelle, les preuves sont plus complexe étant donné qu'il faut prendre en compte tous les contextes possibles. Heureusement, la bisimulation est correcte, et donc nous pouvons utiliser la bisimulation, qui est plus souple, afin de prouver la congruence barbelée des processus. La bisimilarité est aussi complète pour quasiment tous les processus.

Dans cette thèse, nous étudions comment différents systèmes de type affectent les équivalences décrites précédemment. Les systèmes de type permettent d'ajouter des informations aux programmes pour prévenir les erreurs : ainsi, une fonction qui attend un entier ne doit pas recevoir une chaîne de caractères à la place. Les types apportent aussi des informations sur le comportement des programmes qui rendent possible certaines optimisations pour les compilateurs. Par exemple, la linéarité est une discipline qui indique que l'objet doit être utilisé une seule fois. Cette information peut être utilisée pour faciliter le ramassage de miettes.

Certains résultats d'équivalence ne sont valides que sous certaines conditions, conditions qui peuvent être formulées par une discipline de typage. Par exemple, l'équivalence entre les processus P_1 et P_2 est vraie en l'absence de parallélisme. Dans ce travail, nous définissons un système de type pour la séquentialité où les processus sont considérés soit comme actifs, soit comme passifs. Garantir une exécution séquentielle représente alors simplement l'absence de deux processus actifs en même temps. Ainsi, l'équivalence entre P_1 et P_2 est valide lorsque l'on impose la propriété de séquentialité aux processus. En plus du système de type pour la séquentialité (Chapter 4), nous avons aussi développé un système de type pour les références (Chapter 3) et un garantissant une exécution bien-parenthésée (Chapter 5) dans le π-calcul. Quand le langage est typé, certains contextes sont interdits car ils ne sont pas typables; comme tous les processus ne sont pas typables, il en va de même pour les contextes. En conséquence, la définition de la congruence barbelée devient plus lâche. La bisimulation standard non typée peut toujours être utilisée mais risque de rater les nouvelles égalités obtenues grâce aux contraintes de typage. Pour récupérer le lien entre la bisimulation et la congruence barbelée, nous devons adapter la bisimulation au système de type utilisé. Nous le faisons pour les différents systèmes proposés. Intuitivement, lorsque l'on définit une bisimulation typée, certaines actions ne doivent pas être considérées sous certains typages car il n'existe pas de contexte avec le bon type permettant d'effectuer l'action μ. Finalement, nous présentons un exemple d'application de ces systèmes de type en fournissant un encodage totalement abstrait, soit correct et complet, pour un langage avec des fonctions d'ordre supérieur et des références : $\lambda^{\mathtt{ref}}$ (Chapter 6). Pour cela, nous partons du système de type pour l'exécution bien parenthésée et la bisimulation correspondante. Elle doit cependant être adaptée pour tenir compte des termes à divergence différée, des termes bloqués mais qui divergent dès qu'ils sont mis en présence d'un contexte. Cela entraîne la définition de la notion de bisimulation avec divergence pour le π-calcul.

Riassunto in italiano

In questa tesi vengono studiate equivalenze comportamentali tra programmi. La nozione di equivalenza comportamentale si basa sull'idea che due programmi sono equivalenti se agiscono in modo simile. Questi programmi possono essere di vari tipi: funzioni, processi, ecc. L'equivalenza comportamentale che è maggiormente studiata è l'equivalenza contestuale [36], in cui due programmi equivalenti possono essere scambiati in ogni contesto, senza alterare il risultato finale. In una situazione in cui si sostituisce un programma con un altro programma, più efficente, si può dimostrare che una tale ottimizzazione non introduce nessun bug a patto di poter certificare che i due programmi sono equivalenti.

Partendo dall'idea generale secondo la quale due programmi equivalenti dovrebbero avere lo stesso comportamento, ci sono tante possibilità per decidere quali comportamenti sono osservabili o meno. Si possono ad esempio considerare i due programmi seguenti:

$$P_1 \stackrel{\text{def}}{=} x := x + 1; x := x + 1 \qquad\qquad P_2 \stackrel{\text{def}}{=} x := x + 2$$

P_1 incrementa due volte il valore di x, mentre il secondo programma aggiunge 2 a x. Dopo aver eseguito questi due programmi, il valore di x dovrebbe essere lo stesso. È quindi lecito considerare che questi due programmi sono equivalenti? Un punto di vista è di dire che dato che P_2 effettua meno operazioni, il tempo di esecuzione può essere usato per considerare questi due programmi differenti. Le equivalenze che consideriamo in questo documento sono *deboli*, il che significa che non dipendono dal tempo: se due processi sono diversi soltanto perchè calcolano lo stesso risultato ma uno è più lento dell'altro, allora sono equivalenti. Le equivalenze deboli permettono di analizzare ottimizzazioni. Se si dimostra che i due programmi qui sopra sono equivalenti, allora si può sempre usare P_2, che è più rapido, senza alterare il comportamento del programma globale.

Anche senza considerare il tempo, P_1 e P_2 possono essere considerati diversi. In questo esempio, non abbiamo spiegato quali contesti possono essere usati per interagire con P_1 e P_2. Supponiamo che questi programmi vengano eseguiti in parallelo con un programma che stampa il valore di x, in qualsiasi momento durante l'esecuzione. Se il valore iniziale di x è 0, allora per P_2 si può osservare 0 o 2, mentre per P_1, è anche possibile osservare 1 se il valore di x viene stampato tra le due assegnazioni.

In questo lavoro si studiano equivalenze comportamentali per il π-calculus, un linguaggio in cui i processi interagiscono tra di loro trasmettendo messaggi. Due linguaggi di programmazione possono essere messi in relazione tramite una *traduzione*, che è una funzione da un linguaggio sorgente verso un linguaggio oggetto. Nel nostro caso, il linguaggio oggetto è il π-calculus. Funzioni, ad esempio, possono essere rappresentate da un processo che si comporta come un server. Il server aspetta di ricevere un messaggio, che rappresenta la chiamata alla funzione, e che contiene l'argomento della funzione, per poi eseguire il corpo della funzione. Il messaggio comporta un secondo argomento, un nome, che indica su quale canale deve essere mandato il risultato della funzione. Riferimenti, o altri costrutti della programmazione imperativa, possono anche essere rappresentati nel π-calculus. Un messaggio può essere considerato come un dato in memoria. Ricevendo un tale messaggio e rimandandone uno sullo stesso canale, un processo legge il contenuto del riferimento e lo modifica con un nuovo valore.

Una traduzione è interessante quando le equivalenze nel linguaggio sorgente e nel linguaggio oggetto possono essere associate usando la traduzione. Quando le traduzioni di due programmi del linguaggio sorgente sono equivalenti soltanto se sono equivalenti nel linguaggio oggetto, si dice che la traduzione è *corretta*. Questa proprietà garantisce che tutte le equivalenze che si possono dimostrare nel linguaggio oggetto, cioè il π-calculus, sono valide per i programmi del linguaggio sorgente. Viceversa, la *completezza* della traduzione significa che tutte le equivalenze nel linguaggio sorgente possono essere dimostrate traducendo verso il π-calculus, e provando l'equivalenza dei programmi tradotti. Ottenere la completezza è difficile quando si traduce verso il π-calculus, perchè si tratta di un linguaggio molto espressivo.

Tornando alla traduzione dei riferimenti, se un processo riceve un messaggio senza rimandarne uno, le communicazioni su questo canale possono essere bloccate su questo canale, il che non ha senso dal punto di vista dei riferimenti (nel linguaggio sorgente). Dal punto di vista della correttezza, bisogna tener conto del fatto che i contesti nel linguaggio oggetto non hanno soltanto comportamenti che vengono dalla traduzione di un contesto nel linguaggio sorgente.

Nel π-calculus, i processi possono fare certe azioni, determinando quali azioni il processo potrà fare dopo. Queste azioni sono una transizione autonoma, o l'emissione di un messaggio, o la ricezione di

un messaggio. In questa tesi, si studia la tecnica della bisimulazione, che è un equivalenza comportamentale in cui tutte le azioni possono essere usate per fare distinzioni tra processi. Intuitivamente, se un processo P fa un azione μ diventando un processo P', un processo Q equivalente (o *bisimile*) deve essere anche lui capace di fare un azione μ, diventando un processo Q' che deve essere bisimile a P'. Questa equivalenza viene paragonata ad una equivalenza contestuale, la *congruenza barbed*. Quest'ultima si può generalmente definire in modo più naturale, imponendo che i processi rimangano equivalenti in ogni contesto, ma rende le dimostrazioni più difficili, dato che è necessario fare una dimostrazione per ogni contesto. Fortunatamente, la bisimulazione è sempre corretta, e più facile da usare; si può dunque usare la bisimulazione per stabilire che due processi sono congruenti barbed. La bisimulazione è anche completa per quasi tutti i processi.

In questa tesi vengono studiati vari sistemi di tipi che raffinano le equivalenze descritte qui sopra. I sistemi di tipi aggiungono informazioni ai programmi in modo da evitare errori, ad esempio una funzione con un argomento intero non può essere chiamata con una stringa. I tipi possono anche dare informazioni sul comportamento dei programmi, il che permette ai compilatori di fare più ottimizzazioni. Ad esempio, la linearità, che impone di usare ogni oggetto una volta esattamente, può essere usata per facilitare la garbage collection.

Certe equivalenze tra processi sono valide sotto certe condizioni, che possono essere formulate usando un sistema di tipi. È il caso per l'equivalenza dei programmi P_1 e P_2, che è valida se si vieta il parallelismo. In questo lavoro definiamo un sistema di tipi per la sequenzialità, in cui i processi sono attivi o passivi. Per garantire la sequenzialità, si verifica che non si possono mai avere due componenti attivi contemporaneamente. L'equivalenza tra P_1 e P_2 è valida quando si impone la proprietà di sequenzialità. Oltre al sistema per la squenzialità (parte 4), presentiamo anche un sistema per i riferimenti (parte 3) e uno per un esecuzione *well-bracketed* nel π-calculus (parte 5). Tramite il tipaggio, certi contesti sono eliminati, in quanto non tipabili. Di conseguenza, la congruenza barbed diventa meno discriminante. La bisimulazione standard, non tipata, può essere usata per stabilire equivalenze, ma non permette di convalidare certe equivalenze che sono garantite dai vincoli di tipaggio. Per avere la coincidenza tra bisimulazione e congruenza barbed, è necessario adattare la bisimulazione al sistema di tipi. È quello che viene fatto in questa tesi per i vari sistemi di tipi che presentiamo. Intuitivamente, nella definizione della bisimulazione, non si considerano certe azioni μ, perchè non esiste un contesto tipato che renda possibile l'azione μ.

Infine, si presenta un esempio di applicazione di questi sistemi di tipi, studiando una traduzione completamente astratta (*fully abstract)*, cioè corretta e completa, per un linguaggio con funzioni di ordine superiore e riferimenti: λ ref (parte 6). Per ottenere questo risultato, si parte dal sistema di tipi per il well-bracketing, e dalla bisimulazione definita per questo sistema di tipi. Quest'ultima deve essere adattata per tener conto dei lambda termini con divergenza ritardata, che sono termini bloccati ma che generano una divergenza appena sono messi in un contesto. Questo ci porta a introdurre una nozione di bisimulazione con divergenza nel π-calculus.

Contents

1 Introduction **8**

2 Background **12**
2.1 π-calculus . . . 12
2.2 Bisimulations and up-to techniques . . . 13
2.2.1 Bisimulations . . . 13
2.2.2 Up-to techniques . . . 16
2.3 Typing and sorting . . . 18

3 References in the π-calculus **21**
3.1 Asynchronous π-calculus . . . 21
3.2 Reference names in Aπ . . . 22
3.2.1 Types and contextual equivalences with reference names . . . 23
3.2.2 Behavioural equivalences with reference names . . . 25
3.2.3 Bisimulation with reference names . . . 26
3.2.4 Examples . . . 28
3.3 Application: a π-calculus with references . . . 29
3.3.1 Syntax and semantics of $\pi^{\mathtt{ref}}$ . . . 29
3.3.2 Mapping $\pi^{\mathtt{ref}}$ onto the Asynchronous π-calculus . . . 30
3.3.3 Behavioural equivalence in $\pi^{\mathtt{ref}}$: examples . . . 31

4 Sequentiality **33**
4.1 Type system . . . 33
4.2 Behavioural equivalence . . . 36
4.2.1 Sequential bisimilarity . . . 36
4.2.2 Completeness for output-controlled names . . . 37
4.2.3 Examples . . . 42
4.3 Sequential references . . . 42
4.3.1 Combining sequentiality with the type system for references . . . 42
4.3.2 Examples . . . 43

5 Well-Bracketing **45**
5.1 Internal π-calculus: Iπ . . . 45
5.2 Well-bracketing . . . 46
5.2.1 Type system . . . 46
5.2.2 The well-bracketing property on traces . . . 49
5.3 Behavioural equivalences . . . 50
5.3.1 Typed barbed equivalence, well-bracketed bisimulation and soundness . . . 50
5.3.2 Completeness . . . 50
5.4 Extension to Aπ . . . 55

6 Full Abstraction for a Higher-Order Language with References **57**
6.1 Encoding a higher-order language with references in Iπ . . . 57
6.1.1 Definition of $\lambda^{\mathtt{ref}}$ and encoding in Iπ . . . 57
6.1.2 References in $\lambda^{\mathtt{ref}}$ and well-bracketing . . . 60
6.1.3 Technical results about the encoding . . . 61
6.1.4 Additional examples . . . 63
6.2 Full abstraction . . . 64
6.2.1 Normal form bisimilarity . . . 64
6.2.2 A π-calculus characterisation of contextual equivalence in $\lambda^{\mathtt{ref}}$ . . . 66
6.3 Up-to techniques for $\approx_{\mathrm{div}}$ in Iπ, and applications . . . 68
6.3.1 A new up-to technique for Iπ: up-to body . . . 68
6.3.2 Examples of other equivalences in $\lambda^{\mathtt{ref}}$ . . . 69

7 Conclusion **71**

A Proofs for Chapter 3 **76**
A.1 Definitions and results about Aπ with references . . . 76
A.1.1 Type system for output receptiveness: proof of subject reduction . . . 76
A.1.2 Proofs about $\approx_{\mathrm{ip}}$ . . . 77
A.2 Characterisation of $\cong^{Arn}$ using $\approx_{\mathrm{r}}$ . . . 78
A.2.1 Soundness . . . 78
A.2.2 Completeness . . . 80
A.3 Results and examples for Section 3.3 . . . 81
A.3.1 Properties of the encoding . . . 81
A.3.2 Additional material for the examples in Section 3.3.3 . . . 81

B Proofs for Chapter 6 **85**
B.1 A simplified example with wb-bisimulation (Section 6.1.2) . . . 85
B.2 Additional material for Section 6.1.3 . . . 87
B.3 Additional material for Section 6.2.2 . . . 90

Chapter 1

Introduction

In this thesis, we study behavioural equivalences between programs. The notion of behavioural equivalence revolves around the idea that two programs should be considered equal if they act in a similar way. These programs can be of various forms: functions, processes, etc. The most studied behavioural equivalence is contextual equivalence [36], where equivalent programs could be used interchangeably in any context, without altering the overall result. When replacing a program by a more efficient one, having a reliable way to show that both programs are indeed equivalent can ensure that performing this optimisation will not create bugs.

Based on the general idea that equivalent programs should exhibit the same behaviour, there is still a lot of freedom with respect to what behaviour should or should not be considered observable. For instance, considering the following two programs:

$$P_1 \stackrel{\text{def}}{=} x := x + 1; x := x + 1 \qquad\qquad P_2 \stackrel{\text{def}}{=} x := x + 2$$

P_1 increments the value of x twice, while the second adds 2 to x. In the end, for both, the value stored at x should be identical. Should these programs be considered equivalent? One may argue that since P_2 does less operations, the time required to run them both could be used to distinguish between the two programs. However, the equivalences we use in this work are called *weak* in the sense that they are time-irrelevant: if the only difference between two processes is that one takes longer to compute the result, then they are still equivalent. Weak equivalences are suited to work on optimising programs. If we can prove that the two programs are equivalent, then we can always use P_2 that is faster, without the risk of breaking the existing code.

But even without taking time into account, P_1 and P_2 may not be equivalent. In this example, we did not explain what contexts could be used to interact with P_1 and P_2. Suppose that we can run these programs in parallel with one which prints the value of x. This can occur at any time during the computation. Suppose x initially has the value 0, then with P_2, either 0 or 2 can be printed depending on whether the print occurs before or after the assignment. On the other hand, with P_1, it is also possible to print 1 if the print is executed between the two assignments.

Our approach is to study behavioural equivalence in the π-calculus, a language where processes can interact with each other by sending and receiving messages. Relating programming languages is done by using a translation, or *encoding*, i.e. a function from a source language to a target language. In our case, the target language is the π-calculus. For instance, functions can be represented by a process that acts as a server. It is waiting to receive a message, modelling a call of the function with its argument and then executing its body. In the message, there is an additional argument, a name used to know to which channel the result should be sent once the function has finished computing the result. References and other imperative constructs can also be translated into the π-calculus. A message that is sent can be seen as a piece of data. By receiving that message and sending back a new one at the same channel, a process reads the content of the reference and updates it with a new value.

A translation is interesting when the equivalences for both the source and the target language can be related using the translation. When processes which are translations of source programs are only equivalent if they are also equivalent in the source language, we say that the encoding is *sound*. This property ensures that all equivalence results obtained in the target language, e.g. the π-calculus, are

correct with respect to the equivalence in the source language. The converse property, i.e. *completeness*, guarantees that every equivalence from the source language can be recovered by translating the programs into the π-calculus and proving the equivalence there. Obtaining completeness is difficult when encoding in π as the language is very expressive. Regarding the encoding of references sketched earlier, if a process receives a message without emitting a new one afterwards, this essentially prevents further communications on that channel which is not possible with references. For a sound encoding, the contexts in the target language may exhibit behaviours that do not correspond to the translation of a behaviour in the source language.

In the π-calculus, processes can perform actions which determine what the process becomes next. Such actions are either an internal computation, sending a message, or receiving a message. We will be interested in the bisimulation technique, which is a behavioural equivalence that uses all actions to distinguish between processes. Intuitively, if a process P becomes the process P' when performing the action μ, any equivalent – or *bisimilar* – process Q should also be able to perform an action μ that results in a process Q' which must be bisimilar to P'. This equivalence is compared to a contextual equivalence named *barbed congruence*. While the latter is often a more natural equivalence, as processes remain equivalent in any context, this makes proofs much harder as we need a proof for every possible context. Fortunately, bisimulation is always sound, and thus we can use bisimulation, which is more tractable, to prove that processes are barbed congruent. It is also complete on almost all processes.

In this thesis, we study how various type systems affect the equivalences we have described. Type systems add information to programs to prevent errors, e.g. a function that awaits an integer should not receive a string instead. Types can also give information about the behaviour of programs which enables more optimisations for compilers. For instance, linearity, the discipline for which objects must be used exactly once, can be used to facilitate garbage collection.

Some equivalences only hold under some conditions, which can be formulated as a typing discipline. The equivalence between P_1 and P_2 is such a case, as it holds in the absence of parallelism. In this work, we define a type system for sequentiality where processes are considered either active or passive. Ensuring a sequential execution then simply means that there can never be two active processes at the same time. The equivalence between P_1 and P_2 holds when we impose the sequentiality property on processes. Along with the type system for sequentiality (Chapter 4), we also develop a type system for references (Chapter 3) and one for a well-bracketed execution (Chapter 5) in the π-calculus. When the language is typed, certain contexts are ruled out as ill-typed; as not all processes can be typed, neither can all contexts. As a consequence, the definition of barbed congruence becomes coarser. The standard untyped bisimulation can still be used to prove equivalences but may miss some new equalities created by typing constraints. To recover the link between bisimulation and barbed congruence, we need to adapt the bisimulation to the type system used. We do so for the various type systems we present. Intuitively, when defining the bisimulation, some actions μ should not be considered under certain typing as there exists no context with the correct type that can lead to μ being performed.

We begin by presenting the π-calculus in Section 2.1. We will use various subcalculi of the π-calculus in this thesis, which will be defined with more details when needed. We introduce the standard equivalences along with the notion of bisimulation (Section 2.2) and a general framework explaining how it can be adapted in the presence of type systems (Section 2.3).

We introduce in Chapter 3 how references can be implemented in the asynchronous variant of π-calculus, Aπ (Section 3.1). This is done by first presenting our type system (Section 3.2) which imposes constraints on *reference names*, a subset of names representing references. Specifically, we design two bisimulations, reference bisimulation and bisimulation with an inductive predicate. We then show an application of this type system by considering a π-calculus extended with explicit references and imperative constructs, $\pi^{\mathtt{ref}}$, (Section 3.3) and show how to encode $\pi^{\mathtt{ref}}$ in Aπ.

With references and full concurrency, contexts are very expressive, and few equivalences can be proved. Thus, we shift our focus into ensuring a sequential execution in Chapter 4. We define a type system for sequentiality for which the encoding of references can be typed in Section 4.1. We then study the corresponding equivalence leading us to the notion of sequential bisimulation in Section 4.2. We briefly explain how the two systems for sequentiality and references can easily be combined in Section 4.3. Using sequential bisimulations with inductive predicate, we are able to prove results similar to the equivalence of P_1 and P_2 above.

The notion of well-bracketing is a refinement of sequentiality. It builds upon the distinction of

between two kinds of communication: requests (or questions) and answers to those requests. Although well-bracketing can have a meaning in a concurrent setting, during a sequential execution, it consists of always answering to the last unanswered request. It is best expressed in the Internal π-calculus, where all messages sent correspond to private names (Section 5.1). The type system and corresponding bisimulation, wb-bisimulation, are given following the same general schema in Sections 5.2 and 5.3.

Finally, we look at an application for these type systems by providing a fully abstract encoding of a λ-calculus with references: $\lambda^{\mathtt{ref}}$ in Chapter 6. The encoding is described in Section 6.1. Wb-bisimilarity is still too strong because of deferred divergent terms, stuck terms which are expected to diverge. We thus have to adapt wb-bisimulation to handle divergence leading to the notion of bisimulation with divergence (Section 6.2).

Related Works

Techniques like logical relations have been used to prove properties on programs by defining the relations inductively on the types of the program considered, including equivalence of programs [40]. Its extension, Kripke logical relations [39], allows reasoning on programs with state. Relations are indexed by a possible world, which constrains the heap. This means that programs are related when they behave "the same" under any heaps satisfying the constraints from the world. By making worlds evolve over time [3], and by further specifying these evolutions (using *private transitions* and *inconsistent states*), a fully abstract logical relation for a ML-like language was defined [9].

The bisimulation technique has been studied for lambda-calculus and its extensions. Applicative bisimilarity is fully abstract for the lambda-calculus [1]. However, this method does not extend when introducing state. By allowing values to be stored and reused multiple times, a full abstraction result is obtained using environmental bisimulation, the environment storing disclosed values [50]. The open version of applicative bisimulation for call-by-value, normal form bisimulation [28], is also too strong compared to contextual equivalence. It requires both state and control operator, like in the $\lambda\mu\rho$-calculus, to be fully abstract. Full abstraction can be recovered using environment to store values and also contexts [6, 26].

A proof of bisimilarity requires a relation which must be a bisimulation. This can become bothersome when some pairs are easily bisimilar. The use of up-to techniques allows to reduce the size of the relation, by only requiring the relation to be included in a bisimulation. For instance, using up-to bisimilarity [33], the processes only need to be bisimilar to processes related by the relation. Up-to techniques have been studied to know when they are sound and can be combined soundly. One such class of techniques are safe functions [48]. The use of the companion, the largest such function, allows for a more unified reasoning [42]. Additionally, some functions that are not sound up-to techniques can still be used in some cases without leading to unsound reasoning. The same analysis for combining always sound techniques and the others was done by Biernacki et al. [7]. Hur et al. [19] defines tools to incrementally build the relation instead of defining it in full first. This parametrised coinduction allows accumulation of pairs of processes which can be used soundly later after a step has been done. This ensures that the coinduction hypothesis is only used under a "semantic guard". The method was refined to enable unguarded use of the accumulated knowledge if that guard condition was previously fulfilled [55].

The π-calculus has served as a model for multiple higher order languages. Milner first encoded the standard λ-calculus for both call-by-name and call-by-value [35]. For the former, the equivalence induced by the bisimilarity in the π-calculus corresponds to the equality of Lévy-Longo trees [30, 49]. On the other hand, the one for (refined) encoding for call-by-name is normal form bisimilarity [28] extended to handle η-equality. By introducing types in the π-calculus, Berger et al. [5] obtain a fully abstract encoding w.r.t contextual equivalence for a functional language, PCF [13]. Similar full abstraction holds between a linear version System F and a polymorphic session π-calculus, with two encodings going in each direction [54]. There are also sound encodings for languages with first-order references and parallelism with call-by-name, e.g. Concurrent Idealissed ALGOL [8, 44] .

The π-calculus bears some similarity with operational game semantics [23]. Game semantics is a denotational semantics where programs are represented by a *strategy*. For sequential programs, various capabilities or their absence in programs (like state or control) are modelled by adding constraints on those strategies (e.g. innocence, well-bracketing) [2]. In particular, game semantics model has been defined for RefML [37]. On a more operational setting, Laird [27] developed a trace semantics where transitions resemble actions from game semantics, intuition which was confirmed later [21]. This approach

makes it easier to prove equivalences using automated techniques [26, 22].

In typed settings, the bisimulation technique must be adapted, taking the type into account. One such type system for the π-calculus is I/O-types [38] which separate the input and output capabilities of a channel. A fully abstract typed bisimilarity for I/O types with subtyping was proposed by Hennessy [14]. The notion of receptiveness for which an input must be "immediately" available, was formalised as a type system by Sangiorgi [45] for two cases, if the input appears once (linear receptiveness) or is replicated (uniform receptiveness). For each, a sound and complete bisimulation is given.

Chapter 2

Background

2.1 π-calculus

π-calculus processes are defined using an infinite set of names, ranged over by $a, b, \ldots$. These names represent channels that are used by processes to interact with each other, sending and receiving messages (i.e. other names) via these channels. We use a tilde, like in $\widetilde{b}$, for (possibly empty) tuples of names.

The standard syntax of the π-calculus for *processes*, and *guarded processes* (or summations) is given below [52]:

$$\begin{array}{rcl} P, Q & ::= & P \mid Q \mid (\boldsymbol{\nu} a)P \mid !a(\widetilde{b}).\, P \mid G \\ G, G' & ::= & \mathbf{0} \mid \overline{a}\langle\widetilde{b}\rangle.\, P \mid a(\widetilde{b}).\, P \mid \tau.\, P \mid [a = b]G \mid G + G' \end{array}$$

$\mathbf{0}$ stands for the empty process which does nothing. There are three kinds of prefixes: the output $\overline{a}\langle\widetilde{b}\rangle.\, P$ sends the names $\widetilde{b}$ via the channel a before continuing as P; the input $a(\widetilde{b})$ receives names via a and continues as P with $\widetilde{b}$ being distinct placeholders for the names it has received; finally $\tau.\, P$ performs one step of internal computation before executing the process P. In prefixes $\overline{a}\langle\widetilde{b}\rangle.\, P$ and $a(\widetilde{b}).\, P$, name a is the *subject* and $\widetilde{b}$ are the *objects*. The matching $[a = b]G$ behaves as G only if the two names are equal and does nothing otherwise. The process $G + G'$ behaves as G or G'. For instance, the process $\overline{a}\langle b\rangle.\, P + c(x).\, Q$ may either send b via a and then proceed as P or receive a name via c and proceed as Q. The parallel composition of P and Q, i.e. $P \mid Q$, allows both processes to be executed and possibly to interact with each other. The restriction $(\boldsymbol{\nu} a)P$ creates a private name only usable in P. A replicated input $!a(\widetilde{b}).\, P$ behaves as $a(\widetilde{b}).\, P$ in parallel with itself an unbounded number of times. This means that such process can always receive names at a, each time creating a new copy of P.

Variants of these operators exist. The replication can be generalised to all process '$!P$' instead of just input. The semantics of $!P$ is more complex as the multiple copies of P may interact with each other. Nevertheless, this generalisation can be simulated with our calculus as $(\boldsymbol{\nu} a)(!a().\, (P \mid \overline{a}\langle\rangle.\, \mathbf{0}) \mid \overline{a}\langle\rangle.\, \mathbf{0})$. One can also add the operator of mismatching $[a \neq b]$ which acts opposite to the matching. Its addition would not affect the work done here. The main usage of the matching is for proofs of completeness in Sections 3 and 4. It does not play a major role otherwise. In Section 5 when the Internal π-calculus is introduced, it is removed as all names exchanged are private. We work with the polyadic π-calculus: channels may carry more than one name. This will be needed in Section 5 as we require some channels (function names) to always carry a continuation name. This can be done easily using polyadicity without removing the capability to send other names with these channels.

Names $\widetilde{b}$ in $a(\widetilde{b}).\, P$, $!a(\widetilde{b}).\, P$ and a in $(\boldsymbol{\nu} a)P$ are bound in P. We identify processes up to alpha conversion with respect to these binders, e.g. we consider $(\boldsymbol{\nu} b)a(c).\, \overline{c}\langle b\rangle$ and $(\boldsymbol{\nu} d)a(e).\, \overline{e}\langle d\rangle$ equal.

We write $\mathrm{fn}(P)$ the set of free names of P and we write $P\{{}^{b}\!/_{a}\}$ for the result of replacing every free occurrence of name a by b in P in a capture-avoiding way. This may require using alpha-conversion, e.g. $(\boldsymbol{\nu} b)\overline{a}\langle b\rangle\{{}^{b}\!/_{a}\} = (\boldsymbol{\nu} c)\overline{a}\langle c\rangle\{{}^{b}\!/_{a}\} = (\boldsymbol{\nu} c)\overline{b}\langle c\rangle$.

Notations. We often write $a.\, P$ and $\overline{a}.\, P$ when the object of a prefix is the empty tuple, and $a(_).\, P$ for $a(\widetilde{b}).\, P$ with $\widetilde{b} \notin \mathrm{fn}(P)$. We also omit $\mathbf{0}$ after a prefix. We use $\sum_{i\in I} G_i$ (resp. $\prod_{i\in I} P_i$) for $G_{i_1} + \cdots + G_{i_n}$ (resp. $P_{i_1} \mid \ldots \mid P_{i_n}$) where $I = \{i_1, \ldots, i_n\}$. We write $(\boldsymbol{\nu}\widetilde{a})P$ for a sequence of such restrictions. Parallel

composition and sum have the lowest precedence among operators meaning $(\boldsymbol{\nu} a)P \mid Q$ is $((\boldsymbol{\nu} a)P) \mid Q$, $a(x).\, P \mid Q$ is $(a(x).\, P) \mid Q$. Also, $G + G' \mid P$ is $(G + G') \mid P$ (the other option is not accepted by the grammar).

Contexts, ranged over by C, are processes containing a single occurrence of a special constant, the hole (written $[\cdot]$), as if it was added to the grammar of P, Q. The *static* contexts, ranged over by E, have the form $(\boldsymbol{\nu}\widetilde{a})(P \mid [\cdot])$. We write $C[P]$ for the process obtained by replacing the hole by P in C.

Structural congruence is the smallest equivalence that contains the axioms of Figure 2.1 and is also a congruence, i.e. $P \equiv Q$ implies $C[P] \equiv C[Q]$. It contains standard rules for parallel composition and sum which are symmetric, associative and admit $\mathbf{0}$ as neutral element. Structural congruence is used to define the usual reduction-based semantics of the π-calculus which we do not use. Nevertheless, structural congruence provides simple laws that are verified for all (reasonable) behavioural equivalences.

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad G + \mathbf{0} \equiv G \qquad G + G' \equiv G' + G$$

$$G + (G' + G'') \equiv (G + G') + G'' \qquad\qquad !a(\widetilde{b}).\, P \equiv a(\widetilde{b}).\, P \mid !a(\widetilde{b}).\, P$$

$$P \mid (\boldsymbol{\nu} a)Q \equiv (\boldsymbol{\nu} a)(P \mid Q) \text{ if } a \notin \mathrm{fn}(P) \qquad (\boldsymbol{\nu} a)(\boldsymbol{\nu} b)P \equiv (\boldsymbol{\nu} b)(\boldsymbol{\nu} a)P \qquad (\boldsymbol{\nu} a)\mathbf{0} \equiv \mathbf{0} \qquad [a = a]G \equiv G$$

Figure 2.1: Structural congruence in the π-calculus

Thanks to structural congruence, we can describe static contexts by the following grammar:

$$E ::= [\cdot] \mid (\boldsymbol{\nu} a)E \mid E \mid P$$

The semantics of the π-calculus is given by a Labelled Transition System (LTS) and is presented in Figure 2.2. Symmetric rules for Par, Sum and Comm have been omitted. Multiple LTS exist for the π-calculus with their pros and cons. The one we use is often called the *early* LTS. Statements are of the form $P \xrightarrow{\mu} P'$ meaning P evolves into P' by performing the action μ. Actions are defined as follows:

$$\mu ::= \tau \mid a\langle\widetilde{b}\rangle \mid (\boldsymbol{\nu}\widetilde{c})\overline{a}\langle\widetilde{b}\rangle$$

They represent an internal action (τ), the reception of names $\widetilde{b}$ at a ($a\langle\widetilde{b}\rangle$), or the sending of names $\widetilde{b}$ at a, with $\widetilde{c}$ being names that were private to the process. By opposition with the τ action, the others are called *visible* actions. As for prefixes, for these input and output actions, we say that a is the subject and $\widetilde{b}$ are the objects.

The LTS allows us to derive the transitions $\overline{a}\langle b\rangle.\, P + c(x).\, Q \xrightarrow{\overline{a}\langle b\rangle} P$ and $\overline{a}\langle b\rangle.\, P + c(x).\, Q \xrightarrow{c\langle d\rangle} Q\{{}^{d}/_{x}\}$, formalising the informal definition of the sum operator.

In $a\langle\widetilde{b}\rangle$, the notation uses square brackets as in the output to stress the fact that the names are already chosen, hence the name *early* for the LTS, and correspond to free names, while in $a(\widetilde{b}).\, P$ the names $\widetilde{b}$ are bound. The notion of free names is extended to actions as follows: $\mathrm{fn}(\tau) = \emptyset, \mathrm{fn}(a\langle\widetilde{b}\rangle) = \{a\} \cup \widetilde{b}$ and $\mathrm{fn}((\boldsymbol{\nu}\widetilde{c})\overline{a}\langle\widetilde{b}\rangle) = \{a\} \cup (\widetilde{b} \setminus \widetilde{c})$.

We will often write $\rightarrow$ for $\xrightarrow{\tau}$. We do not need to define a reduction-based semantics.

Our work focuses on weak equivalence, meaning we consider that internal transitions are not observable. This leads to the notion of weak transitions which correspond to the initial transitions up to internal actions. We define weak transitions as $\stackrel{\mu}{\Rightarrow} \stackrel{\text{def}}{=} \Rightarrow \xrightarrow{\mu} \Rightarrow$ where the weak reductions is defined as $\Rightarrow \stackrel{\text{def}}{=} \rightarrow^*$. This definition is unsatisfactory for $\stackrel{\tau}{\Rightarrow}$ which still forces at least one internal action. To ensure that internal transitions cannot be observed, we write $\stackrel{\widehat{\mu}}{\Rightarrow} \stackrel{\text{def}}{=} \Rightarrow$ if $\mu = \tau$ and $\stackrel{\mu}{\Rightarrow}$ otherwise. By opposition to weak transitions, standard transitions defined in Figure 2.2 are also called *strong*.

2.2 Bisimulations and up-to techniques

2.2.1 Bisimulations

Intuitively, two processes should be considered as equivalent when they have the same behaviour in every context. In the π-calculus, the reference behavioural equivalence is the context-closure of barbed bisim-

$$\textsc{Inp}\ \frac{}{a(\widetilde{b}).\,P \xrightarrow{a\langle\widetilde{c}\rangle} P\{\widetilde{c}/\widetilde{b}\}} \qquad \textsc{Out}\ \frac{}{\overline{a}\langle\widetilde{b}\rangle.\,P \xrightarrow{\overline{a}\langle\widetilde{b}\rangle} P} \qquad \textsc{Open}\ \frac{P \xrightarrow{(\boldsymbol{\nu}\widetilde{c})\overline{a}\langle\widetilde{b}\rangle} P'}{(\boldsymbol{\nu}d)P \xrightarrow{(\boldsymbol{\nu}d,\widetilde{c})\overline{a}\langle\widetilde{b}\rangle} P'}\text{if } d \in \mathrm{fn}((\boldsymbol{\nu}\widetilde{c})\overline{a}\langle\widetilde{b}\rangle) \setminus \{a\}$$

$$\textsc{Rep}\ \frac{a(\widetilde{b}).\,P \xrightarrow{\mu} P'}{!a(\widetilde{b}).\,P \xrightarrow{\mu} P' \mid !a(\widetilde{b}).\,P} \qquad \textsc{Res}\ \frac{P \xrightarrow{\mu} P'}{(\boldsymbol{\nu}a)P \xrightarrow{\mu} (\boldsymbol{\nu}a)P'}\text{if } a \notin \mathrm{fn}(\mu) \cup \mathrm{bn}(\mu) \qquad \textsc{Par}\ \frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}\text{if } \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$$

$$\textsc{Comm}\ \frac{P \xrightarrow{a\langle\widetilde{b}\rangle} P' \qquad Q \xrightarrow{(\boldsymbol{\nu}\widetilde{c})\overline{a}\langle\widetilde{b}\rangle} Q'}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{c})(P' \mid Q')}\text{if } \widetilde{c} \cap \mathrm{fn}(P) = \emptyset \qquad \textsc{Match}\ \frac{P \xrightarrow{\mu} P'}{[a=a]P \xrightarrow{\mu} P'} \qquad \textsc{Sum}\ \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'}$$

Figure 2.2: Early Labelled Transition Semantics for π-calculus

ulation. The definition of barbed bisimulation uses the reduction relation $\Rightarrow$ along with an observation predicate $\Downarrow_\alpha$ with $\alpha = a$ or $\overline{a}$ for any name a, which detects the possibility of performing an action along a.

Definition 1. We write $P \downarrow_a$ (resp. $P \downarrow_{\overline{a}}$) when there is an input (resp. output) μ with subject a such that $P \xrightarrow{\mu} P'$.

$P \Downarrow_a$ and $P \Downarrow_{\overline{a}}$ are defined similarly using $P \stackrel{\mu}{\Rightarrow} P'$.

We say that P has a (weak) *barb* at α when $P \downarrow_\alpha$ (resp. $P \Downarrow_\alpha$).

This notion can also be characterised without referring to the LTS using structural congruence.

Lemma 2. $P \downarrow_a$ iff $P \equiv (\boldsymbol{\nu}\widetilde{b})(a(\widetilde{c}).\,Q + G \mid R)$ for some $\widetilde{b}, \widetilde{c}, Q, G, R$ with $a \notin \widetilde{b}$.

$P \downarrow_{\overline{a}}$ iff $P \equiv (\boldsymbol{\nu}\widetilde{b})(\overline{a}\langle\widetilde{c}\rangle.\,Q + G \mid R)$ for some $\widetilde{b}, \widetilde{c}, Q, G, R$ with $a \notin \widetilde{b}$.

We prefer however the definition using transitions as this extends naturally to a typed setting.

Definition 3 (Barbed bisimulation)**.** A relation $\mathcal{R}$ on processes is a *barbed bisimulation* if whenever $P\ \mathcal{R}\ Q$:

1. $P \downarrow_\alpha$ implies $Q \Downarrow_\alpha$
2. $P \to P'$ implies that there is Q' such that $Q \Rightarrow Q'$ and $P'\ \mathcal{R}\ Q'$
3. and symmetrically for Q.

Barbed bisimilarity, $\dot{\approx}$, is the largest barbed bisimulation.

Clause 2. can be represented by the following diagram:

$$\begin{array}{ccc} P & \mathcal{R} & Q \\ \downarrow & & \Downarrow \\ P' & \mathcal{R} & Q' \end{array}$$

In this definition and the other bisimulations we shall introduce below, the conditions on P in Clause 1. and 2. are given using the strong version of transitions and barbs. This is done so as to simplify proofs of equivalence between processes. Modifying the two clauses with $P \Downarrow_\alpha$ and $P \Rightarrow P'$ does not affect the notion of barbed bisimulation and thus barbed bisimilarity.

Barbed bisimilarity is a rather coarse equivalence, equating for instance $\overline{a}.\,P$ and $\overline{a}.\,Q$ for all P, Q, so we close it with contexts. A relation $\mathcal{R}$ is a congruence when it is closed by context, i.e. for any P, Q, C, $P\ \mathcal{R}\ Q$ implies $C[P]\ \mathcal{R}\ C[Q]$.

Definition 4. *Barbed congruence*, noted $\simeq_c$, is the largest congruence included in barbed bisimilarity, i.e. $P \simeq_c Q$ iff for all C, $C[P] \dot{\approx} C[Q]$.

Barbed equivalence, noted $\simeq$, is defined similarly, by restricting the quantification over all static contexts E instead.

In this document, we focus on barbed equivalence (as opposed to barbed congruence) because it is simpler. Notably, we do not need to consider issues of closure of the labelled bisimulations under name substitutions in order .

A variant of barbed equivalence (resp. barbed congruence) is *reduction-closed barbed equivalence* (resp. reduction-closed barbed congruence), written $\cong$ (resp. $\cong_c$), which is the largest relation closed by all static contexts (resp. congruence) that is itself a barbed bisimulation [18]. By definition, reduction-closed barbed equivalence is included in barbed equivalence. These equivalences coincide on "usual processes", that is image-finite processes (Definition 9). In the case of the Asynchronous π-calculus detailed in Chapter 3, they coincide on all processes [12]. In this work, we will use both in different situations, choosing the one that is more suited. More precisely, reduction-closed barbed equivalence is used in Chapter 3, while barbed equivalence is used in Chapters 4 and 5. The former usually leads to simpler completeness proof, however in Chapters 4 and 5, we would need an equivalent of Lemma 8 which cannot be typed and so we choose the latter equivalence.

Due to the universal quantification over all contexts, proving that two processes are barbed equivalent is often difficult. A solution is to use another equivalence, easier to prove, that equates the same processes. We use a coinductive technique called bisimulation to do so. As we presented the early LTS, it is sometimes called *early bisimulation*.

Definition 5 (Bisimulation)**.** A relation $\mathcal{R}$ on processes is a *bisimulation* if whenever $P \mathcal{R} Q$ and $P \xrightarrow{\mu} P'$, there is Q' such that $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $P' \mathcal{R} Q'$; and symmetrically on the transitions from Q. *Bisimilarity*, noted $\approx$, is the largest bisimulation.

Bisimilarity (and similarly for all the other bisimilarities we define) is well-defined as the union of all bisimulations is also a bisimulation. As the bisimilarity contains all bisimulations, to prove that two processes are bisimilar, we simply need to provide a relation $\mathcal{R}$ and show that it is a bisimulation. This is the case for structural congruence.

Lemma 6. If $P \equiv Q$, then $P \approx Q$.

We can prove this result by showing that $\equiv$ is a bisimulation. This means proving that if $P \equiv Q$ and $P \xrightarrow{\mu} P'$, then there exists Q' such that $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $P' \equiv Q'$. As structural congruence is symmetric, the symmetrical conditions on Q are immediate. In fact, it is possible to prove a stronger result and always take Q' such that $Q \xrightarrow{\mu} Q'$.

Intuitively, a reduction involving a process and a context is similar to a process performing the action leading to that reduction. This is the (intuitive) reason why bisimilarity coincides with barbed equivalence for most processes.

To prove that bisimilarity is included in barbed equivalence, we first observe that bisimilarity is included in barbed bisimilarity and then show that bisimilarity is a congruence w.r.t static contexts.

Lemma 7. If $P \approx Q$, then $E[P] \approx E[Q]$.

Proof. For this, we prove that the relation $\mathcal{R}$ defined by $\{(E[P], E[Q]) \mid \forall E, P, Q, P \approx Q\}$ is a bisimulation.

Take some processes P, Q with $P \approx Q$ and some static context $E = (\boldsymbol{\nu} \widetilde{b})([\cdot] \mid T)$. Whenever $E[P] \xrightarrow{\mu} R$, using the rules of Figure 5.1, we have multiple cases according to whether rule Par or rule Comm (or their symmetric versions) is used.

We present only the case where the rule Comm is used. This means $\mu = \tau$ and $P \xrightarrow{a\langle \widetilde{b'} \rangle} P'$. Thus, there exists T' such that $T \xrightarrow{(\boldsymbol{\nu} \widetilde{c}) \overline{a} \langle \widetilde{b'} \rangle} T'$, and $E[P] \xrightarrow{\tau} E'[P']$ with $E' = (\boldsymbol{\nu} \widetilde{b}, \widetilde{c})([\cdot] \mid T')$. As $P \approx Q$, we know that $Q \stackrel{a\langle \widetilde{b'} \rangle}{\Longrightarrow} Q'$ and $P' \approx Q'$. We decompose $Q \stackrel{a\langle \widetilde{b'} \rangle}{\Longrightarrow} Q'$ as $Q \Rightarrow Q_0 \xrightarrow{a\langle \widetilde{b'} \rangle} Q_1 \Rightarrow Q'$. By applying rule Par and rule Res, we have that $E[Q] \Rightarrow E[Q_0]$ and $E'[Q_1] \Rightarrow E'[Q']$. Similarly, using rule Comm and rule Res, we have $E[Q_0] \xrightarrow{\tau} E'[Q_1]$. All in all, $E[Q] \Rightarrow E'[Q']$ and $E'[P'] \mathcal{R} E'[Q']$.

□

These 2 lemmas ensure that bisimilarity is included in reduction-closed barbed equivalence and thus also included in barbed equivalence, i.e. $\approx \subseteq \cong \subseteq \simeq$. That barbed equivalence implies bisimilarity is proved in a different way, depending on the variant of barbed equivalence we work with.

For reduction-closed barbed equivalence, it is required to show that barbed equivalence is also a bisimulation. The proof relies on the lemma below to handle bound names extruded via an output:

Lemma 8 ([12])**.** If $(\boldsymbol{\nu} a)(P \mid \overline{s}\langle a\rangle) \cong (\boldsymbol{\nu} a)(Q \mid \overline{s}\langle a\rangle)$ for s fresh, then $P \cong Q$.

On the other hand, barbed equivalence in the plain (untyped) π can be proved to coincide with early bisimilarity on image-finite processes (to which most of the processes one would like to write belongs), exploiting the n-approximants of bisimilarity (Theorem 12‘).

Definition 9. The class of *image-finite processes* is the largest subset $\mathcal{I}$ of processes such that $P \in \mathcal{I}$ implies that, for all μ, the set $\{P' \mid P \stackrel{\mu}{\Rightarrow} P'\}$, quotiented by alpha-conversion, is finite, and that is closed by transition, i.e. $P \in \mathcal{I}$ and $P \stackrel{\mu}{\to} P'$ implies that $P' \in \mathcal{I}$.

Definition 10 (Approximants of bisimilarity)**.** We define a sequence $(\approx^n)_{n\geq 0}$ of relations:

1. $P \approx^0 Q$ for all P, Q.
2. For all n, the relation $\approx^{n+1}$ is defined by: $P \approx^{n+1} Q$ if whenever $P \stackrel{\mu}{\to} P'$ then there is Q' such that $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $P' \approx^n Q'$; and symmetrically on the transitions from Q.

Then $P \approx^\omega Q$ if $P \approx^n Q$ for all n.

Notice that $\approx^0 \supseteq \approx^1 \supseteq \cdots \supseteq \approx^n \supseteq \cdots \supseteq \approx^\omega \supseteq \approx$. On image-finite processes, the last inclusion is an equality:

Lemma 11 ([52, Section 2.4])**.** If P, Q are image-finite, then $P \approx^\omega Q$ iff $P \approx Q$.

The proof that $\simeq$ is included in $\approx$ is usually carried by proving the contrapositive. This means that for all n, if $P \not\approx^n Q$ then there exists a context E such that $E[P] \not\approx E[Q]$. This context is constructed by induction on n. If $P \not\approx^{n+1} Q$, then there is some μ, P' such that $P \stackrel{\mu}{\to} P'$ and for all $Q \stackrel{\mu}{\Rightarrow} Q'$, $P \not\approx^n Q$. Thus, intuitively, we take a context of the form $[\cdot] \mid (\bar{\mu}.\sum_i \tau.(R_i + \overline{z_i})) + \overline{z}$. The outputs at $z, \widetilde{z_i}$ are used as barbs to monitor the interactions of this process, and the R_i correspond to the contexts obtained by induction, one for each Q' such that $Q \stackrel{\mu}{\Rightarrow} Q'$ (there is a finite number of such Q' by image-finiteness).

Theorem 12 ([52, Section 2.4])**.** On image-finite processes, relations $\simeq$ and $\approx$ coincide.

There is an asymmetric variant of bisimilarity, called *expansion* where one process is allowed more internal actions than the other. Its main use is for up-to techniques (see Section 2.2.2). For this purpose, we write $\stackrel{\widehat{\tau}}{\to}$ for $\stackrel{\tau}{\to} \cup =$ and $\stackrel{\widehat{\mu}}{\to}$ for $\stackrel{\mu}{\to}$ otherwise. The transition $\stackrel{\widehat{\tau}}{\to}$ allows for at most one τ instead of exactly one.

Definition 13 (Expansion)**.** We write $\gtrsim$ for the largest relation such that whenever $P \gtrsim Q$:

- if $P \stackrel{\mu}{\to} P'$, then there is Q' such that $Q \stackrel{\widehat{\mu}}{\to} Q'$ and $P' \gtrsim Q'$;
- if $Q \stackrel{\mu}{\to} Q'$, then there is P' such that $P \stackrel{\mu}{\Rightarrow} P'$ and $P' \gtrsim Q'$.

Expansion still has some nice properties from bisimilarity, like containing structural congruence and being closed by static contexts.

2.2.2 Up-to techniques

Up-to techniques are proof techniques for bisimulation developed to reduce the size of the relation used to reason about equivalence between processes [33].

The idea of up-to technique is to replace the condition $P' \ \mathcal{R} \ Q'$ in Definition 5 by $P' \ f(\mathcal{R}) \ Q'$ where f is the up-to technique, leading to the following diagram:

$$\begin{array}{ccc} P & \mathcal{R} & Q \\ \downarrow{\scriptstyle \mu} & & \Downarrow{\scriptstyle \hat{\mu}} \\ P' & f(\mathcal{R}) & Q' \end{array}$$

This defines a *bisimulation up to f*.

Definition 14 (Bisimulation up-to)**.** A relation $\mathcal{R}$ on processes is a *bisimulation up to f* if whenever $P \ \mathcal{R} \ Q$ and $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $P' \ f(\mathcal{R}) \ Q'$; and symmetrically on the transitions from Q.

These functions help to give smaller relations as bisimulation candidates. For instance, given a process P, to prove that $\tau.P \approx P$ using a bisimulation, one can build a relation containing $(\tau.P, P)$ but also all pairs (P', P') with P' obtained from P after any sequence of transitions. Having to include all such pairs in any candidate relation increases their size. Defining `refl` as $\texttt{refl}(\mathcal{R}) = \{(P, P) \text{ for all } P\}$, we have that $\{(\tau.P, P)\}$ is a bisimulation up to `refl`:

$$\begin{array}{ccc} \tau.P & \mathcal{R} & P \\ \downarrow{\scriptstyle \tau} & & \Downarrow \\ P & \texttt{refl}(\mathcal{R}) & P \end{array} \qquad\qquad \begin{array}{ccc} \tau.P & \mathcal{R} & P \\ \Downarrow{\scriptstyle \mu} & & \downarrow{\scriptstyle \mu} \\ P' & \texttt{refl}(\mathcal{R}) & P' \end{array}$$

Not every function can be used. We say that an up-to technique or a function f is *sound*, if any bisimulation up to f is included in the bisimilarity. The reasoning above for $(\tau.P, P)$ is valid because `refl` is sound.

A significant class of sound up-to techniques is the class of *compatible* functions. For this we rely on the notion of progress.

Definition 15 (Progress)**.** A relation $\mathcal{R}$ progresses to a relation S, noted $\mathcal{R} \rightarrowtail S$, if whenever $P \ \mathcal{R} \ Q$ and $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $P' \ S \ Q'$; and symmetrically on the transitions from Q.

Thus, a bisimulation is simply a relation $\mathcal{R}$ that progresses to itself, i.e. $\mathcal{R} \rightarrowtail \mathcal{R}$. Similarly, a bisimulation up-to f is a relation $\mathcal{R}$ such that $\mathcal{R} \rightarrowtail f(\mathcal{R})$. We have $\mathcal{R} \rightarrowtail S$ when we have the following diagram (and the symmetric one) for all $P \ \mathcal{R} \ Q$:

$$\begin{array}{ccc} P & \mathcal{R} & Q \\ \downarrow{\scriptstyle \mu} & & \Downarrow{\scriptstyle \hat{\mu}} \\ P' & S & Q' \end{array}$$

Definition 16 (Compatible function)**.** A function f is *compatible* if f is monotone, i.e. if $\mathcal{R} \subseteq S$ implies $f(\mathcal{R}) \subseteq f(S)$, and $\mathcal{R} \rightarrowtail S$ implies $f(\mathcal{R}) \rightarrowtail f(S)$.

We write $f \cup g$ for the component-wise union of functions, i.e. $(f \cup g)(x) = f(x) \cup g(x)$.

Proposition 17. If f and g are compatible, then $f \cup g$ and $f \circ g$ are compatible too.

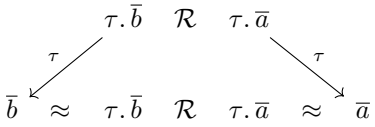
If f is compatible, then f is a sound up-to technique.

