Concurrent Pattern Unification

Thomas Given-Wilson

A dissertation submitted for the Degree of Doctor of Philosophy

in Computing Sciences

Faculty of Engineering and Information Technology

University of Technology, Sydney

Supervisor: Associate Professor Barry Jay

2012

Certificate of Authorship/Originality

I certify that the work in this dissertation has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that this dissertation has been written by me. Any help that I have received in my research work and the preparation of this dissertation has been acknowledged. In addition, I certify that all information sources and literature used are indicated within this dissertation.

Signature of Student

Acknowledgments

Producing a doctoral dissertation is an arduous path whose traversal has been eased greatly by many. To my friends, family, colleagues and associates who have assisted in various ways, I thank you. Unfortunately it is not feasible to acknowledge everyone individually, however there are some who deserve particular mention.

Barry Jay, my supervisor, mentor, greatest supporter and perhaps also my greatest antagonist. My special thanks for being encouraging and supportive, while always remaining unconvinced until both narrative and proof were provided.

I must also thank Daniele Gorla for coming far out of his way to assist in the most technical details and saving me many months of individual study with invaluable insights and guidance.

Also I wish to acknowledge several of my colleagues for sharing their time, thoughts and impressions with me. Particularly (in no special order), Jose Vergara, Sara Mehrabi, Sylvan Rudduck, Christopher Stanton, Julia Prior, Haydn Mearns and Debi Taylor.

A word of thanks to my proof-readers, Virginia Macleod, Debi Taylor and Julia Prior, any remaining grammatical or typographic errors are entirely my responsibility.

Finally, thanks to my assessors for their thorough and insightful feedback.

Contents

Abstract

Ever since Milner showed that Church's λ-calculus can be subsumed by π-calculus, process calculi have been expected to subsume sequential computation. However, Jay & Given-Wilson show that extensional sequential computation as represented by λ-calculus is subsumed by intensional sequential computation characterised by pattern-matching as in SF-calculus. Given-Wilson, Gorla & Jay present a concurrent pattern calculus (CPC) that adapts sequential pattern-matching to symmetric pattern-unification in a process calculus. This dissertation proves that CPC subsumes both intensionality sequential computation and extensional concurrent computation, respectively SF-calculus and π-calculus, to complete a computation square. A behavioural theory is developed for CPC that is then exploited to prove that CPC is more expressive than several representative sequential and concurrent calculi. As part of its greater expressive power, CPC provides a natural language to describe interactions involving information exchange. Augmenting the pattern-matching language **bondi** to implement CPC yields a Concurrent **bondi** that is able to support web services that exploit both sequential and concurrent intensionality.

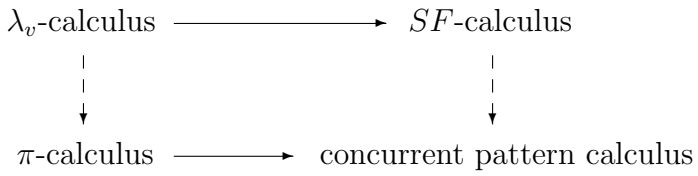
Chapter 1

Introduction

Computation can be characterised in two dimensions: *extensional* versus *intensional*; and