Most standard up-to techniques are compatible. We give some examples of such techniques. `refl`, `ctxt`, $\mathcal{F}_{\gtrsim}$ are some up-to techniques called up-to identity, up-to context and up-to expansion respectively.

$$id(\mathcal{R}) \stackrel{\text{def}}{=} \mathcal{R} \qquad \texttt{refl}(\mathcal{R}) \stackrel{\text{def}}{=} \{(P, P) \mid \forall P\} \qquad \texttt{ctxt}(\mathcal{R}) \stackrel{\text{def}}{=} \{(E[P], E[Q]) \mid \forall E, P, Q \text{ s.t. } P \ \mathcal{R} \ Q\}$$

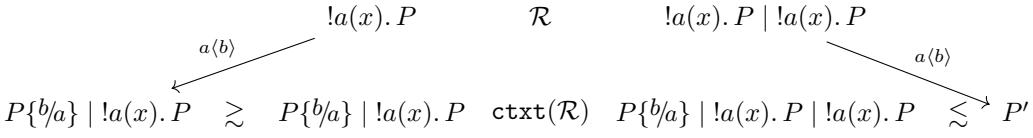
$$\mathcal{F}_{\gtrsim}(\mathcal{R}) \stackrel{\text{def}}{=} \gtrsim \mathcal{R} \lesssim$$

Unfortunately, the function $\mathcal{F}_{\approx}$, with $\mathcal{F}_{\approx}(\mathcal{R}) \stackrel{\text{def}}{=} \approx\mathcal{R}\approx$ is not sound [51]. Take $\mathcal{R} = \{(\tau.\,\overline{b}, \tau.\,\overline{a})\}$. We have $\mathcal{R} \rightarrowtail \mathcal{F}_{\approx}(\mathcal{R})$. Indeed, we have the following diagram:



However, we have $\tau.\,\overline{a} \not\approx \tau.\,\overline{b}$. The asymmetry of the expansion ensures that processes in $\mathcal{F}_{\gtrsim}(\mathcal{R})$ are "slower" than those in $\mathcal{R}$, preventing such unsound reasoning.

Example 18. We prove that $!a(x).\,P \approx !a(x).\,P \mid !a(x).\,P$. Without up-to techniques, a bisimulation containing these two processes can be of the form $\{(!a(x).\,P \mid Q,\, !a(x).\,P \mid !a(x).\,P \mid Q) \mid \forall Q\}$. Using up-to context and up-to expansion, we can reduce the relation to a single pair, namely $(!a(x).\,P,\, !a(x).\,P \mid !a(x).\,P)$. The process $!a(x).\,P$ may only do an input transition, $!a(x).\,P \xrightarrow{a\langle b\rangle} P\{b/a\} \mid !a(x).\,P$. On the other hand, the right process can do the input using any of the two replications. However, in both cases, we have $!a(x).\,P \mid !a(x).\,P \xrightarrow{a\langle b\rangle} \equiv P\{b/a\} \mid !a(x).\,P \mid !a(x).\,P$. As $\equiv$ is included in $\gtrsim$, we end up with the following diagram:



By taking $E = P\{b/a\} \mid [\cdot]$, we indeed have that $P\{b/a\} \mid !a(x).\,P \;\mathtt{ctxt}(\mathcal{R})\; P\{b/a\} \mid !a(x).\,P \mid !a(x).\,P$.

We study in more details the up-to techniques in the π-calculus and some congruence properties in the note [43] which is not presented in this thesis.

2.3 Typing and sorting

The calculi in this work will be typed. For simplicity we define our type systems as refinements of the most basic type system for π-calculus, namely Milner's *sorting* [32], in which names are partitioned into a collection of *types* (or sorts), and a sorting function maps types onto a list of types. If a name type S is mapped onto the types $\widetilde{T}$, this means that names in S when used as channels may only carry names in $\widetilde{T}$.

This type system prevents us from writing ill formed processes, like $\overline{a}\langle b\rangle \mid a(x, y)$, where the arity of the names does not match, even after performing some communications. For instance, while in $a(x).\,\overline{x}\langle b\rangle \mid \overline{a}\langle c\rangle.\,c(y, z)$, we have a, x that carry one name, and c that carries two, as both x and c are used as names that can be sent/received via a, this leads to an ill-typed process: $a(x).\,\overline{x}\langle b\rangle \mid \overline{a}\langle c\rangle.\,c(y, z) \rightarrow \overline{c}\langle b\rangle \mid c(y, z)$. Using a sorting, x and c should have the same sort, thus solving this issue.

We assume that there is a sorting system under which all processes we manipulate are well-typed. For more complex type systems, we always write $\Delta \vdash P$ when process P is well-typed under the typing Δ, and similarly for other objects, such as contexts. In this work, the type environment Δ is a simple structure, e.g. an integer or a set of names.

Notations. We will often use different names to distinguish between sorts. $a, b, c, \ldots$ always range over "the most generic names" in a sense that will be specified in each section. In Chapter 3, $\ell, \ell', \ldots$ range over names representing references. Similarly, in Chapter 5, $p, q, r, \ldots$ range over names following a strict discipline and are called *continuation* names.

Since we work in a typed setting, equivalences must be adapted too. For instance, the continuation names in Chapter 5 are *linear*, meaning they can only appear once in input and once in output. Under such constraints, there exists no typing Δ such that $\Delta \vdash \overline{p} \mid p \mid p$ as p is used twice as input. In that context, transitions must take into account the constraints imposed by typing. For instance, for continuation names, the transition $\overline{p} \mid p \xrightarrow{\overline{p}}$ should not be allowed as it cannot correspond to an interaction

with another process: the context should be able to perform an input at p, while the process $\overline{p} \mid p$ already as the unique input capability.

Much like the typing of the tested processes affects which transitions can be observed, this also affects the definition of barbs.

Definition 19. We write $\Delta \vDash P \Downarrow_\alpha$ if Δ is a typing for P (i.e. $\Delta \vdash P$ holds), there is an output (resp. input) action μ with subject a if $\alpha = \overline{a}$ (resp. $\alpha = a$) s.t. $P \stackrel{\mu}{\Rightarrow} P'$, and such a transition is observable under the typing Δ.

$\Delta \vDash P \downarrow_\alpha$ is defined similarly.

The meaning of 'observable under a typing' will depend on the specific type system adopted; in the case of the plain sorting, all transitions are observable.

Having typed processes, in the definition of barbed equivalence we may only test processes with contexts that respect the typing of the processes.

Definition 20. C is a Γ/Δ *context* if $\Gamma \vdash C$ can be derived, using the type system for the processes and the extra rule $\dfrac{}{\Delta \vdash [\cdot]}\ \textsc{Hole}$ for the hole.

Similarly, P is a Δ-*process* if $\Delta \vdash P$. We also write $\Delta \vdash P, Q$ when both P and Q are Δ-processes. The type systems we describe have the additional property that typing is invariant under reduction, i.e. if $\Delta \vdash P$ and $P \to P'$ then $\Delta \vdash P'$. This property allows us to define a barbed bisimulation for each type environment:

Definition 21 (Typed barbed bisimulation, equivalence, and congruence)**.** *Barbed* Δ*-bisimilarity* is the largest relation $\dot{\approx}^\Delta$ on Δ-processes s.t. $P \dot{\approx}^\Delta Q$ implies:

1. $\Delta \vDash P \downarrow_\alpha$ implies $\Delta \vDash Q \Downarrow_\alpha$
2. $P \to P'$ implies that there exists Q' such that $Q \Rightarrow Q'$ and $P' \dot{\approx}^\Delta Q'$
3. and symmetrically for Q.

Two Δ-processes P and Q are *barbed equivalent at* Δ, written $P \simeq^\Delta Q$, if for each Γ/Δ static context E it holds that $E[P] \dot{\approx}^\Gamma E[Q]$. *Barbed congruence at* Δ, $\simeq_c^\Delta$, is defined in the same way but employing all Γ/Δ contexts (rather than only the static ones).

For reduction-closed barbed equivalence (resp. reduction-closed barbed congruence), we must include the typing in the relation, i.e. the relation is ternary instead of binary. A *typed process relation* is a set of triplets (Δ, P, Q) with $\Delta \vdash P, Q$. We write $\Delta \vDash P \mathcal{R} Q$ when $(\Delta, P, Q) \in \mathcal{R}$.

Another necessary condition for observable transitions is to enable the resulting process to be typed. However, the typing may differ from the original process when doing transitions other than reductions. In this work, we will write $[\Delta; P] \xrightarrow{\mu} [\Delta'; P']$ when $\Delta \vdash P$ holds, $P \xrightarrow{\mu} P'$ is a transition observable under the typing Δ and $\Delta' \vdash P'$ holds. With this notation, we can adapt bisimulation to a typed setting.

Definition 22 (Typed bisimulation)**.** A typed process relation $\mathcal{R}$ is a *typed bisimulation* if whenever $\Delta \vDash P \mathcal{R} Q$ and $[\Delta; P] \xrightarrow{\mu} [\Delta'; P']$, then there is Q' such that $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $\Delta' \vDash P' \mathcal{R} Q'$; and symmetrically on the transitions from Q.

For the type systems we present, when we write $[\Delta; P] \xrightarrow{\mu} [\Delta'; P']$, the type Δ' is unique given Δ and μ. It is also ensured that any process with type Δ performing action μ can be typed with Δ', thus in proofs of typed bisimulation, we do not need to check that $\Delta' \vdash Q'$.

We use various symbols $\approx_{\mathrm{r}}, \approx_{\mathrm{s}}, \ldots$ to denote the different typed bisimilarities we use throughout this work. For a typed bisimilarity $\mathcal{R}$, we often write $P \, \mathcal{R}^\Delta \, Q$ when $(\Delta, P, Q) \in \mathcal{R}$ by analogy with binary relations such as barbed equivalence at Δ. Figure 2.3 collects the notions of behavioural equivalence in the π-calculus that are introduced in this document.

To write examples, we sometimes use basic data values such as integers and booleans. When doing this, we have to adapt the sorting but this does not raise any additional difficulty.

Untyped π-calculus (Chapter 2):

$\xrightarrow{\mu}, \stackrel{\mu}{\Rightarrow}, \xrightarrow{\widehat{\mu}}, \stackrel{\widehat{\mu}}{\Rightarrow}$	Strong and weak transitions
$P \downarrow_\alpha, P \Downarrow_\alpha$	Strong and weak barb
$\equiv$	Structural congruence
$\stackrel{\cdot}{\approx}$	Barbed bisimilarity
$\simeq$	Barbed equivalence
$\simeq_c$	Barbed congruence
$\cong$	Reduction-based barbed equivalence
$\cong_c$	Reduction-based barbed congruence
$\approx$	Bisimilarity
$\gtrsim$	Expansion
$\Delta \vdash P$	Typing, generic version
$[\Delta; P] \xrightarrow{\mu} [\Delta'; P']$	Typed-allowed transition, generic version

Asynchronous π-calculus and references (Chapter 3):

$\ell \lhd n, \ell \rhd (n), \ell \bowtie m(n)$	Imperative operations (write, read, swap)
$\Delta \vdash_{\mathrm{r}} P$	Type system for reference names
$\simeq^a$	Asynchronous barbed equivalence
$\approx_a$	Asynchronous bisimilarity
$\cong^{Arn}$	Reduction-closed barbed equivalence with reference names
$\approx_{\mathrm{r}}$	Reference bisimilarity
$\approx_{\mathrm{ip}}$	Bisimilarity with inductive predicate
$\cong^{ref}$	Reduction-closed barbed equivalence in $\pi^{\mathtt{ref}}$

Sequentiality (Chapter 4):

$\eta \vdash_{\mathrm{s}} P$	Type system for sequentiality
$\simeq^\eta$	Sequential barbed equivalence at η
$\approx_{\mathrm{s}}^\eta$	Sequential bisimilarity at η
$\Delta; \eta \vdash_{\mathrm{rs}} P$	Type system for sequential references
$\approx_{\mathrm{rs}}^{\Delta;\eta}$	Sequential bisimilarity with inductive predicate

Well-bracketing (Chapter 5):

$\rho \vdash_{\mathrm{wb}} P$, $\rho \models_{\mathrm{wb}} P$	Type system for well-bracketing without/with clean stacks
$\simeq^\sigma$	Well-bracketed barbed equivalence
$\approx_{\mathrm{wb}}^\sigma$	Wb-bisimilarity
$\approx_{\mathrm{div}}$	Bisimilarity with divergence (Chapter 6)

Figure 2.3: List of symbols

Chapter 3

References in the π-calculus

To describe how references operate in the π-calculus, we first need to introduce the Asynchronous π-calculus.

3.1 Asynchronous π-calculus

An important subcalculus of π is the Asynchronous π-calculus, Aπ [4]. It is characterised by forcing $\mathbf{0}$ as the only continuation for an output (which we do not write). Outputs no longer guard processes so this change is made syntactically explicit in the grammar:

$$\begin{array}{rcl} P, Q & ::= & \overline{a}\langle\widetilde{b}\rangle \mid P \mid Q \mid (\boldsymbol{\nu} a)P \mid !a(\widetilde{b}).\, P \mid G \\ G, G' & ::= & \mathbf{0} \mid a(\widetilde{b}).\, P \mid \tau.\, P \mid [a = b]G \mid G + G' \end{array}$$

This has the consequence that outputs cannot be used immediately in a sum or after a matching. In that context, an output is implicitly sent as soon as it appears unguarded, but stays until it is received.

The behavioural equivalence is adapted in consequence. As outputs do not guard processes, it intuitively implies that a process cannot detect if its outputs have been used. Thus an environment cannot directly detect whether or not an input has been made. To express this distinction in Aπ, barbs are restricted to outputs only (thus of the form $P \downarrow_{\overline{a}}$). Barbed equivalence (resp. congruence), noted $\simeq^a$ (resp. $\simeq^a_c$), is defined as usual using static contexts (resp. all contexts) from Aπ.

One critical example of the impact of this change is the law $\mathbf{0} \simeq^a a(x).\, \overline{a}\langle x\rangle$. In standard, synchronous, π-calculus, even without observing the barb at a, we could distinguish the two by using the context $[\cdot] \mid \overline{a}\langle b\rangle.\, \overline{c}$. The output at c becomes observable only if a communication is done at a. This, however, cannot be done using an asynchronous context, so as the output at a is resent as soon as it is used in the communication, it is not distinguishable from an internal action. However, synchronous bisimilarity can distinguish between the two by looking at the input transition. *Asynchronous bisimulation* was designed to recover this equivalence.

Definition 23 (Asynchronous Bisimulation)**.** A relation $\mathcal{R}$ on processes is an *asynchronous bisimulation* if whenever $P\ \mathcal{R}\ Q$ and $P \xrightarrow{\mu} P'$, then one of these two clauses holds:

1. there is Q' such that $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $P'\ \mathcal{R}\ Q'$;
2. $\mu = a\langle\widetilde{b}\rangle$ and there is Q' such that $Q \mid \overline{a}\langle\widetilde{b}\rangle \Rightarrow Q'$ and $P'\ \mathcal{R}\ Q'$.

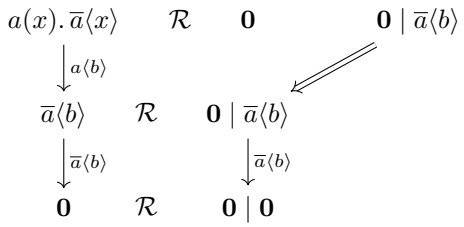
and symmetrically on the transitions from Q.

Asynchronous bisimilarity, $\approx_a$, is the largest bisimulation.

The main distinction is the second clause for input transitions, in which Q is not forced to perform an input, but does internal actions using the output that should have been used.

Example 24. $\mathbf{0} \approx_a a(x).\, \overline{a}\langle x\rangle$.

Proof. Take $\mathcal{R} \stackrel{\text{def}}{=} \{(\mathbf{0}, a(x).\,\overline{a}\langle x\rangle), (\mathbf{0} \mid \overline{a}\langle x\rangle, \overline{a}\langle x\rangle), (\mathbf{0} \mid \mathbf{0}, \mathbf{0})\}$. We have that $\mathcal{R}$ is an asynchronous bisimulation.



□

The equivalent of Theorem 12 still holds in that setting:

Theorem 25 ([4])**.** On image-finite asynchronous processes, relations $\simeq^a$ and $\approx_a$ coincide.

3.2 Reference names in Aπ

The π-calculus has been advocated as a model to interpret, and give semantics to, languages with higher-order features. Often these languages make use of forms of references (and hence viewing a store as set of references). This therefore requires representations of references using the names of the π-calculus. There are strong similarities between the names of the π-calculus and the references of imperative languages. This is evident in the denotational semantics of these languages: the mathematical techniques employed in modelling the π-calculus (e.g. [53, 11]) were originally developed for the semantic description of references. Yet names and references behave rather differently: receiving from a name is destructive — it consumes a value — whereas reading from a reference is not; a reference has a unique location, whereas a name may be used by several processes both in input and in output; etc. These differences make it unclear if and how interesting properties of imperative languages can be proved via a translation into the π-calculus.

Previous implementations of references in the π-calculus model a reference cell as a server accepting requests on two names, representing reading and writing respectively [44]. In comparison, the Asynchronous π-calculus allows one to provide a simpler representation of references, where a reference ℓ storing a value n is just an output message $\overline{\ell}\langle n\rangle$. A process that wishes to access the reference is supposed to make an input at ℓ and then immediately emit a message at ℓ with the new content of the reference. For instance a process reading on the reference and binding its content to x in the continuation P is

$$\ell(x).\,(\overline{\ell}\langle x\rangle \mid P) \ .$$

This representation of references in Aπ is interesting because of the *asynchronous bisimilarity*. To see why the input clause could be interesting with references, consider a process that performs a *useless read* on a reference ℓ and then continues as some process P_2; in a language with references this would be equivalent to P_2 itself. When written in Aπ, the process with the useless read becomes $P_1 \stackrel{\text{def}}{=} \ell(x).\,(\overline{\ell}\langle x\rangle \mid P_2)$ where x does not appear in P_2. In ordinary bisimilarity, P_1 is immediately distinguished from P_2, as the latter cannot answer the input transition $P_1 \xrightarrow{\ell\langle n\rangle} \overline{\ell}\langle n\rangle \mid P_2$. However, the answer is possible using the input clause, as we have $\overline{\ell}\langle n\rangle \mid P_2 \Rightarrow \overline{\ell}\langle n\rangle \mid P_2$. In fact, using reference bisimilarity, we are able to prove the following law:

$$\ell(x).\,(\overline{\ell}\langle x\rangle \mid P) \ \approx_{\mathrm{r}} \ P \quad x \notin \mathrm{fn}(P) \tag{3.1}$$

We are not aware of studies that investigate the faithfulness of the above representation of references in Aπ. To address this issue, we first define in Section 3.2 a type system in Aπ to capture the intended pattern of usage of names representing references, and we establish proof techniques in this typed calculus. We then consider in Section 3.3 an extension of Aπ with explicit references and operators to manipulate them: $\pi^{\mathtt{ref}}$. We can then show how reference names allow for a faithful representation of references by translating $\pi^{\mathtt{ref}}$ into Aπ and study this translation.

The type system in Aπ is designed to capture the intended pattern of usage of names that represent references, called *reference names*, and in particular the property that there is always a unique output message available at these names. The type system has linearity features similar to π-calculus type systems for locks [24] or for receptiveness [45].

The calculus with references, $\pi^{\mathtt{ref}}$, has constructs for reading from a reference, writing on a reference, and a swap operation for atomically reading on a reference and placing a new value onto it. Modern computer architectures offer hardware instructions similar to swap, e.g. test-and-set or control-and-swap constructs, to atomically check and modify the content of a register. These constructs are important to tame the access to shared resources. In distributed systems, swap can be used to solve the consensus problem with two parallel processes, whereas simple registers cannot [15].

The swap construct is also suggested by the translation of references into Aπ. The pattern for accessing a reference ℓ is $\ell(x).\,(\overline{\ell}\langle n\rangle \mid P)$. This yields four cases, depending on whether x is used in P and whether x is equal to n:

	$n \neq x$	$n = x$
x free in P	swap	read
x not free in P	write	useless read

We establish an operational correspondence between the behaviour of a process in $\pi^{\mathtt{ref}}$ and its encoding in Aπ, and from this we establish full abstraction of the translation of $\pi^{\mathtt{ref}}$ into Aπ with respect to barbed equivalence in the two calculi. We then investigate proof techniques for barbed equivalence in Aπ, based on two forms of labelled bisimilarities. For one bisimilarity we derive both soundness and completeness. This bisimilarity is similar to, but not the same as, asynchronous bisimilarity. For instance, it is defined on 'reference-closed' processes (intuitively, processes in which all references are allocated); therefore inputs on reference names from the tested processes are not visible (because such inputs are supposed to consume the unique output message at that reference that is present in the tested processes). The output clause of bisimilarity on reference names is also different, as we have to make sure that the observer respects the pattern of usage for reference names; thus the observer consuming the output message on a reference name ℓ should immediately re-install an output on ℓ.

The second bisimilarity is more efficient because it does not require processes to be 'reference-closed'. Thus output messages on reference names consumed by the observer need not be immediately re-installed. However sometimes access to a certain reference is needed by a process in order to answer the bisimulation challenge from the other process. And depending on the content of such references, further accesses to other references may be needed. Since we wish to add only the needed references, this introduces an inductive game, in which a player requires a reference and the other player specifies the content of such reference, within the coinductive game of bisimulation. We show that the resulting bisimilarity is sound, and leave completeness as an open problem. Finally, we discuss examples of uses of the bisimilarities.

3.2.1 Types and contextual equivalences with reference names

We use types to formalise the behavioural difference between reference names and plain names in Aπ. The types of the sorting impose a partition on the two sets of names (reference names and plain names). Thus we assume such a sorting, under which all processes are well-typed. We separate the base type system (Milner's sorting) from the typing rules for reference names so as to show the essence of the latter rules. Accordingly, we only present the additional typing constraints for reference names. For simplicity, we will consider the language to be monadic, i.e. names may carry only one name.

We write: `RefTypes` for the set of reference types (i.e. types that contain reference names); $\mathtt{Type}(n)$ is the type of name n; $\mathtt{ObType}(n)$ is the type of the objects of n (i.e. the type of the names that may be carried at n). For example in well-typed processes such as $\overline{n}\langle m\rangle$ and $n(m).\,P$, name m will be of type $\mathtt{ObType}(n)$.

Notations. *We use $\ell, \ldots$ to range over reference names, $a, b, \ldots$ over plain names, $n, m, \ldots$ over the set of all names. Δ ranges over finite sets of reference names. We sometimes write $\Delta - x$ as abbreviation for $\Delta - \{x\}$. Moreover $\Delta_1 \uplus \Delta_2$ is defined only when $\Delta_1 \cap \Delta_2 = \emptyset$, in which case it is $\Delta_1 \cup \Delta_2$; we write Δ, x for $\Delta \uplus \{x\}$.*

Compared to the notation from Chapter 2, $n, m, \ldots$ are used instead of $a, b, \ldots$ to denote all names. The type system is presented in Figure 3.1. Judgements have the form $\Delta \vdash_{\mathrm{r}} P$, where P is an Aπ process.

$$\textsc{ROutN}\ \frac{}{\emptyset \vdash_{\mathrm{r}} \overline{a}\langle m\rangle} \qquad \textsc{RInpN}\ \frac{\emptyset \vdash_{\mathrm{r}} P}{\emptyset \vdash_{\mathrm{r}} a(x).\,P} \qquad \textsc{RResN}\ \frac{\Delta \vdash_{\mathrm{r}} P}{\Delta \vdash_{\mathrm{r}} (\boldsymbol{\nu} a)P} \qquad \textsc{RPar}\ \frac{\Delta_1 \vdash_{\mathrm{r}} P \qquad \Delta_2 \vdash_{\mathrm{r}} Q}{\Delta_1 \uplus \Delta_2 \vdash_{\mathrm{r}} P \mid Q} \qquad \textsc{RRep}\ \frac{\emptyset \vdash_{\mathrm{r}} P}{\emptyset \vdash_{\mathrm{r}} !a(m).\,P}$$

$$\textsc{ROutR}\ \frac{}{\ell \vdash_{\mathrm{r}} \overline{\ell}\langle m\rangle} \qquad \textsc{RInpR}\ \frac{\ell \vdash_{\mathrm{r}} P}{\emptyset \vdash_{\mathrm{r}} \ell(x).\,P} \qquad \textsc{RResR}\ \frac{\Delta, \ell \vdash_{\mathrm{r}} P}{\Delta \vdash_{\mathrm{r}} (\boldsymbol{\nu}\ell)P} \qquad \textsc{RSum}\ \frac{\emptyset \vdash_{\mathrm{r}} G_1 \qquad \emptyset \vdash_{\mathrm{r}} G_2}{\emptyset \vdash_{\mathrm{r}} G_1 + G_2} \qquad \textsc{RNil}\ \frac{}{\emptyset \vdash_{\mathrm{r}} \mathbf{0}} \qquad \textsc{RMatch}\ \frac{\emptyset \vdash_{\mathrm{r}} G}{\emptyset \vdash_{\mathrm{r}} [a = b]G}$$

$$\textsc{RTau}\ \frac{\emptyset \vdash_{\mathrm{r}} P}{\emptyset \vdash_{\mathrm{r}} \tau.\,P}$$

Figure 3.1: Typing conditions for reference names in Aπ processes

Rule ROutR along with Rule RPar ensures that every reference name in Δ appears in the subject of exactly one unguarded output. Rule RResR ensures that new reference names are always in Δ while Rule RInpR ensures that Δ is constant after a communication at a reference name (by re-emitting an output after one has been consumed).

Intuitively, if $\Delta \vdash_{\mathrm{r}} P$, then P must make available the names in Δ *immediately* and *exactly once* in output subject position. We say that ℓ is *output receptive* in P if there is exactly one unguarded output at ℓ. Then $\Delta \vdash P$ holds if

– any $\ell \in \Delta$ is output receptive in P;

– in any subterm of P of the form $(\boldsymbol{\nu}\ell')Q$ or $\ell'(m).\,Q$, name ℓ' is output receptive in Q.

This intuition is formalised in Lemma 26, and in Proposition 28 that relates types and operational semantics.

Typing is important because it allows us to derive the required behavioural equivalences. For instance, allowing parallel composition with the ill-typed process $\ell(x).\,\mathbf{0}$ would invalidate the barbed equivalence between the terms in law (3.1).

In the remainder, it is assumed that all processes are *well typed*, meaning that each process P obeys the underlying sorting system and that there is Δ s.t. $\Delta \vdash_{\mathrm{r}} P$ holds. Two processes P, Q are *type-compatible* if both $\Delta \vdash_{\mathrm{r}} P$ and $\Delta \vdash_{\mathrm{r}} Q$, for some Δ; we write $\Delta \vdash_{\mathrm{r}} P, Q$ in this case. *In the remainder, all relations are on pairs of type-compatible processes. Similarly, all compositions (i.e. of a context with processes) and actions are well-typed.*

The type system satisfies standard properties, like uniqueness of typing ($\Delta \vdash P$ and $\Delta' \vdash_{\mathrm{r}} P$ imply $\Delta = \Delta'$), and preservation by structural congruence ($P \equiv Q$ and $\Delta \vdash_{\mathrm{r}} P$ imply $\Delta \vdash Q$). As claimed above, if $\Delta \vdash P$, then names in Δ are output receptive:

Lemma 26. If $\Delta, \ell \vdash P$ then $P \equiv (\boldsymbol{\nu}\widetilde{n})(\overline{\ell}\langle m\rangle \mid Q)$, with $\ell \notin \widetilde{n}$, and there is no unguarded output at ℓ in Q.

We can now define the type-allowed transitions on top of the standard ones (Figure 2.2).

Definition 27 (Type-allowed transitions)**.** When $\Delta \vdash_{\mathrm{r}} P$, we write $[\Delta; P] \xrightarrow{\mu} [\Delta'; P']$ if $P \xrightarrow{\mu} P'$ and one of the following holds:

1. $\mu \in \{(\boldsymbol{\nu}\ell)\overline{a}\langle\ell\rangle, \ell\langle m\rangle\}$ and $\Delta' = \Delta, \ell$
2. $\mu = \overline{\ell}\langle m\rangle$ or $\mu = (\boldsymbol{\nu} b)\overline{\ell}\langle b\rangle$ and $\Delta', \ell = \Delta$
3. $\mu = (\boldsymbol{\nu}\ell')\overline{\ell}\langle\ell'\rangle$ and $\Delta, \ell' = \Delta', \ell$
4. $\mu \in \{\tau, \overline{a}\langle m\rangle, a\langle m\rangle, (\boldsymbol{\nu} b)\overline{a}\langle b\rangle\}$ and $\Delta = \Delta'$.

We can remark that in the case where $\mu = \ell\langle m\rangle$, we must have $\ell \notin \Delta$, as otherwise the context would not be able to trigger an input (since, by typing, it could not generate an output on ℓ).

Proposition 28 (Subject reduction)**.** If $\Delta \vdash_{\mathrm{r}} P$ and $[\Delta; P] \xrightarrow{\mu} [\Delta'; P']$, then $\Delta' \vdash_{\mathrm{r}} P'$.

Additionally, if $[\Delta; P] \xrightarrow{\ell\langle m\rangle} [\Delta'; P']$ then we have $\Delta' \vdash_{\mathrm{r}} P \mid \overline{\ell}\langle m\rangle$. This shows that the input clause found in asynchronous bisimulation is consistent regarding the typing of processes.

3.2.2 Behavioural equivalences with reference names

As usual in typed calculi, the definitions of the barbed relations take typing into account, so that the composition of a context and a process be well-typed. In the case of reference names, an additional ingredient has to be taken into account, namely the accessibility of reference names. If a process has the possibility of accessing a reference, then a context in which the process is tested should guarantee the availability of that reference. For this, we define the notion of *completing context* and *complete* process. Then, roughly, barbed congruence becomes "barbed congruence under all completing contexts".

A process P is *complete* if each reference name that appears free in P is 'allocated' in P. We write $\mathrm{fn}_{\mathrm{r}}(P)$ for the set of free reference names in P.

Definition 29 (Open references and complete processes)**.** The *open references* of P such that $\Delta \vdash P$ are the names in $\mathrm{fn}_{\mathrm{r}}(P) \setminus \Delta$; similarly the open references of processes $P_1, \ldots, P_n$ is the union of the open references of the P_i's. P is *complete* if it contains no open reference: i.e. $\mathrm{fn}_{\mathrm{r}}(P) \subseteq \Delta$ and $\Delta \vdash_{\mathrm{r}} P$, for some Δ.

A context C is *completing for* P if $C[P]$ is complete.

Note that an Aπ *complete process* might have free reference names, if these are not open references;

Lemma 30. P is complete iff $\emptyset \vdash_{\mathrm{r}} (\boldsymbol{\nu}\widetilde{\ell})P$ where $\widetilde{\ell} \stackrel{\mathrm{def}}{=} \mathrm{fn}_{\mathrm{r}}(P)$.

Completing contexts are the only contexts in which processes should be tested. We constrain the definitions of typed barbed congruence and equivalence accordingly. The corresponding bisimulation, *reference bisimulation*, is also a closure to obtain completing contexts. This makes defining its approximants harder, thus in this section, we use the reduction-closed barbed equivalence as our contextual equivalence.

Definition 31 (Barbed bisimulation and reduction-closed barbed congruence with reference names)**.** A relation $\mathcal{R}$ on processes is a *barbed bisimulation with reference names* if whenever $P\ \mathcal{R}\ Q$ with P, Q complete:

1. $P \downarrow_\alpha$ implies $Q \Downarrow_\alpha$
2. $P \to P'$ implies that there is Q' such that $Q \Rightarrow Q'$ and $P'\ \mathcal{R}\ Q'$
3. and symmetrically for Q.

Reduction-closed barbed congruence, written $\cong_c^{Arn}$, is the largest barbed bisimulation that is closed by contexts, and *reduction-closed barbed equivalence*, written $\cong^{Arn}$, is the largest barbed bisimulation that is closed by static contexts.

This typed barbed equivalence is the behavioural equivalence we are mainly interested in. The reference name discipline weakens the requirements on names (by limiting the number of legal contexts), hence the corresponding typed barbed relation is coarser. We are not aware of existing works in the literature that study the impact of the reference name discipline on behavioural equivalence.

Lemma 32. For all compatible P, Q, $P \cong Q$ (and hence also $P \approx_a Q$) implies $P \cong^{Arn} Q$.

We show in Section 3.2.3 that the inclusion is strict.

3.2.3 Bisimulation with reference names

In this section we present proof techniques for barbed equivalence based on the labelled transition semantics of Aπ. For this we introduce two labelled bisimilarities.

The first form of bisimulation, *reference bisimilarity*, only look at transitions of complete processes; processes that are not complete have to be made so. Intuitively, in this bisimilarity, processes are made complete by requiring a closure of the relation with respect to the (well-typed) addition of output messages at reference names (the 'closure under allocation' below). Moreover, when an observer consumes an output at a reference name, say $\overline{\ell}\langle n\rangle$, then, following the discipline on reference names, he/she has to immediately provide another such output message, say $\overline{\ell}\langle m\rangle$. This is formalised using transition notations such as $P \xrightarrow{\overline{\ell}\langle n\rangle[m]} P'$, which makes a swap on ℓ (reading its original content n and replacing it with m). As a consequence of the appearance of such swap transitions, ordinary outputs at reference names are not observed in the bisimulation. Similarly for inputs at reference names: an input $P \xrightarrow{\ell\langle m\rangle} P'$ from a complete process P is not observed, since it is supposed to interact with the unique output at ℓ contained in P (which exists as P is complete). Finally, an observer should respect the completeness condition by the processes and should not communicate a fresh reference name — to communicate such a reference, say ℓ, an allocation for ℓ (an output message at ℓ) has first to be added.

A relation $\mathcal{R}$ is *closed under allocation* if $P\ \mathcal{R}\ Q$ implies $P \mid \overline{\ell}\langle n\rangle\ \mathcal{R}\ Q \mid \overline{\ell}\langle n\rangle$ for any $\overline{\ell}\langle n\rangle$ such that $P \mid \overline{\ell}\langle n\rangle$ and $Q \mid \overline{\ell}\langle n\rangle$ are well-typed. We write $P \xrightarrow{\overline{\ell}\langle n\rangle[m]} P'$ if $P \xrightarrow{\overline{\ell}\langle n\rangle} P''$ and $P' = P'' \mid \overline{\ell}\langle m\rangle$, for some P''; similarly for $P \xrightarrow{(\boldsymbol{\nu} n)\overline{\ell}\langle n\rangle[m]} P'$. Then, as usual, $P \xRightarrow{\overline{\ell}\langle n\rangle[m]} P'$ holds if $P \Rightarrow P'' \xrightarrow{\overline{\ell}\langle n\rangle[m]} P''' \Rightarrow P'$ for some P'', P''', and similarly for $P \xRightarrow{(\boldsymbol{\nu} n)\overline{\ell}\langle n\rangle[m]} P'$.

We let α range over the actions μ plus the aforementioned 'update actions' $\overline{\ell}\langle n\rangle[m]$ and $(\boldsymbol{\nu} n)\overline{\ell}\langle n\rangle[m]$. We also extend type-allowed transitions for update actions as expected, e.g. $[\Delta; P] \xrightarrow{\overline{\ell}\langle n\rangle[m]} [\Delta', \ell; P' \mid \overline{\ell}\langle m\rangle]$ when $[\Delta; P] \xrightarrow{\overline{\ell}\langle n\rangle} [\Delta'; P']$ and similarly for $(\boldsymbol{\nu} n)\overline{\ell}\langle n\rangle[m]$.

Setting m to be the object of an update action, we write $\Delta \vDash_{\mathrm{r}} \alpha$ when: (i) if the object of α is a free reference name then it is in Δ, and (ii) α is not an input or an output at a reference name.

Definition 33 (Reference bisimilarity)**.** A relation $\mathcal{R}$ closed under allocation is a *reference bisimulation* if whenever $\Delta \vDash_{\mathrm{r}} P\ \mathcal{R}\ Q$ with P, Q complete and $[\Delta; P] \xrightarrow{\alpha} [\Delta'; P']$ with $\Delta \vDash_{\mathrm{r}} \alpha$, then

1. either there exists Q' such that $Q \xRightarrow{\widehat{\alpha}} Q'$ and $\Delta' \vDash_{\mathrm{r}} P'\ \mathcal{R}\ Q'$ for some Q'
2. or α is an input $a\langle m\rangle$ and $Q \mid \overline{a}\langle m\rangle \Rightarrow Q'$ with $\Delta' \vDash_{\mathrm{r}} P'\ \mathcal{R}\ Q'$ for some Q'.
3. and symmetrically for Q.

Reference bisimilarity, written $\approx_{\mathrm{r}}$, is the largest reference bisimulation. We write $P \approx_{\mathrm{r}}^{\Delta} Q$ when $(\Delta, P, Q) \in \approx_{\mathrm{r}}$. We often write $P \approx_{\mathrm{r}} Q$ when the typing can be inferred.

Because reference bisimulation is closed by allocation, ensuring that we can add outputs at any time, it explains why we choose to use reduction-based barbed equivalence which can also add contexts at any time, rather than the standard barbed equivalence where all the context must be added upfront.

We can now show that $\approx_{\mathrm{r}}$ coincides with barbed equivalence. The structure of the proof is standard and is given in Appendix A.2, however some care has to be taken to deal with closure under parallel composition.

Lemma 34. If $P \approx_{\mathrm{r}} Q$, and $\emptyset \vdash_{\mathrm{r}} R$, then $P \mid R \approx_{\mathrm{r}} Q \mid R$.

Proposition 35 (Substitutivity for active contexts)**.** If $P \approx_{\mathrm{r}}^{\Delta} Q$, then $E[P] \approx_{\mathrm{r}}^{\Gamma} E[Q]$ for any static Γ/Δ context E.

As barbed bisimilarity is coarser than reference bisimilarity, the soundness of reference bisimilarity with respect to barbed equivalence follows. To prove the completeness, we need a way to get rid of restrictions. When doing reductions in barbed equivalence, a bound name cannot become free. As this can be achieved using transitions, e.g. with an action $(\boldsymbol{\nu} n)\overline{a}\langle n\rangle$, we need the following lemma to mimic this behaviour using a separate process.

Lemma 36. If $(\boldsymbol{\nu} n)(P \mid \overline{s}\langle n\rangle) \cong^{Arn} (\boldsymbol{\nu} n)(Q \mid \overline{s}\langle n\rangle)$ with s a fresh plain name for P and Q, then $P \cong^{Arn} Q$.

Then we can prove the completeness by showing that barbed equivalence is a reference bisimulation. The proof is standard and is given in Appendix A.2

Theorem 37 (Labelled characterisation)**.** $P \approx_{\mathrm{r}} Q$ iff $P \cong^{Arn} Q$.

In reference bisimilarity, the tested processes are complete: hence all their references must explicitly appear as allocated, and when a reference is accessed, an extension of the store is made so to remain with complete processes (and if such an extension introduces other new references, a further extension is needed). The goal of the bisimilarity $\approx_{\mathrm{ip}}$ below is to allow one to work on processes with open references, and make the extension of the store only when necessary. The definition of the bisimulation exploits an inductive predicate to accommodate finite extensions of the store, one step at a time. This predicate can be thought of as an inductive game, in which the 'verifier' can choose rule Base and close the game, or choose rule Ext and a reference ℓ; in the latter case the 'refuter' chooses the value of m in the forall quantification, which corresponds to the value stored in ℓ.

Definition 38 (Inductive predicate)**.** The predicate $\mathtt{ok}(\Delta, \mathcal{R}, P, Q, \mu)$ (where Δ is a set of names, $\mathcal{R}$ a process relation, P, Q processes, and μ an action) holds if it can be proved inductively from the following two rules:

$$\textsc{Base}\quad \frac{\begin{cases} Q \mid \overline{n}\langle m\rangle \Rightarrow Q' & \text{for } \mu = n\langle m\rangle \\ Q \stackrel{\mu}{\Rightarrow} Q' & \text{otherwise} \end{cases} \qquad P' \mathrel{\mathcal{R}} Q'}{\mathtt{ok}(\Delta, \mathcal{R}, P', Q, \mu)}$$

$$\textsc{Ext}\quad \frac{\ell \notin \Delta \qquad \forall\, m \,:\, \mathtt{ok}((\Delta, \ell), \mathcal{R}, P' \mid \overline{\ell}\langle m\rangle, Q \mid \overline{\ell}\langle m\rangle, \mu)}{\mathtt{ok}(\Delta, \mathcal{R}, P', Q, \mu)}$$

Definition 39 (Bisimilarity with inductive predicate, $\approx_{\mathrm{ip}}$)**.** A relation $\mathcal{R}$ is an *ip-bisimulation* if whenever $\Delta \vDash_{\mathrm{r}} P \mathrel{\mathcal{R}} Q$ and $[\Delta; P] \xrightarrow{\mu} [\Delta'; P']$, we can derive $\mathtt{ok}(\Delta \cup \Delta', \mathcal{R}, P', Q, \mu)$, and symmetrically for the transitions emanating from Q. We write $\approx_{\mathrm{ip}}$ for the largest ip-bisimulation.

The names in $\Delta \cup \Delta'$ are the reference names that appear in output subject position in P' or Q. Therefore, when using rule Ext of the inductive predicate, the condition $\ell \notin \Delta$ ensures us that the message at ℓ can be added without breaking typability.

The following up-to technique allows us to erase common messages on reference names along the bisimulation game.

For this, we use the notation M_s, where s is a finite list of pairs (ℓ,m), to describe parallel compositions of outputs on reference names (i.e. $M_s \stackrel{\text{def}}{=} \prod_{(\ell,m)\in s} \overline{\ell}\langle m\rangle$), and $\Delta_s \vdash_{\mathrm{r}} M_s$ where Δ_s contains all first components of pairs of s. Intuitively, M_s represents a chunk of store. An example is given by $s = \{(\ell_1, m_1), (\ell_2, m_2)\}$ where $\Delta_s = \ell_1, \ell_2$ and $M_s = \overline{\ell_1}\langle m_1\rangle \mid \overline{\ell_2}\langle m_2\rangle$

Definition 40 (ip-bisimulation up to store)**.** An ip-bisimulation *up to store* is defined like ip-bisimulation (Definition 39), using a predicate $\mathtt{ok}'(\Delta \cup \Delta', \mathcal{R}, P', Q, \mu)$. This predicate is defined by a modified version of rule Ext where ok' is used instead of ok, both in the premise and in the conclusion, and by the following modified version of the Base rule:

$$\textsc{Base-Up}\quad \frac{P' \equiv P'' \mid M_s \qquad \begin{cases} Q \mid \overline{n}\langle m\rangle \Rightarrow\equiv Q'' \mid M_s & \text{for } \mu = n\langle m\rangle \\ Q \stackrel{\mu}{\Rightarrow}\equiv Q'' \mid M_s & \text{otherwise} \end{cases} \qquad P'' \mathrel{\mathcal{R}} Q''}{\mathtt{ok}'(\Delta, \mathcal{R}, P', Q, \mu)}$$

Rule Base-Up makes it possible to erase common store components before checking that the processes are related by $\mathcal{R}$.

Proposition 41. If $\mathcal{R}$ is an ip-bisimulation up to store, then $\mathcal{R} \subseteq \approx_{\mathrm{ip}}$.

Proposition 42 (Soundness of $\approx_{\mathrm{ip}}$)**.** We have $\approx_{\mathrm{ip}} \subseteq \approx_{\mathrm{r}}$.

Intuitively, the inclusion holds because an ip-bisimulation is closed by parallel composition with M_s processes. We leave the opposite direction, completeness, as an open issue.

3.2.4 Examples

To present the examples below, we introduce various notations displaying the use of reference names to implement proper references. This intuition is more formally developed in Section 3.3.

$$(\boldsymbol{\nu}\ell = n)P \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell)(\overline{\ell}\langle n\rangle \mid P) \qquad \ell \triangleleft n.\, P \stackrel{\text{def}}{=} \ell(_).\,(\overline{\ell}\langle n\rangle \mid P) \qquad \ell \triangleright (m).\, P \stackrel{\text{def}}{=} \ell(m).\,(\overline{\ell}\langle m\rangle \mid P)$$

The first represents a local reference ℓ with content n, the second writes the name n at ℓ and the third reads the content at ℓ and binds it as m. We can see how these read/write operations affects a local reference by looking at the following transitions: we have $(\boldsymbol{\nu}\ell = n)\ell \triangleleft m.\, P \xrightarrow{\tau} (\boldsymbol{\nu}\ell = m)P$ and $(\boldsymbol{\nu}\ell = n)\ell \triangleright (m).\, P \xrightarrow{\tau} (\boldsymbol{\nu}\ell = n)P\{n/m\}$.

We now give examples of uses of the various forms of labelled bisimulation ($\approx_a, \approx_{\mathrm{r}}, \approx_{\mathrm{ip}}, \approx_{\mathrm{ip}}$ up to store) for Aπ to establish equivalences between processes with references. In some cases, we use the 'up-to structural congruence' ($\equiv$) version of the bisimulations — a standard 'up-to' technique.

The first example is about a form of commutativity for the write construct.

Example 43. We wish to establish $!\ell \triangleleft a.\, \ell \triangleleft b \cong^{Arn} !\ell \triangleleft b.\, \ell \triangleleft a$. For this, we prove the law $!\ell \triangleleft a.\, \ell \triangleleft b \cong^{Arn} !\ell \triangleleft a \mid !\ell \triangleleft b$, which will be enough to conclude, by commutativity of parallel composition. We write

$$P_1 \stackrel{\text{def}}{=} !\ell(_).\,(\overline{\ell}\langle a\rangle \mid \ell(_).\,\overline{\ell}\langle b\rangle) \quad \text{and} \quad P_2 \stackrel{\text{def}}{=} (!\ell(_).\,\overline{\ell}\langle a\rangle) \mid (!\ell(_).\,\overline{\ell}\langle b\rangle).$$

We can derive $P_1 \approx_a P_2$, using the singleton relation $\mathcal{R} \stackrel{\text{def}}{=} \{(P_1, P_2)\}$, and showing that $\mathcal{R}$ is an asynchronous bisimilarity up-to context and structural congruence [41]. We can then conclude by Lemma 32.

The following example shows some benefits of using $\approx_{\mathrm{ip}}$ and $\approx_{\mathrm{ip}}$ up to store in the proof of a property that generalises law (3.1), which involves a 'useless read'.

Example 44. Suppose $\emptyset \vdash_{\mathrm{r}} P_0\ \mathcal{R}\ Q_0$, where $\mathcal{R}$ is an asynchronous bisimulation, $\texttt{ObType}(\ell) \in \texttt{RefTypes}$, and x is a fresh name. Then $\emptyset \vdash_{\mathrm{r}} \ell(x).\,(P_0 \mid \overline{\ell}\langle x\rangle) \approx_{\mathrm{r}} Q_0$.

In general, $\ell(x).\,(P_0 \mid \overline{\ell}\langle x\rangle)$ and Q_0 are not related by $\approx_a$ (take $P_0 = Q_0 = \overline{a}\langle n\rangle$), thus the inclusion in Lemma 32 is strict.

Using the notation M_s from Definition 40, to prove $\ell(x).\,(P_0 \mid \overline{\ell}\langle x\rangle) \approx_{\mathrm{r}} Q_0$ using a reference bisimulation, we need a relation such as

$$\begin{aligned}
\mathcal{R}_1 \stackrel{\text{def}}{=}\ & \{(\ell(x).\,(P_0 \mid \overline{\ell}\langle x\rangle), Q_0)\} \\
& \cup \{(\ell(x).\,(P_0 \mid \overline{\ell}\langle x\rangle) \mid \overline{\ell}\langle \ell'\rangle \mid \overline{\ell'}\langle m\rangle, Q_0 \mid \overline{\ell}\langle \ell'\rangle \mid \overline{\ell'}\langle m\rangle) \mid \text{ for any } m\} \\
& \cup \{(\ell(x).\,(P_0 \mid \overline{\ell}\langle x\rangle) \mid \overline{\ell}\langle \ell'\rangle \mid \overline{\ell'}\langle m\rangle \mid M_s, Q_0 \mid \overline{\ell}\langle \ell'\rangle \mid \overline{\ell'}\langle m\rangle \mid M_s) \mid \text{ for any } m, M_s\} \\
& \cup \{P \mid \overline{\ell}\langle \ell'\rangle \mid \overline{\ell'}\langle m\rangle \mid M_s, Q \mid \overline{\ell}\langle \ell'\rangle \mid \overline{\ell'}\langle m\rangle \mid M_s) \mid \text{ for any } m, M_s, \text{with } P\ \mathcal{R}\ Q\}
\end{aligned}$$

and prove that $\mathcal{R}_1 \cup \mathcal{R}_1^{-1}$ (where $\mathcal{R}_1^{-1}$ is the inverse of $\mathcal{R}_1$) is a reference bisimulation.

We can simplify the proof and avoid the several quantifications in $\mathcal{R}_1$ (in particular on M_s, whose size is arbitrary), and prove that $\mathcal{R}_2$ is an ip-bisimulation, for

$$\begin{aligned}
\mathcal{R}_2 \stackrel{\text{def}}{=}\ & \mathcal{R} \cup \{(P \mid \overline{\ell}\langle m\rangle, Q \mid \overline{\ell}\langle m\rangle), \text{ for any } m, \text{with } P\ \mathcal{R}\ Q\} \\
& \cup \{(\ell(x).\,(P_0 \mid \overline{\ell}\langle x\rangle), Q_0),\ (Q_0, \ell(x).\,(P_0 \mid \overline{\ell}\langle x\rangle))\}.
\end{aligned}$$

The last component of $\mathcal{R}_2$ is dealt with using rule Ext of the inductive predicate (Definition 38), and this brings in the second component (the closure of $\mathcal{R}$ under messages on ℓ).

We can simplify the proof further, by removing such second component, and show that $\mathcal{R}_3$ is an ip-bisimulation up to store, for

$$\mathcal{R}_3 \stackrel{\text{def}}{=} \mathcal{R} \cup \{(\ell(x).\,(P_0 \mid \overline{\ell}\langle x\rangle), Q_0),\ (Q_0, \ell(x).\,(P_0 \mid \overline{\ell}\langle x\rangle))\}.$$

3.3 Application: a π-calculus with references

3.3.1 Syntax and semantics of $\pi^{\mathtt{ref}}$

In this section, we introduce $\pi^{\mathtt{ref}}$, the asynchronous π-calculus extended with primitives to interact with memory locations.

We assume an infinite set $\mathtt{Names}$ of *names* and a distinct infinite set $\mathtt{Refs}$ of *references*. These sets do not contain the special symbol $\star$, that stands for the constant "unit". We use $a, b, c, \ldots, p, q, \ldots$ to range over $\mathtt{Names}$; $\ell, \ldots$ to range over $\mathtt{Refs}$; and $n, m, \ldots, x, y, \ldots$ to range over $\mathtt{All} \stackrel{\text{def}}{=} \mathtt{Names} \cup \mathtt{Refs} \cup \{\star\}$. To make the encoding easier to read, we use $\ell, \ldots$ both for references in $\pi^{\mathtt{ref}}$ and for reference names in Aπ which are a subset of names.

The grammar for the calculus $\pi^{\mathtt{ref}}$ is the following.

$$\begin{array}{rcl} P & ::= & \mathbf{0} \mid a(x).\,P \mid \overline{a}\langle n\rangle \mid !P \mid P_1 \mid P_2 \mid (\boldsymbol{\nu} a)P \mid [n=m]P \\ & & \mid (\boldsymbol{\nu}\ell = n)P \mid \ell \triangleleft n.\,P \mid \ell \triangleright (x).\,P \mid \ell \bowtie n(x).\,P \end{array}$$

The operators in the first line are the standard π-calculus constructs for the inactive process, input, asynchronous output, replication, parallel composition, name restriction, and matching (however matching here is defined on both names and references). In the second line, we find the operators to handle references: reference restriction, or allocation (creating a new reference ℓ with initial value n), write (setting the content of ℓ to n), read (reading in x the value of ℓ), swap (atomically reading on x and replacing the content of the reference with n). These operations correspond to the notations used in Section 3.2.4.

We use the same notations as for standard π-calculus, and we write $(\boldsymbol{\nu}\widetilde{L})$ a sequence of reference allocations (i.e. a piece of store), using L to represent a single allocation such as $\ell = n$. Given the binders $(\boldsymbol{\nu} a)P$ and $(\boldsymbol{\nu}\ell = n)P$ (for a and ℓ, respectively), $a(x).\,P$, $\ell \triangleright (x).\,P$ and $\ell \bowtie n(x)$ (for x), we define $\mathrm{bn}(O)$, $\mathrm{fn}(O)$ as usual, for the *bound* and *free names* of some object O (process, action, etc.) and similarly for the *free* and *bound references* of O ($\mathrm{fr}(O)$, $\mathrm{br}(O)$). The set of *names* of O is defined as the union of its free and bound names; and analogously for *references*.

We shall assume that there is a sorting system under which all processes we manipulate are well-typed. In the remainder we assume that all objects (processes, contexts, actions, etc.) respect a given sorting.

The definition of structural congruence, $\equiv$, is the expected one from the π-calculus, treating the $(\boldsymbol{\nu}\ell = n)$ operator like a restriction with n being free:

$$P \mid (\boldsymbol{\nu}\ell = n)Q \;\equiv\; (\boldsymbol{\nu}\ell = n)(P \mid Q) \text{ if } \ell \notin \mathrm{fr}(Q) \qquad (\boldsymbol{\nu}\ell = n)(\boldsymbol{\nu} a)P \;\equiv\; (\boldsymbol{\nu} a)(\boldsymbol{\nu}\ell = n)P \text{ if } a \neq n$$

$$(\boldsymbol{\nu}\ell = n)\mathbf{0} \;\equiv\; \mathbf{0} \qquad (\boldsymbol{\nu}\ell = n)(\boldsymbol{\nu}\ell' = m)P \;\equiv\; (\boldsymbol{\nu}\ell' = m)(\boldsymbol{\nu}\ell = n)P \text{ if } \ell \neq m \text{ and } \ell' \neq n$$

Contexts, ranged over by C, are process expressions with a hole $[\cdot]$ in it. We write $C[P]$ for the process obtained by replacing the hole in C with P. *Active* (or *static*) *contexts*, ranged over by E, are given by:

$$E \quad ::= \quad [\cdot] \mid E \mid P \mid (\boldsymbol{\nu} a)E \mid (\boldsymbol{\nu}\ell = n)E \;.$$

The reduction relation $\rightarrow$ is presented in Figure 3.2. It uses *active contexts* to isolate the subpart of the term that is active in a reduction. We write $\Rightarrow$ for the 'multistep' version of $\rightarrow$, whereby $P \Rightarrow P'$ if P may become P' after a (possibly empty) sequence of reductions. Rules R-Read, R-Write and R-Swap in Figure 3.2 describe an interaction between the process and a reference ℓ. These rules make use of a store $(\boldsymbol{\nu}\widetilde{L})$; this is necessary because there might be references that depend on ℓ, and as such cannot be moved past the restriction on ℓ. An example is $(\boldsymbol{\nu}\ell = a)(\boldsymbol{\nu}\ell' = \ell)\ell \triangleleft b.\,P$: the write operation is executed by applying rule R-Write, with $(\boldsymbol{\nu}\widetilde{L}) = (\boldsymbol{\nu}\ell' = \ell)$, as the restriction on ℓ' cannot be brought above the restriction on ℓ. We recall that $\mathrm{br}(\boldsymbol{\nu}\widetilde{L})$ are the references bound by the $\boldsymbol{\nu}$.

Our behavioural equivalences will be *reduction-closed barbed congruence* and *reduction-closed barbed equivalence*, a form of bisimulation on reduction that uses closure under contexts and simple observables. In the context closure, however, we make sure that all references mentioned in the tested process have been allocated.

P exhibits a barb at a (so a is in $\mathtt{Names}$), written $P \downarrow_{\overline{a}}$, if $P \equiv (\boldsymbol{\nu}\widetilde{b})(\boldsymbol{\nu}\widetilde{L})(\overline{a}\langle m\rangle \mid P')$ with $a \notin \widetilde{b}$. We write $P \Downarrow_{\overline{a}}$ if $P \Rightarrow P_1$ and $P_1 \downarrow_{\overline{a}}$ for some P_1.

R-Equiv:
$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

R-Ctxt:
$$\frac{P \to P'}{E[P] \to E[Q]}$$

R-Comm:
$$\frac{}{a(x).\,P \mid \overline{a}\langle n\rangle \to P\{n/x\}}$$

R-Read:
$$\frac{\ell, n \notin \mathrm{br}(\boldsymbol{\nu}\widetilde{L})}{(\boldsymbol{\nu}\ell = n)(\boldsymbol{\nu}\widetilde{L})(\ell \triangleright (x).\,P \mid Q) \to (\boldsymbol{\nu}\ell = n)(\boldsymbol{\nu}\widetilde{L})(P\{n/x\} \mid Q)}$$

R-Write:
$$\frac{\ell, n \notin \mathrm{br}(\boldsymbol{\nu}\widetilde{L})}{(\boldsymbol{\nu}\ell = m)(\boldsymbol{\nu}\widetilde{L})(\ell \triangleleft n.\,P \mid Q) \to (\boldsymbol{\nu}\ell = n)(\boldsymbol{\nu}\widetilde{L})(P \mid Q)}$$

R-Swap:
$$\frac{\ell, n, m \notin \mathrm{br}(\boldsymbol{\nu}\widetilde{L})}{(\boldsymbol{\nu}\ell = m)(\boldsymbol{\nu}\widetilde{L})(\ell \bowtie n(x).\,P \mid Q) \to (\boldsymbol{\nu}\ell = n)(\boldsymbol{\nu}\widetilde{L})(P\{m/x\} \mid Q)}$$

Figure 3.2: $\pi^{\mathtt{ref}}$, reduction relation

A process P is *reference-closed* if $\mathrm{fr}(P) = \emptyset$. A context C is *closing on the references* of a process P if $C[P]$ is reference-closed; similarly, C is closing on the references of P, Q if it closing on the references of both P and Q. Since reductions may only decrease the set of free names of a process, the property of being reference-closed is preserved by reductions.

The equivalences are then defined in the same way as in Aπ (Definition 31) with reference-closed processes used in place of complete processes. Barbed equivalence (resp. barbed congruence) in $\pi^{\mathtt{ref}}$ is written $\cong^{ref}$ (resp. $\cong_c^{ref}$).

The restriction to closing contexts (as opposed to arbitrary contexts) yields laws such as

$$\ell \triangleright (x).\,P \;\cong_c^{ref}\; P,$$

whenever $x \notin \mathrm{fn}(P)$. Closing contexts ensure that the reading on ℓ is not blocking, and therefore possible observables in P are visible on both sides.

As the quantification on contexts refers to the free references of the tested processes, transitivity of barbed congruence and equivalence requires some care. As usual in the π-calculus, barbed equivalence is not preserved by the input construct.

3.3.2 Mapping $\pi^{\mathtt{ref}}$ onto the Asynchronous π-calculus

We present the encoding of $\pi^{\mathtt{ref}}$ into Aπ, which follows our notations from Section 3.2.4.

Encoding $\pi^{\mathtt{ref}}$. The reference names from Section 3.2 will be used to represent the references of $\pi^{\mathtt{ref}}$.

The encoding $[\![\cdot]\!]$, from $\pi^{\mathtt{ref}}$ to Aπ, is a homomorphism on all operators (thus, e.g. $[\![P_1 \mid P_2]\!] \stackrel{\mathrm{def}}{=} [\![P_1]\!] \mid [\![P_2]\!]$, and $[\![a(m).\,P]\!] \stackrel{\mathrm{def}}{=} a(m).\,[\![P]\!]$), except for reference constructs for which we have:

$$[\![(\boldsymbol{\nu}\ell = m)P]\!] \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\ell)(\overline{\ell}\langle m\rangle \mid [\![P]\!]) \qquad [\![\ell \triangleleft v.\,P]\!] \stackrel{\mathrm{def}}{=} \ell(_).\,(\overline{\ell}\langle v\rangle \mid [\![P]\!]) \qquad [\![\ell \triangleright (x).\,P]\!] \stackrel{\mathrm{def}}{=} \ell(x).\,(\overline{\ell}\langle x\rangle \mid [\![P]\!])$$

$$[\![\ell \bowtie n(x).\,P]\!] \stackrel{\mathrm{def}}{=} \ell(x).\,(\overline{\ell}\langle n\rangle \mid [\![P]\!])$$

Recall that $\ell(_).\,Q$ stands for an input whose bound name does not appear in Q. In the encoding, an object m stored at reference ℓ is represented as a message $\overline{\ell}\langle m\rangle$. Accordingly, the encoding of a write $\ell \triangleleft v.\,P$ is $\ell(_).\,(\overline{\ell}\langle v\rangle \mid [\![P]\!])$, meaning that the process acquires the current message at ℓ (which is thus not available any more) and replaces it with an output with the new value. The encoding of a read $\ell \triangleright (x).\,P$ follows a similar pattern, this time however the same value is received and emitted: $\ell(x).\,(\overline{\ell}\langle x\rangle \mid P)$. The encoding of swap combines the two patterns.

Validating the Encoding. We now show that the two notions of barbed congruence coincide via the encoding.

Theorem 45 (Operational correspondence)**.** If $P \to P'$, then $[\![P]\!] \to [\![P']\!]$.
Conversely, if $[\![P]\!] \to Q$, then $P \to P'$, with $[\![P']\!] \equiv Q$.

The next lemma shows that, up to asynchronous bisimilarity, we can 'read back' well-typed processes in Aπ, via the encoding, as processes in $\pi^{\mathtt{ref}}$. And similarly for contexts. For some technical reason, we need to use simple types; e.g. the sorting is non-recursive (meaning that the graph that represents the sorting function, in which the nodes are the types, does not contain cycles). It removes processes like $(\boldsymbol{\nu}\ell_1, \ell_2)(\overline{\ell_1}\langle\ell_2\rangle \mid \overline{\ell_2}\langle\ell_1\rangle)$ that have no equivalent in $\pi^{\mathtt{ref}}$.

Lemma 46. If $\emptyset \vdash_{\mathrm{r}} P$, then there exists R in $\pi^{\mathtt{ref}}$ such that $[\![R]\!] \approx_a P$.

Theorem 45 and Lemma 46 are the main ingredients to derive the following theorem:

Theorem 47 (Full abstraction)**.** For any P, Q in $\pi^{\mathtt{ref}}$: $P \cong_c^{ref} Q$ iff $[\![P]\!] \cong_c^{Arn} [\![Q]\!]$;
and similarly $P \cong^{ref} Q$ iff $[\![P]\!] \cong^{Arn} [\![Q]\!]$.

3.3.3 Behavioural equivalence in $\pi^{\mathtt{ref}}$: examples

We present a few examples that illustrate some subtleties of behavioural equivalence in $\pi^{\mathtt{ref}}$. The first example shows that processes may be equivalent even though the store is public and holds different values. (In the example, the reference ℓ is actually restricted, but the process P underneath the restriction, representing an observer, is arbitrary).

Example 48. For any P, we have $P_1 \cong^{ref} P_2$, for

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

In the second example, the assignment on top of P is not blocking, provided that the same assignment is anyhow possible, and provided that the current value of the store can be recorded.

Proof. Let R_1, R_2 be the encodings of P_1, P_2 in the example:

$$\begin{aligned} R_1 &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell)\big(\overline{\ell}\langle a\rangle \mid [\![P]\!] \mid !\ell(_).\,\overline{\ell}\langle a\rangle \mid !\ell(_).\,\overline{\ell}\langle b\rangle\big) \\ R_2 &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell)\big(\overline{\ell}\langle b\rangle \mid [\![P]\!] \mid !\ell(_).\,\overline{\ell}\langle a\rangle \mid !\ell(_).\,\overline{\ell}\langle b\rangle\big) \end{aligned}$$

We then have $R_1 \Rightarrow\equiv R_2$ and $R_2 \Rightarrow\equiv R_1$, which implies $R_1 \approx_a R_2$ (where $\approx_a$ is asynchronous bisimilarity), as $\{(R_1, R_2)\} \cup id$, where $id = \{(P, P)\}$ is the identity relation, is an asynchronous bisimulation up to $\equiv$. We can then conclude by Theorem 47. □

Example 49. We have $P_1 \cong^{ref} P_2$, for

$$P_1 \stackrel{\text{def}}{=} \ell \triangleleft b.\,P \mid !\ell \triangleleft b \mid !\ell \triangleright (x).\,\ell \triangleleft x \qquad\qquad P_2 \stackrel{\text{def}}{=} P \mid !\ell \triangleleft b \mid !\ell \triangleright (x).\,\ell \triangleleft x$$

On the left, it would seem that P runs with a store in which ℓ contains b; whereas on the right, P could also run with the initial store, where ℓ could contain a different value, say a. However the component $!\ell \triangleright (x).\,\ell \triangleleft x$ allows us to store a in x and then write it back later, thus overwriting b.

Proof. Let R_1, R_2 be the encodings of P_1, P_2 in the example:

$$\begin{aligned} R_1 &\stackrel{\text{def}}{=} \ell(_).\,(\overline{\ell}\langle b\rangle \mid [\![P]\!]) \mid !\ell(_).\,\overline{\ell}\langle b\rangle \mid !\ell(x).\,(\overline{\ell}\langle x\rangle \mid \ell(_).\,\overline{\ell}\langle x\rangle) \\ R_2 &\stackrel{\text{def}}{=} [\![P]\!] \mid !\ell(_).\,\overline{\ell}\langle b\rangle \mid !\ell(x).\,(\overline{\ell}\langle x\rangle \mid \ell(_).\,\overline{\ell}\langle x\rangle) \end{aligned}$$

Then for all m, processes $\overline{\ell}\langle m\rangle \mid R_1$ and $\overline{\ell}\langle m\rangle \mid R_2$ are complete. We define

$$\mathcal{R} \stackrel{\text{def}}{=} \{\big(R_1 \mid \overline{\ell}\langle m\rangle \mid B_X,\, R_2 \mid \overline{\ell}\langle m\rangle \mid B_X\big)\}\ ,$$

where $X \stackrel{\text{def}}{=} \{x_1, \ldots, x_n\}$ is a possibly empty finite set of names, and

$$B_X \stackrel{\text{def}}{=} \ell(_).\,\overline{\ell}\langle x_1\rangle \mid \ldots \mid \ell(_).\,\overline{\ell}\langle x_n\rangle$$

Then $\mathcal{R} \cup id$ is an ip-bisimulation.

Reusing the same notations, $\mathcal{R}' \stackrel{\text{def}}{=} \{(R_1 \mid B_X,\ R_2 \mid B_X)\} \cup id$ is an ip-bisimulation up to store: this up-to technique allows us to remove the $\overline{\ell}\langle m\rangle$ particles. □

Example 50. We have $P_s \not\cong^{ref} Q_s$, where

$$P_s \stackrel{\text{def}}{=} (\boldsymbol{\nu} t)\ell \triangleleft b.\,(\overline{t} \mid !t.\,\ell \triangleleft a.\,(\overline{c} \mid \ell \triangleleft b.\,(\overline{t} \mid c))) \qquad Q_s \stackrel{\text{def}}{=} (\boldsymbol{\nu} t)\ell \triangleleft a.\,(\overline{t} \mid !t.\,\ell \triangleleft b.\,(\overline{c} \mid \ell \triangleleft a.\,(\overline{t} \mid c)))$$

The discriminating context being large, the formal discussion is moved in Appendix A.3.2. Intuitively, P_s and Q_s are refinements of the processes in Example 48, in that their initial writes store different values on the reference ℓ, but both processes maintain the capability of writing both values in ℓ. The differences with Example 48 are the additional inputs and outputs on name c, which are generated along the transitions. These allow an observer to distinguish P_s from Q_s by exploiting the swap construct. We informally explain the reason. If the two processes have written the same value, say a, in ℓ, then Q_s has generated the same number of inputs and outputs on c, while P_s must have generated an extra output. An observer can use swap to read the content of ℓ, so to check that the value is indeed a, and write back a fresh name, say e. Now the observer can tell that P_s has an extra output on c: process Q_s cannot add a further output, because this would require overwriting e in ℓ, which can be tested by the observer at the end.

We have seen in Example 48 two equivalent processes whose initial store (a single reference) is different. The equivalence holds intuitively because the values that the two processes can store are the same. Using two references, it is possible to make the example more complex. In Example 51, the processes are equivalent and yet the pairs of values that may be simultaneously stored in the two references are different for the two processes. For each reference separately, the set of possible values is the same. But setting a reference to a certain value implies first having set the other reference to some specific values. (The processes could be distinguished if an observer had the possibility to simultaneously read the two references.)

Example 51. Consider two references ℓ_1, ℓ_2 where booleans (represented as 0,1 below) can be stored. Then for any P, we have $P_1 \cong^{ref} P_2$, where

$$P_1 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell_1 = 0, \ell_2 = 0)(P \mid (\boldsymbol{\nu} t)(\overline{t} \mid !t.\,\ell_1 \triangleleft 1.\,\ell_1 \triangleleft 0.\,\ell_2 \triangleleft 1.\,\ell_2 \triangleleft 0.\,\overline{t}))$$
$$P_2 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell_1 = 0, \ell_2 = 0)(P \mid (\boldsymbol{\nu} t)(\overline{t} \mid !t.\,\ell_1 \triangleleft 1.\,\ell_2 \triangleleft 1.\,\ell_1 \triangleleft 0.\,\ell_2 \triangleleft 0.\,\overline{t}))$$

P_1 and P_2 can write 0 and 1 in references ℓ_1 and ℓ_2, but not in the same order. By doing so, we see that if P_1 loops, the content of ℓ_1 and ℓ_2 will evolve thus: $(0,0) \to (1,0) \to (0,0) \to (0,1) \to (0,0)$, while for P_2 the loop is different: $(0,0) \to (1,0) \to (1,1) \to (0,1) \to (0,0)$.

In particular, P_2 can always go through the state $(1,1)$, independently of the transitions of P, while P_1 cannot, in general, reach this state.

The example above relies on the fact that the domain of possible values for ℓ_1 and ℓ_2 is finite. A more sophisticated example, without such assumption, is given in the Appendix A.3.2.

Chapter 4

Sequentiality

In this section we study sequentiality. 'Sequentiality' intuitively indicates the existence of a single thread of computation. Our main objectives are to define a type system ensuring a sequential execution (Section 4.1), and then to examine its impact on behavioural equivalence, by adapting the bisimulation proof technique to this typed calculus (Section 4.2). Finally, in Section 4.3, we briefly explore how combining this type system with the one for references from Chapter 3 enables further reasoning and more interesting examples to be proven equivalent.

4.1 Type system

As mentioned above, intuitively, sequentiality ensures us that at any time at most one interaction can occur in a system; i.e. there is a single computation thread. A process that holds the thread decides what the next interaction can be. We say that such a process is *active*, while a process that does not hold the thread is *inactive*. An active process does so by offering a single particle (input or output) that controls the thread. The process may offer multiple particles, but only one of them may control the thread. The control on the thread attached to a particle is determined by the subject name of that particle. A given name may exercise the control on the thread either in output or in input; in the former case we say that the name is *output-controlled*, in the latter case the name is *input-controlled*. For instance, suppose that x, y, z are output-controlled and u, v are input-controlled. Then the following process correctly manages the thread and will indeed be typable in our type system:

$$P \stackrel{\text{def}}{=} u.\,(\overline{x} \mid y.\,\overline{x}) \mid z.\,\overline{y} \mid \overline{v}$$

The initial particles in P are $u, z, \overline{v}$; however only u controls the thread, as z is output-controlled and v is input-controlled. When the input at u is consumed, the new particles $\overline{x}, y$ are available, where $\overline{x}$ now controls the thread, as both names x, y are output-controlled. An external process that consumes the particle $\overline{x}$ will acquire the control over the thread. For instance, a process such as $Q \stackrel{\text{def}}{=} \overline{u} \mid x.\,Q'$ initially does not hold the thread; in the parallel composition $P \mid Q$, after the two interactions at u and x, the control on the thread will be acquired by Q':

$$P \mid Q \rightarrow\!\!\rightarrow (y.\,\overline{x} \mid z.\,\overline{y} \mid \overline{v}) \mid Q'$$

Now Q' will decide on the next interaction; for instance, it may offer an output at y or z, or an input at v. It may offer at most one of these, though it may offer other particles that do not control the thread.

Sequentiality however does not exclude non-determinism. An output particle $\overline{a}\langle b\rangle$ that owns the thread may have the possibility of interacting with different input processes at a (and symmetrically for input processes owning the thread). Indeed we also admit internal non-determinism (i.e, processes such as $\tau.\,P + \tau.\,Q$ that may chose to reduce either to P or to Q without interactions with the environment), both in active and in inactive processes.

Notations. *In the remainder, x, y, z range over output-controlled names, u, v, w over input-controlled names; we recall that a, b, c range over the set of all names.*

In the π-calculus, programming with output-controlled names is natural. A typical usage of such names is to activate other processes, like a server waiting for requests. This discipline revolves around the idea that when a process sends a message, it expects the receiver to act upon reception, fulfilling the request. In a sequential setting, this also means that the process must wait to receive an answer before continuing.

On the other hand, being input-controlled requires to be active while receiving, which can seem counter-intuitive. Input-controlled names are used to model the references names from Chapter 3 which will be discussed in more details in Section 4.3. We present here some other cases where input-controlled names naturally appear.

From polyadic to monadic. We recall the main clauses for the translation from polyadic processes to monadic processes [52, Section 3.1]:

$$[\![a(b_1,\dots,b_n).\,P]\!] \stackrel{\text{def}}{=} a(u).\,u(b_1).\dots.\,u(b_n).\,[\![P]\!] \qquad [\![\overline{a}\langle b_1,\dots,b_n\rangle.\,P]\!] \stackrel{\text{def}}{=} (\boldsymbol{\nu} u)\overline{a}\langle u\rangle.\,\overline{u}\langle b_1\rangle.\dots.\,\overline{u}\langle b_n\rangle.\,[\![P]\!]$$

Sending multiple names $b_1,\dots,b_n$ is simulated by sending a private channel u, which is then used to send all names in sequence. If we perform this translation in a sequential setting, the private channel u must be input-controlled, whether the channel used for the original communication a is input-controlled or not. In the π-calculus, this translation is sound but not complete, as there exist polyadic processes that are equivalent but whose encodings are not. Restricting the calculus with sequentiality removes some counter examples, so it could be interesting to see how the addition of sequentiality would affects the soundness/completeness of the translation.

Milner's original encoding of the lazy-λ-calculus. Milner defined and studied various encodings of the λ-calculus in the π-calculus. Our encoding in Chapter 6 is built upon one. Here we present another one, solely to illustrate the notion of input-controlled name. The encoding is given as follows [34]:

$$[\![x]\!]_u \stackrel{\text{def}}{=} \overline{x}\langle u\rangle \qquad [\![\lambda x.\,M]\!]_u \stackrel{\text{def}}{=} u(x,v).\,[\![M]\!]_v \qquad [\![MN]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu} v)([\![M]\!]_v \mid (\boldsymbol{\nu} x)(\overline{v}\langle x,u\rangle.\,!x(w).\,[\![N]\!]_w)$$

The encoding is parametrised by a name u. Intuitively, $[\![M]\!]_u$ is computing a function, which can then be called by sending its argument at u. We observe that the encoding of a λ-term is a sequential process, when u, v, w are input-controlled while x is output-controlled. Indeed, in the encoding of the application MN, as N should not reduce, the process can immediately send the argument to the encoding of M. The corresponding output at v is inactive, waiting for the computation of M to reduce into a function.

The type system for sequentiality is presented in Figure 4.1. Judgements are of the form $\eta \vdash_{\text{s}} P$, for $\eta \in \{0,1\}$. A judgement $1 \vdash_{\text{s}} P$ indicates that P owns the thread, i.e. P is *active*, and $0 \vdash_{\text{s}} P$ otherwise, i.e. P is *inactive*.

We recall that we only present the additional typing constraints given by sequentiality, assuming the existence of a sorting under which all processes are well-typed.

Some remarks on the rules in Figure 4.1: a rule with a double conclusion is an abbreviation for more rules with the same premises but separate conclusions. The continuation of an input always owns control on the thread; the input itself may or may not have the control (rules SInpI and SInpO) (this is a choice we make, as the dual approach seems less interesting [29]). A τ-prefix is neutral w.r.t. the thread. The rule for parallel composition makes sure that the control on the thread is granted to only one of the components; in contrast, in the rule for sum, the control is maintained for both summands. Operators $\mathbf{0}$ and match cannot own the thread; this makes sure that the thread control is always exercised.

We present some behavioural properties that highlight the meaning of sequentiality. A reduction $P \to P'$ is an *interaction* if it has been obtained from a communication between an input and an output (formally, its derivation in the LTS of Figure 2.2 uses rule AComm). A pair of an unguarded input and an unguarded output at the same name form an *interaction redex*. In a sequential system, one may not find two disjoint interaction redexes.

Proposition 52. Whenever $\eta \vdash_{\text{s}} P$, there exist no $P_1, P_2, \widetilde{a}$ such that $P \equiv (\boldsymbol{\nu}\widetilde{a})(P_1 \mid P_2)$ with $P_1 \xrightarrow{\tau} P_1'$ and $P_2 \xrightarrow{\tau} P_2'$, and both these transitions are interactions.

$$\textsc{SInpI}\ \frac{1 \vdash_{\mathrm{s}} P}{1 \vdash_{\mathrm{s}} u(\widetilde{a}).\, P} \qquad \textsc{SInpO}\ \frac{1 \vdash_{\mathrm{s}} P}{0 \vdash_{\mathrm{s}} x(\widetilde{a}).\, P,\ !x(\widetilde{a}).\, P} \qquad \textsc{SOutO}\ \frac{0 \vdash_{\mathrm{s}} P}{1 \vdash_{\mathrm{s}} \overline{x}\langle\widetilde{a}\rangle.\, P} \qquad \textsc{SOutI}\ \frac{0 \vdash_{\mathrm{s}} P}{0 \vdash_{\mathrm{s}} \overline{u}\langle\widetilde{a}\rangle.\, P} \qquad \textsc{SRes}\ \frac{\eta \vdash_{\mathrm{s}} P}{\eta \vdash_{\mathrm{s}} (\boldsymbol{\nu} a)P} \qquad \textsc{SNil}\ \frac{}{0 \vdash_{\mathrm{s}} \mathbf{0}}$$

$$\textsc{SPar}\ \frac{\eta_1 \vdash_{\mathrm{s}} P \qquad \eta_2 \vdash_{\mathrm{s}} Q}{\eta_1 + \eta_2 \vdash_{\mathrm{s}} P \mid Q}\ \eta_1 + \eta_2 \leq 1 \qquad \textsc{SSum}\ \frac{\eta \vdash_{\mathrm{s}} G_1 \qquad \eta \vdash_{\mathrm{s}} G_2}{\eta \vdash_{\mathrm{s}} G_1 + G_2} \qquad \textsc{STau}\ \frac{\eta \vdash_{\mathrm{s}} P}{\eta \vdash_{\mathrm{s}} \tau.\, P} \qquad \textsc{SMat}\ \frac{0 \vdash_{\mathrm{s}} G}{0 \vdash_{\mathrm{s}} [a = b]G}$$

Figure 4.1: The typing rules for sequentiality

An inactive process may not perform interactions:

Proposition 53. If $0 \vdash_{\mathrm{s}} P$, then P does not have an interaction redex; i.e, there is no P' with $P \xrightarrow{\tau} P'$ and this transition is an interaction.

An inactive process may however perform τ-reductions, notably to resolve internal choices. In other words such internal choices represent internal matters for a process, orthogonal with respect to the overall interaction thread. The possibility for inactive processes to accommodate internal choices will be important in our completeness proof (Section 4.2.2). However, an inactive process may only perform a finite number of τ-reductions. A process P *is τ-divergent* if it can perform an infinite sequence of reductions, i.e. there are $P_1, P_2, \ldots$, with $P \to P_1 \to P_2 \ldots P_n \to \ldots$.

Proposition 54. If $0 \vdash_{\mathrm{s}} P$ then P is not τ-divergent.

In contrast, an active process may be τ-divergent, through sequences of reductions containing infinitely many interactions.

Sequentiality imposes constraints on the interactions that a 'legal' (i.e. well-typed) context may undertake with a process. For the definition of barbed bisimulation and equivalence we must therefore define the meaning of observability. The following definition of *type-allowed transitions* shows what such 'legal' interactions can be.

Definition 55 (Type-allowed transitions)**.** When $\eta \vdash_{\mathrm{s}} P$, we write $[\eta; P] \xrightarrow{\mu} [\eta'; P']$ if $P \xrightarrow{\mu} P'$ and one of the following holds:

1. $\mu = \tau$ and $\eta' = \eta$
2. $\mu = x\langle\widetilde{a}\rangle$, and $\eta = 0$ and $\eta' = 1$
3. $\mu = u\langle\widetilde{a}\rangle$ and $\eta = \eta' = 1$
4. $\mu = (\boldsymbol{\nu}\widetilde{a})\overline{x}\langle\widetilde{b}\rangle$, and $\eta = 1$ and $\eta' = 0$
5. $\mu = (\boldsymbol{\nu}\widetilde{a})\overline{u}\langle\widetilde{b}\rangle$, and $\eta = \eta' = 0$

Note that in Clause 1. above, in the case of an interaction, we necessarily have $\eta = 1$.

This definition encompasses both which transitions are allowed and what the resulting type should be. In fact, omitting the resulting type, we have that transitions are allowed if one of the following clauses holds:

(a) $\eta = 0$ (Clauses 2. and 5.)

(b) $\mu = \tau$ (Clause 1.)

(c) $\eta = 1$ and $\mu = u\langle\widetilde{a}\rangle$ for some $u, \widetilde{a}$ or $\mu = (\boldsymbol{\nu}\widetilde{a})\overline{x}\langle\widetilde{b}\rangle$ for some $\widetilde{a}, x, \widetilde{b}$. (Clauses 3. and 4.)

Clause (a) says that all interactions between an inactive process and the context are possible; this holds because the context is active and may therefore decide on the next interaction with the process. Clause (b) says that internal reductions may always be performed. Clause (c) says that the only visible

actions observable in active processes are those carrying the thread; this holds because the observer is inactive, and it is therefore up to the process to decide on the next interaction.

Similarly, the resulting type η' only changes when the subject is an output-controlled name. We have the following:

(a) if $\mu = x\langle\widetilde{a}\rangle$, then $\eta' = 1$. (Clause 2.)

(b) if $\mu = (\boldsymbol{\nu}\widetilde{a})\overline{x}\langle\widetilde{b}\rangle$, then $\eta' = 0$. (Clause 4.)

(c) otherwise $\eta' = \eta$. (Clauses 1., 3. and 5.)

In all three cases, the resulting process P' can indeed be typed with η'.

Theorem 56 (Subject Reduction)**.** If $\eta \vdash_{\mathrm{s}} P$ and $[\eta; P] \xrightarrow{\mu} [\eta'; P']$ then $\eta' \vdash_{\mathrm{s}} P'$.

Weak type-allowed transition are defined as expected, exploiting the invariance of typing under reductions: $[\eta; P] \stackrel{\mu}{\Rightarrow} [\eta'; P']$ holds if there are P_0, P_1 with $P \Rightarrow P_0$, $[\eta; P_0] \xrightarrow{\mu} [\eta'; P_1]$ and $P_1 \Rightarrow P'$.

4.2 Behavioural equivalence

4.2.1 Sequential bisimilarity

To tune Definition 21 of barbed bisimulation and equivalence to the setting of sequentiality, we have to specify the meaning of observables. An observable $\eta \vDash_{\mathrm{s}} P \Downarrow_\alpha$ holds if there are P' and an action μ such that $[\eta; P] \stackrel{\mu}{\Rightarrow} [\eta'; P']$ with $\alpha = a$ if μ is an input at a and $\alpha = \overline{a}$ if μ is an output at a. Following Definition 20, in barbed equivalence, the legal contexts are the η/η' static contexts. We write barbed equivalence at η as $\simeq^{\eta}$, and sometimes call it *sequential barbed equivalence at* η. Thus $P \simeq^{\eta} Q$ holds if $\eta \vdash_{\mathrm{s}} P, Q$ and $E[P] \dot{\approx}^{\eta'} E[Q]$, for any η' and any η'/η static context E.

About the reduction-closed variant of barbed equivalence. The reason why completeness proofs for reduction-closed barbed equivalence may be simpler than with standard barbed equivalence is that the testing context may be incrementally adjusted, after every interaction step with the tested processes. This however requires the existence of special components in the contexts to handle the fresh names generated by the tested processes. Specifically, the task of these components is to ensure that new pieces of contexts, added later, will be able to access such fresh names. Thus, we need to prove lemmas like Lemma 8. However, these components represent parallel threads, and break the sequentiality disciplines. For this reason in the section we cannot appeal to reduction-closed forms of barbed equivalence, remaining within the standard notions and therefore requiring an image-finiteness condition.

We are now ready to define the labelled bisimilarity to be used on sequential processes, which is our main proof technique for barbed equivalence. A *typed process relation* is a set of triplets (η, P, Q) with $\eta \vdash_{\mathrm{s}} P, Q$.

Definition 57 (Sequential Bisimulation)**.** A typed process relation $\mathcal{R}$ is a *sequential bisimulation* if whenever $(\eta, P, Q) \in \mathcal{R}$ and $[\eta; P] \xrightarrow{\mu} [\eta'; P']$, there is Q' such that $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $(\eta', P', Q') \in \mathcal{R}$. Moreover, the converse holds on the transitions from Q. Processes P and Q are *sequentially bisimilar at* η, written $P \approx_{\mathrm{s}}^{\eta} Q$, if $(\eta, P, Q) \in \mathcal{R}$ for some sequential bisimulation $\mathcal{R}$.

Ordinary bisimilarity, restricted to processes having the same typing, is included in the sequential one. So it is also a proof technique for sequential barbed equivalence at η. Indeed, from any bisimulation, we may construct a sequential bisimulation by taking the related pairs of well-typed processes in relation.

Proposition 58. For $\eta \vdash_{\mathrm{s}} P, Q$, if $P \approx Q$ then also $P \approx_{\mathrm{s}}^{\eta} Q$.

The inclusion is strict: all the examples in Section 4.2.3 fail to hold for $\approx$.

Theorem 59 (Soundness)**.** If $P \approx_{\mathrm{s}}^{\eta} Q$, then $P \simeq^{\eta} Q$.

As usual, the proof of Theorem 59 relies on the preservation of $\approx_{\mathrm{s}}^{\eta}$ under parallel composition, which requires some care in order to enforce sequentiality. This is ensured by typability. Theorem 59 allows us to use the labelled bisimilarity $\approx_{\mathrm{s}}^{\eta}$ as a proof technique for typed barbed equivalence.

This proof technique is also complete, assuming only output-controlled names (i.e. the thread may only be exercised by output particles, not by the input ones), as we show below. The case with input-controlled names is open.

4.2.2 Completeness for output-controlled names

In this section, we will only consider processes without input-controlled names. For the proof of completeness, we introduce the notion of *singular* process. Intuitively, a singular process always keeps the thread. Formally, the set of singular processes is the largest set $\mathcal{T}$ of processes such that for all $P \in \mathcal{T}$, we have $1 \vdash_{\mathrm{s}} P$ and whenever $[1; P] \xrightarrow{\mu} [\eta; P']$, then $P' \in \mathcal{T}$.

A simple example of singular process is $\mathbf{0}_1 \stackrel{\text{def}}{=} (\boldsymbol{\nu} z)\overline{z}$, which is an active process without transition. For processes with only output-controlled names, singular processes can be characterised in various ways:

Lemma 60. For a process P with only output-controlled names, the following clauses are equivalent:

1. P is singular.
2. $P \approx_{\mathrm{s}}^{1} \mathbf{0}_1$.
3. For all static 1/1 context E, we have $E[P] \approx_{\mathrm{s}}^{1} P$.
4. $1 \vdash_{\mathrm{s}} P$ and for all α, $1 \vdash_{\mathrm{s}} P \not\Downarrow_{\alpha}$.

Proof.

- 1 $\implies$ 2: Taking $\mathcal{T}$ the set of singular processes, we have that $\mathcal{R} = \{(1, P, \mathbf{0}_1) \mid P \in \mathcal{T}\}$ is a sequential bisimulation. Indeed, on one side, $\mathbf{0}_1$ has no transition. On the other side, if $[\eta; P] \xrightarrow{\mu} [\eta'; P']$, then by definition $\eta = \eta' = 1$ and $P' \in \mathcal{T}$. As P only contains output-controlled names, we must also have that $\mu = \tau$. We can then conclude as $\mathbf{0}_1 \Rightarrow \mathbf{0}_1$.

- 2 $\implies$ 3: Using the above implication, it is sufficient to show that $E[\mathbf{0}_1]$ is singular. Then we have $E[\mathbf{0}_1] \approx_{\mathrm{s}}^{1} \mathbf{0}_1$ and $E[P] \approx_{\mathrm{s}}^{1} E[\mathbf{0}_1] \approx_{\mathrm{s}}^{1} \mathbf{0}_1 \approx_{\mathrm{s}}^{1} P$ as $\approx_{\mathrm{s}}$ is a congruence.

  We show that $\mathcal{T} \stackrel{\text{def}}{=} \{E[\mathbf{0}_1] \mid E \text{ is a } 1/1 \text{ static context}\}$ is a set of singular processes. Writing E as $(\boldsymbol{\nu}\widetilde{a})([\cdot] \mid T)$, we have that $0 \vdash_{\mathrm{s}} T$. Thus, when $[1; E[\mathbf{0}_1]] \xrightarrow{\mu} [\eta; Q]$, by looking at the LTS, we must have that $(\boldsymbol{\nu}\widetilde{a})T \xrightarrow{\mu} (\boldsymbol{\nu}\widetilde{a'})T'$ with $Q = (\boldsymbol{\nu}\widetilde{a'})(\mathbf{0}_1 \mid T')$, meaning $Q \in \mathcal{T}$. As by Theorem 56, we also have $1 \vdash_{\mathrm{s}} Q$, we can conclude.

- 3 $\implies$ 4: First, we have $1 \vdash_{\mathrm{s}} P$. We then reason by contradiction. If $P \Downarrow_{\alpha}$ then we must have $\alpha = \overline{x}$. Take $E = x(\widetilde{a}).\,\overline{y} \mid \mathbf{0}$ for a fresh name y. We have $E[P] \Downarrow_{\overline{y}}$ while $P \not\Downarrow_{\overline{y}}$. Thus, we have $E[P] \not\mathrel{\dot{\approx}}^{1} P$ which implies $E[P] \not\approx_{\mathrm{s}}^{1} P$.

- 4 $\implies$ 1: We show that the set $\mathcal{T} \stackrel{\text{def}}{=} \{P' \mid P \Rightarrow P'\}$ is a set of singular processes. Notice that for all P' with $P \Rightarrow P'$, we have $1 \vdash_{\mathrm{s}} P'$ and for all α, $P' \not\Downarrow_{\alpha}$.

  Take $P' \in \mathcal{T}$ and $[1; P'] \xrightarrow{\mu} [\eta; Q]$. As P' has no barbs, we have $\mu = \tau$. Therefore $\eta = 1$ and $P \Rightarrow P' \to Q$ so $Q \in \mathcal{T}$.

□

This lemma suffices to show that the set of singular processes is closed by static context and by $\dot{\approx}^{1}$. However, this result does not extend to general processes. For instance, we have $u.\,\mathbf{0}_1 \not\approx_{\mathrm{s}}^{1} \mathbf{0}_1$ despite $u.\,\mathbf{0}_1$ being singular.

We can now describe the completeness proof. While the overall structure of the proof is standard, the technical details are specific to sequentiality. As usual, we rely on a stratification of bisimilarity and approximants.

Definition 61 (Approximants of sequential bisimilarity)**.** We define a sequence $(\approx_{\mathrm{s}}^{\eta;n})_{n\geq 0}$ of typed relations:

1. $P \approx_{\mathrm{s}}^{\eta;0} Q$ if $\eta \vdash_{\mathrm{s}} P, Q$.
2. For $1 \leq i$, relation $\approx_{\mathrm{s}}^{\eta;i}$ is defined by: $P \approx_{\mathrm{s}}^{\eta;i} Q$ if whenever $[\eta; P] \xrightarrow{\mu} [\eta'; P']$, there is Q' such that $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $P' \approx_{\mathrm{s}}^{\eta';i-1} Q'$; and symmetrically for the transitions of Q.
3. Then $P \approx_{\mathrm{s}}^{\eta;\omega} Q$ if $P \approx_{\mathrm{s}}^{\eta;i} Q$ for all i.

Notice that $\approx_{\mathrm{s}}^{\eta;0} \supseteq \approx_{\mathrm{s}}^{\eta;1} \supseteq \cdots \supseteq \approx_{\mathrm{s}}^{\eta;n} \supseteq \cdots \supseteq \approx_{\mathrm{s}}^{\eta;\omega} \supseteq \approx_{\mathrm{s}}^{\eta}$.

On image-finite processes (Definition 9), sequential bisimilarity, $\approx_{\mathrm{s}}^{\eta}$, coincides with the intersection of its approximants.

Lemma 62. If P, Q are image-finite, $P \approx_{\mathrm{s}}^{\eta;\omega} Q$ iff $P \approx_{\mathrm{s}}^{\eta} Q$.

Because of the sequentiality constraints, if $1 \vdash_{\mathrm{s}} P$ and $0 \vdash_{\mathrm{s}} R$, whenever $1 \vDash_{\mathrm{s}} P \mid R \downarrow_{\alpha}$, we must have that $1 \vDash_{\mathrm{s}} P \downarrow_{\alpha}$. When comparing active processes, the context is not and so cannot use barbs as easily to monitor its interactions with process. For this purpose, we introduce the notation $S \rhd z$ for $\sum_{s\in S} s(\widetilde{b}).\,\overline{z}$ where z is a name and S is a set of names. As we only consider output-controlled names in this section, this means that $s, s', \ldots$ are output-controlled (in addition to $x, y, z, \ldots$).

The main ingredient for proving completeness is then given by the following proposition.

Proposition 63. Given $n \geq 0$, suppose $\eta \vdash_{\mathrm{s}} P, Q$, and $P \not\approx_{\mathrm{s}}^{\eta;n} Q$. Suppose further that both P and Q only use output-controlled names and are image-finite. Then there exists a guarded process R such that for any $\widetilde{a}, S$ with $\widetilde{a} \subseteq \mathrm{fn}(P) \cup \mathrm{fn}(Q) \subseteq S$ and any fresh names z, w, if we define

$$\overline{R} \stackrel{\mathrm{def}}{=} \begin{cases} R + \overline{z} & \text{when } 0 \vdash_{\mathrm{s}} P, Q \\ R + S \rhd z + S \rhd w & \text{when } 1 \vdash_{\mathrm{s}} P, Q \end{cases}$$

and $E_0 \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid \overline{R})$, we have the following:

1. either $E_0[P'] \not\dot{\approx}^1 E_0[Q]$ for all P' such that $P \Rightarrow P'$
2. or $E_0[P] \not\dot{\approx}^1 E_0[Q']$ for all Q' such that $Q \Rightarrow Q'$.

We present here the main aspects which are specific to the case of sequential bisimilarity before giving the proof.

We reason by contradiction to establish that $E_0[P]$ and $E_0[Q]$ are not barbed bisimilar. To respect typing, the definition of E_0 requires some care.

The case when $\eta = 0$ (tested processes are inactive) is rather standard: the context E_0 is of the form $(\boldsymbol{\nu}\widetilde{a})([\cdot] \mid R + \overline{z})$, for some fresh z, and some "tester process" R. The barb at z allows us to detect when the tested process interacts with R.

The delicate case is when $\eta = 1$ (tested processes are active): the context must be inactive and hence cannot have an unguarded output at z. We use in this case a 1/1 context of the form $E_0 \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid R + G)$. Process R is the tester process, and G is $S \rhd z + S \rhd w$, defined for some fresh z, w and some set S containing $\mathrm{fn}(P) \cup \mathrm{fn}(Q)$. G satisfies the following property: for any P_0, and for any x, if $1 \vDash_{\mathrm{s}} P_0 \Downarrow_{\overline{x}}$ then $1 \vDash_{\mathrm{s}} P_0 \mid G \Downarrow_{\overline{z}}$ and $1 \vDash_{\mathrm{s}} P_0 \mid G \Downarrow_{\overline{w}}$. Thus, as soon as P_0 exhibits some barb, P_0 cannot interact with R without removing the barb at z or at w, which allows us to reason as in the case $\eta = 0$.

The proof schema above relies on having P_0 active and exhibiting a barb, in order to be able to detect the interactions. This condition is not fulfilled for singular processes, which must be handled separately. Since we restrict to output-controlled names, we can use Lemma 60. If on the contrary input-controlled names are allowed, a process may exhibit a barb while being singular, which prevents us from finding a G with the same property and thus using the same reasoning.

Proof. We reason by induction on n. For $n = 0$, there is nothing to prove.

For $n > 0$, suppose that $P \not\approx_{\mathrm{s}}^{\eta;n} Q$. We may therefore suppose that there exists μ such that $[\eta; P] \xrightarrow{\mu} [\eta'; P']$ and for all Q' with $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$, we have $P' \not\approx_{\mathrm{s}}^{\eta';n-1} Q'$. We note $\{Q_i\}$, for $i \in I$, the set of all such Q'. This set is finite by hypothesis. We also write $S' = \bigcup_{i\in I} \mathrm{fn}(Q_i) \cup \mathrm{fn}(P')$, which is also a finite set.

We show that clause 2. holds (in the symmetrical case, where we consider a transition from $[\eta; Q]$, we establish the first clause).

We first consider the case where P' is singular. The reasoning is simpler in this case.

First case: P' is singular.

Since P' is singular, $\eta' = 1$. For all i, as $P' \not\approx^1_{\mathrm{s}} Q_i$, we have $Q_i \Downarrow_{\overline{z_i}}$ for some z_i. By definition, S' contains all such z_i's.

We consider the transition from P to P'. By Theorem 56 (Subject Reduction), since P' is singular, P' is active, so μ cannot be an output. Hence there are two possible cases:

- when $\mu = \tau$, we can take
$$R \stackrel{\text{def}}{=} \mathbf{0}.$$
We then have to show that for any $z, w, \widetilde{a}, S, Q'$ with $Q \Rightarrow Q'$, taking $\overline{R} = \mathbf{0} + S \rhd z + S \rhd w$ (since $\eta = 1$) and $E_0 = (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid \overline{R})$, we have $E_0[P] \dot{\not\approx}^1 E_0[Q']$. We reason by contradiction. As we have $E_0[P] \to E_0[P']$, there exists a process T such that $E_0[Q'] \Rightarrow T$ and $E_0[P'] \dot{\approx}^1 T$.
Since $E_0[Q'] \Rightarrow T$, T can be of 3 forms, up to $\equiv$:
  - $E_0[Q_i]$ for some $i \in I$ (with $Q' \Rightarrow Q_i$);
  - $(\boldsymbol{\nu}\widetilde{a})(Q'' \mid \overline{z})$.
  - $(\boldsymbol{\nu}\widetilde{a})(Q'' \mid \overline{w})$.

  $E_0[P']$ is singular by Lemma 60 and thus as no barb. On the other hand, for all i, as $\mathrm{fn}(Q_i) \subseteq S' \subseteq S$, we have $E_0[Q_i] \Downarrow_{\overline{z}}$ and $E_0[Q_i] \Downarrow_{\overline{w}}$. All forms of T have a barb, so T cannot be barbed bisimilar to $E_0[P']$.

- when $\mu = x\langle\widetilde{b}\rangle$, given a fresh name z', we define
$$R \stackrel{\text{def}}{=} \overline{x}\langle\widetilde{b}\rangle.\, S' \rhd z'.$$
We then have to show that for any $z, \widetilde{a}, S, Q'$, with $Q \Rightarrow Q'$,
$$E_0[P] \dot{\not\approx}^1 E_0[Q'],$$
by setting $E_0 = (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid R + \overline{z})$, since in this case $\eta = 0$ (by Theorem 56).
We write $E_1 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{a}, \widetilde{b})([\cdot] \mid S' \rhd z')$. As before, we have $E_0[P] \to E_1[P']$ meaning there should exist T such that $E_0[Q'] \Rightarrow T$ and $E_1[P'] \dot{\approx}^1 T$.
We observe that T can be of 3 forms:
  - $E_0[Q'']$ (with $Q' \Rightarrow Q''$) that has a barb at $\overline{z}$,
  - $E_1[Q_i]$ for some $i \in I$ (with $Q' \xrightarrow{x\langle\widetilde{b}\rangle} Q_i$)
  - $(\boldsymbol{\nu}\widetilde{a})(Q'' \mid \overline{z'})$

  By Lemma 60, $E_1[P']$ has no barb. As $\mathrm{fn}(Q_i) \subseteq S'$, we have $E_1[Q_i] \Downarrow_{\overline{z'}}$, meaning all forms of T have a barb, hence a contradiction.

Second case: P' is not singular. By Lemma 60, we know that either P' is inactive, or $P' \Downarrow_{\overline{z}}$ for some z. As above, we rely on Theorem 56 to reason by cases according to the transition from P to P'.

We note R_i for the process that we obtain, by induction, for each pair (P', Q_i).

We first show that for any $\widetilde{z_i}, \widetilde{w_i}, \widetilde{a}, S, Q_i$ with $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q_i$, taking $\overline{R_i}$ accordingly with η' so
$$\overline{R_i} \stackrel{\text{def}}{=} \begin{cases} R_i + \overline{z_i} & \text{when } 0 \vdash_{\mathrm{s}} P', Q_i \\ R + S' \rhd z_i + S' \rhd w_i & \text{when } 1 \vdash_{\mathrm{s}} P', Q_i \end{cases}$$
and $F = (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid \sum_{i \in I} \tau.\, \overline{R_i})$, we have
$$F[P'] \dot{\not\approx}^1 F[Q_i] \qquad (\star)$$

We only do the proof when $1 \vdash_{\mathrm{s}} P', Q_i$, the other case being simpler. First, as P' is not singular, we have $F[P'] \Downarrow_{\overline{z_i}}$ for all i, and similarly for w_i. Thus, the same holds for $F[Q_i]$.

We have two cases depending on the clause of the proposition that holds, by induction, for (P', Q_i):

1. if clause 1. holds, then $F[Q_i] \to E_i[Q_i]$ with

$$E_i \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid \overline{R_i})$$

This means there is T with $F[P'] \Rightarrow T$ and $T \stackrel{\cdot}{\approx}^1 E_i[Q_i]$.

As we have $E_i[Q_i] \Downarrow_{z_i}$, T can only be of 3 forms:

- either $F[P'']$ with $P' \Rightarrow P''$,
- or $T \equiv E_i[P'']$ with $P' \Rightarrow P''$,
- or $T \equiv (\boldsymbol{\nu}\widetilde{a})(P'' \mid \overline{z_i})$ for some P''.

The first case is not possible as $E_i[Q_i] \not\Downarrow_{z_j}$ for $j \neq i$, and the third as $E_i[Q_i] \Downarrow_{w_i}$. This only leaves the second case.

Up to structural congruence (which is included in $\stackrel{\cdot}{\approx}^1$), we can suppose that in $E_i[P'']$ and $E_i[Q_i]$, $\widetilde{a}$ contains only names from $\text{fn}(P', Q_i)$. We are in a situation where we can apply the induction hypothesis, and this is in contradiction with $E_i[P''] \stackrel{\cdot}{\approx}^1 E_i[Q_i]$ as $P' \Rightarrow P''$.

2. if clause 2. holds, we have similarly that $F[P'] \to E_i[P']$. The same reasoning, with the role of P' and Q_i swapped, is sufficient to conclude.

We can now prove the desired result by looking at the possible cases for μ:

- when $\mu = \tau$ and $1 \vdash_{\text{s}} P'$, given fresh names $(z_i)_{i\in I}$, we define

$$R \stackrel{\text{def}}{=} \tau.\sum_{i\in I} \tau.\,(R_i + S' \rhd z_i + S' \rhd w_i),$$

and $E_0 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid R + S \rhd z + S \rhd w)$.

We reason by contradiction to show that for all $z, w, \widetilde{a}, S, Q'$ with $Q \Rightarrow Q'$, we have $E_0[P] \not\stackrel{\cdot}{\approx}^1 E_0[Q']$.

We have $E_0[P] \to\!\!\to F[P']$, so there should exist T such that $E_0[Q'] \Rightarrow T$ and $F[P'] \stackrel{\cdot}{\approx}^1 T$.

As P' is active and not singular, by Lemma 60, that $P' \Downarrow_{\overline{z'}}$ for some $z' \in S'$ (P' is active because $1 \vdash_{\text{s}} P'$).

So $F[P'] \Downarrow_{\overline{z_i}}$ for all $i \in I$ but not $F[P'] \Downarrow_{\overline{z}}$. Therefore B' can be of 2 forms:

- $E_0[Q_i]$ for some $i \in I$ (with $Q' \Rightarrow Q_i$).
- $F[Q_i]$ for some $i \in I$ (with $Q' \Rightarrow Q_i$).

However, as $S' \subseteq S$, if $E_0[Q_i] \Downarrow_{\overline{z_i}}$, then we must have $E_0[Q_i] \Downarrow_{\overline{z}}$ which rules out the first case.

Thus, $F[P'] \stackrel{\cdot}{\approx}^1 F[Q_i]$ which is absurd by $(\star)$.

- when $\mu = (\boldsymbol{\nu}\widetilde{b^2})\overline{x}\langle\widetilde{b^2}, \widetilde{b^1}\rangle$, given fresh names $z', z'', (z_i)_{i\in I}$, we note $\widetilde{b^2} = b_1^2, \ldots, b_n^2$ and $\widetilde{b^1} = b_1^1, \ldots b_m^1$. Then we set:

$$R \stackrel{\text{def}}{=} x(\widetilde{c^2}, \widetilde{c^1}).\,\big(\,\overline{z'} + G + [c_1^1 = b_1^1] \ldots [c_m^1 = b_m^1]\tau.\,(\sum_{i\in I} \tau.\,(R_i + \overline{z_i}))\,\big)$$

$$G \stackrel{\text{def}}{=} \sum_{j\leq n} \sum_{b\in \text{fn}(P,Q)} [c_j^2 = b]\overline{z''}$$

and $E_0 = (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid R + S \rhd z + S \rhd w)$.

Then $E_0[P] \to A$ with

$$A \equiv (\boldsymbol{\nu}\widetilde{a}, \widetilde{b^2})(P' \mid \overline{z'} + G + \tau.\,(\sum_{i\in I} \tau.\,(R_i + \overline{z_i})))$$

Thus there exists T such that $E_0[Q] \Rightarrow T$ and $A \mathrel{\dot{\approx}}^1 T$. The process A has a weak barb at $\overline{z'}$, $\overline{z_i}$ for all $i \in I$ but not at $\overline{z}$ nor $\overline{z''}$. Thus, T can only be of the form $(\boldsymbol{\nu}\widetilde{a}, \widetilde{b^2})(Q_i \mid \overline{z'}+G+\tau.(\sum_{i\in I}\tau.(R_i+\overline{z_i})))$ for some $i \in I$ (meaning that $Q' \stackrel{\mu}{\Rightarrow} Q_i$).

As $A \to F[P']$ with $F[P'] \Downarrow_{\overline{z_i}}$ for all $i \in I$ but not $F[P'] \Downarrow_{\overline{z'}}$, we must have that $T \Rightarrow F[Q_{i'}]$ with $Q_i \Rightarrow Q_{i'}$ and $F[P'] \mathrel{\dot{\approx}}^1 F[Q_{i'}]$. This is absurd by $(\star)$.

The reasoning in the remaining cases is very similar to the proofs given above. We only sketch how the proof can be derived.

- when $\mu = x\langle\widetilde{b}\rangle$, P is inactive, and P' is active. Given fresh names $(z_i, w_i)_{i\in I}$, we set

$$R \stackrel{\text{def}}{=} \overline{x}\langle\widetilde{b}\rangle.\sum_{i\in I}\tau.(R_i + S' \rhd z_i + S' \rhd w_i).$$

  We have $E_0[P] \to F[P']$. As P' is not singular, we can conclude with a similar proof.

- when $\mu = \tau$ and $0 \vdash_{\mathrm{s}} P'$, given fresh names $(z_i)_{i\in I}$, we pose

$$R \stackrel{\text{def}}{=} \sum_{i\in I}\tau.(R_i + \overline{z_i})$$

  and we can conclude.

□

Theorem 64 (Completeness on output-controlled names)**.** For all image-finite processes P, Q that only use output-controlled names, and for all η, if $P \simeq^{\eta} Q$ then $P \approx^{\eta}_{\mathrm{s}} Q$.

Proof. We suppose $P \simeq^{\eta} Q$ and P, Q are image-finite.
We reason by contradiction:
If $P \not\approx^{\eta}_{\mathrm{s}} Q$, then by Lemma 62, $P \not\approx^{\eta;\omega}_{\mathrm{s}} Q$.
Thus there exists n such that $P \not\approx^{\eta;n}_{\mathrm{s}} Q$.
By Proposition 63, there exists some static context E such that $E[P] \not\mathrel{\dot{\approx}}^{\eta} E[Q]$.
So $P \not\simeq^{\eta} Q$, which is absurd. □

Remark 65. We believe that the same result could be obtained if input-controlled names were allowed but not free at any point. Intuitively, the reasoning for the proof fails when the process is able to perform visible transitions at input-controlled name. However, allowing internal communication on input-controlled seems to not influence the result. For instance, in the encoding of $\lambda^{\mathtt{ref}}$ from Chapter 6 where references are encoded as input-controlled names, no such name is free, even after performing transitions. A proof of this extension would require restricting the sorting so that output-controlled names may only send other output-controlled names, and only look at processes with no free input-controlled names.

On the definition of barbs Using our general definition for typed barbs, an active process P can have two kinds of barbs: $1 \vDash_{\mathrm{s}} P \Downarrow_{\overline{x}}$ and $1 \vDash_{\mathrm{s}} P \Downarrow_{u}$. Barbs for inputs are needed to distinguish the processes $u.\,\mathbf{0}_1$ and $\mathbf{0}_1$ which are not sequentially bisimilar. However, it may seem unnatural to be able to observe these inputs. Although the transition is allowed by the typing, it implies that the environment is able to observe the input at u despite being inactive both before and after the communication. Apart from breaking the completeness, the results we have shown in this section still hold if we forbid to observe inputs at input-controlled names.

The result above can be adapted to the Asynchronous π-calculus in which it was originally presented [17]. In Aπ, as inputs are not considered for barbs, the discussion above no longer applies. In that setting, the completeness is still open and additional laws can be proved, e.g. $u.\,v.\,P$ and $v.\,u.\,P$ are equivalent.

4.2.3 Examples

Example 66. An unguarded occurrence of an input at an input-controlled name becomes the only observation that can be made in a process. This yields the following equalities

$$\begin{array}{lcll} u.\,P \mid x.\,Q & \approx_{\mathrm{s}}^{1} & u.\,(P \mid x.\,Q) & \\ u.\,P \mid \overline{v}.\,Q & \approx_{\mathrm{s}}^{1} & u.\,(P \mid \overline{v}.\,Q) & \text{for } u \neq v \\ \overline{x}.\,P \mid \overline{u}.\,Q & \approx_{\mathrm{s}}^{1} & \overline{x}.\,(P \mid \overline{u}.\,Q) & \\ \overline{x}.\,P \mid y.\,Q & \approx_{\mathrm{s}}^{1} & \overline{x}.\,(P \mid y.\,Q) & \text{for } x \neq y \end{array}$$

For the first equality, one shows that

$$\{(1, u.\,P \mid x.\,Q, u.\,(P \mid x.\,Q))\} \cup id\,,$$

where id is the (typed) identity relation, is a sequential bisimulation. One proves the other equalities similarly.

By sequentiality, it is also not possible to access parts of processes that would require a simultaneous/parallel reception. The following example illustrates this.

Example 67. Consider the process

$$P \stackrel{\text{def}}{=} (\boldsymbol{\nu} z')(!x.\,(z'.\,\overline{z} \mid \mathbf{0}_1) \mid !y.\,\overline{z'}).$$

The output at z becomes observable if both an input at x and an input at y are consumed, so that the internal reduction at z' can take place. However the input at x acquires the thread, preventing a further immediate input at y; similarly for the input at y. Indeed we have $P \approx_{\mathrm{s}}^{0} x.\,\mathbf{0}_1 + y.\,\mathbf{0}_1 \stackrel{\text{def}}{=} Q$ (we recall that $\mathbf{0}_1 = (\boldsymbol{\nu} z)\overline{z}$). To prove the equality, we can use $\{P, Q\} \cup \approx_{\mathrm{s}}^{1}$, which is easily proved to be a sequential bisimulation. The derivatives of P and Q are singular (and stable, i.e. unable to reduce further) processes; therefore they are in $\approx_{\mathrm{s}}^{1}$, as discussed in Lemma 60.

Under the ordinary bisimilarity, P and Q are distinguished because the sequence of transitions $P \xrightarrow{x} \xrightarrow{y} \xrightarrow{\tau} \xrightarrow{\overline{z}}$ cannot be matched by Q.

Example 68. This example informally discusses why sequentiality can help reducing the number of pairs of processes to examine in a bisimulation proof. Suppose we wish to prove the equality between the two processes

$$\begin{array}{lcl} P_1 & \stackrel{\text{def}}{=} & \overline{x} \mid y.\,R \mid u.\,Q \\ P_2 & \stackrel{\text{def}}{=} & \overline{x} \mid u.\,Q \mid y.\,R \end{array}$$

and suppose such processes are typable under the sequentiality system. The equivalence between P_1 and P_2 comes from a commutativity of parallel composition. We may therefore use ordinary bisimilarity, as this is a sound proof technique for typed barbed equivalence. One may prove, more generally, that parallel composition is commutative and derive the equality. However suppose we wish to use the bisimulation method, concretely, on P and Q. The two processes have 3 initial transitions (with labels $\overline{x}, y$, and u), and the subtrees of transitions emanating from such derivatives have similar size. Under ordinary bisimulation, we have to examine all the states in the 3 subtrees.

Under sequential bisimulation, however, only the input at y is observable (it is the only one carrying the thread), thus removing 2 of the 3 initial subtrees. Further pruning may be possible later on, again exploiting the fact that under sequentiality only certain transitions are observable.

4.3 Sequential references

4.3.1 Combining sequentiality with the type system for references

The two type systems we have studied so far can be combined to define a typed π-calculus containing references and sequentiality. As these systems are quite independent, the resulting definitions of equivalence are constructed naturally from their respective definitions in Chapter 3 and Section 4.2. We briefly explain how they are defined before showing some examples.

Since we use the setting of Chapter 3, we work with the Asynchronous π-calculus, and also assume the calculus to be monadic for simplicity. We assume that all references names are input-controlled, as a read or write – i.e. an action requiring the thread – starts by an input. We write $\Delta; \eta \vdash_{\mathrm{rs}} P$ when P can be typed with Δ using the type system for references, and with η using the type system for sequentiality. Thus, $\Delta; \eta \vdash_{\mathrm{rs}} P$ holds when both $\Delta \vdash_{\mathrm{r}} P$ and $\eta \vdash_{\mathrm{s}} P$ hold. Notice that $\Delta; \eta \vdash_{\mathrm{rs}} P$ could also be defined inductively using inference rules in a simple manner. For instance, we would have:

$$\frac{\ell; 1 \vdash_{\mathrm{rs}} P}{\emptyset; 1 \vdash_{\mathrm{rs}} \ell(n).\, P}$$

Barbed bisimulation and barbed equivalence are defined as in Chapter 3 using complete processes and completing contexts typed using both systems. We also impose the typing conditions from both systems on barbs. As the calculus is asynchronous, barbs may only be of the form $\Delta, \eta \vDash_{\mathrm{rs}} P \Downarrow_{\overline{a}}$, with a being output-controlled when $\eta = 1$ and input-controlled when $\eta = 0$.

Definition 69 (Barbed bisimulation, equivalence)**.** *Barbed* $\Delta; \eta$*-bisimulation* is the largest relation $\dot{\approx}^{\Delta;\eta}$ on $\Delta; \eta$-processes s.t. for all complete processes P, Q, $P \dot{\approx}^{\Delta;\eta} Q$ implies:

1. whenever $P \to P'$ then there exists Q' such that $Q \Rightarrow Q'$ and $P' \dot{\approx}^{\Delta;\eta} Q'$;
2. for each name a, $\Delta; \eta \vDash_{\mathrm{rs}} P \Downarrow_{\overline{a}}$ iff $\Delta; \eta \vDash_{\mathrm{rs}} Q \Downarrow_{\overline{a}}$.

and symmetrically for the transitions from Q.

Barbed equivalence, noted $\simeq^{\Delta;\eta}$ is the largest barbed bisimulation that is a congruence for static contexts.

Here we only describe a sound bisimulation technique. As both sections use both variants of barbed equivalence, it is not obvious how to define a complete bisimulation in that case. Nevertheless, the technique we describe is sufficient to prove equivalences between programs which only hold in presence of both disciplines, sequentiality and references. We use a variant of the bisimilarity with inductive predicate. The predicate itself is the same as in Section 3.2.3, only adapted to record the additional typing information for $\Delta'; \eta' \vdash_{\mathrm{rs}} P' \mathcal{R} Q'$.

Definition 70 (Sequential Bisimilarity with inductive predicate, $\approx_{\mathrm{rs}}$)**.** We write $\approx_{\mathrm{rs}}^{\Delta;\eta}$ for the largest relation where whenever $P \approx_{\mathrm{rs}}^{\Delta;\eta} Q$ and $[\Delta; \eta; P] \xrightarrow{\mu} [\Delta'; \eta'; P']$, we can derive $\mathtt{ok}(\Delta \cup \Delta', \eta', \mathcal{R}, P', Q, \mu)$ and symmetrically for Q.

Lemma 71 (Soundness)**.** If $P \approx_{\mathrm{rs}}^{\Delta;\eta} Q$, then $P \simeq^{\Delta;\eta} Q$.

4.3.2 Examples

We recall the notations to describe operations on references:

$$\ell \triangleleft n.\, P \stackrel{\mathrm{def}}{=} \ell(_).\, (\overline{\ell}\langle n\rangle \mid P) \qquad \ell \triangleright (m).\, P \stackrel{\mathrm{def}}{=} \ell(m).\, (\overline{\ell}\langle m\rangle \mid P) \qquad \ell \bowtie n(m).\, P \stackrel{\mathrm{def}}{=} \ell(m).\, (\overline{\ell}\langle n\rangle \mid P)$$

In both examples, we will take an arbitrary process R satisfying $\emptyset; 1 \vdash_{\mathrm{rs}} R$.

Example 72. This example shows that reading or writing on a free reference is not subject to interferences from the outside, as these operations require the thread:

$$\begin{array}{rcl} \overline{\ell}\langle n\rangle \mid \ell \triangleright (m).\, R & \approx_{\mathrm{rs}}^{\ell;1} & \overline{\ell}\langle n\rangle \mid R\{n/m\} \\ \overline{\ell}\langle n\rangle \mid \ell \triangleleft m.\, R & \approx_{\mathrm{rs}}^{\ell;1} & \overline{\ell}\langle m\rangle \mid R \\ \overline{\ell}\langle n\rangle \mid \ell \bowtie n'(m).\, R & \approx_{\mathrm{rs}}^{\ell;1} & \overline{\ell}\langle n'\rangle \mid R\{n/m\} \end{array}$$

Indeed, in each law, with P (resp. Q) being the process on the left-hand (resp. right-hand) side, we have $P \to Q$ and this transition is the only one allowed for P.

These laws do not hold for previous equivalences, take $R = \overline{m}$:

- Without the typing for references, P may perform an input at ℓ. Thus for instance, the first law does not hold as $[1; P] \xrightarrow{\ell\langle m'\rangle} \xrightarrow{\overline{m'}} [0; \overline{\ell}\langle n\rangle \mid \overline{\ell}\langle m'\rangle]$ while $Q \mid \overline{\ell}\langle m'\rangle$ can only perform an output at m.

- Without sequentiality, the output at ℓ can be done. For the second law, $[\ell; P] \xrightarrow{\overline{\ell}\langle n\rangle} [\emptyset; \ell \triangleleft m.\, R]$ while Q cannot do the same transition as $n \notin \mathrm{fn}(Q)$.

Example 73 (Optimised access)**.** The previous laws can then be used to justify optimisation when performing two consecutive read and/or write operations on the reference.

$$\begin{array}{rcll}
\ell \triangleleft n.\, \ell \triangleleft m.\, R & \approx_{\mathrm{rs}}^{\emptyset;1} & \ell \triangleleft m.\, R & \\
\ell \triangleleft n.\, \ell \triangleright (m).\, R & \approx_{\mathrm{rs}}^{\emptyset;1} & \ell \triangleleft n.\, R\{{}^{n}\!/_{m}\} & \\
\ell \triangleright (m).\, \ell \triangleright (m').\, R & \approx_{\mathrm{rs}}^{\emptyset;1} & \ell \triangleright (m).\, R\{{}^{m}\!/_{m'}\} & \\
\ell \triangleright (m).\, \ell \triangleleft n.\, R & \approx_{\mathrm{rs}}^{\emptyset;1} & \ell \bowtie n(m).\, R & \\
\ell \triangleright (m).\, \ell \triangleleft m.\, R & \approx_{\mathrm{rs}}^{\emptyset;1} & R & \text{if } m \notin \mathrm{fn}(R)
\end{array}$$

For all but the last equivalence, both processes may only perform an input at ℓ. The resulting processes can then be proved equivalent using the laws from Example 72.

The last law requires the use of rule Ext. Indeed, when $[\emptyset; 1; R] \xrightarrow{\mu} [\Delta; \eta; R']$, and $\ell \notin \Delta$, we use rule Ext and show that for all n, $\ell \triangleright (m).\, \ell \triangleleft m.\, R \mid \overline{\ell}\langle n\rangle \stackrel{\mu}{\Rightarrow} R' \mid \overline{\ell}\langle n\rangle$.

Example 74 (Independence)**.** As only one process may access the content of references, this also implies that operations on distinct references can be rescheduled without altering the resulting behaviour. For instance, we can prove:

$$\ell \triangleleft n.\, \ell' \triangleright (m).\, R \approx_{\mathrm{rs}}^{\emptyset;1} \ell' \triangleright (m).\, \ell \triangleleft n.\, R \quad \text{if } n \neq m$$

Proof. Take $\mathcal{R}$ as

$$\begin{array}{lll}
\{(\emptyset; 1, & \ell \triangleleft n.\, \ell' \triangleright (m).\, R, & \ell' \triangleright (m).\, \ell \triangleleft n.\, R), \\
(\ell; 1, & \overline{\ell}\langle n\rangle \mid \ell' \triangleright (m).\, R, & \overline{\ell}\langle n'\rangle \mid \ell' \triangleright (m).\, \ell \triangleleft n.\, R), \\
(\ell'; 1, & \overline{\ell'}\langle m'\rangle \mid \ell \triangleleft n.\, \ell' \triangleright (m).\, R, & \overline{\ell'}\langle m\rangle.\, \ell \triangleleft n.\, R)\} \\
\cup id & &
\end{array}$$

. We have that $\mathcal{R}$ is a sequential bisimulation with inductive predicate. □

Chapter 5

Well-Bracketing

5.1 Internal π-calculus: Iπ

We present another subcalculus of π: the Internal π-calculus, Iπ [47]. In Iπ, all names sent by an output are bound, i.e. all outputs are of the form $(\boldsymbol{\nu}\widetilde{b})\overline{a}\langle\widetilde{b}\rangle$. This can be expressed simply by introducing a bound output $\overline{a}(\widetilde{b})$, that is, an output which carries the bound names $\widetilde{b}$ used in P. The full syntax is given by the following grammar:

Processes	$P, Q ::= \overline{a}(\widetilde{b})\ P \mid\ !a(\widetilde{b}).\, P \mid\ P \mid Q \mid\ (\boldsymbol{\nu}a)P \mid\ G \mid\ K\lfloor\widetilde{a}\rfloor$
	$G, G' ::= \mathbf{0} \mid\ \overline{a}(\widetilde{b}).\, P \mid\ a(\widetilde{b}).\, P \mid\ \tau.\, P \mid\ G + G'$
Recursive definitions	$K \triangleq (\widetilde{a})\ P$

As in Iπ only bound outputs are used, there is no need for performing a substitution upon performing a communication. Using alpha-conversion is enough to ensure that the bound names sent coincide with the bound names received. This also removes the possibility of having aliasing, making the matching construct useless. The calculus features synchronous and asynchronous outputs, $\overline{a}(\widetilde{b}).\, P$ and $\overline{a}(\widetilde{b})\ P$ respectively. In the π-calculus, the latter can be encoded as $(\boldsymbol{\nu}\widetilde{b})(\overline{a}\langle\widetilde{b}\rangle \mid P)$, but we need to introduce a specific construct in Iπ [5]. In a bound asynchronous output, the continuation P is not guarded but it can still use the bound names that are sent at a. Similarly to Aπ, asynchronous outputs do not guard processes and so cannot be used inside a sum (synchronous outputs still can).

The standard definition of structural congruence is extended with two laws to reflect the behaviour of asynchronous output:

$$\overline{a}(\widetilde{b})\ (P \mid Q) \equiv P \mid \overline{a}(\widetilde{b})\ Q \text{ if } \mathrm{fn}(P) \cap \widetilde{b} = \emptyset \qquad\qquad \overline{a}(\widetilde{b})\ (\boldsymbol{\nu}c)P \equiv (\boldsymbol{\nu}c)\overline{a}(\widetilde{b})\ P \text{ if } c \notin a, \widetilde{b}$$

Constants are defined as an abstraction $(K \triangleq (\widetilde{a})\ P)$, where $\widetilde{a}$ are distinct names, bound in P. Given an abstraction $(\widetilde{a})\ P$, we note $((\widetilde{a})\ P)\lfloor\widetilde{b}\rfloor$ the process $P\{\widetilde{b}/\widetilde{a}\}$. Thus $K\lfloor\widetilde{b}\rfloor$ represents the process in the definition of K substituted with the names $\widetilde{b}$ (as expressed by the rule Cst in the operational semantics). Constants were not present in the standard π-calculus as they can be expressed using replication. Conversely, replication can be defined using constants. Take $K \triangleq (a)\ a(x).(P \mid K\lfloor a\rfloor)$. Then $K\lfloor a\rfloor$ behaves as $!a(x).\, P$. In Iπ however, replications are less expressive than constants, thus the need to add them into the language.

The full Labelled Transition System for Iπ is given in Figure 5.1. There is no longer a rule Open as all names sent are already bound. The actions are slightly different with μ being either τ, a bound output $\overline{a}(\widetilde{b})$ or a bound input $a(\widetilde{b})$. As we know that the name received are bound (and thus fresh), we use this notation for the input to distinguish it from the standard early input which requires a substitution. The rules AOut and AComm are the asynchronous equivalent of the rules Out and Comm. The rule Async has both the side conditions of rule Par, denoting the fact that the output is in parallel, and rule Res as the names $\widetilde{b}$ are restricted to P.

Ground bisimulation, and the corresponding ground bisimilarity, is a variant of the standard bisimulation in the π-calculus, where the objects of an input must be fresh [52, Section 4.4]. The bisimulation for Iπ using the LTS of Figure 5.1 ensures this property and so it is also called ground bisimulation by extension.

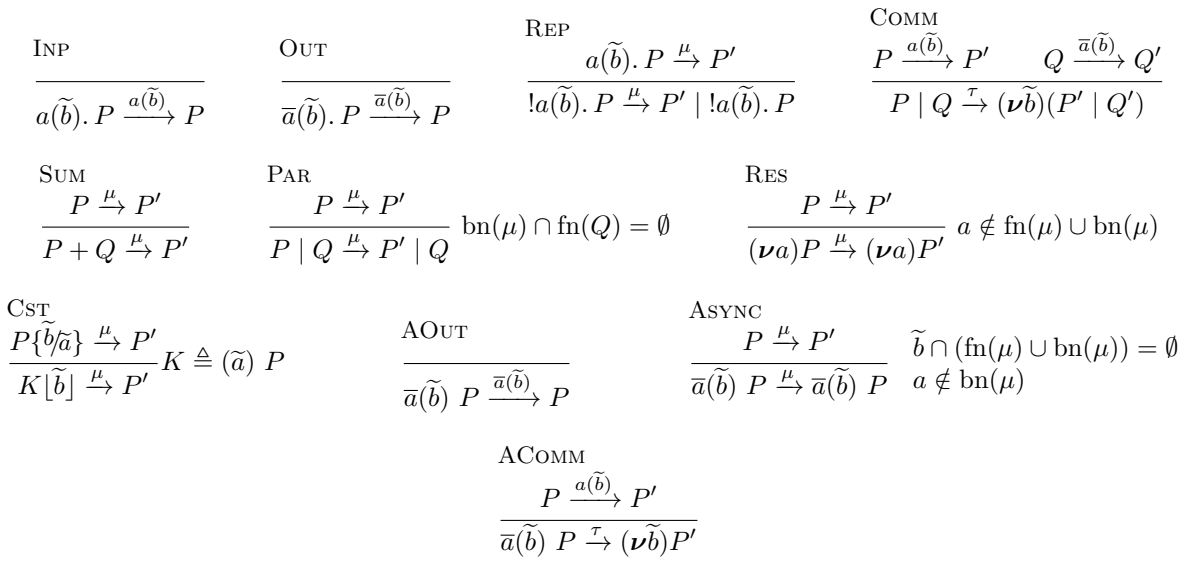


Figure 5.1: Labelled Transition Semantics for Iπ

5.2 Well-bracketing

5.2.1 Type system

We now go beyond sequentiality, so to handle well-bracketing. In languages without control operators, this means that return-call interactions among terms follow a stack-based discipline.

Intuitively, a well-bracketed system is a sequential system offering services. When interrogated, a service, say A, acquires the thread and is supposed to return a final result (unless the computation diverges) thus releasing the thread. During its computation, A may however interrogate another service, say B, which, upon completion of its computation, will return the result to A. In a similar manner, the service B, during its computation, may call yet another service C, and will wait for the return from C before resuming its computation. In any case, the 'return' obligation may not be thrown away or duplicated.

The implementation of this policy requires *continuation names*. For instance, when calling B, process A transmits a fresh name, say p, that will be used by B to return the result to A. Moreover, A waits for such a result, via an input at p. Therefore continuation names are *linear* [25] — they may only be used once — and *input receptive* [45] — the input-end of the name must be made available as soon as the name is created; and they are output-controlled: they carry the thread in output. Additionally, continuations names are asynchronous. This is needed, for technical reasons, in the proof of completeness (Section 5.3.2). The encoding of functions we use in Chapter 6 satisfies this constraint. Although continuation names are asynchronous, the bisimulation we define in Section 5.3.1 is synchronous, in the sense that we have no specific clause for inputs like in Definition 23. It seems non trivial to define an input clause similar to asynchronous bisimulation in Iπ. When $P \xrightarrow{\overline{a}(b)} P'$, we cannot just consider $Q \mid \overline{a}(b) \Rightarrow Q'$ as then b would be free in P' but bound in Q'.

In short, the 'well-bracketing' type system defined in this section refines the type discipline for sequentiality by adding linear-receptive names and enforcing a stack discipline on the usage of such names.

Thus, with well-bracketing, we have three kinds of names: (non-continuation) output-controlled names (ranged over by $x, y, z, \ldots$), input-controlled names (ranged over by $u, v, w, \ldots$), as in the previous section; and continuation names, ranged over by $p, q, r, \ldots$. In the following, we use "output-controlled" to refer specifically to the names $x, y, z, \ldots$, that are not continuation names. As before, names $a, b, c \ldots$ range over the union of output- an input-controlled names.

Input-controlled names do not play a major role in this type system. However, they are needed for the encoding of references in Section 6.1 to be typable. Continuation names may only be sent at output-

controlled names. Indeed, any output at an output-controlled name must carry exactly one continuation name. Without this constraint the type system for well-bracketing would be more complex, and it is unclear whether it would be useful in practice. Additionally, as a consequence of this constraint, sending a continuation means transmitting the thread, so that continuation names somehow represent the thread.

By convention, we assume that, in a tuple of names transmitted over an output-controlled name, the last name is a continuation name. We write $\widetilde{a}, p$ for such a tuple of names.

We present the typing rules for well-bracketing in Figure 5.2. Judgements are of the form $\Gamma; \rho \vdash_{\text{wb}} P$ with ρ being a stack and Γ a set containing pairs of a constant and a stack. The Γ component is used to store a typing invariant for constants, and plays no role in the rules other than WB-REC1 and WB-REC2. In fact, we will often write $\rho \vdash_{\text{wb}} P$ when Γ is empty. A stack is a sequence of *tagged* names, the tag being either $\mathtt{i}, \mathtt{o}$ or $\star$ denoting respectively input, output and both input and output capabilities. Upon creation, a continuation name comes with both capabilities (WB-RESC1), that are used once as input and output (WB-INPC and WB-AOUTC). Rule WB-RESC2 is required as once the communication on p is done, the restriction remains. In WB-OUTOC, a continuation is created and its input capability is passed to P, while its output capability is sent by the communication. This capability can then be used after the input in WB-INPOC. As continuation names are receptive, inputs at such names cannot appear after another prefix (WB-INPOC, WB-INPC, WB-INPIC). Input-controlled names do not interact with continuations, so the stack is not affected (WB-OUTIC, WB-INPIC, WB-RESOCIC). In WB-INPIC, the stack contains a output-tagged continuation name which represents the active thread of the process. We choose to have synchronous outputs at output-controlled names and asynchronous outputs at input-controlled names. This is done with the encoding from Section 6.1 in mind, but our results also hold when allowing both outputs for the two kind of names.

$$\textsc{wb-Nil}\ \frac{}{\Gamma\ ;\ \emptyset \vdash_{\text{wb}} \mathbf{0}} \qquad \textsc{wb-AOutC}\ \frac{\Gamma\ ;\ \sigma \vdash_{\text{wb}} P}{\Gamma\ ;\ p : \mathtt{o}, \sigma \vdash_{\text{wb}} \overline{p}(\widetilde{a})\ P} \qquad \textsc{wb-OutOC}\ \frac{\Gamma\ ;\ p : \mathtt{i}, q : \mathtt{o} \vdash_{\text{wb}} P}{\Gamma\ ;\ q : \mathtt{o} \vdash_{\text{wb}} \overline{x}(\widetilde{a}, p).\, P} \qquad \textsc{wb-AOutIC}\ \frac{\Gamma\ ;\ \sigma \vdash_{\text{wb}} P}{\Gamma\ ;\ \sigma \vdash_{\text{wb}} \overline{u}(\widetilde{a})\ P}$$

$$\textsc{wb-InpC}\ \frac{\Gamma\ ;\ p : \mathtt{o} \vdash_{\text{wb}} P \qquad p \neq q}{\Gamma\ ;\ q : \mathtt{i}, p : \mathtt{o} \vdash_{\text{wb}} q(\widetilde{a}).\, P} \qquad \textsc{wb-InpOC}\ \frac{\Gamma\ ;\ p : \mathtt{o} \vdash_{\text{wb}} P}{\Gamma\ ;\ \emptyset \vdash_{\text{wb}} x(\widetilde{a}, p).\, P,\ !x(\widetilde{a}, p).\, P} \qquad \textsc{wb-InpIC}\ \frac{\Gamma\ ;\ p : \mathtt{o} \vdash_{\text{wb}} P}{\Gamma\ ;\ p : \mathtt{o} \vdash_{\text{wb}} u(\widetilde{a}).\, P}$$

$$\textsc{wb-ResC1}\ \frac{\Gamma\ ;\ \rho, p : \star, \rho' \vdash_{\text{wb}} P}{\Gamma\ ;\ \rho, \rho' \vdash_{\text{wb}} (\boldsymbol{\nu} p) P} \qquad \textsc{wb-ResC2}\ \frac{\Gamma\ ;\ \rho \vdash_{\text{wb}} P}{\Gamma\ ;\ \rho \vdash_{\text{wb}} (\boldsymbol{\nu} p) P}\ p \notin \rho \qquad \textsc{wb-ResOCIC}\ \frac{\Gamma\ ;\ \rho \vdash_{\text{wb}} P}{\Gamma\ ;\ \rho \vdash_{\text{wb}} (\boldsymbol{\nu} y) P,\ (\boldsymbol{\nu} u) P} \qquad \textsc{wb-Tau}\ \frac{\Gamma\ ;\ \rho \vdash_{\text{wb}} P}{\Gamma\ ;\ \rho \vdash_{\text{wb}} \tau.\, P} |\rho| \leq 1$$

$$\textsc{wb-Sum}\ \frac{\Gamma\ ;\ \rho \vdash_{\text{wb}} P \qquad \Gamma\ ;\ \rho \vdash_{\text{wb}} Q}{\Gamma\ ;\ \rho \vdash_{\text{wb}} P + Q} |\rho| \leq 1 \qquad \textsc{wb-Par}\ \frac{\Gamma\ ;\ \rho \vdash_{\text{wb}} P \qquad \Gamma\ ;\ \rho' \vdash_{\text{wb}} Q}{\Gamma\ ;\ \rho'' \vdash_{\text{wb}} P \mid Q}\ \rho'' \in \mathtt{inter}(\rho; \rho')$$

$$\textsc{wb-Rec1}\ \frac{\Gamma, K : \rho\ ;\ \rho \vdash_{\text{wb}} P}{\Gamma\ ;\ \rho\{\widetilde{a}/\widetilde{b}\} \vdash_{\text{wb}} K\lfloor \widetilde{a} \rfloor} K \triangleq (\widetilde{b})\ P \qquad \textsc{wb-Rec2}\ \frac{}{\Gamma, K : \rho\ ;\ \rho\{\widetilde{a}/\widetilde{b}\} \vdash_{\text{wb}} K\lfloor \widetilde{a} \rfloor} K \triangleq (\widetilde{b})\ P$$

Figure 5.2: Type system for well-bracketing

Intuitively, a stack expresses the expected usage of the free continuation names in a process. Stacks are given by the following grammars:

$$\rho\ ::=\ \rho^{\mathtt{o}} \mid \rho^{\mathtt{i}} \qquad \rho^{\mathtt{o}}\ ::=\ p : \mathtt{o}, \rho^{\mathtt{i}} \mid p : \star, \rho^{\mathtt{o}} \mid \emptyset \qquad \rho^{\mathtt{i}}\ ::=\ p : \mathtt{i}, \rho^{\mathtt{o}} \mid \emptyset$$

Moreover, a name may appear at most once in a stack, so we will say that a name is $\mathtt{o}$-*tagged* in ρ when the name appears with tag $\mathtt{o}$ in ρ and similarly for $\mathtt{i}$ and $\star$.

As a $\star$-tagged name represents both output and input capabilities, a stack can be seen as an alternation of input- and output-tagged names. For instance, if we have

$$p_1 : \mathtt{o}, p_2 : \mathtt{i}, p_3 : \star, p_4 : \mathtt{o} \vdash_{\text{wb}} P$$

then $p_1, \ldots, p_4$ are the free continuation names in P; among these, p_1 will be used first, as an output at p_1; then p_2 will be used, in an input interaction with the environment. P possesses both the output and the input capability on p_3, and will use both capabilities by performing a reduction at p_3; the computation for P terminates with an output at p_4.

Thus, to compose two processes P and Q in parallel, the usage of continuations of $P \mid Q$ is an interleaving of the ones of P and Q (WB-PAR). Names with capabilities shared between P and Q can be used as synchronisation point. Formally, the interleaving of two stacks is described by the following definition:

Definition 75 (Interleaving)**.** The interleaving relation is defined as a ternary relation between stacks, written $\rho \in \mathtt{inter}(\rho_1 \ ; \ \rho_2)$, and defined as follows:

- $\emptyset \in \mathtt{inter}(\emptyset; \emptyset)$
- $\rho \in \mathtt{inter}(\rho_1 \ ; \ \rho_2)$ implies $\rho \in \mathtt{inter}(\rho_2 \ ; \ \rho_1)$.
- $\rho \in \mathtt{inter}(\rho_1 \ ; \ \rho_2)$ implies $p : \eta, \rho \in \mathtt{inter}(p : \eta, \rho_1 \ ; \ \rho_2)$, where $\eta \in \{\mathtt{o}, \mathtt{i}, \star\}$ and $p : \eta, \rho_1$ is a stack.
- Whenever $\rho \in \mathtt{inter}(\rho^{\mathtt{i}}; \rho^{\mathtt{o}})$, we have $p : \star, \rho \in \mathtt{inter}(p : \mathtt{o}, \rho^{\mathtt{i}} \ ; \ p : \mathtt{i}, \rho^{\mathtt{o}})$.

Having $\rho \vdash_{\mathrm{wb}} P$ implies that ρ either is empty or ends with an output-tagged name, even though stacks ending with an input-tagged name are allowed by the grammar. Thus, the side condition in rules WB-TAU and WB-SUM prevent ρ from containing an input-tagged name. As for the rules WB-INPOC and WB-INPC, this is to ensure linear receptiveness of these inputs.

As continuation names are used linearly, when both input and output are present in the process, i.e. when the name is $\star$-tagged in the stack, this name should intuitively not be observable. To hide these unobservable names from the type, we introduce the notion of *clean stack*.

Definition 76 (Clean stack)**.** A stack ρ is *clean* if no name appears in ρ $\star$-tagged. We note $\mathtt{c}(\rho)$ the clean stack obtained by removing all $\star$-tagged names from ρ, and $\rho \vDash_{\mathrm{wb}} P$ when there exists ρ' with $\rho' \vdash_{\mathrm{wb}} P$ and $\mathtt{c}(\rho') = \rho$.

Remark 77. If $\rho \vDash_{\mathrm{wb}} P$, then there exists $\widetilde{p}$ such that $\rho \vdash_{\mathrm{wb}} (\boldsymbol{\nu}\widetilde{p})P$.

There is no loss of information when cleaning a stack, it is still possible to recover the original stack:

Lemma 78. For any clean ρ with $\rho \vDash_{\mathrm{wb}} P$, there exists a unique ρ' with $\rho = \mathtt{c}(\rho')$ and $\rho' \vdash_{\mathrm{wb}} P$.

We use clean stacks to define *typed transitions*.

Definition 79. When $\rho \vDash_{\mathrm{wb}} P$, we write $[\rho; P] \xrightarrow{\mu} [\rho'; P']$ if $P \xrightarrow{\mu} P'$, ρ' is a stack and one of the following holds:

1. $\mu = \overline{p}(a)$ and $\rho = p : \mathtt{o}, \rho'$
2. $\mu = p(a)$ and $\rho = p : \mathtt{i}, \rho'$
3. $\mu = \overline{x}(a, p)$ and $\rho' = p : \mathtt{i}, \rho$
4. $\mu = x(a, p)$ and $\rho' = p : \mathtt{o}, \rho$
5. $\mu \in \{u(a), \overline{u}(a), \tau\}$ and $\rho' = \rho$.

In the cases 1 and 3, the stack ρ is of the form $\rho^{\mathtt{o}}$ while ρ' is of the form $\rho^{\mathtt{i}}$, and symmetrically for the cases 2 and 4. This occurs because continuation names are output-controlled. In fact, we have the following lemma:

Lemma 80. If $\rho^{\mathtt{i}} \vdash_{\mathrm{wb}} P$, then $0 \vdash_{\mathrm{s}} P$. Similarly, if $\rho^{\mathtt{o}} \vdash_{\mathrm{wb}} P$ and $\rho^{\mathtt{o}} \neq \emptyset$, then $1 \vdash_{\mathrm{s}} P$.

Another interest of using clean stacks is that the type is not preserved by reduction, as once a communication occurs at a continuation name, it no longer appear in the stack. This behaviour is also observed for linear names [25].

Lemma 81 (Subject reduction)**.** If $[\rho; P] \xrightarrow{\mu} [\rho'; P']$, then $\rho' \vDash_{\mathrm{wb}} P'$.

5.2.2 The well-bracketing property on traces

In this section, we aim at expressing how our system satisfies a well-bracketing property, similar to Proposition 52 for sequentiality. Following game semantics [20], we formalise well-bracketing, that is, the stack-like behaviour of continuation names for well-typed processes, using traces of actions. A trace for such a process is obtained from a sequence of transitions emanating from the process, with the expected freshness conditions to avoid ambiguity among names.

Definition 82 (Trace)**.** A sequence of actions $\mu_1, \ldots, \mu_n$ is a *trace for a process P_0 and a clean stack σ_0* if there are $\sigma_1, \ldots, \sigma_n, P_1, \ldots, P_n$ such that for all $0 \leq j < n$ we have $[\sigma_j; P_j] \xrightarrow{\mu_{j+1}} [\sigma_{j+1}; P_{j+1}]$, where all continuation names appearing as object in μ_{j+1} are fresh (i.e. the names may not appear in any μ_i for $i \leq j$).

The condition in Definition 82 on continuation names ensures us that for actions like $\overline{x}(\widetilde{a}, p)$, name p is fresh, and that after an action $\overline{p}(\widetilde{a})$ (thus after the only allowed interaction at p has been played), name p cannot be reintroduced, e.g. in an action $x(\widetilde{a}, p)$. We simply say that $\mu_1, \ldots, \mu_n$ is a trace, or is a trace for P, when the stack or the process are clear from the context.

Lemma 83. Suppose we have a trace $\mu_1, \ldots, \mu_n$ with $(\sigma_j, P_j)_{0 \leq j \leq n}$ such that $[\sigma_j; P_j] \xrightarrow{\mu_{j+1}} [\sigma_{j+1}; P_{j+1}]$ for $0 \leq j \leq n$ If $\sigma_0 = \sigma_n$, then there exists some ρ_i, such that $\sigma_i = \rho_i, \sigma_0$ for all $0 \leq i \leq n$.

This lemma formalises the idea that when looking at the evolution of the stacks σ_i for a given trace, names inside a stack may not reappear later on if they are removed. Thus, if at some point we obtain a stack σ_i identical to σ_0, then for all the intermediate stacks in between σ_0 and σ_i, we should find the original stack as substack.

The well-bracketing property is best described with the notion of questions and answers.

Definition 84. For a trace $\mu_1, \ldots, \mu_n$, we set $\mu_i \curvearrowright \mu_j$ if $i < j$ and:

1. either $\mu_i = \overline{x}(\widetilde{a}, p)$ and $\mu_j = p(\widetilde{b})$,
2. or $\mu_i = x(\widetilde{a}, p)$ and $\mu_j = \overline{p}(\widetilde{a})$.

Actions μ_i (with a continuation name in object position) are called *questions*, while actions μ_j (with a continuation name in subject position) are called *answers*.

The notion of questions and answers for these actions originates from operational game semantics [27, 37, 21], whose link with the π-calculus is studied in [23].

Remark 85. For a transition $[\sigma; P] \xrightarrow{\mu} [\sigma'; P']$, the value $|\sigma'| - |\sigma|$ is 1 for a question, -1 for an answer, and 0 for an internal action or communication at an input-controlled name (we recall that $|\sigma|$ stands for the number of elements in the stack σ).

Lemma 86 (Uniqueness)**.** Given a trace $\mu_1, \ldots, \mu_n$, if $\mu_i \curvearrowright \mu_j$ and $\mu_{i'} \curvearrowright \mu_{j'}$, then we have ($i = i'$ iff $j = j'$).

A transition is then either an internal transition, or a question, or an answer. A question mentioning a continuation name p is matched by an answer at the same name p. When questions and answers are seen as delimiters ('$[_p$','$]_p$', different for each continuation name), a well-bracketed trace is a substring of a Dyck word.

This can be expressed as follows.

Definition 87 (Well-bracketing)**.** A trace $\mu_1, \ldots, \mu_n$ is *well-bracketed* if for all $i < j$, if μ_i is a question and μ_j is an answer with $\mu_i \not\curvearrowright \mu_k$ and $\mu_k \not\curvearrowright \mu_j$ for all $i < k < j$, then $\mu_i \curvearrowright \mu_j$.

To prove that all traces are well-bracketed (Proposition 89), we need the following property relating questions and answers to stacks.

Lemma 88. Let $\mu_1, \ldots, \mu_n$ be a trace, and $\sigma_0, \ldots, \sigma_n$ be the corresponding stacks, as in Definition 82. Suppose $\sigma_0 = \sigma_n$, and for all i, $|\sigma_i| > |\sigma_0|$. Then μ_1 is a question, μ_n is an answer, and $\mu_1 \curvearrowright \mu_n$.

Proposition 89. Any trace (as by Definition 82) is well-bracketed.

Berger et al. [5] prove a similar property for their type system. Our approach allows us to type a larger set of processes. We can type processes that are not in the encoding of PCF terms, and in particular non-deterministic processes, which makes it easier to define a suitable bisimulation.

5.3 Behavioural equivalences

5.3.1 Typed barbed equivalence, well-bracketed bisimulation and soundness

As in Chapter 4, we define a *wb-typed relation on processes* as a set of triplets (σ, P, Q) with $\sigma \vDash_{\text{wb}} P, Q$.

The definition of $\sigma \vDash_{\text{wb}} P \Downarrow_\alpha$ is as expected. In particular, as σ is clean, we cannot observe barbs at continuation names that are used both in input and output. This type system being a refinement of sequentiality as expressed by Lemma 80, the same issue arises about the reduction-based version of barbed equivalence. In fact, the processes used in Lemma 8 are not even allowed in untyped Iπ. Therefore, *well-bracketed barbed equivalence*, written $\simeq^\sigma$, is defined as follows: $P \simeq^\sigma Q$ holds if $\sigma \vDash_{\text{wb}} P, Q$, and for any σ'/σ static context E, if $\sigma' \vDash_{\text{wb}} E[P], E[Q]$, then $E[P] \dot{\approx}^{\sigma'} E[Q]$.

As we are not using the typing directly to hide unobservable continuation names, we need to add that $E[P]$ and $E[Q]$ can indeed be typed before checking if they are bisimilar. For instance, $E \stackrel{\text{def}}{=} \overline{q}(a) \mid [\cdot]$ is a $q : \mathsf{o}, \sigma/\sigma$ context whenever $q \notin \sigma$. However, even if σ is clean, if we have $q : \star, \sigma \vdash_{\text{wb}} P$, then $\sigma \vDash_{\text{wb}} P$ but $E[P]$ cannot be typed.

We continue by defining the *well-bracketed bisimulation*, which we note wb-bisimulation.

Definition 90 (wb-bisimulation)**.** A wb-typed relation $\mathcal{R}$ on processes is a *wb-bisimulation* if whenever $(\sigma, P, Q) \in \mathcal{R}$ and $[\sigma; P] \xrightarrow{\mu} [\sigma'; P']$, there is Q' with $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $(\sigma', P', Q') \in \mathcal{R}$ and symmetrically for the transitions from Q.

Processes P and Q are *wb-bisimilar at* σ, noted $P \approx^\sigma_{\text{wb}} Q$, if $(\sigma, P, Q) \in \mathcal{R}$ for some wb-bisimulation $\mathcal{R}$.

wb-bisimulation is sound with respect to barbed equivalence for all processes. The main result is preservation of wb-bisimulation by parallel composition:

Lemma 91 (Parallel composition)**.** If $P \approx^\sigma_{\text{wb}} Q$, then for any process R and stacks σ', σ'' such that $\sigma' \vDash_{\text{r}} R$, $\sigma'' \vDash_{\text{wb}} P \mid R$ and $\sigma'' \vDash_{\text{wb}} Q \mid R$ with $\sigma'' \in \texttt{inter}(\sigma; \sigma')$, we have $P \mid R \approx^{\sigma''}_{\text{wb}} Q \mid R$.

In the statement of Lemma 91, we ask explicitly that $P \mid R$ and $Q \mid R$ are well-typed. Indeed, there is no equivalent of rule WB-PAR for statements like $\sigma \vDash_{\text{wb}} P$. As cleaning a stack removes unobservable continuation names, rule WB-PAR does not ensure that the linearity of these names are preserved.

Theorem 92 (Soundness)**.** $\approx^\sigma_{\text{wb}} \subseteq \simeq^\sigma$.

To prove soundness, we show that $\approx^\sigma_{\text{wb}}$ is preserved by all static contexts. We can develop up-to techniques similar to the ones introduced in Section 6.3, like up-to $\gtrsim$ and up-to static context. Such techniques are used in Section 6.1.2.

5.3.2 Completeness

As in Chapter 4, we prove completeness for processes that only use output-controlled names. The overall structure of the proof is similar: the crux of the proof is defining the discriminating static contexts.

However, the additional difficulty is related to receptiveness of continuation names: we cannot use $R + T$, as in Chapter 4, when the tester process R contains an input at a free continuation name. Thus, it is not longer possible to have a process R whose structure mimics the semantic tree as its input at continuation names should be unguarded even though that will not be used until later in the interaction.

The following lemma is a direct consequence of the usage discipline of continuation names.

Lemma 93. If $\sigma \vDash_{\text{r}} P \mid Q \mid R$ and $p \notin \text{fn}(P) \cup \text{fn}(Q) \cup \text{fn}(R)$ then

$$\overline{p}(\widetilde{a}).\, P \mid p(\widetilde{a}).\, Q \mid R \approx^\sigma_{\text{wb}} P \mid Q \mid R.$$

By soundness, this equivalence also holds for barbed equivalence at σ. This lemma implies that the communication at a continuation name does not change the equivalence class of the process.

As wb-typed processes are sequential (by Lemma 80), we can extend the notion of singular processes to wb-typed processes too and reuse most of Lemma 60. Thus, we can prove that:

- if P is singular, so is $E[P]$ for any typed static context E,

- for all α, $\sigma \vDash_{\mathrm{wb}} P \not\Downarrow_\alpha$ (given $\sigma \vDash_{\mathrm{wb}} P$).

Theorem 94 (Completeness)**.** For all image-finite processes P, Q that only use output-controlled names, and for all σ, if $P \simeq^\sigma Q$ then $P \approx^\sigma_{\mathrm{wb}} Q$.

The following proposition is the counterpart, for wb-bisimulation, of Proposition 63, and represents the core of the proof of completeness. It refers to the stratification of wb-bisimulation, which is defined similarly to the stratification of sequential bisimulation.

We often use outputs at output-controlled name as barbs. As they are not meant to be used in communication, we write $\overline{z}$ for $\overline{z}(p).\, p.\, \overline{q}(\widetilde{x})$ where q is determined by the typing. As in Section 4.2.2, $s, s', \dots$ are also used to denote output-controlled names (in addition to $x, y, z, \dots$). For a name z and a set S of output-controlled names, we recall that $S \rhd z$ stands for $\sum_{s \in S} s(\widetilde{b}, q).\, \overline{z}$, where $\overline{z}$ is the notation introduced above. Let $\mathrm{fn_O}(-)$ denote the set of free output-controlled names.

Proposition 95. For all $m \geq 0$, stack $\sigma = \sigma_1, p_1 : \mathtt{o}, q_1 : \mathtt{i}, \dots, p_n : \mathtt{o}$ with $\sigma_1 = \emptyset$ or $\sigma_1 = q : \mathtt{i}$, if $\sigma \vdash_{\mathrm{wb}} P, Q$ and $P \not\approx^{\sigma,m}_{\mathrm{wb}} Q$, then for any fresh $q_n, \widetilde{y_i}$, there exists processes R and $(R_i)_i$ such that for any fresh z, w and names $\widetilde{a}, S$ with $\widetilde{a} \subseteq \mathrm{fn}(P, Q)$ and $\mathrm{fn_O}(P, Q) \cup \widetilde{y_i} \subseteq S$, and all summations $(M_i)_i$, we define:

$$\overline{R} \stackrel{\text{def}}{=} \begin{cases} R + \overline{z} & \text{when } \sigma_1 = q : \mathtt{i} \\ R + S \rhd z + S \rhd w & \text{when } \sigma_1 = \emptyset \end{cases}$$

$$E_0 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid \overline{R} \mid \prod_{i \leq n} p_i(\widetilde{b}).\, \overline{y_i}(r).\, r.\, (M_i + R_i))$$

and we have:

1. either $E_0[P'] \dot{\not\approx}^{q_n : \mathtt{o}} E_0[Q]$ for all P' such that $P \Rightarrow P'$
2. or $E_0[P] \dot{\not\approx}^{q_n : \mathtt{o}} E_0[Q']$ for all Q' such that $Q \Rightarrow Q'$

Proof. First, some remarks:

- If $\sigma_1 = q : \mathtt{i}$, then $q_n : \mathtt{o} \vDash_{\mathrm{wb}} E_0[P] \Downarrow_{\overline{z}}$.
- If $\sigma_1 = \emptyset$, then we have
  - $\sigma \vDash_{\mathrm{wb}} P \Downarrow_\alpha$ for some α iff $q_n : \mathtt{o} \vDash_{\mathrm{wb}} E_0[P] \Downarrow_{\overline{z}}$ iff $q_n : \mathtt{o} \vDash_{\mathrm{wb}} E_0[P] \Downarrow_{\overline{w}}$.
  - $q_n : \mathtt{o} \vDash_{\mathrm{wb}} E_0[P] \Downarrow_{\overline{y_1}}$ iff $\sigma \vDash_{\mathrm{wb}} P \mid R \Downarrow_{\overline{p_1}}$.
- and similarly for Q.

As continuation names are asynchronous, we have that if $P \xrightarrow{\overline{p}(\widetilde{b})} \xrightarrow{\mu} P'$ and $\mathrm{n}(\mu) \cap \widetilde{b} = \emptyset$, then $P \xrightarrow{\mu} \xrightarrow{\overline{p}(\widetilde{b})} \equiv P'$. This fact is used mainly to show replace $\stackrel{\overline{p}(\widetilde{b})}{\Longrightarrow}$ with $\Rightarrow \xrightarrow{\overline{p}(\widetilde{b})}$.

In the proof, processes R_i consist of sums of processes, all of which are as in the case for $\mu = \overline{p_1}(\widetilde{b})$. Their definitions require the use of two fresh names, written c and z'. The first name also appears once in R. We assume that these names are taken distinct from each other, even when using our induction hypothesis yielding multiple R^j, R^j_i. Additionally, we assume that M_i cannot perform the same barbs as R_i. This is ensured in the proof because of the freshness of the names c and z'.

We reason by induction on m as usual. For $m = 0$, there is nothing to prove.

For $m > 0$, suppose that $P \not\approx^{\sigma;m}_{\mathrm{wb}} Q$. We may therefore suppose that there exists μ such that $[\sigma; P] \xrightarrow{\mu} [\sigma'; P']$ and for all Q' with $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$, we have $P' \not\approx^{\sigma';m-1}_{\mathrm{wb}} Q'$. We note $\{Q_j\}$ for $j \in J$ the set of all such Q'. This set is finite by hypothesis. We also write $S' = \bigcup_{j \in J} \mathrm{fn_O}(Q_j) \cup \mathrm{fn_O}(P') \cup \widetilde{y_i}$, which is also a finite set. We show that clause 2. holds. (in the symmetrical case, where we consider a transition from $[\sigma; Q]$, we establish the first clause).

We distinguish two cases, according to whether P' is singular or not.

First case: P' is singular.

Since P' is singular, $\sigma'_1 = \emptyset$. For all j, as $P' \not\approx^{\sigma'}_{\mathrm{wb}} Q_j$, we have $Q_j \Downarrow_{\overline{s}}$ for some $s \in S'$ or $Q_j \Downarrow_{\overline{p_1}}$.

By Lemma 81, there are three possible cases:

- when $\mu = \tau$ and $\sigma' = \sigma$. We set $R \stackrel{\text{def}}{=} \mathbf{0}$ and $R_i \stackrel{\text{def}}{=} \mathbf{0}$. We then have to show that for any $z, \widetilde{a}, S, (M_i), Q'$ with $Q \Rightarrow Q'$, taking $\overline{R} = R + S \rhd z + S \rhd w$ (since $\sigma_1 = \emptyset$) and $E_0 = (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid \overline{R} \mid \prod_{i \leq n} p_i(\widetilde{b}).\, \overline{y_i}(r).\, r.\, (M_i + R_i))$, we have $E_0[P] \mathrel{\dot{\not\approx}}^{q_n : \mathtt{o}} E_0[Q']$.

  We reason by contradiction. First, we have $E_0 \equiv (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid S \rhd z + S \rhd w \mid \prod_{i \leq n} p_i(\widetilde{b}).\, \overline{y_i}(r).\, r.\, M_i)$. As $\mathrm{fn}_{\mathrm{O}}(Q_j) \subseteq S$, we have $E_0[Q_j] \Downarrow_{\overline{z}}$ and similarly for w. On the other hand, $E_0[P']$ is singular by Lemma 60 and thus has no barb. As we have $E_0[P] \to E_0[P']$, there exist T such that $E_0[Q'] \Rightarrow T$ and $E_0[P'] \mathrel{\dot{\approx}}^{q_n : \mathtt{o}} T$. We observe then that T can be of 4 forms:

  - $E_0[Q_j]$ for some $j \in J$ (with $Q' \Rightarrow Q_j$)
  - $(\boldsymbol{\nu}\widetilde{a})(Q'' \mid \overline{z})$
  - $(\boldsymbol{\nu}\widetilde{a})(Q'' \mid \overline{w})$
  - $(\boldsymbol{\nu}\widetilde{a})(Q'' \mid \overline{R} \mid \overline{y_1}(r).\, r.\, M_1 \mid R')$

  All forms have a barb, so T cannot be barbed bisimilar to $E_0[P']$.

- when $\mu = q(\widetilde{x})$, given a fresh name z', we define

$$R \stackrel{\text{def}}{=} \overline{q}(\widetilde{x}).\, S' \rhd z' \qquad\qquad R_i \stackrel{\text{def}}{=} \mathbf{0}$$

  We then show that for any $z, \widetilde{a}, S, (M_i), Q'$ with $Q \Rightarrow Q'$ by setting $E_0 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid R + \overline{z} \mid \prod_{i \leq n} p_i(\widetilde{b}).\, \overline{y_i}(r).\, r.\, M_i)$, we have $E_0[P] \mathrel{\dot{\not\approx}}^{q_n : \mathtt{o}} E_0[Q']$.

  We write $E_1 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{a}, \widetilde{x})([\cdot] \mid S' \rhd z' \mid \prod_{i \leq n} p_i(\widetilde{b}).\, \overline{y_i}(r).\, r.\, M_i)$. As $\mathrm{fn}_{\mathrm{O}}(Q_j) \subseteq S'$, we have $q_n : \mathtt{o} \vDash_{\mathrm{wb}} E_1[Q_j] \Downarrow_{\overline{z'}}$. Similarly to before, $E_0[P] \to E_1[P']$, so there exists T such that $E_0[Q'] \Rightarrow T$ and $E_1[P'] \mathrel{\dot{\approx}}^{q_n : \mathtt{o}} T$. By Lemma 60, $E_1[P']$ has no barb. On the other hand, T can be of 4 forms:

  - $E_0[Q'']$ with a barb at $\overline{z}$
  - $E_1[Q_j]$ for some $j \in J$ (with $Q' \stackrel{q(\widetilde{x})}{\Longrightarrow} Q_j$) with a barb at $\overline{z'}$
  - $(\boldsymbol{\nu}\widetilde{a})(Q'' \mid \overline{z'})$ with a barb at $\overline{z'}$
  - $(\boldsymbol{\nu}\widetilde{a})(Q'' \mid \overline{y_1}(r).\, r.\, M_1 \mid R')$ with a barb at $\overline{y_1}$

  We thus have a contradiction.

- when $\mu = x(\widetilde{b}, p_0)$, given fresh names z' and y_0, we define

$$R \stackrel{\text{def}}{=} \overline{x}(\widetilde{b}, p_0).\, (p_0(\widetilde{b'}).\, \overline{y_0} \mid S' \rhd z') \qquad\qquad R_i \stackrel{\text{def}}{=} \mathbf{0}$$

  The proof is identical to the previous case with

$$E_1 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{a}, \widetilde{b}, p_0)([\cdot] \mid S' \rhd z' \mid p_0(\widetilde{b}).\, \overline{y_0} \mid \prod_{i \leq n} p_i(\widetilde{b}).\, \overline{y_i}(r).\, r.\, M_i)$$

  and the last case having a barb at $\overline{y_0}$ instead of $\overline{y_1}$.

Second case: if P' is not singular. We note R^j and R^j_i for the processes we obtain, by induction, for each pair (P', Q_j).

As for the proof of completeness for sequential bisimilarity, we prove an intermediate result: for any $\widetilde{z_j}, \widetilde{w_j}, \widetilde{a}, S', Q_j, (M_i)$ with $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q_j$, we define

$$\overline{R^j} \stackrel{\text{def}}{=} \begin{cases} R^j + \overline{z_j} & \text{when } \sigma'_1 = q : \mathtt{i} \\ R^j + S' \rhd z_j + S' \rhd w_j & \text{when } \sigma'_1 = \emptyset \end{cases}$$

$$F \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid \sum_{j \in J} \tau.\, \overline{R^j} \mid \prod_{i \leq n} p_i(\widetilde{b}).\, \overline{y_i}(r).\, r.\, (M_i + \sum_j R^j_i)$$

and we have

$$F[P'] \mathrel{\dot{\not\approx}}^{q_n : \mathtt{o}} F[Q_j] \qquad (\star)$$

We prove the second case, the first being similar but with less cases.

We have two cases depending on the clause of the proposition that holds, by induction, for (P', Q_j):

1. if clause 1. holds, then $F[Q_j] \to E_j[Q_j]$ with

$$E_j \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid \overline{R^j} \mid \prod_{i\leq n} p_i(\widetilde{b}).\,\overline{y_i}(r).\,r.\,(M_i' + R_i^j)) \qquad\qquad M_i' \stackrel{\text{def}}{=} M_i + \sum_{j'\neq j} R_i^{j'}$$

This means there is T with $F[P'] \Rightarrow T$ and $T \stackrel{\cdot}{\approx}^{q_n:\mathtt{o}} E_j[Q_j]$. As we have $E_j[Q_j] \Downarrow_{\overline{z_j}}$, $E_j[Q_j] \Downarrow_{\overline{w_j}}$ and $E_j[Q_j] \not\Downarrow_{\overline{z_{j'}}}$ for $j' \neq j$, we must have that

- either $T \equiv E_j[P'']$ with $P' \Rightarrow P''$,
- or $T \equiv (\boldsymbol{\nu}\widetilde{a},\widetilde{b})(P'' \mid \overline{R^j} \mid \overline{y_1}(r).\,r.\,(M_1'+R_1^j) \mid \prod_{2\leq i\leq n} p_i(\widetilde{b}).\,\overline{y_i}(r).\,r.\,(M_i'+R_i^j))$ with $P' \Rightarrow \xrightarrow{\overline{p_1}(\widetilde{b})} P''$

Because of Lemma 93, the process in the second case is wb-bisimilar (and thus barbed bisimilar) to the first one.

Up to structural congruence, we can suppose that in $E_j[P'']$ and $E_j[Q_j]$, $\widetilde{a}$ contains only names from $\mathrm{fn}(P') \cup \mathrm{fn}(Q_j)$. We are in a situation where we can apply the induction hypothesis, and this is in contradiction with $E_j[P''] \stackrel{\cdot}{\approx}^{q_n:\mathtt{o}} E_j[Q_j]$ as $P' \Rightarrow P''$.

2. if clause 2. holds, we have similarly that $F[P'] \to E_j[P']$. The same reasoning, with the role of P' and Q_j swapped, is sufficient to conclude.

We can now examine the different possibilities for the transition along μ.

- $\mu = x(\widetilde{b},p_0)$, then $\sigma_1 = q_0 : \mathtt{i}$ and $\sigma' = p_0 : \mathtt{o},\sigma$. By induction, we obtain processes R_i^j for $0 \leq i \leq n$. Given fresh names $(z_j,w_j)_{j\in J}$ and y_0, we define

$$R \stackrel{\text{def}}{=} \overline{x}(\widetilde{b},p_0).\,(\sum_{j\in J} \tau.\,(R_j + S' \rhd z_j \mid S' \rhd w_j) \mid p_0(\widetilde{b}).\,\overline{y_0}(r).\,r.\sum_j R_0^j) \qquad\qquad R_i = \sum_j R_i^j$$

We then have to show that for any $z,\widetilde{a},S,Q',(M_i)$ with $Q \Rightarrow Q'$ by setting $E_0 = (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid R + \overline{z} \mid \prod_{1\leq i\leq n} p_i(\widetilde{b}).\,\overline{y_i}(r).\,r.\,(M_i + R_i))$, we have $E_0[P] \not\approx^{q_n:\mathtt{o}} E_0[Q']$.

We reason by contradiction. We have $E_0[P] \to F[P']$ with the restrictions of F being $\widetilde{a},\widetilde{b},p_0$.

Thus, there exists T such that $E_0[Q'] \Rightarrow T$ and $F[P'] \stackrel{\cdot}{\approx}^{q_n:\mathtt{o}} T$. As P' is not singular, $P' \Downarrow_\alpha$ with $\alpha = \overline{z}$ or $\alpha = \overline{p_0}$, so $F[P'] \Downarrow_{\overline{z_j}}$ for all z_j. We also have $F[P'] \not\Downarrow_{\overline{z}}$. This implies that T can only be of 2 forms:

  - $F[Q_j]$ with $Q' \xRightarrow{x(\widetilde{b},p_0)} Q_j$ for some $j \in J$
  - or $(\boldsymbol{\nu}\widetilde{a},\widetilde{b},p_0,\widetilde{b'})(Q'' \mid \sum_j \tau.\,\overline{R^j} \mid \overline{y_0}(r).\,r.\sum_j R_0^j \mid \prod_{1\leq i\leq n} p_i(\widetilde{b}).\,\overline{y_i}(r).\,r.\,(M_i+R_i))$ with $Q' \xRightarrow{x(\widetilde{b},p_0)} \xrightarrow{\overline{p_0}(\widetilde{b'})} Q''$

By Lemma 93, the process in the second case is barbed bisimilar to the first, which is in contradiction with $(\star)$.

- $\mu = q(\widetilde{b})$, then $\sigma_1 = q : \mathtt{i}$ and $\sigma' = \sigma_2$. By induction, we obtain processes R_i^j for $1 \leq i \leq n$. Given fresh names $(z_j,w_j)_{j\in J}$, we define

$$R \stackrel{\text{def}}{=} \tau.\,\overline{q}(\widetilde{b}).\sum_{j\in J} \tau.\,(R_j + S' \rhd z_j + S' \rhd w_j) \qquad\qquad R_i \stackrel{\text{def}}{=} \sum_j R_i^j$$

We then have to show that for any $z,\widetilde{a},S,Q',(M_i)$ with $Q \Rightarrow Q'$ by setting $E_0 = (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid R + \overline{z} \mid \prod_{1\leq i\leq n} p_i(\widetilde{b}).\,\overline{y_i}(r).\,r.\,(M_i + R_i))$, we have $E_0[P] \not\approx^{q_n:\mathtt{o}} E_0[Q']$.

We have $E_0[P] \to\!\!\!\!\to F[P']$ with the restrictions of F being $\widetilde{a},\widetilde{b}$. Thus, there exists T such that $E_0[Q'] \Rightarrow T$ and $F[P'] \stackrel{\cdot}{\approx}^{q_n:\mathtt{o}} T$. As P' is not singular, so $F[P'] \Downarrow_{\overline{z_j}}$ for all z_j. We also have $F[P'] \not\Downarrow_{\overline{z}}$. This implies that T can only be of 3 forms:

– $(\boldsymbol{\nu}\widetilde{a})(Q'' \mid \overline{q}(\widetilde{b}).\sum_j \tau.\,\overline{R^j} \mid \prod_{1\le i\le n} p_i(\widetilde{b}).\,\overline{y_i}(r).\,r.\,(M_i+R_i))$

– or $F[Q_j]$ with $Q' \stackrel{q(\widetilde{b})}{\Longrightarrow} Q_j$ for some $j\in J$

– or $(\boldsymbol{\nu}\widetilde{a},\widetilde{b},\widetilde{b'})(Q'' \mid \sum_j \tau.\,\overline{R^j} \mid \overline{y_1}(r).\,r.\,(M_1+R_1) \mid \prod_{2\le i\le n} p_i(\widetilde{b}).\,\overline{y_i}(r).\,r.\,(M_i+R_i))$ with $Q' \stackrel{q(\widetilde{b})}{\Longrightarrow}\xrightarrow{\overline{p_1}(\widetilde{b'})} Q''$

By Lemma 93, the processes in the first and last cases are wb-bisimilar to the second case, which is in contradiction with $(\star)$.

- $\mu = \overline{x}(\widetilde{b},q)$. Given fresh names $(z_j)_{j\in J}$, we define

$$R \stackrel{\text{def}}{=} x(\widetilde{b},q).\sum_{j\in J}\tau.\,(R^j+\overline{z_j}) \qquad\qquad R_i = \sum_j R_i^j$$

We then have to show that for any $z,w,\widetilde{a},S,Q',(M_i)$ with $Q\Rightarrow Q'$ by setting $E_0 = (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid R+S\rhd z+S\rhd w \mid \prod_{1\le i\le n} p_i(\widetilde{b}).\,\overline{y_i}(r).\,r.\,(M_i+R_i))$, we have $E_0[P] \mathrel{\dot{\approx}^{q_n:\mathtt{o}}} E_0[Q']$.

We have $E_0[P]\to F[P']$ with the restrictions of F being $\widetilde{a},\widetilde{b},q$. Thus, there exists T such that $E_0[Q']\Rightarrow T$ and $F[P'] \mathrel{\dot{\approx}^{q_n:\mathtt{o}}} T$. $F[P']\Downarrow_{\overline{z_j}}$ for all z_j and we have $F[P']\not\Downarrow_{\overline{z}}$. T can only of the form $F[Q_j]$ with $Q' \stackrel{\overline{x}(\widetilde{b},q)}{\Longrightarrow} Q_j$ for some $j\in J$ yielding a contradiction with $(\star)$.

- $\mu = \overline{p}(\widetilde{b})$. By subject reduction, we must have $p=p_1$. Given fresh names y_1,c,z' we define

$$R \stackrel{\text{def}}{=} y_1(r).\,\overline{r}.\,c(r').\,\overline{r'} \qquad R_1 \stackrel{\text{def}}{=} (\tau.\,\overline{c}(r').\,r'.\sum_j \tau.\,(R^j+\overline{z_j}))+\overline{z'} \qquad R_i \stackrel{\text{def}}{=} \sum_j R_i^j$$

The input at y_1 is used in R to allow the computation to proceed. In the previous cases, we often used the output at y_1 to block the computation after an unobservable communication at a continuation name has been done. The continuation sent at y_1, r is used immediately so that the thread is owned by M_1+R_1. Intuitively, M_1 is a sum of processes similar to R_1 but with different barbs (with other c,z'). They are accumulated in the proof of $(\star)$ when defining M_i'. The input at c in R then enables the computation to continue on the only branch we are interested in.

We then have to show that for any $z,w,\widetilde{a},S,Q',(M_i)$ with $Q\Rightarrow Q'$ by setting $E_0 = (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid R+S\rhd z+S\rhd w \mid \prod_{1\le i\le n} p_i(\widetilde{b}).\,\overline{y_i}(r).\,r.\,(M_i+R_i))$, we have $E_0[P] \mathrel{\dot{\approx}^{q_n:\mathtt{o}}} E_0[Q']$.

We have $E_0[P] \to\to\to (\boldsymbol{\nu}\widetilde{a})(P' \mid c(r').\,\overline{r'} \mid \overline{c}(r').\,r'.\sum_j \tau.\,(R^j+\overline{z_j}) \mid \prod_{2\le i\le n} p_i(\widetilde{b}).\,\overline{y_i}(r).\,r.\,(M_i+R_i)) \stackrel{\text{def}}{=} E_1[P']$. Thus, there exists T such that $E_0[Q']\Rightarrow T$ and $E_1[P'] \mathrel{\dot{\approx}^{q_n:\mathtt{o}}} T$.

We have $E_1[P']\Downarrow_{\overline{c}}$ and $E_1[P']\not\Downarrow_{\overline{z'}}$. As $M_1\not\Downarrow_{\overline{c}}$ by hypothesis, we must have that T is of the form $E_1[Q_j]$ with $Q \stackrel{\overline{p_1}(\widetilde{b})}{\Longrightarrow} Q_j$ for some $j\in J$.

Then $E_1[P']\to\to F[P']$, so there exists T' such that $E_1[Q_j]\Rightarrow T'$ and $F[P'] \mathrel{\dot{\approx}^{q_n:\mathtt{o}}} T'$. As $F[P']\Downarrow_{\overline{z_j}}$ for all $j\in J$ and $F[P']\not\Downarrow_{\overline{c}}$, we have that T' is of the form $F[Q_{j'}]$ with $Q_j\Rightarrow Q_{j'}$ for some $j'\in J$. We have a contradiction with $(\star)$.

- $\mu=\tau$ and $\sigma_1=\emptyset$. Given fresh names $(z_j,w_j)_{j\in J}$, we define $R \stackrel{\text{def}}{=} \tau.\sum_{j\in J}\tau.\,(R_j+S'\rhd z_j+S'\rhd w_j)$ and $R_i=\sum_j R_i^j$, and we can conclude.

- $\mu=\tau$ and $\sigma_1 = q:\mathtt{i}$. Given fresh names $(z_j)_{j\in J}$, we define $R \stackrel{\text{def}}{=} \tau.\sum_{j\in J}\tau.\,(R_j+\overline{z_j})$ and $R_i=\sum_j R_i^j$, and we can conclude.

□

5.4 Extension to Aπ

We briefly explain how the type system we have studied can be adapted from Iπ to Aπ. It was indeed introduced for Aπ in [17]. In a calculus where free names can be communicated, like Aπ, recursion is not required as it can be encoded using replication. Matching is allowed, but not on continuation names. Thus, the typing rules do not have a Γ component. We present here the main changes to the rules, that is the rules for free outputs and matching:

$$\text{WB-OUTC'}\ \frac{}{p : \mathsf{o} \vdash_{\mathrm{wb}} \overline{p}\langle\widetilde{a}\rangle} \qquad \text{WB-OUTOC'}\ \frac{}{p : \mathsf{o} \vdash_{\mathrm{wb}} \overline{x}\langle\widetilde{a}, p\rangle} \qquad \text{WB-OUTIC'}\ \frac{}{\emptyset \vdash_{\mathrm{wb}} \overline{u}\langle\widetilde{a}\rangle} \qquad \text{WB-MATCH}\ \frac{\emptyset \vdash_{\mathrm{wb}} P}{\emptyset \vdash_{\mathrm{wb}} [a = b]P}$$

The main difference can be seen in rule WB-OUTOC'. We know that for an input $x(\widetilde{a}, p)$ receive the output capability at p. As the name p is free in $\overline{x}\langle\widetilde{a}, p\rangle$, the process should have the output capability at p to send it. To compare the two systems, we can look at the two equivalent processes $(\boldsymbol{\nu}\widetilde{a}, p)(\overline{x}\langle\widetilde{a}, p\rangle \mid p(\widetilde{b}).\, Q)$ and $\overline{x}(\widetilde{a}, p).\, p(\widetilde{b}).\, Q$ in Aπ and Iπ respectively. When $q : \mathsf{o} \vdash_{\mathrm{wb}} Q$, both processes can be typed with $q : \mathsf{o}$ using their respective type system. Being able to send free continuation names allows more freedom on how continuations are used. In Section 5.2, we explain how a service B called by a service A may call another service C and wait for the return from C before resuming and eventually returning to A. If instead B sends to C the continuation name it received from A, it intuitively delegates to C the task of returning to A.

By the linearity and the absence of matching for continuation names, sending a free continuation name is contextually equivalent to sending to fresh name and "forwarding back" the result. This intuitively represents tail call optimisation and cannot be expressed easily in Iπ.

Lemma 96. $\overline{x}\langle\widetilde{a}, p\rangle \simeq_c^{p:\mathsf{o}} (\boldsymbol{\nu}q)(\overline{x}\langle\widetilde{a}, q\rangle \mid q(\widetilde{b}).\, \overline{p}\langle\widetilde{b}\rangle)$.

This property originates from [45, Theorem 6.1] and allows us to look at processes which only send bound continuation names. We call such processes *discreet*. This notion can also be extended to transitions.

Definition 97 (Discreet processes and transitions)**.** A process P is *discreet* if any free continuation name $p \in \mathrm{fn}(P)$ may not appear in the object of an output, and, in any sub-process $x(\widetilde{a}, q).\, Q$, the same holds for q in Q.

A typed transition $[\sigma; P] \xrightarrow{\mu} [\sigma'; P']$ is discreet if any continuation name in the object of μ is not free in σ (and hence also in P).

When restricting to discreet processes and transitions, an analogue of the results from Section 5.2.2 holds.

The asynchronous clause for the bisimulation is more complex when the input is an output-controlled name. As we want the processes to be discreet, when $\mu = x\langle\widetilde{a}, p\rangle$, we cannot use $Q \mid \overline{x}\langle\widetilde{a}, p\rangle$ as in Definition 23. In the light of Lemma 96, we use $(\boldsymbol{\nu}q)(\overline{x}\langle\widetilde{a}, q\rangle \mid q(\widetilde{b}).\, \overline{p}\langle\widetilde{b}\rangle)$ instead.

Definition 98 (Asynchronous wb-bisimulation)**.** A wb-typed relation $\mathcal{R}$ on discreet processes is an asynchronous wb-bisimulation if whenever $(\sigma, P, Q) \in \mathcal{R}$ and $[\sigma; P] \xrightarrow{\mu} [\sigma'; P']$ is discreet, then one of the three following clauses holds:

- there is Q' with $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $(\sigma', P', Q') \in R$
- $\mu = x\langle\widetilde{a}, p\rangle$ and for some fresh q, there is Q' such that $Q \mid (\boldsymbol{\nu}q)(\overline{x}\langle\widetilde{a}, q\rangle \mid q(\widetilde{b}).\, \overline{p}\langle\widetilde{b}\rangle) \Rightarrow Q'$ and $(\sigma', P', Q') \in \mathcal{R}$.
- $\mu = u\langle\widetilde{a}\rangle$ and there is Q' with $Q \mid \overline{u}\langle\widetilde{a}\rangle \Rightarrow Q'$ and $(\sigma', P', Q') \in \mathcal{R}$,

and symmetrically for the transitions on Q.

Using asynchronous wb-bisimulation, we can prove the soundness and completeness results of Section 5.3 with respect to the typed barbed equivalence (using asynchronous static contexts).

We comment on the difference for the proof of completeness. In Iπ, we need to have some processes R_i guarded by p_i in order to use the names received by the communication at p_i. As names can be

forwarded, it is possible to use a process of the form $p_i(\widetilde{b}).\,\overline{y_i}\langle\widetilde{b}, q_i\rangle$. Thus, it is possible to have the processes R_i inside the main process R as long as they are guarded by an input at y_i.

Following the same notations, for some fresh $\widetilde{y_i}, S \supseteq \mathrm{fn}_{\mathrm{O}}(P) \cup \mathrm{fn}_{\mathrm{O}}(Q) \cup \widetilde{y_i}$ and fresh z. we have a context of the form:

$$\overline{R} \stackrel{\text{def}}{=} \begin{cases} R + \overline{z} & \text{when } \sigma_1 = q : \mathtt{i} \\ R + S \rhd z + S \rhd w & \text{when } \sigma_1 = \emptyset \end{cases}$$

$$E_0 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{a})([\cdot] \mid \overline{R} \mid \prod_{i \leq n} p_i(\widetilde{b}).\,\overline{y_i}\langle\widetilde{b}, q_i\rangle)$$

The proof has a similar structure as the one presented in the proof of Theorem 94. However, being expressed in Aπ, the processes R are more similar to the ones presented in the proof of Theorem 64. The case for $\mu = x\langle\widetilde{a}, p\rangle$ needs extra care due to the presence of the adapted input clause.

We refer to [17] for the statement and proof of these results.

Chapter 6

Full Abstraction for a Higher-Order Language with References

In this part, we define $\lambda^{\mathtt{ref}}$, a higher-order language with references, and study its encoding in Iπ (Section 6.1). This encoding yields a full abstraction result between contextual equivalence in $\lambda^{\mathtt{ref}}$ and a refinement of wb-bisimulation we introduce: *bisimulation with divergence* (Section 6.2. We then study how up-to techniques, existing and new, can be used to simplify bisimulation proofs for this new equivalence (Section 6.3).

6.1 Encoding a higher-order language with references in Iπ

6.1.1 Definition of $\lambda^{\mathtt{ref}}$ and encoding in Iπ

We define the syntax of the λ-calculus with references, written $\lambda^{\mathtt{ref}}$:

Terms:	$M, N ::= V \mid M\ N \mid \ell \coloneqq M; N \mid\ !\ell \mid \mathtt{new}\ \ell \coloneqq V\ \mathtt{in}\ M$
Values:	$V, W ::= x \mid \lambda x.\, M$
Evaluation contexts:	$E, F ::= [\cdot] \mid E\ M \mid V\ E \mid \ell \coloneqq E; M$

Note that we use here E, F to range over *evaluation contexts* in $\lambda^{\mathtt{ref}}$, and to range over *static context* in the π-calculus. This does not raise any ambiguity in the sequel, and we shall see that the former are translated into the latter by our encoding. Free variables for terms and contexts are defined as usual with $\lambda x.\, M$ as binder. We write $M\{^{V}/_{x}\}$ for the usual capture-avoiding substitution of x by V in M. And we let $\int$ range over simultaneous substitutions $\{^{V_1}/_{x_1}\} \ldots \{^{V_n}/_{x_n}\}$ where $x_1, \ldots, x_n$ are pairwise different variables. As in the π-calculus, we use $\lambda_.\, M$ for $\lambda x.\, M$ where $x \notin \mathrm{fn}(M)$. In examples, we often use $M; N$ for $(\lambda_.\, N)M$ which computes M discard its return value and continues as N.

Free references for terms and contexts, noted $\mathrm{fr}(M)$ and $\mathrm{fr}(E)$, are defined similarly, with $\mathtt{new}\ \ell \coloneqq V\ \mathtt{in}\ M$ binding ℓ in M. Notice that in $\lambda^{\mathtt{ref}}$, ℓ is not a value, meaning that in $\mathtt{new}\ \ell \coloneqq V\ \mathtt{in}\ M$, ℓ is local to the term M. To give access to a local reference, M may pass functions $\lambda x.\, !\ell$ and $\lambda x.\, \ell \coloneqq x$ instead of ℓ.

A store, noted $h, g, \ldots$, is a partial function with finite domain from references to values. We write $\emptyset$ for the empty store and $\mathrm{dom}(h)$ for the set of references on which h is defined. For any ℓ, V, we write $h \uplus \ell = V$ for the store h extended with a reference ℓ containing V and $h[\ell \coloneqq V]$ for the store h with the content of ℓ updated to the value V.

The semantics are defined on *configurations* $\langle h \mid M \rangle$ where h is a store. We assume that for all configurations $\langle h \mid M \rangle$, we have $\mathrm{fr}(M) \subseteq \mathrm{dom}(h)$ and for all $\ell \in \mathrm{dom}(h)$, $\mathrm{fr}(h(\ell)) \subseteq \mathrm{dom}(h)$. This assumption ensures that any free reference used in terms is defined in h.

Reductions between configurations are defined as follows:

$$\langle h \mid (\lambda x.\, M)V \rangle \to \langle h \mid M\{^{V}/_{x}\} \rangle \qquad (\beta)$$

$$\langle h \mid !\ell \rangle \to \langle h \mid h(\ell) \rangle \qquad \text{(Read)}$$

$$\langle h \mid \ell \coloneqq V; M \rangle \to \langle h[\ell \coloneqq V] \mid M \rangle \qquad \text{(Write)}$$

$$\langle h \mid \texttt{new}\ \ell \coloneqq V\ \texttt{in}\ M \rangle \to \langle h \uplus \ell = V \mid M \rangle \qquad \text{(Alloc)}$$

$$\langle h \mid E[M] \rangle \to \langle g \mid E[N] \rangle \qquad \text{if } \langle h \mid M \rangle \to \langle g \mid N \rangle \qquad \text{(Eval)}$$

A term M in a configuration $\langle h \mid M \rangle$ which cannot reduce is called a normal form. A normal form can either be a value, or a stuck term of the form $E[yV]$.

Lemma 99. For any $\langle h \mid M \rangle$ we have:

1. either $\langle h \mid M \rangle \to \langle h' \mid M' \rangle$ for some $\langle h' \mid M' \rangle$
2. or M is a value
3. or M is of the form $E[yV]$.

We write $\Rightarrow$ for $\to^*$ and we say that $\langle h \mid M \rangle$ and $\langle g \mid N \rangle$ *co-terminate* when $\langle h \mid M \rangle \Rightarrow \langle h' \mid M' \rangle$ with M' being a normal form iff $\langle g \mid N \rangle \Rightarrow \langle g' \mid N' \rangle$ with N' being a normal form. A term (resp. evaluation context) is *closed* (resp. *reference-closed*) if its set of free variables (resp. free references) is empty. A substitution $\int$ is *closing terms* $M, N, \ldots$ if $M\int, N\int, \ldots$ are closed. As closed terms do not have free variables, the normal form they may reduce to can only be a value, more specifically a function like $\lambda x.\, M$. The following equivalence coincides with a more standard contextual equivalence quantified over all contexts [31].

Definition 100. Two reference-closed terms are contextually equivalent, written $M \asymp N$, if for all closing substitutions $\int$ and reference-closed evaluation contexts E, $\langle \emptyset \mid E[M]\int \rangle$ and $\langle \emptyset \mid E[N]\int \rangle$ co-terminate.

Milner [35] described the first encodings from the λ-calculus to the π-calculus for both call-by-name and call-by-value. These encodings are shown to be sound with respect to contextual equivalence, enabling the use of the π-calculus as a model to prove equivalence between λ-terms. In sequential languages like the λ-calculus or extensions thereof, contextual equivalence is often considered as the canonical equivalence.

Nevertheless, for sequential languages, while such equivalences in the π-calculus lead to sound encodings with respect to contextual equivalence, completeness does not hold, due to the parallel nature of the π-calculus. Full abstraction is obtained for stronger equivalences in the source language, generally based on trees: Lévy-Longo Trees for the call-by-name λ-calculus [49], and η-eager normal form bisimilarity for call-by-value [10]. More generally, we are not aware of full abstraction results relating a contextual equivalence in a sequential language and a labelled bisimilarity between process terms in the encoding. Existing results either use a finer relation than contextual equivalence in the source language, or obtain full abstraction for a contextually-defined equivalence in the π-calculus [5, 54].

We now give more details about the encoding from $\lambda^{\texttt{ref}}$ to the π-calculus, described in Figure 6.1. Milner's original encoding of the untyped call-by-value λ-calculus does not handle equivalences between terms with free variables well. This issue is amended in [10] by moving to the Internal π-calculus, where only fresh names can be transmitted. For instance, the resulting encodings of $(\lambda z.\, z)(xV)$ and xV are equivalent, which is not the case for Milner's original encoding. The encoding of a term $[\![M]\!]$ is an abstraction as used for recursive definition in Section 5.1. The encoding is parametrised by a continuation name p that is used to send the result of the computation. To express this parametricity, we write $[\![M]\!]_p$ in place of $[\![M]\!]\lfloor p \rfloor$. The names sent by the continuation correspond to values that are encoded using a second encoding $[\![V]\!]^{\texttt{v}}$. Intuitively, a function is encoded as a server which receives its argument and a continuation name, then computes a value which is sent back using the continuation name provided. Thus, in the encoding of $[\![MN]\!]$, the value corresponding to M is computed first and the name y for calling the value is sent at q, then the one for N is computed and the name w is sent at r, finally we send w at y to call the function. The encoding for imperative construct follows the encoding of $\pi^{\texttt{ref}}$ from Section 3.3.

To analyse the encoding, we define a simple sorting with 3 sorts: $\mathtt{F}$,$\mathtt{R}$,$\mathtt{C}$. The sorting function Σ is defined by $\Sigma(\mathtt{F}) = (\mathtt{F}, \mathtt{C})$ and $\Sigma(\mathtt{R}) = \Sigma(\mathtt{C}) = \mathtt{F}$. We call *function names*, ranged over with $w, x, y, z, \ldots$, names in $\mathtt{F}$, *reference names*, ranged over with $\ell, \ell', \ldots$, names in $\mathtt{R}$, and *continuation names*, ranged over with $p, q, r, \ldots$, names in $\mathtt{C}$. In fact, they correspond to the output-controlled, input-controlled and continuation names from Chapter 5 but with a defined arity. These names also correspond to their usage in the encoding.

We use two constants, noted $\rhd_{\mathtt{F}}$ and $\rhd_{\mathtt{C}}$, defined as follows:

$$\begin{array}{ll} \rhd_{\mathtt{C}} \triangleq (p,q)\ p(x).\,\overline{q}(y)\ y \rhd_{\mathtt{F}} x & \text{with } p,q \text{ being continuation names} \\ \rhd_{\mathtt{F}} \triangleq (x,y)\ !x(z,p).\,\overline{y}(w,q)\ (q \rhd_{\mathtt{C}} p \mid w \rhd_{\mathtt{F}} z) & \text{with } x,y \text{ being function names} \end{array}$$

These constants represent a link, also called forwarder or dynamic wire [46], which transforms outputs at the first name into outputs at the second, and we write $p \rhd_{\mathtt{C}} q$ in place of $\rhd_{\mathtt{C}} \lfloor p, q \rfloor$, and similarly for $\rhd_{\mathtt{F}}$. Since sending free names is impossible in Iπ, a forwarder also creates additional forwarder for each of the names sent. As the usage of forwarders only differs in the linearity of continuation names and the arity, we use the same symbol $\rhd$ to denote both $\rhd_{\mathtt{F}}$ and $\rhd_{\mathtt{C}}$.

Functions

$$[\![V]\!] \stackrel{\text{def}}{=} (p)\ \overline{p}(y)\ [\![V]\!]^{\mathtt{v}}_{y} \qquad\qquad [\![xV]\!] \stackrel{\text{def}}{=} (p)\ \overline{x}(z,q).\,([\![V]\!]^{\mathtt{v}}_{z} \mid q \rhd p)$$

$$[\![(\lambda x.\,N)V]\!] \stackrel{\text{def}}{=} (p)\ (\boldsymbol{\nu} y, w)([\![\lambda x.\,N]\!]^{\mathtt{v}}_{y} \mid [\![V]\!]^{\mathtt{v}}_{w} \mid \overline{y}(w', r').\,(w' \rhd w \mid r' \rhd p))$$

$$[\![VM]\!] \stackrel{\text{def}}{=} (p)\ (\boldsymbol{\nu} y)([\![V]\!]^{\mathtt{v}}_{y} \mid (\boldsymbol{\nu} r)([\![M]\!]_{r} \mid r(w).\,\overline{y}(w', r').\,(w' \rhd w \mid r' \rhd p)))$$

$$[\![MN]\!] \stackrel{\text{def}}{=} (p)\ (\boldsymbol{\nu} q)([\![M]\!]_{q} \mid q(y).\,(\boldsymbol{\nu} r)([\![N]\!]_{r} \mid r(w).\,\overline{y}(w', r').\,(w' \rhd w \mid r' \rhd p)))$$

Imperative constructs

$$[\![!\ell]\!] \stackrel{\text{def}}{=} (p)\ \ell(w).\,(\overline{\ell}(y)\ y \rhd w \mid \overline{p}(z)\ z \rhd w) \qquad\qquad [\![\ell \coloneqq V; N]\!] \stackrel{\text{def}}{=} (p)\ \ell(_).\,(\overline{\ell}(y)\ [\![V]\!]^{\mathtt{v}}_{y} \mid [\![N]\!]_{p})$$

$$[\![\ell \coloneqq M; N]\!] \stackrel{\text{def}}{=} (p)\ (\boldsymbol{\nu} q)([\![M]\!]_{q} \mid q(w).\,\ell(_).\,(\overline{\ell}(y)\ y \rhd w \mid [\![N]\!]_{p}))$$

$$[\![\mathtt{new}\ \ell \coloneqq V\ \mathtt{in}\ N]\!] \stackrel{\text{def}}{=} (p)\ (\boldsymbol{\nu} q)([\![V]\!]_{q} \mid q(z).\,(\boldsymbol{\nu}\ell)(\overline{\ell}(y)\ y \rhd z \mid [\![N]\!]_{p}))$$

Configurations

$$[\![h]\!] \stackrel{\text{def}}{=} \prod_{\ell_0 \in \widetilde{\ell}} (\overline{\ell_0}(y)\ [\![h(\ell_0)]\!]^{\mathtt{v}}_{y}) \text{ with } \widetilde{\ell} = \mathrm{dom}(h) \qquad\qquad [\![\langle h \mid N \rangle]\!] \stackrel{\text{def}}{=} (p)\ (\boldsymbol{\nu}\widetilde{\ell})([\![h]\!] \mid [\![N]\!]_{p})$$

where $[\![V]\!]^{\mathtt{v}}$ is defined as:

$$[\![\lambda x.\,N]\!]^{\mathtt{v}} \stackrel{\text{def}}{=} (y)\ !y(x,q).\,[\![N]\!]_{q} \qquad\qquad [\![x]\!]^{\mathtt{v}} \stackrel{\text{def}}{=} (y)\ y \rhd x$$

Figure 6.1: Encoding of terms and configurations into Iπ

Our results are based on the optimised version of the encoding of [10], where administrative reductions introduced in the encoding of the application are removed. In the optimised encoding, a stuck term like xV is translated into a process that cannot reduce. Additionally, the translation of an evaluation context always yields a static context, which is not the case in the encoding of [10]. These two properties lead to a tight statement of operational correspondence (Proposition 105) : the encoding of a $\lambda^{\mathtt{ref}}$-term can perform an internal communication only if the source term has a reduction.

Remark 101. In the encoding of Figure 6.1, outputs are always followed by processes guarded by an input at names bound by that output. This is a particular case, where the asynchrony does not affect

the behaviour. Indeed, in the Iπ-calculus, we have the following law:

$$\overline{a}(y).\, y(\widetilde{b}).\, P \sim \overline{a}(y)\ y(\widetilde{b}).\, P$$

and similarly with $!y(\widetilde{b}).\, P$. As y is bound, the input at y cannot be done before the output that sends y, whether this output is synchronous or not. Thus, outputs at continuation names are asynchronous so that the encoding of a term can be typed, and similarly with references names to be consistent with Chapter 3. In fact, the type system for references for Aπ can be adapted to Iπ and our encoding could still be typed.

6.1.2 References in $\lambda^{\mathtt{ref}}$ and well-bracketing

Consider the two following $\lambda^{\mathtt{ref}}$ programs:

$$M_1 \stackrel{\text{def}}{=} \mathtt{new}\ \ell := 0\ \mathtt{in}\ \lambda y.\, \ell := 0; y(); \ell := 1; y(); !\ell$$
$$M_2 \stackrel{\text{def}}{=} \lambda y.\, y(); y(); 1$$

Function M_2 makes two calls to an external function y and returns 1. The other term, M_1, between the two calls, modifies a local reference ℓ, which is then used to return the final result. Intuitively, equivalence between the two functions holds because: (i) the reference ℓ in M_1 represents a local state, not accessible from an external function; (ii) computation respects well-bracketing (e.g. the language does not have control operators like call/cc). (with such an operator, it would be possible to reinstall the value 0 at ℓ, and obtain 0 as final result).

Note that we are outside Iπ because we are using integers 0,1 and unit (). Adapting our setting to simple types can be done by having the sorts $\mathtt{F}, \mathtt{R}, \mathtt{C}$ and forwarders $\rhd_{\mathtt{C}}$,$\rhd_{\mathtt{F}}$ for each type T, the forwarders being defined inductively on T instead of being recursive constants.

For instance, with $M; N$ standing for $(\lambda_ N)M$, we would have:

$$[\![y(); M]\!]_p \stackrel{\text{def}}{=} (\boldsymbol{\nu} x, r)(!x(_, p).\, [\![M]\!]_p \mid (\boldsymbol{\nu} q)\overline{y}\langle \star, q\rangle.\, q(u).\, \overline{r}\langle u\rangle \mid r(u).\, \overline{x}\langle u, p\rangle)$$

Here, y is a function from unit to unit, thus u may only be instantiated with $\star$ which is the constant unit. For clarity, we remove the use of unit, e.g. writing $\overline{y}\langle q\rangle$ and $\overline{q}$ instead. This process can be optimised by performing the deterministic communications:

$$[\![y(); M]\!]_p \gtrsim \overline{y}(q).\, q.\, [\![M]\!]_p$$

Below are the translations of M_1 and M_2, following the encoding of functions and references defined above, using the notations for references from Section 3.2.4, and the optimisation presented above:

$$[\![M_1]\!]_{p'} \gtrsim (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid \overline{p'}(x)\ !x(y, p).\, \ell \lhd 0.\, \overline{y}(q).\, q.\, \ell \lhd 1.\, \overline{y}(r).\, r.\, \ell \rhd (n).\, \overline{p}\langle n\rangle)$$
$$[\![M_2]\!]_{p'} \gtrsim \overline{p'}(x).\, !x(y, p).\, \overline{y}(q).\, q.\, \overline{y}(r).\, r.\, \overline{p}\langle 1\rangle$$

$[\![M_2]\!]_{p'}$ has a unique transition, $[\![M_2]\!]_{p'} \xrightarrow{(\boldsymbol{\nu} x)\overline{p'}\langle x\rangle} [\![M_2]\!]^{\mathtt{v}}_x$. Let $Q = [\![M_2]\!]^{\mathtt{v}}_x$ and similarly, let P be the unique derivative from $[\![M_1]\!]_{p'}$. The equivalence between $[\![M_1]\!]_{p'}$ and $[\![M_2]\!]_{p'}$ follows immediately from $P \approx^{\emptyset}_{\text{wb}} Q$. To prove the latter, we exhibit a relation $\mathcal{R}$ containing the triple $(\emptyset, P, Q)$ and show that $\mathcal{R}$ is a wb-bisimulation up-to deterministic reductions and static context.

We need to introduce some notations in order to reason about the sub-processes which are generated by calls to the functions in the encodings of M_1 and M_2. To simplify notations, we assume that names extruded from the two outputs at y are in the set $\{q_i \mid i \in I\}$ and $\{r_j \mid j \in J\}$ respectively for some set I (resp. J). Continuation names received on x will be noted p_i or p_j depending on how many output prefixes at y they are underneath. For instance, in the processes below P^1_i use a continuation name p_i while P^2_j use p_j.

$$P^1_i = q_i.\, \overline{y}(r).\, r.\, \overline{p_i}\langle 1\rangle \qquad P^2_j = r_j.\, \overline{p_j}\langle 1\rangle \qquad Q_0 = !x(y, p).\, \ell \lhd 0.\, \overline{y}(q).\, q.\, \ell \lhd 1.\, \overline{y}(r).\, r.\, \ell \rhd (n).\, \overline{p}\langle n\rangle$$

$$Q^1_i = q_i.\, \ell \lhd 1.\, \overline{y}(r).\, r.\, \ell \rhd (n).\, \overline{p_i}\langle n\rangle \qquad Q^2_j = r_j.\, \ell \rhd (n).\, \overline{p_j}\langle n\rangle$$

Given an ordered list s of the indices in $I \uplus J$, we write $\sigma(s)$ for the stack defined inductively as follows:

$$\sigma(\emptyset) = \emptyset \qquad \sigma(i;s) = q_i : \mathtt{i}, p_i : \mathtt{o}, \sigma(s) \qquad \sigma(j;s) = r_j : \mathtt{i}, p_j : \mathtt{o}, \sigma(s)$$

We can then define the relation:

$$\begin{aligned}\mathcal{R} \stackrel{\text{def}}{=} \{&(\emptyset, P, Q), \quad (\emptyset, P, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid Q_0)),\\ &(\sigma(i;s),\ P \mid \prod_{i\in I} P_i^1 \mid \prod_{j\in J} P_j^2,\ (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid Q_0 \mid \prod_{i\in I} Q_i^1 \mid \prod_{j\in J} Q_j^2)),\\ &(\sigma(i;s),\ P \mid \prod_{i\in I} P_i^1 \mid \prod_{j\in J} P_j^2,\ (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid Q_0 \mid \prod_{i\in I} Q_i^1 \mid \prod_{j\in J} Q_j^2)),\\ &(\sigma(j;s),\ P \mid \prod_{i\in I} P_i^1 \mid \prod_{j\in J} P_j^2,\ (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid Q_0 \mid \prod_{i\in I} Q_i^1 \mid \prod_{j\in J} Q_j^2))\\ \}&\end{aligned}$$

$\mathcal{R}$ is a wb-bisimulation up-to $\gtrsim$ and up-to static context.

For an example of the importance of well-bracketing, $\mathcal{R}$ contains the triple $(\sigma(2;1), (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid Q_0 \mid Q_2^1 \mid Q_1^2), P \mid P_2^1 \mid P_1^2)$ which expands to:

$$\begin{array}{lcll}\big(q_2 : \mathtt{i}, p_2 : \mathtt{o}, r_1 : \mathtt{i}, p_1 : \mathtt{o}, & P & \mid q_2.\,\overline{y}(r).\,r.\,\overline{p_2}\langle 1\rangle & \mid r_1.\,\overline{p_1}\langle 1\rangle,\\ & (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid Q_0 & \mid q_2.\,\ell \triangleleft 1.\,\overline{y}(r).\,r.\,\ell \triangleright (n).\,\overline{p_2}\langle n\rangle & \mid r_1.\,\ell \triangleright (n).\,\overline{p_1}\langle n\rangle\big)\end{array}$$

Without the well-bracketing constraint, the first process in the triple could perform an input at r_1, an internal transition, and finally an output $\overline{p_1}\langle 0\rangle$. The second process cannot emit 0, which would allow us to distinguish P and Q. With well-bracketing, since r_1 is not on top of the stack in the triple, the initial transition on r_1 is ruled out.

In Appendix B.1, we present a simpler example to show the impact of up-to techniques in reducing the size of the relation.

6.1.3 Technical results about the encoding

For any reference-closed configurations $\langle h \mid M\rangle$, we have $p : \mathtt{o} \vdash_{\text{wb}} [\![\langle h \mid M\rangle]\!]_p$.

We extend the encoding to evaluation contexts as shown in Figure 6.2. When an evaluation context

$$[\![[\cdot]]\!] \stackrel{\text{def}}{=} [\cdot] \qquad [\![FN]\!] \stackrel{\text{def}}{=} (p)\ (\boldsymbol{\nu}q)([\![F]\!]_q \mid q(y).\,(\boldsymbol{\nu}r)([\![N]\!]_r \mid r(w).\,\overline{y}(w', r').\,(w' \triangleright w \mid r' \triangleright p)))$$

$$[\![\ell := F; N]\!] \stackrel{\text{def}}{=} (p)\ (\boldsymbol{\nu}q)([\![F]\!]_q \mid q(w).\,\ell(_).\,(\overline{\ell}(y)\ y \triangleright w \mid [\![N]\!]_p))$$

$$[\![VF]\!] \stackrel{\text{def}}{=} (p)\ (\boldsymbol{\nu}y)([\![V]\!]^{\mathtt{v}}_y \mid (\boldsymbol{\nu}r)([\![F]\!]_r \mid r(w).\,\overline{y}(w', r').\,(w' \triangleright w \mid r' \triangleright p)))$$

Figure 6.2: Encoding of evaluation contexts

is applied to a non-value term, its encoding correspond to encode the context first and then apply it to the encoding of the term:

Lemma 102. For any E M such that M is not a value, we have $[\![E[M]]\!]_p = [\![E]\!][\![M]\!]]_p$.

Proof. We proceed by induction on E:

1. when $E = [\cdot]$, the result is immediate.
2. when $E = FN$, we have

$$[\![E[M]]\!]_p = (\boldsymbol{\nu}q)([\![F[M]]\!]_q \mid q(y).\,(\boldsymbol{\nu}r)([\![N]\!]_r \mid r(w).\,\overline{y}(w', r').\,(w' \triangleright w \mid r' \triangleright p))).$$

By induction, we have that $[\![F[M]]\!]_q = [\![F]\!][[\![M]]\!]_q$ and thus:

$$\begin{aligned}[\![E[M]]\!]_p &= (\boldsymbol{\nu} q)([\![F]\!][[\![M]]\!]_q \mid q(y).\,(\boldsymbol{\nu} r)([\![N]\!]_r \mid r(w).\,\overline{y}(w', r').\,(w' \rhd w \mid r' \rhd p))) \\ &= [\![E]\!][[\![M]]\!]_p\end{aligned}$$

3. when $E = VF$, we have

$$[\![E[M]]\!]_p = (\boldsymbol{\nu} y, r)([\![V]\!]^{\mathrm{v}}_y \mid [\![F[M]]\!]_r \mid r(w).\,\overline{y}(w', r').\,(w' \rhd w \mid r' \rhd p)).$$

   By induction, we have that $[\![F[M]]\!]_r = [\![F]\!][[\![M]]\!]_r$.
   So $[\![E[M]]\!]_p = (\boldsymbol{\nu} y, r)([\![V]\!]^{\mathrm{v}}_y \mid [\![F]\!][[\![M]]\!]_r \mid r(w).\,\overline{y}(w', r').\,(w' \rhd w \mid r' \rhd p)) = [\![E]\!][[\![M]]\!]_p$.

4. when $E = \ell \coloneqq F; N$, then $[\![E[M]]\!]_p = (\boldsymbol{\nu} q)([\![F[M]]\!]_q \mid q(w).\,\ell(_).\,(\overline{\ell}(y)\ y \rhd w \mid [\![N]\!]_p))$.
   By induction, we have that $[\![F[M]]\!]_q = [\![F]\!][[\![M]]\!]_q$.
   So $[\![E[M]]\!]_p = (\boldsymbol{\nu} q)([\![F]\!][[\![M]]\!]_q \mid q(w).\,\ell(_).\,(\overline{\ell}(y)\ y \rhd w \mid [\![N]\!]_p)) = [\![E]\!][[\![M]]\!]_p$. □

Because we distinguish between values and non-values in the encoding, the previous lemma does not hold when M is a value. In that case, the encoding performs some "optimisations". To relate the two processes, we need to establish that on the encoding, forwarders act like substitutions.

Lemma 103. We have

1. $(\boldsymbol{\nu} x)([\![M]\!]_p \mid x \rhd y) \gtrsim [\![M\{y/\!_x\}]\!]_p$
2. $(\boldsymbol{\nu} x)([\![V]\!]^{\mathrm{v}}_z \mid x \rhd y) \gtrsim [\![V\{y/\!_x\}]\!]^{\mathrm{v}}_z$
3. $(\boldsymbol{\nu} p)([\![M]\!]_p \mid p \rhd q) \gtrsim [\![M]\!]_q$
4. $(\boldsymbol{\nu} y)([\![V]\!]^{\mathrm{v}}_y \mid x \rhd y) \gtrsim [\![V]\!]^{\mathrm{v}}_x$

The proof is given in Appendix B.2. As a result of the optimisation, we have the following lemma when applying a context to a value:

Lemma 104. For any value V and context E, $[\![E]\!][[\![V]]\!]_p \gtrsim [\![E[V]]\!]_p$.

Proof. We proceed by induction on E:

- when $E = [\cdot]$, this is trivial.
- When $E = (\lambda x.\,M)\ [\cdot]$, we have

$$\begin{aligned}[\![E]\!][[\![V]]\!]_p &= (\boldsymbol{\nu} y, r)([\![\lambda x.\,M]\!]^{\mathrm{v}}_y \mid [\![V]\!]_r \mid r(w).\,\overline{y}(w', r').\,(w' \rhd w \mid r' \rhd p)) \\ &\quad \to (\boldsymbol{\nu} y, w)([\![\lambda x.\,M]\!]^{\mathrm{v}}_y \mid [\![V]\!]^{\mathrm{v}}_w \mid \overline{y}(w', r').\,(w' \rhd w \mid r' \rhd p)) = [\![E[V]]\!]_p\end{aligned}$$

  As this transition is deterministic by construction, we obtain that $[\![E]\!][[\![V]]\!]_p \gtrsim [\![E[V]]\!]_p$. This also holds for the three following cases.

- When $E = x\ [\cdot]$, we have

$$\begin{aligned}[\![E]\!][[\![V]]\!]_p &= (\boldsymbol{\nu} y, r)([\![x]\!]^{\mathrm{v}}_y \mid [\![V]\!]_r \mid r(w).\,\overline{y}(w', r').\,(w' \rhd w \mid r' \rhd p)) \\ &\quad \to^2 (\boldsymbol{\nu} y, w, w', r')([\![x]\!]^{\mathrm{v}}_y \mid \overline{x}(w'', r'').\,(w'' \rhd w' \mid r'' \rhd r') \mid [\![V]\!]^{\mathrm{v}}_w \mid w' \rhd w \mid r' \rhd p) \\ &\quad \gtrsim (\boldsymbol{\nu} y)[\![x]\!]^{\mathrm{v}}_y \mid \overline{x}(w'', r'').\,(\boldsymbol{\nu} w, w', r')(w'' \rhd w' \mid w' \rhd w \mid [\![V]\!]^{\mathrm{v}}_w \mid r'' \rhd r' \mid r' \rhd p) \\ &\quad \gtrsim (\boldsymbol{\nu} w)(\overline{x}(w'', r'').\,([\![V]\!]^{\mathrm{v}}_{w''} \mid r'' \rhd p)) \equiv [\![xV]\!]_p\end{aligned}$$

- When $E = [\cdot]\ M$, we have

$$\begin{aligned}[\![E]\!][[\![V]]\!]_p &= (\boldsymbol{\nu} q)([\![V]\!]_q \mid q(y).\,(\boldsymbol{\nu} r)([\![N]\!]_r \mid r(w).\,\overline{y}(w', r').\,(w' \rhd w \mid r' \rhd p))) \\ &\quad \to (\boldsymbol{\nu} y)([\![V]\!]^{\mathrm{v}}_y \mid (\boldsymbol{\nu} r)([\![N]\!]_r \mid r(w).\,\overline{y}(w', r').\,(w' \rhd w \mid r' \rhd p))) = [\![xV]\!]_p\end{aligned}$$

- When $E = \ell \coloneqq [\cdot]; M$, we have

$$\begin{aligned}[\![E]\!][\![V]\!]_p &= (\boldsymbol{\nu} q)([\![V]\!]_q \mid q(w).\, \ell(_).\, (\overline{\ell}(y)\ y \rhd w \mid [\![N]\!]_p)) \\ &\to (\boldsymbol{\nu} w)([\![V]\!]^{\mathtt{v}}_w \mid \ell(_).\, (\overline{\ell}(y)\ y \rhd w \mid [\![N]\!]_p)) \\ &\gtrsim \ell(_).\, (\overline{\ell}(y)\ (\boldsymbol{\nu} w)([\![V]\!]^{\mathtt{v}}_w \mid y \rhd w) \mid [\![N]\!]_p) \\ &\gtrsim \ell(_).\, (\overline{\ell}(y)\ [\![V]\!]^{\mathtt{v}}_y \mid [\![N]\!]_p) = [\![E[V]]\!]_p\end{aligned}$$

- when $E = F\ M$ or $V\ F$ or $\ell \coloneqq F; M$ with $F \neq [\cdot]$, then $F[V]$ is not a value, so $[\![E[V]]\!]_p = [\![E']\!][\![F[M]]\!]_p$ for some E' and the result follows by induction as $\gtrsim$ is a congruence. □

We can now establish operational correspondence.

Proposition 105 (Untyped Operational Correspondence)**.** For any M, h with $\mathrm{dom}(h) = \widetilde{\ell}$ and fresh q_0, $[\![\langle h \mid M\rangle]\!]_{q_0}$ has exactly one immediate transition, and exactly one of the following clauses holds:

1. $\langle h \mid M\rangle \to \langle h' \mid N\rangle$ and $[\![\langle h \mid M\rangle]\!]_{q_0} \xrightarrow{\tau} P$ with $P \gtrsim [\![\langle h' \mid N\rangle]\!]_{q_0}$
2. M is a value, $[\![\langle h \mid M\rangle]\!]_{q_0} \xrightarrow{\overline{q_0}(x_0)} P$ and $P = (\boldsymbol{\nu}\widetilde{\ell})([\![h]\!] \mid [\![M]\!]^{\mathtt{v}}_{x_0})$.
3. M is of the form $E_0[yV_0]$ for some E_0, y and V_0, and we have $[\![\langle h \mid M\rangle]\!]_{q_0} \xrightarrow{\overline{y}(x_0,p_0)} P$ with $P \gtrsim (\boldsymbol{\nu}\widetilde{\ell})([\![h]\!] \mid [\![V_0]\!]^{\mathtt{v}}_{x_0} \mid p_0(z).\, [\![E_0[z]]\!]_{q_0})$.

In the first case, the τ transition is deterministic, so we can prove that the following holds:

$$[\![\langle h \mid M\rangle]\!]_{q_0} \gtrsim [\![\langle h' \mid N\rangle]\!]_{q_0}$$

6.1.4 Additional examples

Using the existing equivalences in Iπ, we can prove that the encodings of two $\lambda^{\mathtt{ref}}$-terms are equivalent. The equivalence and preorder used here are finer than bisimilarity with divergence, our main equivalence to study $\lambda^{\mathtt{ref}}$ programs in the π-calculus, which is sound as stated in Section 6.2.2. These examples show how the standard theory of the π-calculus is enough to prove interesting properties of $\lambda^{\mathtt{ref}}$-terms. These properties do not require us to use the triples defined in Section 6.2.2 nor to take into account the possibility of deferred divergence.

Example 106 (Unused reference)**.** For any reference ℓ, value V and any term M with $\ell \notin \mathrm{fr}(M)$, we have $[\![\mathtt{new}\ \ell \coloneqq V\ \mathtt{in}\ M]\!]_p \gtrsim [\![M]\!]_p$.

Proof. We write

$$\begin{aligned}[\![\mathtt{new}\ \ell \coloneqq V\ \mathtt{in}\ M]\!]_p &\gtrsim [\![\langle \emptyset \mid \mathtt{new}\ \ell \coloneqq V\ \mathtt{in}\ M\rangle]\!]_p \\ &\gtrsim [\![\langle \ell = V \mid M\rangle]\!]_p && \text{by Proposition 105} \\ &\gtrsim (\boldsymbol{\nu}\ell)(\overline{\ell}(y)\ [\![V]\!]^{\mathtt{v}}_y) \mid [\![M]\!]_p \gtrsim [\![M]\!]_p\end{aligned}$$

We use $\gtrsim$ to perform a deterministic reduction and then to use simple laws and remove inaccessible processes. This result can be extended in presence of store by congruence, giving $[\![\langle h \mid \mathtt{new}\ \ell \coloneqq V\ \mathtt{in}\ M\rangle]\!]_p \gtrsim [\![\langle h \mid M\rangle]\!]_p$. □

Example 107 (One-use context)**.** Let $f_1 \stackrel{\mathrm{def}}{=} \lambda x.\, \mathtt{if}\ !\ell = \mathtt{tt}\ \mathtt{then}\ \ell \coloneqq \mathbf{ff}; \mathtt{tt}\ \mathtt{else}\ \mathbf{ff}$ and $f_2 \stackrel{\mathrm{def}}{=} \lambda x.\, \mathtt{tt}$ and $E = [\cdot]\ (x \lambda y.\, y)$. Then $[\![\mathtt{new}\ \ell \coloneqq \mathtt{tt}\ \mathtt{in}\ E[f_1]]\!]_p \approx [\![E[f_2]]\!]_p$

Proof. By Proposition 105 and congruence of $\gtrsim$ we have:

$$\begin{aligned}[\![E[f_2]]\!]_p &\gtrsim \overline{x}(z,q).\, ([\![\lambda y.\, y]\!]^{\mathtt{v}}_z \mid q(w).\, [\![f_2 w]\!]_p) \\ &\gtrsim \overline{x}(z,q).\, ([\![\lambda y.\, y]\!]^{\mathtt{v}}_z \mid q(w).\, [\![\mathtt{tt}]\!]_p)\end{aligned}$$

Using standard laws for $\gtrsim$, we relate the encoding of the first program to the same process:

$$\begin{aligned}
[\![\texttt{new}\ \ell \coloneqq \mathbf{tt}\ \texttt{in}\ E[f_1]]\!]_p &\gtrsim (\boldsymbol{\nu}\ell)([\![\ell = \mathbf{tt}]\!] \mid \overline{x}(z,q).\,([\![\lambda y.\, y]\!]^{\mathtt{v}}_z \mid q(w).\,[\![f_1 w]\!]_p)) \\
&\gtrsim \overline{x}(z,q).\,([\![\lambda y.\, y]\!]^{\mathtt{v}}_z \mid q(w).\,[\![\langle \ell = \mathbf{tt} \mid f_1 w\rangle]\!]_p) \\
&\gtrsim \overline{x}(z,q).\,([\![\lambda y.\, y]\!]^{\mathtt{v}}_z \mid q(w).\,[\![\langle \ell = \mathbf{ff} \mid \mathbf{tt}\rangle]\!]_p) \\
&\gtrsim \overline{x}(z,q).\,([\![\lambda y.\, y]\!]^{\mathtt{v}}_z \mid q(w).\,[\![\mathbf{tt}]\!]_p)
\end{aligned}$$

□

6.2 Full abstraction

We present here the characterisation of contextual equivalence in $\lambda^{\mathtt{ref}}$ using a labelled bisimilarity in the π-calculus. To define our labelled bisimulation, we rely on a refinement of wb-bisimulation.

This study allows us to understand the expressiveness of wb-bisimulation and to improve it. As mentioned in [6], to capture contextual equivalence in $\lambda^{\mathtt{ref}}$, *deferred divergent terms* need to be taken into account. Intuitively such terms hide a divergence behind stuck terms, like e.g. in $(\lambda y.\,\Omega)(xV)$, where Ω is an always diverging term and V is a value: the stuck term xV prevents the β-reduction yielding Ω to be applied. Still, this term is contextually equivalent to Ω. We define an equivalent notion for typable π-terms, called π-divergence, which is used to refine wb-bisimulation yielding a notion of *bisimulation with divergence* in the π-calculus.

We do not prove full abstraction with respect to contextual equivalence in $\lambda^{\mathtt{ref}}$ directly, but rather exploit its characterisation using *normal form bisimilarity* [6]. Normal form bisimulations in [6], which we call nfb in the sequel, and bisimulations with divergence in the π-calculus are strongly connected, and we can establish the following result:

> If $\mathcal{R}$ is a nfb, then the encoding of $\mathcal{R}$ is a bisimulation with divergence up to expansion in the π-calculus. (6.1)

This property allows us to prove that bisimilarity with divergence is complete w.r.t. nfb. Thanks to (6.1), when proving equivalences of $\lambda^{\mathtt{ref}}$-terms, the π-calculus machinery running under the hood is almost invisible: the bisimulations we manipulate are mainly built using (the encoding of) $\lambda^{\mathtt{ref}}$-terms. This makes π-calculus a powerful environment to prove equivalences between $\lambda^{\mathtt{ref}}$ programs. We illustrate this power via several examples throughout the paper. In some cases, we are able to compare $\lambda^{\mathtt{ref}}$ programs using equivalences which are finer, and simpler to use, than bisimulation with divergence.

6.2.1 Normal form bisimilarity

To define normal form bisimulations, we need to introduce the notion of *triples*, used in [6].

We use a tilde, like in $\widetilde{V_i}$, to denote a (possibly empty) tuple. Triples are of the form $(\widetilde{V_i}, \sigma, c)$ and $(\widetilde{V_i}, \sigma, h)$ where: $\widetilde{V_i}$ is a tuple of values, σ is a stack of evaluation contexts, c is a configuration and h is a store. These are the elements being compared in a normal form bisimulation. We provide some comments about the role of triples.

The tuple $\widetilde{V_i}$ stores the values accumulated by the environment, and we write $(\widetilde{V_i}, V)$ for the tuple $\widetilde{V_i}$ extended with the additional value V. A stack of evaluation contexts σ corresponds to interrupted contexts that must be executed to complete the computation. $\odot$ stands for the empty stack and $E :: \sigma$ for the stack obtained by adding E on top of σ. The store is accessible by all the other objects of the triple, values or contexts – in $(\widetilde{V_i}, \sigma, c)$, the store is part of c.

Intuitively, a triple of the form $(\widetilde{V_i}, \sigma, c)$ is active, meaning it is computing up until a normal form term is obtained, at which point the computation proceeds to triples of the form $(\widetilde{V_i}, \sigma, h)$ where the environment is carrying on the computation, deciding to call one of the accumulated functions or to resume the computation by evaluating the context at the top of the stack.

For instance, in the triple $(\widetilde{V_i}, \sigma, \langle h \mid E[yV]\rangle)$, the environment is about to get access to V and the function corresponding to y will run before eventually (in absence of divergence) returning the result that will be used by E. Thus, the "next" triple is $((\widetilde{V_i}, V), E :: \sigma, h)$: value V and context E are added

to the corresponding tuple and stack, and h is kept identical while the environment is deciding for the next move. This evolution can be seen in Definitions 109 and 110.

Following [26], to define normal form bisimulation, we first need to introduce an auxiliary relation which performs an immediate beta reduction in a term of the form VW whenever possible.

Definition 108 (Relation $\succ$)**.** We write $VW \succ N$ when $V = \lambda y.\, N'$ and $N = N'\{^{W}/_{y}\}$ or when $V = z$ and $N = VW$.

We say that a term is deferred diverging if it hides a diverging behaviour behind a stuck term, e.g. $(\lambda x.\, \Omega)(yV)$. This notion can be extended to triples to capture all diverging behaviours, including deferred ones, as defined below. Intuitively, a triple is diverging if it contains a diverging, possibly deferred, context or configuration.

Formally, the set of diverging triples is defined coinductively using a diverging set.

Definition 109 (Diverging set)**.** A set S of triples is *diverging* if the two following conditions hold:

1. $(\widetilde{V_i}, \sigma, c) \in S$ implies
   (a) if $c \to c'$, then $(\widetilde{V_i}, \sigma, c') \in S$
   (b) if $c = \langle h \mid V \rangle$, then $\sigma \neq \emptyset$ and $((\widetilde{V_i}, V), \sigma, h) \in S$
   (c) if $c = \langle h \mid E[yV] \rangle$, then $((\widetilde{V_i}, V), E :: \sigma, h) \in S$
2. $(\widetilde{V_i}, \sigma, h) \in S$ implies
   (a) for every j and fresh x, $(\widetilde{V_i}, \sigma, \langle h \mid M \rangle) \in S$ with $V_j x \succ M$
   (b) if $\sigma = E :: \sigma'$, then $(\widetilde{V_i}, \sigma', \langle h \mid E[x] \rangle) \in S$ for x fresh

We note λ_{div} the largest diverging set.

We say that a relation $\mathcal{R}$ is a *triples relation* if the two elements of any pair of triples in $\mathcal{R}$ both contain a tuple of values and stack of the same size, and either both contain a configuration or both contain a store.

We can now define the normal form bisimulation:

Definition 110 (nfb)**.** A triples relation $\mathcal{R}$ is a normal form bisimulation when there exists a diverging set S such that:

1. $(\widetilde{V_i}, \sigma_1, c)\ \mathcal{R}\ (\widetilde{W_i}, \sigma_2, d)$ implies
   (a) if $c \to c'$, then $d \Rightarrow d'$ and $(\widetilde{V_i}, \sigma_1, c')\ \mathcal{R}\ (\widetilde{W_i}, \sigma_2, d')$
   (b) if $c = \langle h \mid V \rangle$, then either
      i. $d \Rightarrow \langle g \mid W \rangle$ and $((\widetilde{V_i}, V), \sigma_1, h)\ \mathcal{R}\ ((\widetilde{W_i}, W), \sigma_2, g)$, or
      ii. $\sigma_1 \neq \emptyset$ and $((\widetilde{V_i}, V), \sigma_1, h) \in S$
   (c) if $c = \langle h \mid E[yV] \rangle$, then either
      i. $d \Rightarrow \langle g \mid F[yW] \rangle$ and $((\widetilde{V_i}, V), E :: \sigma_1, h)\ \mathcal{R}\ ((\widetilde{W_i}, W), F :: \sigma_2, g)$, or
      ii. $((\widetilde{V_i}, V), E :: \sigma_1, h) \in S$
   (d) and symmetrically for d
2. $(\widetilde{V_i}, \sigma_1, h)\ \mathcal{R}\ (\widetilde{W_i}, \sigma_2, g)$ implies
   (a) for every j and fresh x, $(\widetilde{V_i}, \sigma_1, \langle h \mid M \rangle)\ \mathcal{R}\ (\widetilde{W_i}, \sigma_2, \langle g \mid N \rangle)$ with $V_j x \succ M$ and $W_j x \succ N$
   (b) if $\sigma_1 = E :: \sigma_1'$ and $\sigma_2 = F :: \sigma_2'$, then $(\widetilde{V_i}, \sigma_1', \langle h \mid E[x] \rangle)\ \mathcal{R}\ (\widetilde{W_i}, \sigma_2', \langle g \mid F[x] \rangle)$ for x fresh

Normal form bisimilarity, $\approx_\lambda$, is the largest normal form bisimulation.

In [6], the notion of bisimulation, which we will call *set-nf bisimulation*, is similar to the relation we introduce in Definition 110. The main differences are the following:

- The environment is composed of a set instead of a tuple. This change has consequences regarding redundancy.
- In the Clause 2.(a) of Definition 110, we use the same index for both environments. This is not possible when using plain sets. Thus, in [6] both environments are stored in a single set containing pairs of values to make the pairing explicit. In the end, bisimulation states are not pairs but quadruples $(\mathcal{E}, \Sigma, c, d)$ or $(\mathcal{E}, \Sigma, h, g)$, with $\mathcal{E}$ being a set of pairs of values and Σ being a stack of pairs of contexts.
- set-nf bisimulation corresponds to both Definitions 109 and 110
- In the clause corresponding to Clause 2.(a) (both for Definitions 109 and 110), the use of $\succ$ is removed.

These distinctions are made to highlight the operational correspondence of the encoding (Theorem 113) and the bisimulation with divergence (Definition 118). We show however that the bisimilarity resulting from both definitions are identical. Formally, we write $\approx^s_\lambda$ set-nf bisimilarity, and `set` (resp. `tup`) the trivial conversion from tuples to sets (resp. from sets to tuple), `zip` the conversion from pairs of tuples (resp. pairs of stacks) to tuples of pairs (resp. stacks of pairs).

Lemma 111. If $(\widetilde{V_i}, \sigma_1, c) \approx_\lambda (\widetilde{W_i}, \sigma_2, d)$ then $(\mathtt{set}(\mathtt{zip}(\widetilde{V_i}, \widetilde{W_i})), \mathtt{zip}(\sigma_1, \sigma_2), c, d) \in \approx^s_\lambda$.
Conversely, if $(\mathcal{E}, \Sigma, c, d) \in \approx^s_\lambda$, then $(\pi_1(\mathtt{tup}(\mathcal{E})), \pi_1(\Sigma), c) \approx_\lambda (\pi_2(\mathtt{tup}(\mathcal{E})), \pi_2(\Sigma), d)$.

The proof uses the notion of set-nf bisimulation up to `red` which is an up-to technique defined in [6].

Proof. If $\mathcal{R}$ is a normal form bisimulation, then after conversion $\mathcal{R}$ is a set-nf bisimulation up to `red`.

If $\mathcal{R}$ is a set-nf bisimulation, then after conversion $\mathcal{R}$ is included in a normal form bisimulation (namely, the closure of $\mathcal{R}$ by duplicate in the environment). □

We can thus rely on the following result.

Theorem 112 (Full abstraction [6, Theorems 3,4])**.** For all $\lambda^{\mathtt{ref}}$-terms M, N, we have the following: $(\emptyset, \odot, \langle \emptyset \mid M \rangle) \approx_\lambda (\emptyset, \odot, \langle \emptyset \mid N \rangle)$ iff $M \asymp N$.

This means that to prove that two reference-closed terms are contextually equivalent, we can provide a normal form bisimulation $\mathcal{R}$ relating the two terms, and its corresponding diverging set S.

6.2.2 A π-calculus characterisation of contextual equivalence in $\lambda^{\mathtt{ref}}$

We can now move on to show our full abstraction result. To do so, we first extend the encoding to the triples defined in Section 6.2.1. This leads to an operational correspondence similar to Proposition 105 but for triples (Theorem 113). Thanks to triples, it is possible to state Theorem 113 without using explicit Iπ constructs, so that each transition relates the encoding of triples. The bisimulation with divergence can then be defined and shown fully abstract using mainly the operational correspondence theorem.

We describe in Figure 6.3 the encoding of triples (Section 6.2.1). It builds on the encoding in Figure 6.1, with values being encoded as expected. To encode σ, every evaluation context E in σ is encoded as the process $p(z).[\![E[z]]\!]_q$ so that, intuitively, $E[z]$ can be executed as soon as the input at p is triggered. Both $[\![(\widetilde{V_i}, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}$ and $[\![(\widetilde{V_i}, \sigma, h)]\!]_{\widetilde{x};\widetilde{pq}}$ are typable with stacks that we write $\rho_{q_0,\widetilde{pq}}$ and $\rho_{\widetilde{pq}}$ respectively.

Theorem 113 (Operational Correspondence)**.**
We relate transitions for the encoding of both kind of triples:
When $[\rho; [\![(\widetilde{V_i}, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \xrightarrow{\mu} [\rho'; P]$ with $\widetilde{x} = \widetilde{x_i}$ and $\widetilde{pq} = p_1, q_1, \ldots, p_n, q_n$ then:

1. either $c \to c'$, $\mu = \tau$ and $P \gtrsim [\![(\widetilde{V_i}, \sigma, c')]\!]_{\widetilde{x};q_0,\widetilde{pq}}$
2. or $c = \langle h \mid V_0 \rangle$, $\mu = \overline{q_0}(x_0)$ and $P \gtrsim [\![((\widetilde{V_i}, V_0), \sigma, h)]\!]_{x_0,\widetilde{x};\widetilde{pq}}$
3. or $c = \langle h \mid E_0[y\ V_0] \rangle$, $\mu = \overline{y}(x_0, p_0)$ and $P \gtrsim [\![((\widetilde{V_i}, V_0), E_0 :: \sigma, h)]\!]_{x_0,\widetilde{x};p_0,q_0,\widetilde{pq}}$

$$[\![\widetilde{V_i}]\!]_{\widetilde{x}} \stackrel{\text{def}}{=} \prod_i [\![V_i]\!]^{\mathrm{v}}_{x_i} \qquad \text{with } \widetilde{x} = \widetilde{x_i}$$

$$[\![E_1 :: \ldots :: E_n]\!]_{\widetilde{pq}} \stackrel{\text{def}}{=} \prod_{i \le n} p_i(z).\, [\![E_i[z]]\!]_{q_i} \qquad \text{with } \widetilde{pq} = p_1, q_1, \ldots, p_n, q_n$$

$$[\![(\widetilde{V_i}, \sigma, h)]\!]_{\widetilde{x};\widetilde{pq}} \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{\ell})([\![\widetilde{V_i}]\!]_{\widetilde{x}} \mid [\![\sigma]\!]_{\widetilde{pq}} \mid [\![h]\!]) \qquad \text{with } \widetilde{\ell} = \mathrm{dom}(h)$$

$$[\![(\widetilde{V_i}, \sigma, \langle h \mid M\rangle)]\!]_{\widetilde{x};q_0,\widetilde{pq}} \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{\ell})([\![\widetilde{V_i}]\!]_{\widetilde{x}} \mid [\![\sigma]\!]_{\widetilde{pq}} \mid [\![h]\!] \mid [\![M]\!]_{q_0}) \qquad \text{with } \widetilde{\ell} = \mathrm{dom}(h)$$

Figure 6.3: Encoding for triples

When $[\rho; [\![(\widetilde{V_i}, \sigma, h)]\!]_{\widetilde{x};\widetilde{pq}}] \xrightarrow{\mu} [\rho'; P]$ with $\widetilde{x} = \widetilde{x_i}$ and $\widetilde{pq} = p_1, q_1, \ldots, p_n, q_n$ then:

1. either $\mu = x_j(z, q_0)$ and $P \gtrsim [\![(\widetilde{V_i}, \sigma, \langle h \mid N\rangle)]\!]_{\widetilde{x};q_0,\widetilde{pq}}$ with $V_j z \succ N$.
2. or $\sigma = E_1 :: \sigma'$ and $\mu = p_1(z)$ and $P \gtrsim [\![(\widetilde{V_i}, \sigma', \langle h \mid E_1[z]\rangle)]\!]_{\widetilde{x};q_1,p_2,q_2,\ldots,p_n,q_n}$

The following result is useful for the full abstraction proof.

Corollary 114. If $[\![(\widetilde{V_i}, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}} \Rightarrow P'$, then there exists a configuration c' with $c \Rightarrow c'$ and $P' \gtrsim [\![(\widetilde{V_i}, \sigma, c')]\!]_{\widetilde{x};q_0,\widetilde{pq}}$.

Note that we rely on untyped expansion, which does not make any assumption about sequentiality of processes, in Theorem 113 and Corollary 114. This shows the robustness of the encoding.

The following example shows that by contrast with the encoding of configurations, the τ transition in the first case of Theorem 113 is not deterministic.

Example 115. $[\![(\{\lambda z.\, \ell \coloneqq z\}, \odot, \langle \ell = y \mid !\ell\rangle)]\!]_{x_1;q_0} \not\gtrsim [\![(\{\lambda z.\, \ell \coloneqq z\}, \odot, \langle \ell = y \mid y\rangle)]\!]_{x_1;q_0}$.

Recall that $\gtrsim$ stands for the untyped expansion relation. As the π-calculus is concurrent, the encoding of $\lambda z.\, \ell \coloneqq z$ may be executed to change the content of ℓ before the read is executed. However, we can recover this result by using $\gtrsim_{wb}$ which forbids the concurrent transitions. This makes $\gtrsim_{wb}$ useful to handle reductions for triples.

We now introduce the notion of divergence for π-terms. This leads to the definition of bisimulation with divergence which coarsens $\approx_{\mathrm{wb}}$ to account for divergent terms. The induced equivalence for $\lambda^{\mathtt{ref}}$-terms corresponds to nfb.

A *wb-typed set* is a set of pairs (ρ, P) with ρ clean and $\rho \vDash_{\mathrm{wb}} P$.

Definition 116 (π_{div})**.** A wb-typed set S is π-divergent if whenever we have $(\rho, P) \in S$, then $\rho \neq \emptyset$ and for all μ, ρ', P' with $[\rho; P] \xrightarrow{\mu} [\rho'; P']$, we have $(\rho', P') \in S$.

We write π_{div} for the largest π-divergent set.

Intuitively, the computation ends when the stack gets empty, meaning there is no pending continuation. Processes in π_{div} thus correspond to processes which cannot terminate, hence the name of π-divergence.

The following example shows that the divergence of a process depends on the stack used to type it.

Example 117. Let us consider

$$P \stackrel{\text{def}}{=} (\boldsymbol{\nu}x)(p_1(z).\, \overline{x}(y, r_1).\, r_1 \rhd q_1 \mid p_2(z).\, (\overline{q_2}(y) \mid x(y', r).\, \overline{r}(z')\ \mathbf{0})),$$

$$\rho_1 = p_1 : \mathtt{i}, q_1 : \mathtt{o}, p_2 : \mathtt{i}, q_2 : \mathtt{o}, \qquad \rho_2 = p_2 : \mathtt{i}, q_2 : \mathtt{o}, p_1 : \mathtt{i}, q_1 : \mathtt{o}.$$

We have both $\rho_1 \vDash_{\mathrm{wb}} P$ and $\rho_2 \vDash_{\mathrm{wb}} P$.

$[\rho_1; P] \xrightarrow{p_1(z)} [\rho'; P']$ is the only typed-allowed transition and P' has no typed-allowed transition. Thus, $(\rho_1, P) \in \pi_{\mathrm{div}}$.

On the other hand, $[\rho_2; P] \xrightarrow{p_2(z)} \xrightarrow{\overline{q_2}(y)} \xrightarrow{p_1(z_0)} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\overline{q_1}(z')} [\emptyset; P']$. So $(\rho_2, P) \notin \pi_{\mathrm{div}}$.

Definition 118. A wb-typed relation $\mathcal{R}$ is a *bisimulation with divergence* if there exists a π-divergent set S such that whenever we have $(\sigma, P, Q) \in \mathcal{R}$ and $[\sigma; P] \xrightarrow{\mu} [\sigma'; P']$, then one of the following holds:

1. there exists Q' with $Q \stackrel{\widehat{\mu}}{\Rightarrow} Q'$ and $(\sigma', P', Q') \in \mathcal{R}$;
2. μ is an output and $(\sigma', P') \in S$

and symmetrically for Q. We write $\approx_{\text{div}}$ for the largest bisimulation with divergence. We write $P \approx^{\rho}_{\text{div}} Q$ when $(\rho, P, Q) \in \approx_{\text{div}}$.

Bisimilarity with divergence is coarser than wb-bisimilarity and thus also coarser than the untyped bisimilarity.

To establish soundness, we rely on Theorem 113 (operational correspondence). The set of triples whose encoding is π-diverging is itself diverging, so we can prove that the relation induced by $\approx_{\text{div}}$ is a nfb.

Theorem 119 (Soundness)**.** If $[\![M]\!]_p \approx^{p:\mathtt{o}}_{\text{div}} [\![N]\!]_p$, then $M \asymp N$.

The completeness is proved by showing that the encoding of a divergent set is π-divergent up to $\gtrsim$ and then Property (6.1) from the introduction, namely that the encoding of a nfb is a bisimulation with divergence up to $\gtrsim$.

Theorem 120 (Completeness)**.** If $M \asymp N$, then $[\![M]\!]_p \approx^{p:\mathtt{o}}_{\text{div}} [\![N]\!]_p$.

6.3 Up-to techniques for $\approx_{\text{div}}$ in Iπ, and applications

Up-to techniques are usually defined as functions from relations to relations. This makes standard up-to techniques like up-to expansion or up-to context adaptable to our typed setting as in [17]. However, as π-divergence is defined coinductively, up-to techniques can also be defined to enhance the proofs of divergence. We define similar up-to techniques for sets that can be exploited for π-divergent sets. For instance, in order for up-to context to be sound, we must forbid contexts where the hole is guarded by a replicated input. Indeed, replicated input may only be typed with an empty set so they cannot be diverging. This is similar to $\lambda^{\mathtt{ref}}$, where Ω is divergent but $\lambda x.\,\Omega$ is not. Thanks to up-to techniques, to prove that two processes are equivalent, we can give $\mathcal{R}$, a bisimulation with divergence up to using S, a π-divergent set up to.

We also introduce a new up-to technique, to which we return below; this technique is *compatible*, and thus fits in the existing theory of up-to techniques for the π-calculus.

This is used in the equivalences proven in Section 6.3.2.

6.3.1 A new up-to technique for Iπ: up-to body

In Iπ, functions are encoded as a replicated process of the form $!x(y,p).\,T$, which we denote T^x. When identical calls result in the same value being sent, it creates copies of that value, leading to processes of the form $T^x \mid T^y \mid \ldots \mid T^z$, with $x, y, \ldots, z$ being all different names. This behaviour makes bisimulations infinite, as they would need to contain processes with an arbitrary number of processes in parallel, despite all of them sharing the same body. We introduce a new technique, up-to body, which allows us to remove these duplicated copies. Indeed, it is sound to keep only one copy when comparing processes: any discriminating interaction with the environment involving multiple copies can be mimicked by a similar interaction with only one copy. The up-to body technique is defined by the following rule:

$$\frac{(\rho,\ E[T_1^x],\ F[T_2^x]) \in \mathcal{R}}{(\rho,\ E[T_1^z \mid T_1^x],\ F[T_2^z \mid T_2^x]) \in \mathtt{body}(\mathcal{R})} x, z \notin \mathrm{n}(E) \cup \mathrm{n}(F) \cup \mathrm{fn}(T_1) \cup \mathrm{fn}(T_2)$$

Up-to body differs from up-to context because we keep the possibly different static contexts, E and F. These contexts correspond to private resources shared among the copies. In our case, the private resource is the local store, but this technique can be exported to the plain π-calculus. The technique for sets is defined similarly, with $(\rho, E[T^z \mid T^x]) \in \mathtt{body}(S)$ whenever $(\rho, E[T^x]) \in S$ and $x, z \notin \mathrm{n}(E) \cup \mathrm{fn}(T)$.

Using up-to body, it is possible to prove the analogue of (6.1) from Section 6.2 but using normal form bisimulations from [6] instead of the formulation we have given in Definition 110.

We now present a simple example that demonstrates the use of up-to body.

Example 121. $\texttt{new}\ \ell \coloneqq z\ \texttt{in}\ \lambda x.\,\lambda y.\,!\ell \asymp \lambda x.\,\lambda y.\,z$

Proof. We reason using the soundness of the encoding, and define

$$\begin{aligned}\mathcal{R} = \{&(\rho_q, [\![\texttt{new}\ \ell \coloneqq z\ \texttt{in}\ \lambda x.\,\lambda y.\,!\ell]\!]_q, [\![\lambda x.\,\lambda y.\,z]\!]_q),\\ &(\emptyset, [\![(\lambda x.\,\lambda y.\,!\ell,), \odot, \ell = z]\!]_{x_0}, [\![(\lambda x.\,\lambda y.\,z,), \odot, \emptyset]\!]_{x_0})\\ &(\emptyset, [\![(\lambda x.\,\lambda y.\,!\ell, \lambda y.\,!\ell), \odot, \ell = z]\!]_{x_0,x_1}, [\![(\lambda x.\,\lambda y.\,z, \lambda y.\,z), \odot, \emptyset]\!]_{x_0,x_1})\}.\end{aligned}$$

We can show that $\mathcal{R}$ is a bisimulation with divergence up to $\gtrsim_{wb}$, context and body. The up-to body technique makes it possible to stop the relation after the third pair. Without up-to body, we would need pairs like $(\emptyset, [\![(\lambda x.\,\lambda y.\,!\ell, \lambda y.\,!\ell, \lambda y.\,!\ell), \odot, \ell = z]\!]_{x_0,x_1,x_2}, [\![(\lambda x.\,\lambda y.\,z, \lambda y.\,z, \lambda y.\,z), \odot, \emptyset]\!]_{x_0,x_1,x_2})$ and so on. □

6.3.2 Examples of other equivalences in $\lambda^{\texttt{ref}}$

We now present an example that involves the encoding of triples, but does not require us to take into account deferred divergences. To validate the laws below, we thus rely on $\approx_{\text{wb}}$ which is included in $\approx_{\text{div}}$ (Definition 118).

Example 122 (Optimised access)**.** Two consecutive read and/or write operations can be transformed into an equivalent single operation.

For any pair (f_1, f_2) from the following (() is the unique inhabitant of the unit type):

$$\begin{aligned}&\big(f_1 = \lambda().\,\ell \coloneqq !\ell, && f_2 = \lambda().\,()\big)\\ &\big(f_1 = \lambda n.\,\lambda m.\,\ell \coloneqq n; \ell \coloneqq m, && f_2 = \lambda n.\,\lambda m.\,\ell \coloneqq m\big)\\ &\big(f_1 = \lambda n.\,\ell \coloneqq n; !\ell, && f_2 = \lambda n.\,\ell \coloneqq n; n\big)\\ &\big(f_1 = \lambda().\,\texttt{let}\ x = !\ell\ \texttt{in}\ \texttt{let}\ y = !\ell\ \texttt{in}\ M, && f_2 = \lambda().\,\texttt{let}\ x = !\ell\ \texttt{in}\ M\{x/y\}\big)\end{aligned}$$

we have for any E, y, V_0, we have $[\![\texttt{new}\ \ell \coloneqq V_0\ \texttt{in}\ E[y\ f_1]]\!]_{q_1} \approx_{\text{wb}} [\![\texttt{new}\ \ell \coloneqq V_0\ \texttt{in}\ E[y\ f_2]]\!]_{q_1}$.

Proof. First, we have that $[\![((f_1,), E :: \odot, \ell \coloneqq V_0)]\!]_{x;p_1,q_1} \approx_{\text{wb}} [\![((f_2,), E :: \odot, \ell \coloneqq V_0)]\!]_{x;p_1,q_1}$ using $(f_i,)$ to denote the singleton containing f_i. Indeed, we define a relation $\mathcal{R}$ as follows:

$$\mathcal{R} = \{(\rho_{\widetilde{pq}},\ [\![((\widetilde{V_i}, f_1), \sigma, h \uplus \ell = V)]\!]_{\widetilde{x};\widetilde{pq}}, [\![((\widetilde{V_i}, f_2), \sigma, h \uplus \ell = V)]\!]_{\widetilde{x};\widetilde{pq}}) \mid \text{ for all } \widetilde{V_i}, \sigma, h, V, \widetilde{x}, \widetilde{pq}\}$$

and we show that $\mathcal{R}$ is a wb-bisimulation up to $\gtrsim_{wb}$ and static context.

Then we use Theorem 113 to show

$$[\![\texttt{new}\ \ell \coloneqq V_0\ \texttt{in}\ E[x\ f_i]]\!]_{q_1} \gtrsim [\![(\emptyset, \odot, \langle \ell = V_0 \mid E[y\ f_i]\rangle)]\!]_{q_1} \xrightarrow{\overline{y}(x,p_1)} \gtrsim [\![(f_i,), E :: \odot, \ell = V_0)]\!]_{x;p_1,q_1}$$

with the output being the only transition that the intermediate process can perform. □

The proof of this law could become even simpler by adopting the type system of [16]: we could prove prove directly that $[\![f_1]\!]_p$ and $[\![f_2]\!]_p$ are equivalent.

Relation $\mathcal{R}$ above would have the same base if we were to reason in the source language. If we were to show $\texttt{new}\ \ell \coloneqq V\ \texttt{in}\ E[y\ f_1] \asymp \texttt{new}\ \ell \coloneqq V\ \texttt{in}\ E[y\ f_2]$ using nfb, we would need to add triples with the same normal form term on both sides.

We present some examples involving deferred divergence from the literature.

Example 123. $\langle \emptyset \mid x\ V\ \Omega\rangle \asymp \langle \emptyset \mid \Omega\rangle$, where V is a value and Ω is an always diverging term.

Proof. Take $\mathcal{R} = \{(\rho_q, [\![\langle \emptyset \mid x\ V\ \Omega\rangle]\!]_q, [\![\langle \emptyset \mid \Omega\rangle]\!]_q)\}$ and $S = \{(\rho_r, [\![\Omega]\!]_r)\}$.

S is π-divergent, so $\gtrsim_{wb}\texttt{ctxt}(S)$ is too, where `ctxt` stands for the up-to context technique. Then we show that $\mathcal{R}$ is a bisimulation with divergence up to context with $\gtrsim_{wb}\texttt{ctxt}(S)$ as the π-divergent set.

$[\rho_q; [\![\langle \emptyset \mid x\ V\ \Omega\rangle]\!]_q] \xrightarrow{\overline{x}(y,p)} [\rho_{p,q}; P]$ is the only type-allowed transition.

By Theorem 113, we know that $P \gtrsim [\![(\{V\}, [\cdot]\ \Omega :: \odot, \emptyset)]\!]_{y;pq}$.

As $(\rho_{p,q}, [\![(\{V\}, [\cdot]\ \Omega :: \odot, \emptyset)]\!]_{y;pq}) \in \texttt{ctxt}(S)$, we indeed have $(\rho_{p,q}, P) \in \gtrsim_{wb}\texttt{ctxt}(S)$. □

Example 124 (Example 9 from [6])**.**

$$V_1 = \lambda x.\, \texttt{if}\ !\ell\ \texttt{then}\ \Omega\ \texttt{else}\ k \coloneqq \mathbf{tt} \qquad W_1 = \lambda x.\, \Omega$$

$$V_2 = \lambda f.\, f\ V_1; \texttt{if}\ !k\ \texttt{then}\ \Omega\ \texttt{else}\ \ell \coloneqq \mathbf{tt} \qquad W_2 = \lambda f.\, f\ W_1$$

We have $\texttt{new}\ \ell \coloneqq \mathbf{ff}\ \texttt{in}\ \texttt{new}\ k \coloneqq \mathbf{ff}\ \texttt{in}\ V_2 \asymp W_2$.

Proof. Given a context E, we write E^n for $E :: E :: \ldots :: E :: \odot$ with n occurrences of E.

Let $E \stackrel{\text{def}}{=} (\lambda().\, \texttt{if}\ !k\ \texttt{then}\ \Omega\ \texttt{else}\ \ell \coloneqq \mathbf{tt})[\cdot]$, and $F \stackrel{\text{def}}{=} [\cdot]$. We define $\mathcal{R}$ as

$$\begin{array}{lll}
\{(\rho_q, & [\![\texttt{new}\ \ell \coloneqq \mathbf{ff}\ \texttt{in}\ \texttt{new}\ k \coloneqq \mathbf{ff}\ \texttt{in}\ V_2]\!]_q, & [\![W_2]\!]_q), \\
(\emptyset, & [\![(V_2,), \odot, \ell = \mathbf{ff} \uplus k = \mathbf{ff}]\!]_{x_2}, & [\![(W_2,), \odot, \emptyset]\!]_{x_2})\} \bigcup \\
\{(\rho_{\widetilde{pq}}, & [\![(V_1, V_2), E^n, \ell = \mathbf{ff} \uplus k = \mathbf{ff}]\!]_{\widetilde{x_i};\widetilde{pq}}, & [\![(W_1, W_2), F^n, \emptyset]\!]_{\widetilde{x_i};\widetilde{pq}}), \\
(\rho_{q_0,\widetilde{pq}}, & [\![(V_1, V_2), E^n, \langle \ell = \mathbf{ff} \uplus k = \mathbf{tt} \mid () \rangle]\!]_{\widetilde{x_i};q_0,\widetilde{pq}}, & [\![(W_1, W_2), F^n, \langle \emptyset \mid \Omega \rangle]\!]_{\widetilde{x_i};q_0,\widetilde{pq}}), \\
(\rho_{\widetilde{pq}}, & [\![(V_1, V_2), E^n, \ell = \mathbf{tt} \uplus k = \mathbf{ff}]\!]_{\widetilde{x_i};\widetilde{pq}}, & [\![(W_1, W_2), F^n, \emptyset]\!]_{\widetilde{x_i};\widetilde{pq}}) \\
 & & \mid \text{ for all } \widetilde{pq}, n \text{ with } |\widetilde{pq}| = 2n\ \}
\end{array}$$

and S as $\{(\rho_q, [\![\Omega]\!]_q),\ (\rho_{\widetilde{pq}}, [\![(V_1, V_2), E^n, \ell = \mathbf{ff} \uplus k = \mathbf{tt}]\!]_{\widetilde{x_i};\widetilde{pq}})$ for all $\widetilde{pq}, n$ with $|\widetilde{pq}| = 2n\}$.

$\mathcal{R}$ is a bisimulation up to $\gtrsim_{wb}$, context and body. Multiple calls to V_1, W_1 create multiples continuations, but thanks to up-to body, multiples calls to V_2, W_2 do not create multiples copies of V_1, W_1. All in all, we have the same number of pairs in the candidate bisimulation relation as in the relation in [6]. □

By modifying this example so as to allocate the references inside V_2, we get Example 15 from [26]. In that case, V_2 does not have free reference names. This makes it possible to use up-to parallel composition between the encoding of V_2 and V_1 preventing multiples calls to V_2. This usage of up-to parallel composition is similar to the up-to separation technique introduced in [26].

Chapter 7

Conclusion

This work shows various type systems in the π-calculus: reference names, sequentiality and well-bracketing. For each system, we define a corresponding bisimulation that is sound and complete (up to some technicalities) with respect to barbed equivalence in Chapters 3,4 and 5 and with respect to contextual equivalence in $\lambda^{\mathtt{ref}}$ in Chapter 6. This enables reasoning for proving equivalences on the corresponding typed calculi. We also present how these equivalences can be applied, by combining them in Section 4.3 or by obtaining a full abstraction result in Chapter 6. These systems follow rather distinct disciplines, which are expressed in different subcalculi of π: reference names are asynchronous, while continuation names are best explained with internal mobility.

The higher-order language with references we use in Chapter 6, $\lambda^{\mathtt{ref}}$, uses local references in the sense that they are created locally but never returned as values. Extending this language to general references, i.e. references that can be passed along channels, raises several issues. First, there does not seem to be a good equivalent of the forwarder for reference names. This means that these names may have to be sent directly, free, and not abstracted away behind a bound name. Intuitively, this happens as in a sequential setting, testing the extensional equality between references (i.e. they store the "same" value) is sufficient to determine if these references are equal. Second, as actions related to reference names can occur (and not only internal communications), the bisimulation must be modified by including the type system for references of Chapter 3.

Currently, the type system with references is not well connected with the other systems. When the bisimulations are expressed mainly using type-allowed transitions, combining them while remaining sound, as done in Section 4.3, does not seem to raise additional difficulties. However, due to the difference in the barbed equivalence used, it is not clear how a bisimulation can be proved complete in the settings of Section 4.3. The safest bet would be using standard barbed equivalence. Understanding the effect of reference names and more generally output receptiveness may help recovering the completeness of sequential bisimilarity in presence of input-controlled names. Indeed, both examples for input-controlled names given in Section 4.1 seem to have an output which is receptive to some extent: in Milner's encoding, the names u, v are used linearly and the output is immediately available; and although there are multiple outputs at u in the encoding from the polyadic to the monadic π-calculus, there is always exactly one output available from the moment the name is extruded to the context until all outputs are consumed. While Lemma 60 does not hold in general with input-controlled names, it may be possible to prove it by adding some extra conditions like output receptiveness.

Additionally, having a full abstraction with respect to contextual equivalence in $\lambda^{\mathtt{ref}}$ shows that our typing constraints considerably reduce the expressiveness of the calculus. The typed processes have similar behaviours as $\lambda^{\mathtt{ref}}$ programs, although not quite, as π can still express non-determinism. Nevertheless, it seems reasonable to expect that with a more refined type system, typable processes may only be equivalent to the ones in the image of the encoding, similar to the result obtained for PCF [5].

Bibliography

[1] S. Abramsky. The lazy λ-calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1987.

[2] Samson Abramsky and Guy Mccusker. Game semantics. *Computational Logic*, 01 2000.

[3] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 340–353. ACM, 2009.

[4] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous pi-calculus. *Theor. Comput. Sci.*, 195(2):291–324, 1998.

[5] Martin Berger, Kohei Honda, and Nobuko Yoshida. Sequentiality and the pi-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2001.

[6] Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. A complete normal-form bisimilarity for state. In Mikolaj Bojanczyk and Alex Simpson, editors, *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11425 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2019.

[7] Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. Diacritical companions. In Barbara König, editor, *Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2019, London, UK, June 4-7, 2019*, volume 347 of *Electronic Notes in Theoretical Computer Science*, pages 25–43. Elsevier, 2019.

[8] S. Brookes. The essence of parallel algol. In *Proc. LICS'96*. IEEE, 1996.

[9] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.*, 22(4-5):477–528, 2012.

[10] Adrien Durier, Daniel Hirschkoff, and Davide Sangiorgi. Eager Functions as Processes (long version). working paper or preprint, December 2021.

[11] M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully abstract model for the π-calculus. *Inf. Comput.*, 179(1):76–117, 2002.

[12] Cédric Fournet and Georges Gonthier. A hierarchy of equivalences for asynchronous calculi. *J. Log. Algebraic Methods Program.*, 63(1):131–173, 2005.

[13] Carl A. Gunter. *Semantics of programming languages - structures and techniques*. Foundations of computing. MIT Press, 1993.

[14] Matthew Hennessy and Julian Rathke. Typed behavioural equivalences for processes in the presence of subtyping. *Math. Struct. Comput. Sci.*, 14(5):651–684, 2004.

[15] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 276–290, 1988.

[16] Daniel Hirschkoff, Enguerrand Prebet, and Davide Sangiorgi. On the representation of references in the pi-calculus. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020*, volume 171 of *LIPIcs*, pages 34:1–34:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[17] Daniel Hirschkoff, Enguerrand Prebet, and Davide Sangiorgi. On sequentiality and well-bracketing in the π-calculus. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–13. IEEE, 2021.

[18] K. Honda and N. Yoshida. On reduction-based process semantics. *TCS*, 152(2):437–486, 1995.

[19] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 193–206. ACM, 2013.

[20] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: i, ii, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[21] Guilhem Jaber. Operational nominal game semantics. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2015.

[22] Guilhem Jaber. Syteci: automating contextual equivalence for higher-order programs with references. *Proc. ACM Program. Lang.*, 4(POPL):59:1–59:28, 2020.

[23] Guilhem Jaber and Davide Sangiorgi. Games, mobile processes, and functions. In *30th EACSL Annual Conference on Computer Science Logic (CSL 2022).*, Göttingen, Germany, February 2022.

[24] N. Kobayashi. A partially deadlock-free typed process calculus. *Transactions on Programming Languages and Systems*, 20(2):436–482, 1998. A preliminary version in *12th Lics Conf.* IEEE Computer Society Press 128–139, 1997.

[25] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.

[26] Vasileios Koutavas, Yu-Yang Lin, and Nikos Tzevelekos. From bounded checking to verification of equivalence via symbolic up-to techniques. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, volume 13244 of *Lecture Notes in Computer Science*, pages 178–195. Springer, 2022.

[27] James Laird. A fully abstract trace semantics for general references. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007.

[28] Søren B. Lassen. Eager normal form bisimulation. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pages 345–354. IEEE Computer Society, 2005.

[29] Titouan Leclercq. Equivalence de programmes ccs typés. Internship Report, 2021.

[30] Giuseppe Longo. Set-theoretical models of lambda-calculus: theories, expansions, isomorphisms. *Annals of Pure and Applied Logic*, 24(2):153 – 188, 1983.

[31] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *J. Funct. Program.*, 1(3):287–327, 1991.

[32] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

[33] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[34] Robin Milner. Functions as processes. In Mike Paterson, editor, *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, July 16-20, 1990, Proceedings*, volume 443 of *Lecture Notes in Computer Science*, pages 167–180. Springer, 1990.

[35] Robin Milner. Functions as processes. *Math. Struct. Comput. Sci.*, 2(2):119–141, 1992.

[36] J.H. Morris and Project MAC (Massachusetts Institute of Technology). *Lambda-calculus Models of Programming Languages*. MAC-TR. Massachusetts Institute of Technology, Project MAC; available from the Clearinghouse for Federal Scientific and Technical Information, Springfield, Va., 1968.

[37] Andrzej S. Murawski and Nikos Tzevelekos. Game semantics for good general references. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, pages 75–84. IEEE Computer Society, 2011.

[38] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

[39] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.

[40] G. D. Plotkin. Lambda Definability and Logical Relations. Memorandum SAI-RM-4, University of Edinburgh, 1973.

[41] D. Pous and D. Sangiorgi. *Advanced Topics in Bisimulation and Coinduction (D. Sangiorgi and J. Rutten editors)*, chapter Enhancements of the coinductive proof method. Cambridge University Press, 2011.

[42] Damien Pous. Coinduction all the way up. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 307–316. ACM, 2016.

[43] Enguerrand Prebet. On Up-to Context Techniques in the π-calculus. working paper or preprint, December 2021.

[44] C. Röckl and D. Sangiorgi. A pi-calculus process semantics of concurrent idealised ALGOL. In *Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99*, volume 1578 of *Lecture Notes in Computer Science*, pages 306–321. Springer, 1999.

[45] D. Sangiorgi. The name discipline of uniform receptiveness. *Theor. Comput. Sci.*, 221(1-2):457–493, 1999.

[46] Davide Sangiorgi. Locality and interleaving semantics in calculi for mobile processes. *Theor. Comput. Sci.*, 155(1):39–83, 1996.

[47] Davide Sangiorgi. pi-calculus, internal mobility, and agent-passing calculi. *Theor. Comput. Sci.*, 167(1&2):235–274, 1996.

[48] Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, 1998.

[49] Davide Sangiorgi. Lazy functions and mobile processes. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 691–720. The MIT Press, 2000.

[50] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), Proceedings*, pages 293–302. IEEE Computer Society, 2007.

[51] Davide Sangiorgi and Robin Milner. The problem of "weak bisimulation up to". In Rance Cleaveland, editor, *CONCUR '92, Third International Conference on Concurrency Theory, Stony Brook, NY, USA, August 24-27, 1992, Proceedings*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 1992.

[52] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[53] I. Stark. A fully abstract domain model for the pi-calculus. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 36–42. IEEE Computer Society, 1996.

[54] Bernardo Toninho and Nobuko Yoshida. On polymorphic sessions and functions: A tale of two (fully abstract) encodings. *ACM Trans. Program. Lang. Syst.*, 43(2):7:1–7:55, 2021.

[55] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 71–84. ACM, 2020.

Appendix A

Proofs for Chapter 3

A.1 Definitions and results about Aπ with references

A.1.1 Type system for output receptiveness: proof of subject reduction

We prove subject reduction, which we first recall:

Proposition 28 (Subject reduction)**.** If $\Delta \vdash_{\mathrm{r}} P$ and $[\Delta; P] \xrightarrow{\mu} [\Delta'; P']$, then $\Delta' \vdash_{\mathrm{r}} P'$.

Proof. We reason depending on the action μ:

- For $\mu = n\langle m\rangle$, we have $P \equiv (\boldsymbol{\nu}\widetilde{a}, \widetilde{\ell})(n(x).P_1 \mid P_2)$ and $P' \equiv (\boldsymbol{\nu}\widetilde{a}, \widetilde{\ell})(P_1\{m/x\} \mid P_2)$ for some $\widetilde{a}, \widetilde{\ell}, P_1, P_2$ with $m \notin \widetilde{a} \cup \widetilde{\ell}$.

  We take $\Delta_1 \vdash P_1$ and $\Delta_2 \vdash P_2$. This means $\Delta_2 = \Delta \uplus \widetilde{\ell}$. Depending on whether n is a reference name or not, we have that $\Delta_1 = n$ or $\Delta_1 = \emptyset$ respectively. In both cases, $\Delta_1 \vdash P_1\{m/x\}$ and $\Delta_1 \uplus \Delta_2 \vdash P_1\{m/x\} \mid P_2$. Thus $\Delta' = \Delta_1 \uplus \Delta$, meaning that $\Delta' = \Delta, n$ if n is a reference name and $\Delta' = \Delta$ otherwise.

- For $\mu = \overline{n}\langle m\rangle$, we have $P \equiv (\boldsymbol{\nu}\widetilde{a}, \widetilde{\ell})(\overline{n}\langle m\rangle \mid P_2)$ and $P' \equiv (\boldsymbol{\nu}\widetilde{a}, \widetilde{\ell})P_2$ for some $\widetilde{a}, \widetilde{\ell}, P_1, P_2$ with $n, m \notin \widetilde{a} \cup \widetilde{\ell}$.

  We take $\Delta_1 \vdash \overline{n}\langle m\rangle$ and $\Delta_2 \vdash P_2$. This means $\Delta_1 \uplus \Delta_2 = \Delta \uplus \widetilde{\ell}$, and $\Delta_2 = \Delta' \uplus \widetilde{\ell}$. As $n \notin \widetilde{\ell}$, $\Delta' = \Delta \setminus \Delta_1$. Thus $\Delta' = \Delta - \ell$ if n is a reference name and $\Delta' = \Delta$ otherwise.

- For $\mu = (\boldsymbol{\nu}m)\overline{n}\langle m\rangle$, we have $P \equiv (\boldsymbol{\nu}\widetilde{a}, \widetilde{\ell}, m)(\overline{n}\langle m\rangle \mid P_2)$ and $P' \equiv (\boldsymbol{\nu}\widetilde{a}, \widetilde{\ell})P_2$ for some $\widetilde{a}, \widetilde{\ell}, P_1, P_2$ with $n \notin \widetilde{a} \cup \widetilde{\ell} \cup \{m\}$. With the same notation, we have that $\Delta_2 = \Delta' \uplus \widetilde{\ell}$, and if m is a plain name then $\Delta_1 \uplus \Delta_2 = \Delta \uplus \widetilde{\ell}$ and $\Delta_1 \uplus \Delta_2 = \Delta \uplus \widetilde{\ell}, m$ otherwise. Thus we have four cases for Δ' shown in the table below:

$n \backslash m$	plain	reference
plain	Δ	Δ, m
reference	$\Delta \setminus n$	$\Delta, m \setminus n$

- For $\mu = \tau$, we look at the interaction that has occurred. This can be mimicked using two transitions, one for the output and one for the input for which we have already proven the resulting typing.
  - $P \xrightarrow{\overline{a}\langle m\rangle} \xrightarrow{a\langle m\rangle} P'$, it is straightforward.
  - $P \xrightarrow{(\boldsymbol{\nu}b)\overline{a}\langle b\rangle} \xrightarrow{a\langle b\rangle} P''$ with $P' = (\boldsymbol{\nu}b)P''$. We have $\Delta \vdash P''$ then $\Delta \vdash (\boldsymbol{\nu}b)P''$.
  - $P \xrightarrow{(\boldsymbol{\nu}\ell)\overline{a}\langle \ell\rangle} \xrightarrow{a\langle \ell\rangle} P''$ with $P' = (\boldsymbol{\nu}\ell)P''$. We have $\Delta, \ell \vdash P''$ then $\Delta \vdash (\boldsymbol{\nu}\ell)P''$.
  - $P \xrightarrow{\overline{\ell}\langle m\rangle} \xrightarrow{\ell\langle m\rangle} P'$. We have $\ell \notin \Delta$ after the output, so we can subject reduction for the input transition.
  - $P \xrightarrow{(\boldsymbol{\nu}b)\overline{\ell}\langle b\rangle} \xrightarrow{\ell\langle b\rangle} P''$ with $P' = (\boldsymbol{\nu}b)P''$. We have $\Delta \vdash P''$ then $\Delta \vdash (\boldsymbol{\nu}b)P''$

– $P \xrightarrow{(\boldsymbol{\nu}\ell')\overline{\ell}\langle\ell'\rangle} \xrightarrow{\ell\langle\ell'\rangle} P''$ with $P' = (\boldsymbol{\nu}\ell')P''$. We have $\Delta, \ell' \vdash P''$ then $\Delta \vdash (\boldsymbol{\nu}\ell')P''$

□

A.1.2 Proofs about $\approx_{\text{ip}}$

We show soundness of $\approx_{\text{ip}}$-bisimulation up to store with respect to $\approx_{\text{ip}}$-bisimilarity, and of $\approx_{\text{ip}}$-bisimilarity with respect to reference bisimilarity.

Proof of Proposition 41. We show that

$$\mathcal{R}' \stackrel{\text{def}}{=} \{P \mid M_s, Q \mid M_s \;\big|\; P \,\mathcal{R}\, Q \text{ for any } M_s\}$$

is an $\approx_{\text{ip}}$-bisimulation.

If $P \mid M_s \,\mathcal{R}\, Q \mid M_s$ and $P \mid M_s \xrightarrow{\mu} \tilde{P}$, we distinguish the sub-processes of $\tilde{P}$ that have changed:

1. If $P \mid M_s \xrightarrow{\mu} P' \mid M_s$, then $P \xrightarrow{\mu} P'$, and $\mathtt{ok}'(\Delta \cup \Delta', \mathcal{R}, P', Q, \mu)$. We show by induction on the proof of $\mathtt{ok}'(\Delta \cup \Delta', \mathcal{R}, P', Q, \mu)$ that $\mathtt{ok}((\Delta \uplus \Delta_s) \cup (\Delta' \uplus \Delta_s), \mathcal{R}', P' \mid M_s, Q \mid M_s, \mu)$. First note that $(\Delta \uplus \Delta_s) \cup (\Delta' \uplus \Delta_s) = (\Delta \cup \Delta') \uplus \Delta_s$. In short, we prove that $\mathtt{ok}'(\Delta, \mathcal{R}, P', Q, \mu)$ implies $\mathtt{ok}(\Delta \uplus \Delta_s, \mathcal{R}', P' \mid M_s, Q \mid M_s, \mu)$.

   - (Base-Up) $P' = P'' \mid M_t$, $Q \stackrel{\mu}{\Rightarrow} Q'' \mid M_t$ (or $Q \mid \overline{n}\langle m\rangle \Rightarrow Q'' \mid M_t$ for $\mu = n\langle m\rangle$) and $P'' \,\mathcal{R}\, Q''$. Then $P'' \mid M_t \mid M_s \mathcal{R}' Q'' \mid M_t \mid M_s$ and $Q \mid M_s \stackrel{\mu}{\Rightarrow} Q'' \mid M_t \mid M_s$, so we can conclude with rule Base.
   - (Ext) We use an induction on the size of s.
     - If s is empty, then $\ell \notin \Delta \uplus \Delta_s$, and we can apply rule Ext.
     - If $\ell \notin \Delta \uplus \Delta_s$, we can apply rule Ext as before. Otherwise, $M_s = \overline{\ell}\langle m\rangle \mid M_{s'}$ for some m, s'. Moreover, we know that $\mathtt{ok}'((\Delta, \ell), \mathcal{R}, P' \mid \overline{\ell}\langle m\rangle, Q \mid \overline{\ell}\langle m\rangle, \mu)$. Thus, by induction, $\mathtt{ok}((\Delta, \ell \uplus \Delta_{s'}), \mathcal{R}', P' \mid \overline{\ell}\langle m\rangle \mid M_{s'}, Q \mid \overline{\ell}\langle m\rangle \mid M_{s'}, \mu)$.

2. If $P \mid M_s \xrightarrow{\tau} P' \mid M_{s'}$, then there exists an input action $\mu' = \ell\langle m\rangle$ such that $P \xrightarrow{\mu'} P'$, and $\mathtt{ok}'(\Delta \cup \Delta', \mathcal{R}, P', Q, \mu)$. We show by induction on the proof of $\mathtt{ok}'(\Delta \cup \Delta', \mathcal{R}, P', Q, \mu)$, that $\mathtt{ok}((\Delta \uplus \Delta_s) \cup (\Delta' \uplus \Delta_{s'}), \mathcal{R}', P' \mid M_{s'}, Q \mid M_s, \mu)$. First note that $M_s \equiv \overline{\ell}\langle m\rangle \mid M_{s'}$ and $\Delta \uplus \Delta_s = \Delta' \uplus \Delta_{s'} = (\Delta \cup \Delta') \uplus \Delta_{s'}$. In short, we prove that $\mathtt{ok}'(\Delta, \mathcal{R}, P', Q, \mu)$ implies $\mathtt{ok}(\Delta \uplus \Delta_{s'}, \mathcal{R}', P' \mid M_{s'}, Q \mid M_s, \mu)$.

   - (Base-Up) $P' = P'' \mid M_t$ and $Q \mid \overline{\ell}\langle m\rangle \Rightarrow Q'' \mid M_t$, and $P'' \,\mathcal{R}\, Q''$.
     Then $P'' \mid M_t \mid M_{s'} \,\mathcal{R}'\, Q'' \mid M_t \mid M_{s'}$ and $Q \mid M_s \equiv Q \mid \overline{\ell}\langle m\rangle \mid M_{s'} \stackrel{\tau}{\Rightarrow} Q'' \mid M_t \mid M_{s'}$, so we can conclude with rule Base.
   - (Ext) We use ℓ' for the name used in that rule here. We use an induction on the size of s'.
     - If s' is empty, then $\ell \notin \Delta \uplus \Delta_{s'}$, and we can apply rule Ext.
     - If $\ell \notin \Delta \uplus \Delta_{s'}$, we can apply rule Ext as before. Otherwise, $M_{s'} = \overline{\ell'}\langle m'\rangle \mid M_{t'}$ for some m, t'. Moreover, we know that $\mathtt{ok}'((\Delta, \ell'), \mathcal{R}, P' \mid \overline{\ell'}\langle m'\rangle, Q \mid \overline{\ell'}\langle m'\rangle, \mu)$. Thus, by induction, $\mathtt{ok}((\Delta, \ell' \uplus \Delta_{t'}), \mathcal{R}, P' \mid \overline{\ell'}\langle m'\rangle \mid M_{t'}, Q \mid \overline{\ell'}\langle m'\rangle \mid M_{t'}, \mu)$.

3. If $P \mid M_s \xrightarrow{\mu} P \mid M_{s'}$, then μ is an output and $Q \mid M_s \xrightarrow{\mu} Q \mid M_{s'}$ so we can apply rule Base.

□

Corollary 125. As $\approx_{\text{r}}$ is an $\approx_{\text{r}}$-bisimulation up to store, it is closed by parallel composition of M_s.

Lemma 126. For any $\Delta \vdash P, Q$ and $\ell \notin \text{fn}_{\text{r}}(P) \cup \text{fn}_{\text{r}}(Q)$, and for all m, $P \mid \overline{\ell}\langle m\rangle \approx_{\text{ip}} Q \mid \overline{\ell}\langle m\rangle$ implies $P \approx_{\text{ip}} Q$.

This is true in particular for complete processes P, Q and any $\ell \notin \Delta$.

Proof. First notice that $P \mid \overline{\ell}\langle m\rangle \approx_{\text{ip}} Q \mid \overline{\ell}\langle m\rangle$ iff $P \mid \overline{\ell'}\langle m\rangle \approx_{\text{ip}} Q \mid \overline{\ell'}\langle m\rangle$ for any ℓ' fresh.

We show that $\{(P,Q)$ s.t. $P \mid \overline{\ell}\langle m\rangle \approx_{\text{ip}} Q \mid \overline{\ell}\langle m\rangle$ for any fresh ℓ and any $m\}$ is an $\approx_{\text{ip}}$-bisimulation.

When $P \xrightarrow{\mu} P'$, we distinguish if ℓ appears in μ:

- If $\ell \notin \mu$, then $P \mid \overline{\ell}\langle m\rangle \xrightarrow{\mu} P' \mid \overline{\ell}\langle m\rangle$ and $\mathtt{ok}((\Delta \cup \Delta', \ell), \approx_{\text{ip}}, P' \mid \overline{\ell}\langle m\rangle, Q \mid \overline{\ell}\langle m\rangle, \mu)$. We reason by induction on this predicate.
  - (Base) Then $Q \mid \overline{\ell}\langle m\rangle \overset{\mu}{\Rightarrow} Q' \mid \overline{\ell}\langle m\rangle$ and $Q \overset{\mu}{\Rightarrow} Q'$. Thus we conclude with rule Base.
  - (Ext) If $\ell' \notin \Delta, \ell$, then we can apply rule Ext.
- If $\ell \in \mu$, then we consider $P \mid \overline{\ell'}\langle m\rangle$ and $Q \mid \overline{\ell'}\langle m\rangle$ with ℓ' fresh and $\ell' \neq \ell$, and do the same proof.

□

A consequence of this lemma is that to prove $P \approx_{\text{ip}} Q$, we may assume that rule Ext is never used with ℓ fresh.

Proof of Proposition 42. $\approx_{\text{r}}$ is closed by allocation by Corollary 125.

For any P, Q complete:

- If $P \approx_{\text{ip}} Q$ and $P \xrightarrow{\mu} P'$, then by Lemma 126, we know $\mathtt{ok}(\Delta, \approx_{\text{ip}}, P', Q, \mu)$ using rule Base, so $Q \overset{\mu}{\Rightarrow} Q'$ and $P' \approx_{\text{ip}} Q'$.
- If $P \xrightarrow{\overline{\ell}\langle n\rangle[m]} P'$ (resp. $(\boldsymbol{\nu} n)\overline{\ell}\langle n\rangle[m]$), then as before but for $\mu = \overline{\ell}\langle n\rangle$ (resp. $\mu = (\boldsymbol{\nu} n)\overline{\ell}\langle n\rangle$), we have $P \xrightarrow{\mu} P''$ and $Q \overset{\mu}{\Rightarrow} Q''$ with $P'' \approx_{\text{ip}} Q''$, and $P' = P'' \mid \overline{\ell}\langle m\rangle$. But then we have $Q \xRightarrow{\overline{\ell}\langle n\rangle[m]} Q'$ (resp. $(\boldsymbol{\nu} n)\overline{\ell}\langle n\rangle[m]$) with $Q' = Q'' \mid \overline{\ell}\langle m\rangle$ and $P' \approx_{\text{ip}} Q'$ so we are done.

□

A.2 Characterisation of $\cong^{Arn}$ using $\approx_{\text{r}}$

A.2.1 Soundness

Reference bisimulation up to $\equiv$. Up-to techniques ease the task of proving bisimilarity between processes. Informally, the general idea is to use an extra relation (for instance $\equiv$), and when we need to prove that $P \mathcal{R} Q$, instead of proving that $P' \mathcal{R} Q'$ (for some P', Q' that satisfy the required conditions), we show $P' \equiv \mathcal{R} \equiv Q'$. This often leads to smaller relations, which are easier to check.

We say that a relation $\mathcal{R}$ is *$\equiv$-closed under allocation* if $P \mathcal{R} Q$ implies $P \mid \overline{\ell}\langle n\rangle \equiv\mathcal{R}\equiv Q \mid \overline{\ell}\langle n\rangle$ for any $\overline{\ell}\langle n\rangle$ such that $P \mid \overline{\ell}\langle n\rangle$ and $Q \mid \overline{\ell}\langle n\rangle$ are well-typed.

Definition 127 (Reference Bisimulation up to $\equiv$)**.** A relation $\mathcal{R}$ that is $\equiv$-closed under allocation is a *reference bisimulation up to* $\equiv$ if whenever $P \mathcal{R} Q$ with P, Q complete, $\Delta \vdash P, Q$ and $P \xrightarrow{\alpha} P'$ with $\Delta \vdash \alpha$, we have

1. either there exists Q' such that $Q \overset{\widehat{\alpha}}{\Rightarrow} Q'$ and $P' \equiv\mathcal{R}\equiv Q'$ for some Q'
2. or α is an input $a\langle m\rangle$ and $Q \mid \overline{a}\langle m\rangle \Rightarrow Q'$ with $P' \equiv\mathcal{R}\equiv Q'$ for some Q'.
3. and symmetrically for Q.

Proposition 128. If $\mathcal{R}$ is a reference bisimulation up to $\equiv$, then $\mathcal{R} \subseteq \approx_{\text{r}}$.

Proof. $\equiv\mathcal{R}\equiv$ is a reference bisimulation and $\mathcal{R} \subseteq \equiv\mathcal{R}\equiv$. □

Lemma 129. If $P \approx_{\text{r}} Q$, then $(\boldsymbol{\nu} n)P \approx_{\text{r}} (\boldsymbol{\nu} n)Q$.

Proof. $\mathcal{R} \overset{\text{def}}{=} \{((\boldsymbol{\nu} n)P, (\boldsymbol{\nu} n)Q)$ s.t. $P \approx_{\text{r}} Q\} \cup \approx_{\text{r}}$ is a reference bisimulation up to $\equiv$. □

Definition 130 (Bisimulation up to restriction and up to $\equiv$)**.** A relation $\mathcal{R}$ $\equiv$-closed under allocation is a *reference bisimulation up to restriction and up to* $\equiv$ if whenever $P\ \mathcal{R}\ Q$ with P, Q complete, $\Delta \vdash P, Q$ and $P \xrightarrow{\alpha} P'$ with $\Delta \vdash \alpha$, then

1. either there exists Q' such that $Q \stackrel{\widehat{\alpha}}{\Rightarrow} Q'$, $P' \equiv (\boldsymbol{\nu}\widetilde{n})P''$, $Q' \equiv (\boldsymbol{\nu}\widetilde{n})Q''$ with $P''\ \mathcal{R}\ Q''$ for some $P'', Q', Q'', \widetilde{n}$
2. or α is an input $a\langle m\rangle$ and $Q \mid \overline{a}\langle m\rangle \Rightarrow Q'$ with $P' \equiv (\boldsymbol{\nu}\widetilde{n})P''$, $Q' \equiv (\boldsymbol{\nu}\widetilde{n})Q''$ with $P''\ \mathcal{R}\ Q''$ for some $P'', Q', Q'', \widetilde{n}$.
3. and symmetrically for Q.

Lemma 131. If $\mathcal{R}$ is a reference bisimulation up to restriction and up to $\equiv$, then $\mathcal{R} \subseteq \approx_{\mathrm{r}}$.

Proof. $\mathcal{R}' \stackrel{\text{def}}{=} \{((\boldsymbol{\nu}\widetilde{n})P, (\boldsymbol{\nu}\widetilde{n})Q) \text{ s.t. } P\ \mathcal{R}\ Q\}$ is a reference bisimulation up to $\equiv$. □

The following lemma uses notation M_s, which has been introduced before Definition 40.

Lemma 132 (Extractable store)**.** Let $\Delta \vdash P$, then $P \equiv (\boldsymbol{\nu}\widetilde{n})(M_s \mid P')$ with $\emptyset \vdash P'$ for some M_s.

Proof. We reason by induction on the structure of P. There are two cases depending on the size of Δ.

- If $\Delta = \emptyset$, then nothing has to be done.
- If $\Delta = \Delta', \ell$, then by Lemma 26, $P \equiv (\boldsymbol{\nu}\widetilde{n})(\overline{\ell}\langle m\rangle \mid Q)$ with $\Delta', \Delta'' \vdash Q$. By induction, $Q \equiv (\boldsymbol{\nu}\widetilde{n'})(M_s \mid Q')$ with $\emptyset \vdash Q'$. Therefore, $P \equiv (\boldsymbol{\nu}\widetilde{n}, \widetilde{n'})(M_{s'} \mid Q')$ with $M_{s'} = \overline{\ell}\langle m\rangle \mid M_s$.

□

For $\ell \vdash P$, this lemma can be strengthened to $P \equiv (\boldsymbol{\nu}\widetilde{n})(\overline{\ell}\langle m\rangle \mid M_s \mid P')$ with $\emptyset \vdash P'$.
We can now prove substitutivity for $\approx_{\mathrm{r}}$ under parallel composition.

Lemma 34. If $P \approx_{\mathrm{r}} Q$, and $\emptyset \vdash_{\mathrm{r}} R$, then $P \mid R \approx_{\mathrm{r}} Q \mid R$.

Proof. We show that $\mathcal{R}$ is a bisimulation up to restriction, with

$$\mathcal{R} \stackrel{\text{def}}{=} \{(P \mid R, Q \mid R) \text{ s.t. } P \approx_{\mathrm{r}} Q, \emptyset \vdash R\}$$

- $\mathcal{R}$ is closed by allocation.
- Suppose $P \mid R$ and $Q \mid R$ are complete, and $P \mid R \xrightarrow{\alpha} \tilde{P}$ with $\Delta \vdash \alpha$ we distinguish according to the last rule used (Par, Comm or Close)
  - If $P \mid R \xrightarrow{\alpha} P' \mid R$, first note that P, Q are complete, so either $Q \stackrel{\alpha}{\Rightarrow} Q'$ and $P' \approx_{\mathrm{r}} Q'$ or $\alpha = a\langle m\rangle$ and $Q \mid \overline{a}\langle m\rangle \Rightarrow Q'$ and $P' \approx_{\mathrm{r}} Q'$. In both cases, we have $P' \mid R \approx_{\mathrm{r}} Q' \mid R$.
  - If $P \mid R \xrightarrow{\alpha} P \mid R'$, then $Q \mid R \xrightarrow{\alpha} Q \mid R'$. For $\alpha = \tau, a\langle m\rangle, \overline{a}\langle m\rangle, (\boldsymbol{\nu}b)\overline{a}\langle b\rangle$, we have $\emptyset \vdash R'$. The only remaining case is when $\alpha = (\boldsymbol{\nu}\ell)\overline{a}\langle\ell\rangle$ (by typing, R cannot perform an output on a reference). In that case, $\ell \vdash R'$. Thus $R' \equiv (\boldsymbol{\nu}\widetilde{n})(M_s \mid R'')$ with $\emptyset \vdash R''$. By definition $P \mid M_s \approx_{\mathrm{r}} Q \mid M_s$ hence $P \mid M_s \mid R''\ \mathcal{R}\ Q \mid M_s \mid R''$, which is sufficient as $P \mid R' \equiv (\boldsymbol{\nu}\widetilde{n})(P \mid M_s \mid R'')$ and $Q \mid R' \equiv (\boldsymbol{\nu}\widetilde{n})(Q \mid M_s \mid R'')$.
  - If $P \mid R \xrightarrow{\tau} P' \mid R'$, we distinguish according to the action performed by P:
    - For $P \xrightarrow{\overline{a}\langle n\rangle} P'$ or $P \xrightarrow{a\langle n\rangle} P'$, then $R \xrightarrow{a\langle n\rangle} R'$ and $R \xrightarrow{\overline{a}\langle n\rangle} R'$ respectively, so $\emptyset \vdash R'$. The remaining part of the proof is standard π-calculus reasoning.
    - For $P \xrightarrow{\overline{\ell}\langle n\rangle} P'$, then $R \xrightarrow{\ell\langle n\rangle} R'$ with $\ell \vdash R'$. By Lemma 132, $R' \equiv (\boldsymbol{\nu}\widetilde{n})(\overline{\ell}\langle m\rangle \mid M_s \mid R'')$ with $\emptyset \vdash R''$. By definition, $P \mid M_s \approx_{\mathrm{r}} Q \mid M_s$. Moreover both processes are complete and $P \mid M_s \xrightarrow{\overline{\ell}\langle n\rangle[m]} P' \mid \overline{\ell}\langle m\rangle \mid M_s$. So $Q \mid M_s \xRightarrow{\overline{\ell}\langle n\rangle[m]} Q'$ and $P' \mid \overline{\ell}\langle m\rangle \mid M_s \approx_{\mathrm{r}} Q'$. As all names in subject position in M_s are fresh for Q, we have $Q \mid R \stackrel{\tau}{\Rightarrow} \equiv (\boldsymbol{\nu}\widetilde{n})(Q' \mid R'')$. Moreover we have $P' \mid R' \equiv (\boldsymbol{\nu}\widetilde{n})(P' \mid \overline{\ell}\langle m\rangle \mid R'')$, thus we are done.
  - If $P \mid R \xrightarrow{\tau} (\boldsymbol{\nu}n)(P' \mid R')$, then the reasoning is similar.

□

A.2.2 Completeness

We prove completeness. For this, we need the following lemmas.

Lemma 36. If $(\boldsymbol{\nu} n)(P \mid \overline{s}\langle n\rangle) \cong^{Arn} (\boldsymbol{\nu} n)(Q \mid \overline{s}\langle n\rangle)$ with s a fresh plain name for P and Q, then $P \cong^{Arn} Q$.

Proof. We show that the following relation $\mathcal{R}$ is included in barbed equivalence.

$$\mathcal{R} = \{(P,Q) \mid (\boldsymbol{\nu} n)(P \mid \overline{s}\langle n\rangle) \cong^{Arn} (\boldsymbol{\nu} n)(Q \mid \overline{s}\langle n\rangle) \text{ with } s \text{ fresh}\}$$

We will note $P_1 = (\boldsymbol{\nu} n)(P \mid \overline{s}\langle n\rangle)$ and $Q_1 = (\boldsymbol{\nu} n)(Q \mid \overline{s}\langle n\rangle)$

- If $P \to P'$, then $P_1 \to (\boldsymbol{\nu} n)(P' \mid \overline{s}\langle n\rangle)$ so $Q_1 \Rightarrow Q_2$ with $(\boldsymbol{\nu} n)(P' \mid \overline{s}\langle n\rangle) \cong^{Arn} Q_2$. But we have $Q_2 \equiv (\boldsymbol{\nu} n)(Q' \mid \overline{s}\langle n\rangle)$ and $Q \Rightarrow Q'$.
- If $P \downarrow_{\overline{a}}$, then we have two cases:
  - $a \neq n$, then $P_1 \downarrow_{\overline{a}}$ so $Q_1 \Downarrow_a$ meaning that $Q \Downarrow_a$ as $a \neq s$.
  - $a = n$, then we consider $E \stackrel{\text{def}}{=} [\cdot] \mid s(x).x(_).\overline{s'}$ for a fresh s'. $E[P_1] \to\!\!\to (\boldsymbol{\nu} n)(P \mid \overline{s'})$ so $(\boldsymbol{\nu} n)(P \mid \overline{s'}) \downarrow_{\overline{s'}}$. Therefore, $E[Q_1] \Rightarrow\!\Rightarrow \Downarrow_{\overline{s'}}$ which just means that $E[Q_1] \Downarrow_{\overline{s'}}$. However this can only be done by doing a communication on n, thus we must have $Q \Downarrow_{\overline{n}}$.
- Take an active context E completing for P and Q, we assume s is fresh for E, then $E' \stackrel{\text{def}}{=} E \mid s(x).\overline{s'}\langle x\rangle$ with s' fresh is also completing for P_1 and Q_1, so $E'[P_1] \cong^{Arn} E'[Q_1]$. We then have $E'[P_1] \to (\boldsymbol{\nu} n)(E[P] \mid \overline{s'}\langle n\rangle)$, so $E'[Q_1] \Rightarrow Q'$ with $(\boldsymbol{\nu} n)(E[P] \mid \overline{s'}\langle n\rangle) \cong^{Arn} Q'$, meaning in particular that $Q' \not\downarrow_{\overline{s}}$ and $Q' \Downarrow_{\overline{s'}}$ which is only possible if $Q' \downarrow_{\overline{s'}}$. Moreover, we have that $E'[Q_1] \to (\boldsymbol{\nu} n)(E[Q] \mid s(x).\overline{s'}\langle x\rangle) \Rightarrow Q'$. The same also apply symmetrically for $E'[Q_1] \to (\boldsymbol{\nu} n)(E[Q] \mid s'(x).\overline{s''}\langle x\rangle) \cong^{Arn} P'$ for some P'. Thus we have $(\boldsymbol{\nu} n)(E[P] \mid \overline{s''}\langle x\rangle) \Rightarrow P' \cong^{Arn} (\boldsymbol{\nu} n)(E[Q] \mid \overline{s''}\langle x\rangle) \Rightarrow Q' \cong^{Arn} (\boldsymbol{\nu} n)(E[P] \mid \overline{s'}\langle x\rangle)$ which implies $(\boldsymbol{\nu} n)(E[P] \mid \overline{s'}\langle x\rangle) \cong^{Arn} (\boldsymbol{\nu} n)(E[Q] \mid \overline{s'}\langle x\rangle)$.

□

Lemma 133. If $P \mid [x = y]\overline{s} \cong^{Arn} Q \mid [x = y]\overline{s}$ with $x \neq y$, then $P \cong^{Arn} Q$.

Proof. We have $[x = y]\overline{s} \approx_a \mathbf{0}$ so $P \mid [x = y]\overline{s} \approx_a P$ and similarly for Q. Thus by Lemma 32, $P \cong^{Arn} P \mid [x = y]\overline{s} \cong^{Arn} Q \mid [x = y]\overline{s} \cong^{Arn} Q$. □

This result can be extended to an arbitrary number of $[x = y]\overline{s}$ in parallel.

Proof of Completeness. We show that $\cong^{Arn}$ is a reference bisimulation:

- It is closed by allocation
- Take P, Q complete with $F \stackrel{\text{def}}{=} \text{fn}(P) \cup \text{fn}(Q)$, $P \cong^{Arn} Q$ and $P \xrightarrow{\alpha} P'$
  1. When $\alpha = \tau$, we take $E \stackrel{\text{def}}{=} [\cdot]$. Then $E[P] \to P'$. So we have $Q \Rightarrow Q'$ with $P' \cong^{Arn} Q'$.
  2. When $\alpha = a(n)$, we take $E \stackrel{\text{def}}{=} [\cdot] \mid \overline{a}\langle n\rangle$. Then $E[P] \to P'$. So we have $Q \mid \overline{a}\langle n\rangle \Rightarrow Q'$ with $P' \cong^{Arn} Q'$.
  3. When $\alpha = \overline{a}\langle n\rangle$, we take $E \stackrel{\text{def}}{=} [\cdot] \mid a(x).[x = n]s \mid \overline{s}$ with s fresh. Then $E[P] \to\!\!\to P'$ with $E[P] \downarrow_{\overline{s}}$ and $P' \not\downarrow_{\overline{s}}$. This implies that $E[Q] \Rightarrow Q'$ with $P' \cong^{Arn} Q'$. So we have $Q' \not\downarrow_{\overline{s}}$, which is only possible if $Q \xRightarrow{\overline{a}\langle n\rangle} Q'$.
  4. When $\alpha = (\boldsymbol{\nu} n)\overline{a}\langle n\rangle$, we take $E \stackrel{\text{def}}{=} [\cdot] \mid a(x).(s \mid \overline{s'}\langle x\rangle \mid \prod_{m\in F}[x = m]\overline{s}) \mid \overline{s}$ with s, s' fresh. Then $E[P] \to\!\!\to (\boldsymbol{\nu} n)(P' \mid \prod_{m\in F}[n = m]\overline{s} \mid \overline{s'}\langle n\rangle)$. This implies that $E[Q] \Rightarrow Q''$ with $(\boldsymbol{\nu} n)(P' \mid \prod_{m\in F}[x = m]\overline{s} \mid \overline{s'}\langle n\rangle) \cong^{Arn} Q''$. As $Q'' \not\downarrow_{\overline{s}}$, we necessarily have $Q'' \equiv (\boldsymbol{\nu} n)(Q' \mid \prod_{m\in F}[n = m]\overline{s} \mid \overline{s'}\langle n\rangle)$. By Lemmas 36 and 133, this means that $P' \cong^{Arn} Q'$. But then $Q \xRightarrow{(\boldsymbol{\nu} n)\overline{a}\langle n\rangle} Q'$ so we can conclude.

5. When $\alpha = \overline{\ell}\langle n\rangle[m]$, we take $E \stackrel{\text{def}}{=} [\cdot] \mid \ell(x).(\overline{\ell}\langle m\rangle \mid [x = n]s) \mid \overline{s}$. Then $E[P] \to\!\!\to P'$ with $P' \not\downarrow_{\overline{s}}$. This implies that $E[Q] \Rightarrow Q'$ with $P' \cong^{Arn} Q'$. As $Q' \not\downarrow_{\overline{s}}$ we have $Q \xrightarrow{\overline{\ell}\langle n\rangle[m]} Q'$.
6. When $\alpha = (\boldsymbol{\nu} n)\overline{\ell}\langle n\rangle[m]$, we take
$E \stackrel{\text{def}}{=} [\cdot] \mid \ell(x).(\overline{\ell}\langle m\rangle \mid s \mid \overline{s'}\langle x\rangle \mid \prod_{m\in F}[x = m]\overline{s}) \mid \overline{s}$. Then
$E[P] \to\!\!\to (\boldsymbol{\nu} n)(P' \mid \prod_{m\in F}[n = m]\overline{s} \mid \overline{s'}\langle n\rangle)$. This implies $E[Q] \Rightarrow Q''$ with $(\boldsymbol{\nu} n)(P' \mid \prod_{m\in F}[n = m]\overline{s} \mid \overline{s'}\langle n\rangle) \cong^{Arn} Q''$. As $Q'' \not\downarrow_{\overline{s}}$, we necessarily have
$Q'' \equiv (\boldsymbol{\nu} n)(Q' \mid \prod_{m\in F}[n = m]\overline{s} \mid \overline{s'}\langle n\rangle)$. By Lemmas 36 and 133, this means $P' \cong^{Arn} Q'$. But then $Q \xrightarrow{(\boldsymbol{\nu} n)\overline{\ell}\langle n\rangle[m]} Q'$ so we are done.

□

A.3 Results and examples for Section 3.3

A.3.1 Properties of the encoding

Lemma 46. If $\emptyset \vdash_{\mathrm{r}} P$, then there exists R in $\pi^{\mathtt{ref}}$ such that $[\![R]\!] \approx_a P$.

Proof. We construct R by induction on the structure of P, we only discuss the two cases below, the other cases are immediate.

- For $\emptyset \vdash (\boldsymbol{\nu}\ell)P$, we know that $\ell \vdash P$ so we have two cases according to Lemma 26:
  - $P \equiv \overline{\ell}\langle m\rangle \mid P'$. Thus $(\boldsymbol{\nu}\ell)P \equiv (\boldsymbol{\nu}\ell)(\overline{\ell}\langle m\rangle \mid P')$ with $\emptyset \vdash P'$. By induction, we have Q' with $[\![Q']\!] \approx_a P'$. Therefore we have $[\![(\boldsymbol{\nu}\ell = m)Q']\!] \approx_a (\boldsymbol{\nu}\ell)P$.
  - $P \equiv (\boldsymbol{\nu} m)(\overline{\ell}\langle m\rangle \mid P')$. We reason by induction on the type of ℓ. If m is a plain name, we can conclude as above with $[\![(\boldsymbol{\nu} m)(\boldsymbol{\nu}\ell = m)Q']\!]$. Otherwise, m is reference name and there exists R such that $(\boldsymbol{\nu} m)(\boldsymbol{\nu}\ell)(\overline{\ell}\langle m\rangle \mid P') \equiv [\![R]\!]$. As $(\boldsymbol{\nu}\ell)P \equiv (\boldsymbol{\nu} m)(\boldsymbol{\nu}\ell)(\overline{\ell}\langle m\rangle \mid P')$, we are done.
- For $\emptyset \vdash \ell(x).P$, we know that $\ell \vdash P$ then
  - either $P \equiv \overline{\ell}\langle m\rangle \mid P'$ with $\emptyset \vdash P'$. By induction, we have $[\![Q']\!] \approx_a P'$ in which case we take $\ell \triangleright (x).Q'$ or $\ell \bowtie m(x).Q'$ depending on whether $m = x$ or not,
  - or $P \equiv (\boldsymbol{\nu} m)(\overline{\ell}\langle m\rangle \mid P')$ and then $\ell(x).P \approx_a (\boldsymbol{\nu} m)\ell(x).(\overline{\ell}\langle m\rangle \mid P')$ and we can refer to the first case.

□

A.3.2 Additional material for the examples in Section 3.3.3

Proof of Example 50. To get an idea of how P_s and Q_s evolve, let us consider first $E \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell = z)[\cdot]$. Then $E[Q_s]$ can reduce to one of the following:

1. $(\boldsymbol{\nu}\ell = z)(\boldsymbol{\nu} t)\ell \triangleleft a.(\overline{t} \mid !t.\ell \triangleleft b.(\overline{c} \mid \ell \triangleleft a.(\overline{t} \mid c)))$
2. $(\boldsymbol{\nu}\ell = a)(\boldsymbol{\nu} t)(\overline{t} \mid !t.\ell \triangleleft b.(\overline{c} \mid \ell \triangleleft a.(\overline{t} \mid c))) \mid c^n \mid \overline{c}^n$
3. $(\boldsymbol{\nu}\ell = a)(\boldsymbol{\nu} t)(\ell \triangleleft b.(\overline{c} \mid \ell \triangleleft a.(\overline{t} \mid c)) \mid !t.\ell \triangleleft b.(\overline{c} \mid \ell \triangleleft a.(\overline{t} \mid c))) \mid c^n \mid \overline{c}^n$
4. $(\boldsymbol{\nu}\ell = b)(\boldsymbol{\nu} t)(\ell \triangleleft a.(\overline{t} \mid c) \mid !t.\ell \triangleleft b.(\overline{c} \mid \ell \triangleleft a.(\overline{t} \mid c))) \mid c^n \mid \overline{c}^{n+1}$.

Similarly, $E[P_s]$ can reduce to those four processes but with the role of a and b swapped. Notice that when $E[Q_s] \Rightarrow Q'$, then there is a correspondence between the value stored in ℓ (i.e. a or b) and the presence of more $\overline{c}$ processes than c processes (or the same number).

We now consider the following context:

$$E_0 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell = z)([\cdot] \mid \ell \bowtie z(x).[x = b]s_0.s_1.(P_{11} \mid P_{12}) \mid \overline{s_0} \mid \overline{s_1})$$

$$P_{11} \stackrel{\text{def}}{=} \ell \triangleright (x).[x = z]s_{11} \mid \overline{s_{11}} \qquad P_{12} \stackrel{\text{def}}{=} c.\ell \triangleright (x).[x = z]s_{12} \mid \overline{s_{12}}$$

with s_0, s_{11}, s_{12} fresh names.

At first $\overline{s_0}$ and $\overline{s_1}$ are the only observables, meaning $E_0[P_s] \downarrow_{\overline{s_0}}$ and $E_0[P_s] \downarrow_{\overline{s_1}}$, but then $E_0[P_s] \to\to\to (\boldsymbol{\nu}\ell = z)((\boldsymbol{\nu}t)(\overline{t} \mid !t.\,\ell \triangleleft a.\,(\overline{c} \mid \ell \triangleleft b.\,(\overline{t} \mid c))) \mid s_1.\,(P_{11} \mid P_{12}) \mid \overline{s_1}) \stackrel{\text{def}}{=} P'$ where the three reductions have been derived using rules R-Write, R-Swap, and R-Comm respectively. Finally, we have $P' \not\Downarrow_{\overline{s_0}}$, whereas $P' \downarrow_{\overline{s_1}}$.

Thus, to avoid the observable $\overline{s_0}$, process $E_0[Q_s]$ must reduce to a process with b stored in ℓ before doing the swap in E_0. This implies that the swap is executed in a state that corresponds to case 4 above. So for any Q' with $E[Q_s] \Rightarrow Q'$ and $Q' \not\downarrow_{\overline{s_0}}$ and $Q' \Downarrow_{\overline{s_1}}$, such process Q' has one of the following forms:

1. $Q'_1 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell = a)((\boldsymbol{\nu}t)(\overline{t} \mid !t.\,\ell \triangleleft b.\,(\overline{c} \mid \ell \triangleleft a.\,(\overline{t} \mid c)) \mid c^n \mid \overline{c}^n) \mid s_1.\,(P_{11} \mid P_{12}) \mid \overline{s_1})$
2. $Q'_2 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell = a)((\boldsymbol{\nu}t)(\ell \triangleleft b.\,(\overline{c} \mid \ell \triangleleft a.\,(\overline{t} \mid c)) \mid !t.\,\ell \triangleleft b.\,(\overline{c} \mid \ell \triangleleft a.\,(\overline{t} \mid c)) \mid c^n \mid \overline{c}^n) \mid s_1.\,(P_{11} \mid P_{12}) \mid \overline{s_1})$
3. $Q'_3 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell = b)((\boldsymbol{\nu}t)(\ell \triangleleft a.\,(\overline{t} \mid c) \mid !t.\,\ell \triangleleft b.\,(\overline{c} \mid \ell \triangleleft a.\,(\overline{t} \mid c)) \mid c^n \mid \overline{c}^{n+1}) \mid s_1.\,(P_{11} \mid P_{12}) \mid \overline{s_1})$
4. $Q'_4 \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell = z)((\boldsymbol{\nu}t)(\ell \triangleleft a.\,(\overline{t} \mid c) \mid !t.\,\ell \triangleleft b.\,(\overline{c} \mid \ell \triangleleft a.\,(\overline{t} \mid c)) \mid c^n \mid \overline{c}^{n+1}) \mid s_1.\,(P_{11} \mid P_{12}) \mid \overline{s_1})$

Then we use either P_{11} or P_{12} depending on the form of Q'. If Q' is of the first three forms, then we use P_{11}.

Indeed, $P' \to\to (\boldsymbol{\nu}\ell = z)((\boldsymbol{\nu}t)(\overline{t} \mid !t.\,\ell \triangleleft a.\,(\overline{c} \mid \ell \triangleleft b.\,(\overline{t} \mid c))) \mid P_{12}) \stackrel{\text{def}}{=} P''$ using rules R-Read and R-Comm respectively. Notice that $P'' \not\Downarrow_{\overline{s_{11}}}$. On the other hand, z does not appear anywhere else than in a matching in Q', thus there is no reduction $Q' \Rightarrow Q''$ with $Q'' \not\downarrow_{\overline{s_{11}}}$ for any Q''.

In the other case, it holds that $Q'_4 \to\to (\boldsymbol{\nu}\ell = z)((\boldsymbol{\nu}t)(\ell \triangleleft a.\,(\overline{t} \mid c) \mid !t.\,\ell \triangleleft b.\,(\overline{c} \mid \ell \triangleleft a.\,(\overline{t} \mid c)) \mid c^n \mid \overline{c}^n) \mid P_{11}) \stackrel{\text{def}}{=} Q''$ using rules R-Comm, R-Read, and R-Comm respectively. Then we have $Q'' \not\Downarrow_{\overline{s_{12}}}$. However, the only output $\overline{c}$ is behind a write $\ell \triangleleft a$ in P'. Thus, there is no $P' \Rightarrow P''$ with $P'' \not\downarrow_{\overline{s_{12}}}$.

We can finally conclude $P_s \not\cong^{ref} Q_s$. □

Proof of Example 51. Recall the definitions of the two processes (we rename the processes that are given in the main text, to ease readability):

$$P \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell_1 = 0, \ell_2 = 0)(R \mid (\boldsymbol{\nu}t)(\overline{t} \mid !t.\,\ell_1 \triangleleft 1.\,\ell_1 \triangleleft 0.\,\ell_2 \triangleleft 1.\,\ell_2 \triangleleft 0.\,\overline{t}))$$
$$Q \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell_1 = 0, \ell_2 = 0)(R \mid (\boldsymbol{\nu}t)(\overline{t} \mid !t.\,\ell_1 \triangleleft 1.\,\ell_2 \triangleleft 1.\,\ell_1 \triangleleft 0.\,\ell_2 \triangleleft 0.\,\overline{t}))$$

To prove their equivalence, we introduce the following processes:

$$P' \stackrel{\text{def}}{=} !t.\,\ell_1(_).\,(\overline{\ell_1}\langle 1\rangle \mid \ell_1(_).\,(\overline{\ell_1}\langle 0\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 1\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 0\rangle \mid \overline{t}))))$$
$$Q' \stackrel{\text{def}}{=} !t.\,\ell_1(_).\,(\overline{\ell_1}\langle 1\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 1\rangle \mid \ell_1(_).\,(\overline{\ell_1}\langle 0\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 0\rangle \mid \overline{t}))))$$

$$\begin{aligned}
P_1 = Q_1 &\stackrel{\text{def}}{=} \overline{t}\\
P_2 &\stackrel{\text{def}}{=} \ell_1(_).\,(\overline{\ell_1}\langle 1\rangle \mid \ell_1(_).\,(\overline{\ell_1}\langle 0\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 1\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 0\rangle \mid \overline{t}))))\\
Q_2 &\stackrel{\text{def}}{=} \ell_1(_).\,(\overline{\ell_1}\langle 1\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 1\rangle \mid \ell_1(_).\,(\overline{\ell_1}\langle 0\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 0\rangle \mid \overline{t}))))\\
P_3 &\stackrel{\text{def}}{=} \ell_1(_).\,(\overline{\ell_1}\langle 0\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 1\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 0\rangle \mid \overline{t})))\\
Q_3 &\stackrel{\text{def}}{=} \ell_2(_).\,(\overline{\ell_2}\langle 1\rangle \mid \ell_1(_).\,(\overline{\ell_1}\langle 0\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 0\rangle \mid \overline{t})))\\
P_4 &\stackrel{\text{def}}{=} \ell_2(_).\,(\overline{\ell_2}\langle 1\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 0\rangle \mid \overline{t}))\\
Q_4 &\stackrel{\text{def}}{=} \ell_1(_).\,(\overline{\ell_1}\langle 0\rangle \mid \ell_2(_).\,(\overline{\ell_2}\langle 0\rangle \mid \overline{t}))\\
P_5 = Q_5 &\stackrel{\text{def}}{=} \ell_2(_).\,(\overline{\ell_2}\langle 0\rangle \mid \overline{t})
\end{aligned}$$

P' and Q' are the encodings of the replicated part of P and Q. Then P_i and Q_i are the processes that can be reached from P' and Q'.

We now show that the relation $\mathcal{R} \cup \mathcal{R}^{-1}$ is an $\approx_{\text{ip}}$-bisimulation where we have:

$$\begin{aligned}\mathcal{R} \stackrel{\text{def}}{=} &\left\{ \begin{array}{c} (\overline{\ell_1}\langle n_1\rangle \mid (\boldsymbol{\nu} t)(P' \mid P_i),\ \overline{\ell_1}\langle n_1'\rangle \mid (\boldsymbol{\nu} t)(Q' \mid Q_j)) \\ \text{for any } n_1, n_1' \in \{0,1\}, i, j \end{array} \right\} \\ \cup &\left\{ \begin{array}{c} (\overline{\ell_2}\langle n_2\rangle \mid (\boldsymbol{\nu} t)(P' \mid P_i),\ \overline{\ell_2}\langle n_2'\rangle \mid (\boldsymbol{\nu} t)(Q' \mid Q_j)) \\ \text{for any } n_2, n_2' \in \{0,1\}, i, j \end{array} \right\} \\ \cup &\left\{ \begin{array}{c} (\overline{\ell_1}\langle n_1\rangle \mid \overline{\ell_2}\langle n_2\rangle \mid (\boldsymbol{\nu} t)(P' \mid P_i),\ \overline{\ell_1}\langle n_1'\rangle \mid \overline{\ell_2}\langle n_2'\rangle \mid (\boldsymbol{\nu} t)(Q' \mid Q_j)) \\ \text{for any } n_1, n_1', n_2, n_2' \in \{0,1\}, i, j \end{array} \right\}\end{aligned}$$

First, note that the only free names appearing in those processes are ℓ_1 and ℓ_2. Thus for any $P\ \mathcal{R}\ Q$, the only actions to consider are $\tau, \ell_i\langle n\rangle$ and $\overline{\ell_i}\langle n\rangle$, for $i = 1, 2$.

For any $P\ \mathcal{R}\ Q$, we have:

- If $P \xrightarrow{\tau} P_0$, then $P_0\ \mathcal{R}\ Q$
- If $P \xrightarrow{\ell_i\langle n\rangle} P_0$, then $P_0\ \mathcal{R}\ Q \mid \overline{\ell_i}\langle n\rangle$
- If $P \xrightarrow{\overline{\ell_i}\langle n\rangle} P_0$, then either $Q \xrightarrow{\overline{\ell_i}\langle n\rangle} Q_0$ and $P_0\ \mathcal{R}\ Q_0$, or $Q \xrightarrow{\overline{\ell_i}\langle 1-n\rangle} Q_0$. In this case, we use rule Ext (from Definition 38) to add the other reference if $\Delta \neq \ell_1, \ell_2$. Then after at most 5 internal transitions (by cycling around the P_i or Q_j), we obtain a process Q_0 that can make the required transition $Q_0 \xrightarrow{\overline{\ell_i}\langle n\rangle} Q_0'$ with $P_0\ \mathcal{R}\ Q_0'$.

As $\mathcal{R} \cup \mathcal{R}^{-1}$ is an $\approx_{\text{ip}}$-bisimulation, we have $\mathcal{R} \subseteq \approx_{\text{r}}$. Moreover, $(\boldsymbol{\nu}\ell_1, \ell_2)([\![R]\!] \mid [\cdot])$ is an active context, so this implies $[\![P]\!] \approx_{\text{r}} [\![Q]\!]$. By Theorems 37 and 47, we can conclude $P \cong^{ref} Q$.

To extend this result to barbed congruence, we notice that for all σ,

1. either $P\sigma = (\boldsymbol{\nu}\ell_1 = 0, \ell_2 = 0)(R\sigma \mid (\boldsymbol{\nu} t)(\overline{t} \mid !t.\, \ell_1 \triangleleft 1.\, \ell_1 \triangleleft 0.\, \ell_2 \triangleleft 1.\, \ell_2 \triangleleft 0.\, \overline{t})$
2. or $P\sigma = (\boldsymbol{\nu}\ell_1 = 0, \ell_2 = 0)(R\sigma \mid (\boldsymbol{\nu} t)(\overline{t} \mid !t.\, \ell_1 \triangleleft 0.\, \ell_1 \triangleleft 0.\, \ell_2 \triangleleft 0.\, \ell_2 \triangleleft 0.\, \overline{t})$
3. or $P\sigma = (\boldsymbol{\nu}\ell_1 = 1, \ell_2 = 1)(R\sigma \mid (\boldsymbol{\nu} t)(\overline{t} \mid !t.\, \ell_1 \triangleleft 1.\, \ell_1 \triangleleft 1.\, \ell_2 \triangleleft 1.\, \ell_2 \triangleleft 1.\, \overline{t})$

As $P \cong_c^{ref} Q$ holds for any R, it also holds for any $R\sigma$, which prove the first case. Moreover, the proof never uses the fact that 0 and 1 are distinct, so we can prove in the same way that cases 2 and 3 hold.

We conclude $P \cong_c^{ref} Q$. □

We now present an additional example, which corresponds to a generalisation of Example 51.

Example 134. Here we remove the assumption that the two references can only hold values 0 and 1. This enables the context to store fresh names in references. If used with the original processes, these are distinguished by using those fresh values to block transition along the lines of Example 50. To make these processes equivalent again, we could add in parallel a buffer as in Example 49. However, by making these additions, we would also enable P_1 to desynchronise the content in ℓ_1 and ℓ_2 and have $(1, 1)$. The solution is to prevent those buffers from writing at a different 'time' than the 'time' they have read. For this we introduce a more complex buffer B_i^j. Consider the following processes:

$$\begin{aligned} B_i^j &\stackrel{\text{def}}{=} r(x^j).\, \mathbf{0} \mid !r(x^j).\, t_i.\, \ell^j \bowtie x^j(y^j).\, (\overline{r}\langle y^j\rangle \mid \overline{t_i}) \\ S_i^j &\stackrel{\text{def}}{=} !t_i.\, \ell^j \triangleright (x^j).\, (\overline{t_i} \mid (\boldsymbol{\nu} r)(\overline{r}\langle x^j\rangle \mid B_i^j)) \end{aligned}$$

$$\begin{aligned} P \stackrel{\text{def}}{=} (\boldsymbol{\nu} t_1, t_2, t_3, t_4)\Big(&\overline{t_1} \mid !t_1.\, \ell^1 \triangleleft 1.\, \overline{t_2} \mid S_1^1 \mid S_1^2 \mid !t_2.\, \ell^1 \triangleleft 0.\, \overline{t_3} \mid S_2^1 \mid S_2^2 \\ &\mid !t_3.\, \ell^2 \triangleleft 1.\, \overline{t_4} \mid S_3^1 \mid S_3^2 \mid !t_4.\, \ell^2 \triangleleft 0.\, \overline{t_1} \mid S_4^1 \mid S_4^2\Big) \\ Q \stackrel{\text{def}}{=} (\boldsymbol{\nu} t_1, t_2, t_3, t_4)\Big(&\overline{t_1} \mid !t_1.\, \ell^1 \triangleleft 1.\, \overline{t_2} \mid S_1^1 \mid S_1^2 \mid !t_2.\, \ell^2 \triangleleft 1.\, \overline{t_3} \mid S_2^1 \mid S_2^2 \\ &\mid !t_3.\, \ell^1 \triangleleft 0.\, \overline{t_4} \mid S_3^1 \mid S_3^2 \mid !t_4.\, \ell^2 \triangleleft 0.\, \overline{t_1} \mid S_4^1 \mid S_4^2\Big) \end{aligned}$$

We have $P \cong^{ref} Q$. If we take $E \stackrel{\text{def}}{=} (\boldsymbol{\nu}\ell^1 = 0)(\boldsymbol{\nu}\ell_2 = 0)[\cdot]$, we have
$E[Q] \rightarrow\!\!\!\!\rightarrow (\boldsymbol{\nu}\ell^1 = 1)(\boldsymbol{\nu}\ell^2 = 1)Q'$ for some Q'. However, there is no sequence of reductions such that $E[P] \Rightarrow (\boldsymbol{\nu}\ell^1 = 1)(\boldsymbol{\nu}\ell^2 = 1)P'$ for any P'.

If we forget all S_i^j's, then these processes are similar to the 'loop' used in the previous example but split into multiple replications. Those S_i^j's help to equate the two processes even if the context can write any value in ℓ_1, ℓ_2.

Process S_i^j can only be activated when $\overline{t_i}$ is available. It then reads the content of ℓ_j to initialise a new buffer B_i^j.

Process B_i^j contains value x_i^j that is the object of $\overline{r}\langle x_i^j\rangle$. Process B_i^j can be stopped by making the communication with the first input on r, or can be used to swap its content with the content of ℓ^j. Note that this swap can only be done when $\overline{t_i}$ is available, so it cannot be used to desynchronise the content in ℓ_1, and ℓ_2.

Appendix B

Proofs for Chapter 6

B.1 A simplified example with wb-bisimulation (Section 6.1.2)

We discuss below a simplified example, which exposes the main difficulties that arise when studying the well-bracketed state change example. The primary simplification consists in using single-use functions, i.e. without replication. As a consequence, the two calls to the external function are split into two separate functions. In ML, this corresponds to the terms N and L below:

$$\begin{array}{rcl} N & \stackrel{\text{def}}{=} & \texttt{let x = ref 0 in } (N_1, N_2) \\ N_1 & \stackrel{\text{def}}{=} & \texttt{fun f -> x := 1; f (); !x} \\ N_2 & \stackrel{\text{def}}{=} & \texttt{fun f -> x := 0; f (); x := 1} \end{array}$$

$$\begin{array}{rcl} L & \stackrel{\text{def}}{=} & (L_1, L_2) \\ L_1 & \stackrel{\text{def}}{=} & \texttt{fun f -> f (); 1} \\ L_2 & \stackrel{\text{def}}{=} & \texttt{fun f -> f (); ()} \end{array}$$

with the constraint that each term may only be called once. (Such a simplification would not work on the well-bracketed state change example, as M_1 and M_2 become equivalent even dropping well-bracketing if used at most once.)

Intuitively,

$$h_2\ \texttt{f}\ ;\ h_1\ \texttt{f}$$

is similar to a single use of M_1, and the same for L and M_2.

The translation of the above terms is as follows:

$$\begin{array}{rcl} [\![N]\!]_{p_0} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu}\ell, x, y)(\overline{\ell}\langle 0\rangle \mid \overline{p_0}\langle x, y\rangle \mid P_1 \mid P_2) \\ P_1 & \stackrel{\text{def}}{=} & x(z, p).\, \ell \triangleleft 1.\, (\boldsymbol{\nu}p')(\overline{z}\langle \star, p'\rangle \mid p'.\, \ell \triangleright (n).\, \overline{p}\langle n\rangle) \\ P_2 & \stackrel{\text{def}}{=} & y(z, q).\, \ell \triangleleft 0.\, (\boldsymbol{\nu}q')(\overline{z}\langle \star, q'\rangle \mid q'.\, \ell \triangleleft 1.\, \overline{q}) \\ [\![L]\!]_{p_0} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu}x, y)(\overline{p_0}\langle x, y\rangle \mid Q_1 \mid Q_2) \\ Q_1 & \stackrel{\text{def}}{=} & x(z, p).\, (\boldsymbol{\nu}p')(\overline{z}\langle \star, p'\rangle \mid p'.\, \overline{p}\langle 1\rangle) \\ Q_2 & \stackrel{\text{def}}{=} & y(z, q).\, (\boldsymbol{\nu}q')(\overline{z}\langle \star, q'\rangle \mid q'.\, \overline{q}) \end{array}$$

Indeed, the translation of an ML program is parametrised upon a continuation name (here p_0).

We introduce a slight abuse of notation, and write $[\![N]\!]_{p_0} \xrightarrow{(\boldsymbol{\nu}x,y)\overline{p_0}\langle x,y\rangle} [\![N]\!]$, and similarly for notation $[\![L]\!]$.

Below we show that $[\![N]\!]$ and $[\![L]\!]$ are equivalent. First however we note that well-bracketing is necessary for this. Sequentiality alone is not sufficient: under the type system for sequentiality we can observe the following trace from $[\![N]\!]$:

$$[\![N]\!] \xrightarrow{x\langle\star,p\rangle} \xRightarrow{(\boldsymbol{\nu}p')\overline{z}\langle\star,p'\rangle} \xrightarrow{y\langle\star,q\rangle} \xRightarrow{\overline{z}\langle\star,q'\rangle} \xrightarrow{p'} \xRightarrow{\overline{p}\langle 0\rangle}$$

This trace is not well-bracketed: function x is called first, but the continuation p is returned before q. Process $[\![L]\!]$ may not produce such a trace — it may only emit 1. However, the above non-well-bracketed trace is the only trace from $[\![N]\!]$ that may produce 0.

We prove $[\![N]\!] \approx^{\emptyset}_{\mathrm{wb}} [\![L]\!]$ by defining a relation, and relying on up-to techniques for wb-bisimulation.

For that, we use the abbreviations

$$\begin{array}{llllll} P'_1 & \stackrel{\text{def}}{=} & p'.\,\ell \triangleright (n).\,\overline{p}\langle n\rangle, & Q'_1 & \stackrel{\text{def}}{=} & p'.\,\overline{p}\langle 1\rangle, \\ P'_2 & \stackrel{\text{def}}{=} & q'.\,\ell \triangleleft 1.\,\overline{q} & Q'_2 & \stackrel{\text{def}}{=} & q'.\,\overline{q} \end{array}$$

This allows us to define the following relation, called $\mathcal{R}$:

$$\begin{array}{llll} \{(\emptyset, & (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P_1 \mid P_2), Q_1 \mid Q_2), & & \\ ((p' : \mathtt{i}, p : \mathtt{o}), & (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P'_1 \mid P_2), Q'_1 \mid Q_2), & ((q' : \mathtt{i}, q : \mathtt{o}), & (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P_1 \mid P'_2), Q_1 \mid Q'_2), \\ ((q' : \mathtt{i}, q : \mathtt{o}, p' : \mathtt{i}, p : \mathtt{o}), & (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P'_1 \mid P'_2), Q'_1 \mid Q'_2), & (\emptyset, & (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P_2), Q_2), \\ ((p' : \mathtt{i}, p : \mathtt{o}, q' : \mathtt{i}, q : \mathtt{o}), & (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P'_1 \mid P'_2), Q'_1 \mid Q'_2), & (\emptyset, & (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P_1), Q_1), \\ ((p' : \mathtt{i}, p : \mathtt{o}), & (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P'_1), Q'_1), & ((q' : \mathtt{i}, q : \mathtt{o}), & (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P'_2), Q'_2), \\ ((q' : \mathtt{i}, q : \mathtt{o}), & (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P'_2), Q'_2), & (\emptyset, & (\boldsymbol{\nu}\ell)\overline{\ell}\langle 1\rangle, \mathbf{0})\} \end{array}$$

$\mathcal{R}$ is a wb-bisimulation up-to deterministic reduction and up-to static context. Ensuring well-bracketing restricts transitions for the two triplets below:

$$((q' : \mathtt{i}, q : \mathtt{o}, p' : \mathtt{i}, p : \mathtt{o}), (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P'_1 \mid P'_2), Q'_1 \mid Q'_2)$$

$$((p' : \mathtt{i}, p : \mathtt{o}, q' : \mathtt{i}, q : \mathtt{o}), (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P'_1 \mid P'_2), Q'_1 \mid Q'_2)$$

Indeed, we may only test the input on q' (resp. p') while both are unguarded in both cases. This is crucial for the first triplet as allowing the input on p' on the first process would lead to a process (after a deterministic reduction) that can send $\overline{p}0$, while the second one cannot emit 0.

Without up-to techniques.

We define some additional relations, which allow us to show the improvement brought by up-to techniques.

For this, we introduce the following notations for processes related to P_1 and Q_1 respectively:

$$P^a_1 \stackrel{\text{def}}{=} \ell \triangleleft 1.\,P^b_1 \qquad P^b_1 \stackrel{\text{def}}{=} (\boldsymbol{\nu}p')(\overline{z}\langle \star, p'\rangle \mid P^c_1) \qquad P^c_1 \stackrel{\text{def}}{=} p'.\,P^d_1 \qquad P^d_1 \stackrel{\text{def}}{=} \ell \triangleright (n).\,\overline{p}\langle n\rangle$$

$$Q^c_1 \stackrel{\text{def}}{=} p'.\,\overline{p}\langle 1\rangle \qquad Q^b_1 \stackrel{\text{def}}{=} (\boldsymbol{\nu}p')(\overline{z}\langle \star, p'\rangle \mid Q^c_1)$$

and similarly for processes P_2 and Q_2.

We can remark that P^c_i is the same as P'_i introduced above to define $\mathcal{R}$.

$$\begin{array}{lll} \mathcal{R}' \stackrel{\text{def}}{=} \{(p : \mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P^a_1 \mid P_2), Q^a_1 \mid Q_2), & & (p : \mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P_1 \mid P^a_2), Q_1 \mid Q^b_2), \\ \quad ((q : \mathtt{o}, p' : \mathtt{i}, p : \mathtt{o}), (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P^c_1 \mid P^a_2), Q^c_1 \mid Q^b_2) & & (p : \mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P^d_1 \mid P_2), \overline{p}\langle 1\rangle \mid Q_2) \\ \quad ((p : \mathtt{o}, q' : \mathtt{i}, q : \mathtt{o}), (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P^a_1 \mid P^c_2), Q^b_1 \mid Q^c_2) & & (q : \mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P_1 \mid P^d_2), Q_1 \mid \overline{q}) \\ \quad ((q : \mathtt{o}, p' : \mathtt{i}, p : \mathtt{o}), (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P^c_1 \mid P^d_2), Q^c_1 \mid \overline{q}) & & (q : \mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P^a_2), Q^b_2) \\ \quad ((p : \mathtt{o}, q' : \mathtt{i}, q : \mathtt{o}), (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P^d_1 \mid P^c_2), \overline{p}\langle 1\rangle \mid Q^c_2) & & (p : \mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P^a_1), Q^b_1) \\ \quad (q : \mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P^d_2), \overline{q}) & & (q : \mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P^d_2), \overline{q}) \\ \quad (p : \mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P^d_1), \overline{p}\langle 1\rangle)\} & & \end{array}$$

$$
\begin{aligned}
\mathcal{R}'' \stackrel{\text{def}}{=} \{ & (p:\mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P_1^b \mid P_2), Q_1^b \mid Q_2), && (p:\mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P_1 \mid P_2^b), Q_1 \mid Q_2^b), \\
& ((q:\mathtt{o}, p':\mathtt{i}, p:\mathtt{o}), (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P_1^c \mid P_2^b), Q_1^c \mid Q_2^b) && (p:\mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid \overline{p}\langle 1\rangle \mid P_2), \overline{p}\langle 1\rangle \mid Q_2) \\
& ((p:\mathtt{o}, q':\mathtt{i}, q:\mathtt{o}), (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P_1^b \mid P_2^c), Q_1^b \mid Q_2^c) && (q:\mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P_1 \mid \overline{q}), Q_1 \mid \overline{q}) \\
& ((q:\mathtt{o}, p':\mathtt{i}, p:\mathtt{o}), (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P_1^c \mid \overline{q}), Q_1^c \mid \overline{q}) && (q:\mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 0\rangle \mid P_2^b), Q_2^b) \\
& ((p:\mathtt{o}, q':\mathtt{i}, q:\mathtt{o}), (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid \overline{p}\langle 1\rangle \mid P_2^c), \overline{p}\langle 1\rangle \mid Q_2^c) && (p:\mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid P_1^b), Q_1^b) \\
& (q:\mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid \overline{q}), \overline{q}) && (p:\mathtt{o}, (\boldsymbol{\nu}\ell)(\overline{\ell}\langle 1\rangle \mid \overline{p}\langle 1\rangle), \overline{p}\langle 1\rangle) \\
& \}
\end{aligned}
$$

We see that $\mathcal{R}$, $\mathcal{R}'$ and $\mathcal{R}''$ are comparable in size. We have that:

- $\mathcal{R} \uplus \mathcal{R}'$ is a wb-bisimulation up-to static context.
- $\mathcal{R} \uplus \mathcal{R}''$ is a wb-bisimulation up-to deterministic τ.
- $\mathcal{R} \uplus \mathcal{R}' \uplus \mathcal{R}''$ is a wb-bisimulation.

B.2 Additional material for Section 6.1.3

Lemma 103. We have

1. $(\boldsymbol{\nu}x)(\llbracket M \rrbracket_p \mid x \rhd y) \gtrsim \llbracket M\{y/x\} \rrbracket_p$
2. $(\boldsymbol{\nu}x)(\llbracket V \rrbracket^{\mathrm{v}}_z \mid x \rhd y) \gtrsim \llbracket V\{y/x\} \rrbracket^{\mathrm{v}}_z$
3. $(\boldsymbol{\nu}p)(\llbracket M \rrbracket_p \mid p \rhd q) \gtrsim \llbracket M \rrbracket_q$
4. $(\boldsymbol{\nu}y)(\llbracket V \rrbracket^{\mathrm{v}}_y \mid x \rhd y) \gtrsim \llbracket V \rrbracket^{\mathrm{v}}_x$

Proof. We reason by induction on the encoding of M or V to prove the four properties in conjunction. Indeed, there are dependencies between these properties which prevent us from treating them separately. This result is proved for the optimised encoding of the plain call-by-value λ-calculus [10] and the addition of the imperative constructs does not change the proof for those cases. So, we only present the new cases involving these imperative constructs, which only appear for the first and third clauses.

1. (a) When $M = !\ell$, $(\boldsymbol{\nu}x)(\llbracket M \rrbracket_p \mid x \rhd y) \sim \llbracket M \rrbracket_p = \llbracket M\{y/x\} \rrbracket_p$

   (b) When $M = \ell \coloneqq M_1; M_2$ and M_1 is not a value,

$$
\begin{aligned}
(\boldsymbol{\nu}x)(\llbracket M \rrbracket_p \mid x \rhd y) &\sim (\boldsymbol{\nu}q)((\boldsymbol{\nu}x)(\llbracket M_1 \rrbracket_q \mid x \rhd y) \mid q(w).\, \ell(_).\, (\overline{\ell}(z)\ z \rhd w \mid (\boldsymbol{\nu}x)(\llbracket M_2 \rrbracket_p \mid x \rhd y))) \\
&\gtrsim (\boldsymbol{\nu}q)(\llbracket M_1\{y/x\} \rrbracket_q \mid q(w).\, \ell(_).\, (\overline{\ell}(z)\ z \rhd w \mid \llbracket M_2\{y/x\} \rrbracket_p)) \quad \text{by induction} \\
&\gtrsim \llbracket M\{y/x\} \rrbracket_p
\end{aligned}
$$

   (c) When $M = \ell \coloneqq V; M_2$,

$$
\begin{aligned}
(\boldsymbol{\nu}x)(\llbracket M \rrbracket_p \mid x \rhd y) &\sim (\boldsymbol{\nu}w)((\boldsymbol{\nu}x)(\llbracket V \rrbracket^{\mathrm{v}}_w \mid x \rhd y) \mid \ell(_).\, (\overline{\ell}(z)\ z \rhd w \mid (\boldsymbol{\nu}x)(\llbracket M_2 \rrbracket_p \mid x \rhd y))) \\
&\gtrsim (\boldsymbol{\nu}w)(\llbracket V\{y/x\} \rrbracket^{\mathrm{v}}_w \mid \ell(_).\, (\overline{\ell}(z)\ z \rhd w \mid \llbracket M_2\{y/x\} \rrbracket_p)) \quad \text{by induction} \\
&\gtrsim \llbracket M\{y/x\} \rrbracket_p
\end{aligned}
$$

   (d) When $M = \mathtt{new}\ \ell \coloneqq V\ \mathtt{in}\ M'$,

$$
\begin{aligned}
(\boldsymbol{\nu}x)(\llbracket M \rrbracket_p \mid x \rhd y) &\sim (\boldsymbol{\nu}q)((\boldsymbol{\nu}x)(\llbracket V \rrbracket_q \mid x \rhd y) \mid q(w).\, (\overline{\ell}(z)\ z \rhd w \mid (\boldsymbol{\nu}x)(\llbracket M' \rrbracket_p \mid x \rhd y))) \\
&\gtrsim (\boldsymbol{\nu}q)(\llbracket V\{y/x\} \rrbracket_q \mid q(w).\, (\overline{\ell}(z)\ z \rhd w \mid \llbracket M'\{y/x\} \rrbracket_p)) \quad \text{by induction} \\
&\gtrsim \llbracket M\{y/x\} \rrbracket_p
\end{aligned}
$$

3 (a) When $M = !\ell$,

$$\begin{aligned}(\boldsymbol{\nu} p)([\![M]\!]_p \mid p \rhd q) &\sim (\boldsymbol{\nu} p)(\ell(w).\,(\overline{\ell}(y)\ y \rhd w \mid \overline{p}(z)\ z \rhd w \mid p \rhd q)) \\ &\gtrsim (\boldsymbol{\nu} p)(\ell(w).\,(\overline{\ell}(y)\ y \rhd w \mid \overline{q}(z)\ z \rhd w) \\ &= [\![M]\!]_q\end{aligned}$$

(b) When $M = \ell \coloneqq M_1; M_2$ and M_1 is not a value,

$$\begin{aligned}(\boldsymbol{\nu} p)([\![M]\!]_p \mid p \rhd q) &\sim (\boldsymbol{\nu} r)([\![M_1]\!]_r \mid r(w).\,\ell(_).\,(\overline{\ell}(z)\ z \rhd w \mid (\boldsymbol{\nu} p)([\![M_2]\!]_p \mid p \rhd q))) \\ &\gtrsim (\boldsymbol{\nu} r)([\![M_1]\!]_r \mid r(w).\,\ell(_).\,(\overline{\ell}(z)\ z \rhd w \mid [\![M_2]\!]_q)) \quad \text{by induction} \\ &= [\![M]\!]_q\end{aligned}$$

(c) When $M = \ell \coloneqq V; M_2$,

$$\begin{aligned}(\boldsymbol{\nu} p)([\![M]\!]_p \mid p \rhd q) &\sim (\boldsymbol{\nu} w)([\![V]\!]^{\mathrm{v}}_w \mid \ell(_).\,(\overline{\ell}(z)\ z \rhd w \mid (\boldsymbol{\nu} p)([\![M_2]\!]_p \mid p \rhd q))) \\ &\gtrsim (\boldsymbol{\nu} w)([\![V]\!]^{\mathrm{v}}_w \mid \ell(_).\,(\overline{\ell}(z)\ z \rhd w \mid [\![M_2]\!]_q)) \quad \text{by induction} \\ &= [\![M]\!]_q\end{aligned}$$

(d) When $M = \mathtt{new}\ \ell \coloneqq V\ \mathtt{in}\ M'$,

$$\begin{aligned}(\boldsymbol{\nu} p)([\![M]\!]_p \mid p \rhd q) &\sim (\boldsymbol{\nu} r)([\![V]\!]_r \mid r(w).\,(\overline{\ell}(z)\ z \rhd w \mid (\boldsymbol{\nu} p)([\![M']\!]_p \mid p \rhd q))) \\ &\gtrsim (\boldsymbol{\nu} r)([\![V]\!]_r \mid r(w).\,(\overline{\ell}(z)\ z \rhd w \mid [\![M']\!]_q)) \quad \text{by induction} \\ &= [\![M]\!]_q\end{aligned}$$

□

Lemma 135 (Validity of β-reduction)**.** For any M, N with $\langle h \mid M\rangle \to \langle h \mid N\rangle$ (using rule (β)), then we have $[\![M]\!]_p \overset{\tau}{\to}\gtrsim [\![N]\!]_p$.

Proof. By Lemmas 102 and 104, if we prove $[\![(\lambda x.\,M)V]\!]_p \overset{\tau}{\to}\gtrsim [\![M\{^V/_x\}]\!]_p$, then the results follows for any (β) reduction by congruence of $\gtrsim$.

First, notice that $[\![(\lambda x.\,M)V]\!]_p \overset{\tau}{\to}\gtrsim (\boldsymbol{\nu} x)([\![M]\!]_p \mid [\![V]\!]^{\mathrm{v}}_x)$ by Lemma 103.

So we prove by induction on the encoding of M that $(\boldsymbol{\nu} x)([\![M]\!]_p \mid [\![V]\!]^{\mathrm{v}}_x) \gtrsim [\![M\{^V/_x\}]\!]_p$. By doing so, we will also prove that $(\boldsymbol{\nu} x)([\![W]\!]^{\mathrm{v}}_y \mid [\![V]\!]^{\mathrm{v}}_x) \gtrsim [\![W\{^V/_x\}]\!]^{\mathrm{v}}_y$.

We present the cases involving imperative constructs.

1. When $M = !\ell$, then $(\boldsymbol{\nu} x)([\![M]\!]_p \mid [\![V]\!]^{\mathrm{v}}_x) \sim [\![M]\!]_p = [\![M\{^V/_x\}]\!]_p$

2. When $M = \ell \coloneqq M_1; M_2$ and M_1 is not a value,

$$\begin{aligned}(\boldsymbol{\nu} x)([\![M]\!]_p \mid [\![V]\!]^{\mathrm{v}}_x) &\sim (\boldsymbol{\nu} q)((\boldsymbol{\nu} x)([\![M_1]\!]_q \mid [\![V]\!]^{\mathrm{v}}_x) \mid q(w).\,\ell(_).\,(\overline{\ell}(z)\ z \rhd w \mid (\boldsymbol{\nu} x)([\![M_2]\!]_p \mid [\![V]\!]^{\mathrm{v}}_x))) \\ &\gtrsim (\boldsymbol{\nu} q)([\![M_1\{^V/_x\}]\!]_q \mid q(w).\,\ell(_).\,(\overline{\ell}(z)\ z \rhd w \mid [\![M_2\{^V/_x\}]\!]_p)) \quad \text{by induction} \\ &\gtrsim [\![M\{^V/_x\}]\!]_p\end{aligned}$$

3. When $M = \ell \coloneqq W; M_2$,

$$\begin{aligned}(\boldsymbol{\nu} x)([\![M]\!]_p \mid [\![V]\!]^{\mathrm{v}}_x) &\sim (\boldsymbol{\nu} w)((\boldsymbol{\nu} x)([\![W]\!]^{\mathrm{v}}_w \mid [\![V]\!]^{\mathrm{v}}_x) \mid \ell(_).\,(\overline{\ell}(z)\ z \rhd w \mid (\boldsymbol{\nu} x)([\![M_2]\!]_p \mid [\![V]\!]^{\mathrm{v}}_x))) \\ &\gtrsim (\boldsymbol{\nu} w)([\![W\{^V/_x\}]\!]^{\mathrm{v}}_w \mid \ell(_).\,(\overline{\ell}(z)\ z \rhd w \mid [\![M_2\{^V/_x\}]\!]_p)) \quad \text{by induction} \\ &\gtrsim [\![M\{^V/_x\}]\!]_p\end{aligned}$$

4. When $M = \mathtt{new}\ \ell \coloneqq W\ \mathtt{in}\ M'$,

$$\begin{aligned}(\boldsymbol{\nu} x)([\![M]\!]_p \mid [\![V]\!]^{\mathrm{v}}_x) &\sim (\boldsymbol{\nu} q)((\boldsymbol{\nu} x)([\![W]\!]_q \mid [\![V]\!]^{\mathrm{v}}_x) \mid q(w).\,(\overline{\ell}(z)\ z \rhd w \mid (\boldsymbol{\nu} x)([\![M']\!]_p \mid [\![V]\!]^{\mathrm{v}}_x))) \\ &\gtrsim (\boldsymbol{\nu} q)([\![W\{^V/_x\}]\!]_q \mid q(w).\,(\overline{\ell}(z)\ z \rhd w \mid [\![M'\{^V/_x\}]\!]_p)) \quad \text{by induction} \\ &\gtrsim [\![M\{^V/_x\}]\!]_p\end{aligned}$$

□

We now prove the validity of the other rules.

Lemma 136. For any h, h', M, N with $\langle h \mid M\rangle \to \langle h' \mid N\rangle$, using rules (Alloc), (Read) or (Write), we have $[\![\langle h \mid M\rangle]\!]_p \xrightarrow{\tau}\gtrsim [\![\langle h' \mid N\rangle]\!]_p$.

Proof.

- $[\![E[\texttt{new}\ \ell \coloneqq V\ \texttt{in}\ M]]\!]_p \xrightarrow{\tau}\gtrsim [\![\langle \ell = V \mid E[M]\rangle]\!]_p$
- We have that $h = h' \uplus \ell = V$, so using Lemma 102, we know that $[\![\langle h \mid E[!\ell]\rangle]\!]_p \equiv (\boldsymbol{\nu}\widetilde{\ell_0})([\![h']\!] \mid \overline{\ell}(y)\ [\![V]\!]^{\mathrm{v}}_y \mid [\![E]\!][\![!\ell]\!]_p)$. Thus, $[\![\langle h \mid E[!\ell]\rangle]\!]_p \xrightarrow{\tau}\gtrsim (\boldsymbol{\nu}\widetilde{\ell_0})([\![h']\!] \mid \overline{\ell}(y)\ [\![V]\!]^{\mathrm{v}}_y \mid [\![E]\!][\![V]\!]_p)$.

  By Lemma 104, we obtain $[\![\langle h \mid E[!\ell]\rangle]\!]_p \xrightarrow{\tau}\gtrsim [\![\langle h \mid E[V]\rangle]\!]_p$
- We have that $h = h' \uplus \ell = V$, so using Lemma 102, we know that $[\![\langle h \mid E[\ell \coloneqq W; M]\rangle]\!]_p \equiv (\boldsymbol{\nu}\widetilde{\ell_0})([\![h']\!] \mid \overline{\ell}(y)\ [\![V]\!]^{\mathrm{v}}_y \mid [\![E]\!][\![\ell = W; M]\!]_p)$.
  Thus, $[\![\langle h \mid E[\ell \coloneqq W; M]\rangle]\!]_p \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{\ell_0})([\![h']\!] \mid \overline{\ell}(y)\ [\![W]\!]^{\mathrm{v}}_y \mid [\![E]\!][\![M]\!]_p)$.

  By Lemma 104, we obtain $[\![\langle h \mid E[\ell \coloneqq W; M]\rangle]\!]_p \xrightarrow{\tau}\gtrsim [\![\langle h[\ell \coloneqq W] \mid E[M]\rangle]\!]_p$ □

Corollary 137. For any configurations c, d, if $c \to d$, then $[\![c]\!]_p \xrightarrow{\tau}\gtrsim [\![d]\!]_p$.

Corollary 137 is used in case 3 of Proposition 105, the following lemma is needed for case 2.

Lemma 138. We have $[\![E_0[y\ V_0]]\!]_{q_0} \gtrsim \overline{y}(x_0, p_0)\ ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, [\![E_0[z]]\!]_{q_0})$.

Proof. We reason by induction on E_0:

1. When $E_0 = [\cdot]$, $[\![y\ V_0]\!]_{q_0} \equiv \overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, [\![z]\!]_{q_0})$
2. When $E_0 = F\ N$, we write Q_{q_0} for $(\boldsymbol{\nu}r)([\![N]\!]_r \mid r(w).\, \overline{z'}(w', r').\, (w' \rhd w \mid r' \rhd q_0))$.

$$\begin{aligned}
[\![F[y\ V_0]\ N]\!]_{q_0} &= (\boldsymbol{\nu}q)([\![F[y\ V_0]]\!]_q \mid q(z').\, Q_{q_0}) \\
&\gtrsim (\boldsymbol{\nu}q)(\overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, [\![F[z]]\!]_q) \mid q(z').\, Q_{q_0}) && \text{by induction} \\
&\sim \overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, (\boldsymbol{\nu}q)([\![F[z]]\!]_q \mid q(z').\, Q_{q_0})) \\
&= \overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, [\![[\cdot]\ N]\!][\![F[z]]\!]_{q_0}) \\
&\gtrsim \overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, [\![E[z]]\!]_{q_0}) && \text{by Lemma 104}
\end{aligned}$$

3. When $E_0 = W\ F$, we write R_{q_0} for $\overline{z'}(w', r').\, (w' \rhd w \mid r' \rhd q_0)$.

$$\begin{aligned}
[\![W\ F[y\ V_0]]\!]_{q_0} &= (\boldsymbol{\nu}z')([\![W]\!]^{\mathrm{v}}_{z'} \mid (\boldsymbol{\nu}r)([\![F[y\ V_0]]\!]_r \mid r(w').\, R_{q_0})) \\
&\gtrsim (\boldsymbol{\nu}z')([\![W]\!]^{\mathrm{v}}_{z'} \mid (\boldsymbol{\nu}r)(\overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, [\![F[z]]\!]_r) \mid r(w').\, R_{q_0})) && \text{by induction} \\
&\sim \overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, (\boldsymbol{\nu}z')([\![W]\!]^{\mathrm{v}}_{z'} \mid (\boldsymbol{\nu}r)([\![F[z]]\!]_r \mid r(w').\, R_{q_0}))) \\
&= \overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, [\![W\ [\cdot]]\!][\![F[z]]\!]_{q_0}) \\
&\gtrsim \overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, [\![E[z]]\!]_{q_0}) && \text{by Lemma 104}
\end{aligned}$$

4. When $E_0 = \ell \coloneqq F; N$, we write Q_{q_0} for $\ell(_).\, (\overline{\ell}(y')\ y' \rhd w \mid [\![N]\!]_{q_0})$.

$$\begin{aligned}
[\![\ell \coloneqq F[y\ V_0]; N]\!]_{q_0} &= (\boldsymbol{\nu}q)([\![F[y\ V_0]]\!]_q \mid q(w).\, Q_{q_0}) \\
&\gtrsim (\boldsymbol{\nu}q)(\overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, [\![F[z]]\!]_q) \mid q(w).\, Q_{q_0}) && \text{by induction} \\
&\sim \overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, (\boldsymbol{\nu}q)([\![F[z]]\!]_q \mid q(w).\, Q_{q_0})) \\
&= \overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, [\![\ell \coloneqq [\cdot]; N]\!][\![F[z]]\!]_{q_0}) \\
&\gtrsim \overline{y}(x_0, p_0).\, ([\![V_0]\!]^{\mathrm{v}}_{x_0} \mid p_0(z).\, [\![E[z]]\!]_{q_0}) && \text{by Lemma 104}
\end{aligned}$$

□

The expansion is only needed because of the optimisation of $[\![E_0[z]]\!]_{q_0}$. Thus, $[\![E_0[y\ V_0]]\!]_{q_0} \xrightarrow{\overline{y}(x_0,p_0)} P'$ is the only transition.

Proposition 105 (Untyped Operational Correspondence)**.** For any M, h with $\mathrm{dom}(h) = \widetilde{\ell}$ and fresh q_0, $[\![\langle h \mid M\rangle]\!]_{q_0}$ has exactly one immediate transition, and exactly one of the following clauses holds:

1. $\langle h \mid M\rangle \to \langle h' \mid N\rangle$ and $[\![\langle h \mid M\rangle]\!]_{q_0} \xrightarrow{\tau} P$ with $P \gtrsim [\![\langle h' \mid N\rangle]\!]_{q_0}$
2. M is a value, $[\![\langle h \mid M\rangle]\!]_{q_0} \xrightarrow{\overline{q_0}(x_0)} P$ and $P = (\boldsymbol{\nu}\widetilde{\ell})([\![h]\!] \mid [\![M]\!]^{\mathtt{v}}_{x_0})$.
3. M is of the form $E_0[yV_0]$ for some E_0, y and V_0, and we have $[\![\langle h \mid M\rangle]\!]_{q_0} \xrightarrow{\overline{y}(x_0,p_0)} P$ with $P \gtrsim (\boldsymbol{\nu}\widetilde{\ell})([\![h]\!] \mid [\![V_0]\!]^{\mathtt{v}}_{x_0} \mid p_0(z).\,[\![E_0[z]]\!]_{q_0})$.

Proof. By case analysis from Lemma 99:

- If M is a value, then there is no τ transition. So the only possible transition is the output on q_0.
- If $M = E[y\ V_0]$, we can use Lemma 138.
- Otherwise there exists h', N with $\langle h \mid M\rangle \to \langle h' \mid N\rangle$ and we can use Corollary 137. □

B.3 Additional material for Section 6.2.2

Remark 139. We shall use the following properties implicitly in the proofs below:

1. $[\![(\emptyset, \odot, c)]\!]_{\emptyset;q_0} \equiv [\![c]\!]_{q_0}$
2. $[\![(\widetilde{V_i}, \sigma, h)]\!]_{\widetilde{x};\widetilde{pq}} \not\xrightarrow{\tau}$

Lemma 140. We write $\rho_{\widetilde{pq}} \stackrel{\text{def}}{=} p_1 : \mathtt{i}, q_1 : \mathtt{o}, \ldots, p_n : \mathtt{i}, q_n : \mathtt{o}$ and $\rho_{q_0,\widetilde{pq}} \stackrel{\text{def}}{=} q_0 : \mathtt{o}, p_1 : \mathtt{i}, q_1 : \mathtt{o}, \ldots, p_n : \mathtt{i}, q_n : \mathtt{o}$, both clean. Then $\rho_{\widetilde{pq}} \vdash_{\mathrm{wb}} [\![(\widetilde{V_i}, \sigma, h)]\!]_{\widetilde{x};\widetilde{pq}}$ and $\rho_{q_0,\widetilde{pq}} \vdash_{\mathrm{wb}} [\![(\widetilde{V_i}, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}$.

Lemma 141 (Validity of rules w.r.t triples)**.** For any $\widetilde{V_i}, \sigma, c, d$ with $c \to d$, then $[\![(\widetilde{V_i}, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}} \xrightarrow{\tau}\gtrsim [\![(\widetilde{V_i}, \sigma, d)]\!]_{\widetilde{x};q_0,\widetilde{pq}}$.

Proof. For rule (β), as, in Lemma 135 the term are related by $\xrightarrow{\tau}\gtrsim$ (and not only the configuration) the result follows by congruence of $\gtrsim$

Otherwise, the proof is similar to Lemma 136. □

Theorem 113 (Operational Correspondence)**.**
We relate transitions for the encoding of both kind of triples:
When $[\rho; [\![(\widetilde{V_i}, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \xrightarrow{\mu} [\rho'; P]$ with $\widetilde{x} = \widetilde{x_i}$ and $\widetilde{pq} = p_1, q_1, \ldots, p_n, q_n$ then:

1. either $c \to c'$, $\mu = \tau$ and $P \gtrsim [\![(\widetilde{V_i}, \sigma, c')]\!]_{\widetilde{x};q_0,\widetilde{pq}}$
2. or $c = \langle h \mid V_0\rangle$, $\mu = \overline{q_0}(x_0)$ and $P \gtrsim [\![((\widetilde{V_i}, V_0), \sigma, h)]\!]_{x_0,\widetilde{x};\widetilde{pq}}$
3. or $c = \langle h \mid E_0[y\ V_0]\rangle$, $\mu = \overline{y}(x_0, p_0)$ and $P \gtrsim [\![((\widetilde{V_i}, V_0), E_0 :: \sigma, h)]\!]_{x_0,\widetilde{x};p_0,q_0,\widetilde{pq}}$

When $[\rho; [\![(\widetilde{V_i}, \sigma, h)]\!]_{\widetilde{x};\widetilde{pq}}] \xrightarrow{\mu} [\rho'; P]$ with $\widetilde{x} = \widetilde{x_i}$ and $\widetilde{pq} = p_1, q_1, \ldots, p_n, q_n$ then:

1. either $\mu = x_j(z, q_0)$ and $P \gtrsim [\![(\widetilde{V_i}, \sigma, \langle h \mid N\rangle)]\!]_{\widetilde{x};q_0,\widetilde{pq}}$ with $V_j z \succ N$.
2. or $\sigma = E_1 :: \sigma'$ and $\mu = p_1(z)$ and $P \gtrsim [\![(\widetilde{V_i}, \sigma', \langle h \mid E_1[z]\rangle)]\!]_{\widetilde{x};q_1,p_2,q_2,\ldots,p_n,q_n}$

Proof. For active processes, no input at names in `F` or `C` are allowed, names in `R` are bound so the only possible transitions are the 3 given in the theorem.

1. This is Lemma 141.

2. We use Lemma 138, and conclude by congruence of $\gtrsim$.

3. We use the same reasoning as in Proposition 105.

For inactive processes:

1. When V_j is a variable, then $N = V_j\ z$ and $P \equiv [\![(\widetilde{V}_i, \sigma, \langle h \mid N\rangle)]\!]_{\widetilde{x};q_0,\widetilde{pq}}$. Otherwise, $V_j = \lambda z.\, N$ and $[\![V_j]\!]^{\mathtt{v}}_{x_j} \xrightarrow{x_j(z,q_0)} [\![V_j]\!]^{\mathtt{v}}_{x_j} \mid [\![N]\!]_{q_0}$.

2. Immediate. □

Lemma 142. π_{div} is closed by $\approx_{\mathrm{wb}}$, meaning that if $P \approx^{\rho}_{\mathrm{wb}} Q$, then $(\rho, P) \in \pi_{\mathrm{div}}$ iff $(\rho, Q) \in \pi_{\mathrm{div}}$.

Lemma 143. The set S defined by

$$S \stackrel{\mathrm{def}}{=} \{(\widetilde{V}_i, \sigma, c) \mid (\rho_{q_0,\widetilde{pq}}, [\![(\widetilde{V}_i, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}) \in \pi_{\mathrm{div}}\} \cup \{(\widetilde{V}_i, \sigma, h) \mid (\rho_{\widetilde{pq}}, [\![(\widetilde{V}_i, \sigma, h)]\!]_{\widetilde{x};\widetilde{pq}}) \in \pi_{\mathrm{div}}\}$$

is diverging.

Proof.

1. $(\widetilde{V}_i, \sigma, c) \in S$ implies

   - if $c \to c'$, then $[\![(\widetilde{V}_i, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}} \xrightarrow{\tau} P$ with $(\rho_{q_0,\widetilde{pq}}, P) \in \pi_{\mathrm{div}}$. By Theorem 113, $P \gtrsim [\![(\widetilde{V}_i, \sigma, c')]\!]_{\widetilde{x};q_0,\widetilde{pq}}$. Thus $(\rho_{q_0,\widetilde{pq}}, [\![(\widetilde{V}_i, \sigma, c')]\!]_{\widetilde{x};q_0,\widetilde{pq}}) \in \pi_{\mathrm{div}}$ so $(\widetilde{V}_i, \sigma, c') \in S$
   - if $c = \langle h \mid V_0\rangle$, then $[\![(\widetilde{V}_i, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}} \xrightarrow{\overline{q_0}(x_0)} P$ with $(\rho_{\widetilde{pq}}, P) \in \pi_{\mathrm{div}}$. Thus, $\rho_{\widetilde{pq}} \vdash_{\mathrm{wb}} P$ with $\rho_{\widetilde{pq}} \neq \emptyset$, meaning $\sigma \neq \odot$. By Theorem 113, $P \gtrsim [\![((\widetilde{V}_i, V_0), \sigma, h)]\!]_{x_0,\widetilde{x};\widetilde{pq}}$. Thus $(\rho_{\widetilde{pq}}, [\![((\widetilde{V}_i, V_0), \sigma, h)]\!]_{x_0,\widetilde{x};\widetilde{pq}}) \in \pi_{\mathrm{div}}$ and $((\widetilde{V}_i, V_0), \sigma, h) \in S$
   - if $c = \langle h \mid E_0[yV_0]\rangle$, then $[\![(\widetilde{V}_i, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}} \xrightarrow{\overline{y}(x_0,p_0)} P$ with $(\rho_{p_0,q_0,\widetilde{pq}}, P) \in \pi_{\mathrm{div}}$. Thus, $\rho_{p_0,q_0,\widetilde{pq}} \vdash_{\mathrm{wb}} P$. By Theorem 113, $P \gtrsim [\![((\widetilde{V}_i, V_0), E_0 :: \sigma, h)]\!]_{x_0,\widetilde{x};p_0,q_0,\widetilde{pq}}$. Thus $(\rho_{p_0,q_0,\widetilde{pq}}, [\![((\widetilde{V}_i, V_0), E_0 :: \sigma, h)]\!]_{x_0,\widetilde{x};p_0,q_0,\widetilde{pq}}) \in \pi_{\mathrm{div}}$ so $((\widetilde{V}_i, V_0), E_0 :: \sigma, h) \in S$ for a fresh z.

2. $(\widetilde{V}_i, \sigma, h) \in S$ implies

   - if $V_j \in \widetilde{V}_i$, then $[\![(\widetilde{V}_i, \sigma, h)]\!]_{\widetilde{x};\widetilde{pq}} \xrightarrow{x_j(z,q_0)} P$ with $(\rho_{q_0,\widetilde{pq}}, P) \in \pi_{\mathrm{div}}$. By Theorem 113, $P \gtrsim [\![(\widetilde{V}_i, \sigma, \langle h \mid N\rangle)]\!]_{\widetilde{x};q_0,\widetilde{pq}}$ with $V_j z \succ M$. Thus we have $(\rho_{q_0,\widetilde{pq}}, [\![(\widetilde{V}_i, \sigma, \langle h \mid M\rangle)]\!]_{\widetilde{x};q_0,\widetilde{pq}}) \in \pi_{\mathrm{div}}$, i.e. $(\widetilde{V}_i, \sigma, \langle h \mid M\rangle) \in S$
   - if $\sigma = E_1 :: \sigma'$, then $[\![(\widetilde{V}_i, E_1 :: \sigma', h)]\!]_{\widetilde{x};\widetilde{pq}} \xrightarrow{p_1(z)} P$ with $(\rho_{q_1,\widetilde{p'q'}}, P) \in \pi_{\mathrm{div}}$, where $\widetilde{pq} = p_1, q_1, \widetilde{p'q'}$. By Theorem 113, $P \gtrsim [\![(\widetilde{V}_i, \sigma', \langle h \mid E_1[z]\rangle)]\!]_{\widetilde{x};q_1,\widetilde{p'q'}}$. Thus, $(\rho_{q_1,\widetilde{p'q'}}, [\![(\widetilde{V}_i, \sigma, \langle h \mid V_j z\rangle)]\!]_{\widetilde{x};q_1,\widetilde{p'q'}}) \in \pi_{\mathrm{div}}$ so $(\widetilde{V}_i, \sigma', \langle h \mid E_1[z]\rangle) \in S$ for a fresh z. □

Corollary 144. If $[\![(\widetilde{V}_i, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}$ (resp. $[\![(\widetilde{V}_i, \sigma, h)]\!]_{\widetilde{x};\widetilde{pq}}$) is in π_{div}, then $(\widetilde{V}_i, \sigma, c)$ (resp. $(\widetilde{V}_i, \sigma, h)$) is in λ_{div}.

Theorem 119 (Soundness)**.** If $[\![M]\!]_p \approx^{p:\mathtt{o}}_{\mathrm{div}} [\![N]\!]_p$, then $M \asymp N$.

Proof. The relation $\mathcal{R}$ defined by

$$\begin{aligned}\mathcal{R} \stackrel{\mathrm{def}}{=} \{((\widetilde{V}_i, \sigma_1, c), (\widetilde{W}_i, \sigma_2, d)) \mid [\![(\widetilde{V}_i, \sigma_1, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}} \approx^{\rho_{q_0,\widetilde{pq}}}_{\mathrm{div}} [\![(\widetilde{V}_i, \sigma_1, d)]\!]_{\widetilde{x};q_0,\widetilde{pq}}\}\\ \cup \{((\widetilde{V}_i, \sigma_1, h), (\widetilde{W}_i, \sigma_2, g)) \mid [\![(\widetilde{V}_i, \sigma_1, h)]\!]_{\widetilde{x};\widetilde{pq}} \approx^{\rho_{\widetilde{pq}}}_{\mathrm{div}} [\![(\widetilde{V}_i, \sigma_1, g)]\!]_{\widetilde{x};\widetilde{pq}}\}\end{aligned}$$

is a bisimulation.

1. Take $(\widetilde{V}_i, \sigma_1, c)\ \mathcal{R}\ (\widetilde{W}_i, \sigma_2, d)$

- if $c \to c'$, then by Theorem 113, $[\rho_{q_0,\widetilde{pq}}; [\![(\widetilde{V_i}, \sigma_1, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \xrightarrow{\tau} [\rho_{q_0,\widetilde{pq}}; P]$ with $P \gtrsim [\![(\widetilde{V_i}, \sigma_1, c')]\!]_{\widetilde{x};q_0,\widetilde{pq}}$
  Thus, $[\rho_{q_0,\widetilde{pq}}; [\![(\widetilde{W_i}, \sigma_2, d)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \xRightarrow{\tau} [\rho_{q_0,\widetilde{pq}}; Q]$ with $P \approx_{\text{div}}^{\rho_{q_0,\widetilde{pq}}} Q$.
  By Corollary 114, this means that $d \Rightarrow d'$ and $Q \gtrsim (\widetilde{W_i}, \sigma_2, d')$. As $\approx_{\text{div}}$ is transitive and $\gtrsim \subseteq \approx_{\text{div}}$, we have $(\widetilde{V_i}, \sigma_1, c')\ \mathcal{R}\ (\widetilde{W_i}, \sigma_2, d')$
- if $c = \langle h \mid V_0 \rangle$, then by Theorem 113, $[\rho_{q_0,\widetilde{pq}}; [\![(\widetilde{V_i}, \sigma_1, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \xrightarrow{\overline{q_0}(x_0)} [\rho_{\widetilde{pq}}; P]$ with $P \gtrsim [\![((\widetilde{V_i}, V_0), \sigma_1, h)]\!]_{x_0,\widetilde{x};\widetilde{pq}}$ so either
  - $[\rho_{q_0,\widetilde{pq}}; [\![(\widetilde{W_i}, \sigma_2, d)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \xRightarrow{\overline{q_0}(x_0)} [\rho_{\widetilde{pq}}; Q]$ with $P \approx_{\text{div}}^{\rho_{\widetilde{pq}}} Q$.
    By Theorem 113, Corollary 114 and Remark 139, this means that $d \Rightarrow \langle g \mid W_0 \rangle$ and $Q \gtrsim [\![((\widetilde{W_i}, W_0), \sigma_2, g)]\!]_{x_0,\widetilde{x};\widetilde{pq}}$. As $\approx_{\text{div}}$ is transitive and $\gtrsim \subseteq \approx_{\text{div}}$, we have $((\widetilde{V_i}, V_0), \sigma_1, h)\ \mathcal{R}\ ((\widetilde{W_i}, W_0), \sigma_2, g)$, or
  - $(\rho_{\widetilde{pq}}, P) \in \pi_{\text{div}}$ Thus, $\widetilde{pq} \neq \emptyset$ meaning that $\sigma_1 \neq \odot$. Moreover, by Lemma 142 and Corollary 144, $((\widetilde{V_i}, V_0), \sigma_1, h) \in \lambda_{\text{div}}$
- if $c = \langle h \mid E_0[yV_0] \rangle$, then by Theorem 113, $[\rho_{q_0,\widetilde{pq}}; [\![(\widetilde{V_i}, \sigma_1, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \xrightarrow{\overline{y}(x_0,p_0)} [\rho_{p_0,q_0,\widetilde{pq}}; P]$ with $P \gtrsim [\![((\widetilde{V_i}, V_0), E_0 :: \sigma_1, h)]\!]_{x_0,\widetilde{x};p_0,q_0,\widetilde{pq}}$
  so either
  - $[\rho_{q_0,\widetilde{pq}}; [\![(\widetilde{W_i}, \sigma_2, d)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \xRightarrow{\overline{y}(x_0,p_0)} [\rho_{p_0,q_0,\widetilde{pq}}; Q]$ with $P \approx_{\text{div}}^{\rho_{p_0,q_0,\widetilde{pq}}} Q$.
    By Theorem 113, Corollary 114 and Remark 139, this means that $d \Rightarrow \langle g \mid F_0[yW_0] \rangle$ and $Q \gtrsim [\![((\widetilde{W_i}, W_0), F_0 :: \sigma_2, g)]\!]_{x_0,\widetilde{x};p_0,q_0,\widetilde{pq}}$. As $\approx_{\text{div}}$ is transitive and $\gtrsim \subseteq \approx_{\text{div}}$, we have $((\widetilde{V_i}, V_0), E_0 :: \sigma_1, h)\ \mathcal{R}\ ((\widetilde{W_i}, W_0), F_0 :: \sigma_2, g)$, or
  - $(\rho_{p_0,q_0,\widetilde{pq}}, P) \in \pi_{\text{div}}$ We also have that $E_0 :: \sigma_1 \neq \odot$. Moreover, by Lemma 142 and Corollary 144,
    so $((\widetilde{V_i}, V_0), E_0 :: \sigma_1, h) \in \lambda_{\text{div}}$.

2. Take $(\widetilde{V_i}, \sigma_1, h)\ \mathcal{R}\ (\widetilde{W_i}, \sigma_2, g)$

   - if $(V_j, W_j) \in (\widetilde{V_i}, \widetilde{W_i})$, then by Theorem 113, $[\rho_{\widetilde{pq}}; [\![(\widetilde{V_i}, \sigma_1, h)]\!]_{\widetilde{x};\widetilde{pq}}] \xrightarrow{x_j(z,q_0)} [\rho_{q_0,\widetilde{pq}}; P]$ with $P \gtrsim [\![(\widetilde{V_i}, \sigma_1, \langle h \mid M \rangle)]\!]_{\widetilde{x};q_0,\widetilde{pq}}$ for a fresh z and $V_j z \succ M$.
     Thus, $[\rho_{\widetilde{pq}}; [\![(\widetilde{W_i}, \sigma_2, g)]\!]_{\widetilde{x};\widetilde{pq}}] \xRightarrow{x_j(z,q_0)} [\rho_{q_0,\widetilde{pq}}; Q]$ with $P \approx_{\text{div}}^{\rho_{q_0,\widetilde{pq}}} Q$.
     By Remark 139, Theorem 113, Corollary 114 and Lemma 141, this means that $Q \approx_{\text{div}}^{\rho_{q_0,\widetilde{pq}}} [\![\widetilde{W_i}, \sigma_2, \langle g \mid N \rangle]\!]_{\widetilde{x};q_0,\widetilde{pq}}$ with $W_j z \succ N$. We then obtain that $(\widetilde{V_i}, \sigma_1, \langle h \mid M \rangle)\ \mathcal{R}\ (\widetilde{W_i}, \sigma_2, \langle g \mid N \rangle)$.
   - if $\sigma_1 = E_1 :: \sigma_1'$ and $\sigma_2 = F_1 :: \sigma_2'$, then then by Theorem 113, $[\rho_{\widetilde{pq}}; [\![(\widetilde{V_i}, \sigma_1, h)]\!]_{\widetilde{x};\widetilde{pq}}] \xrightarrow{p_1(z)} [\rho_{q_1,\widetilde{p'q'}}; P]$ with $\widetilde{pq} = p_1, q_1, \widetilde{p', q'}$ and $P \gtrsim [\![(\widetilde{V_i}, \sigma_1', \langle h \mid E_1[z] \rangle)]\!]_{\widetilde{x};q_1,\widetilde{p'q'}}$ for a fresh z.
     Thus, $[\rho_{\widetilde{pq}}; [\![(\widetilde{W_i}, \sigma_2, g)]\!]_{\widetilde{x};\widetilde{pq}}] \xRightarrow{p_1(z)} [\rho_{q_1,\widetilde{p'q'}}; Q]$ with $P \approx_{\text{div}}^{\rho_{q_1,\widetilde{p'q'}}} Q$.
     By Remark 139, Theorem 113, Corollary 114 and Lemma 141, this means that $Q \approx_{\text{div}}^{\rho_{q_1,\widetilde{p'q'}}} [\![\widetilde{W_i}, \sigma_2, \langle g \mid F_1[z] \rangle]\!]_{\widetilde{x};q_1,\widetilde{p'q'}}$. As $\approx_{\text{div}}$ is transitive, we then have $(\widetilde{V_i}, \sigma_1, \langle h \mid E_1[z] \rangle)\ \mathcal{R}\ (\widetilde{W_i}, \sigma_2, \langle g \mid F_1[z] \rangle)$. □

We can now look at the completeness, first for divergent sets, then for nfb.

Lemma 145. The set S defined by

$$S \stackrel{\text{def}}{=} \{[\![(\widetilde{V_i}, \sigma, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}} \mid (\widetilde{V_i}, \sigma, c) \in \lambda_{\text{div}}\} \cup \{[\![(\widetilde{V_i}, \sigma, h)]\!]_{\widetilde{x};\widetilde{pq}} \mid (\widetilde{V_i}, \sigma, h) \in \lambda_{\text{div}}\}$$

is a π-divergent set up to $\gtrsim$.

Proof. This is a direct consequence of Theorem 113. □

Lemma 146. The relation $\mathcal{R}$ defined by

$$\mathcal{R} \stackrel{\text{def}}{=} \{(\rho_{q_0,\widetilde{pq}}, [\![(\widetilde{V}_i, \sigma_1, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}, [\![(\widetilde{W}_i, \sigma_2, d)]\!]_{\widetilde{x};q_0,\widetilde{pq}}) \mid (\widetilde{V}_i, \sigma_1, c) \approx_\lambda (\widetilde{W}_i, \sigma_2, d)\}$$
$$\cup \{(\rho_{\widetilde{pq}}, [\![(\widetilde{V}_i, \sigma_1, h)]\!]_{\widetilde{x};\widetilde{pq}}, [\![(\widetilde{W}_i, \sigma_2, g)]\!]_{\widetilde{x};\widetilde{pq}}) \mid (\widetilde{V}_i, \sigma_1, h) \approx_\lambda (\widetilde{W}_i, \sigma_2, g)\}$$

is a bisimulation with divergence up to $\gtrsim$.

Proof. Take $(\rho_{q_0,\widetilde{pq}}, [\![(\widetilde{V}_i, \sigma_1, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}, [\![(\widetilde{W}_i, \sigma_2, d)]\!]_{\widetilde{x};q_0,\widetilde{pq}}) \in \mathcal{R}$.
By Theorem 113, we have that

1. either $c \to c'$ and $[\rho_{q_0,\widetilde{pq}}; [\![(\widetilde{V}_i, \sigma_1, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \xrightarrow{\tau} [\rho_{q_0,\widetilde{pq}}; P]$ with $P \gtrsim [\![(\widetilde{V}_i, \sigma_1, c')]\!]_{\widetilde{x};q_0,\widetilde{pq}}$. By assumption, there exists d' such that $d \Rightarrow d'$ and $(\widetilde{V}_i, \sigma_1, c') \approx_\lambda (\widetilde{W}_i, \sigma_2, d')$. So $[\rho_{q_0,\widetilde{pq}}; [\![(\widetilde{W}_i, \sigma_2, d)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \stackrel{\tau}{\Rightarrow} [\rho_{q_0,\widetilde{pq}}; Q]$ with $Q \gtrsim [\![(\widetilde{W}_i, \sigma_2, d')]\!]_{\widetilde{x};q_0,\widetilde{pq}}$ and we can conclude.

2. or $c = \langle h \mid V_0 \rangle$ and $[\rho_{q_0,\widetilde{pq}}; [\![(\widetilde{V}_i, \sigma_1, c)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \xrightarrow{\overline{q_0}(x_0)} [\rho_{\widetilde{pq}}; P]$ with $P \gtrsim [\![((\widetilde{V}_i, V_0), \sigma_1, h)]\!]_{\widetilde{x};\widetilde{pq}}$. By assumption, we have two cases:

   (a) there exists g, W_0 such that $d \Rightarrow \langle g \mid W_0 \rangle$ and $((\widetilde{V}_i, V_0), \sigma_1, h) \approx_\lambda ((\widetilde{W}_i, W_0), \sigma_2, g)$. So $[\rho_{q_0,\widetilde{pq}}; [\![(\widetilde{W}_i, \sigma_2, d)]\!]_{\widetilde{x};q_0,\widetilde{pq}}] \xRightarrow{\overline{q_0}(x_0)} [\rho_{\widetilde{pq}}; Q]$ with $Q \gtrsim [\![((\widetilde{W}_i, W_0), \sigma_2, g)]\!]_{\widetilde{x};\widetilde{pq}}$ and we can conclude.

   (b) or $\sigma_1 \neq \emptyset$ and $((\widetilde{V}_i, V_0), \sigma_1, h) \in \lambda_{\text{div}}$. Then we conclude as $P \gtrsim [\![((\widetilde{V}_i, V_0), \sigma_1, h)]\!]_{\widetilde{x};\widetilde{pq}}$ and $(\rho_{\widetilde{pq}}, [\![((\widetilde{V}_i, V_0), \sigma_1, h)]\!]_{\widetilde{x};\widetilde{pq}}) \in \pi_{\text{div}}$.

3. The last case is treated similarly.

The case for $(\rho_{\widetilde{pq}}, [\![(\widetilde{V}_i, \sigma_1, h)]\!]_{\widetilde{x};\widetilde{pq}}, [\![(\widetilde{W}_i, \sigma_2, g)]\!]_{\widetilde{x};\widetilde{pq}}) \in \mathcal{R}$ is handle similarly. □

Theorem 120 (Completeness)**.** If $M \asymp N$, then $[\![M]\!]_p \approx^{p:\mathsf{o}}_{\text{div}} [\![N]\!]_p$.

Proof. As for all M, p, we have $[\![M]\!]_p \equiv [\![(\emptyset, \odot, \langle \emptyset \mid M \rangle)]\!]_p$, it is an immediate consequence of Lemma 146. □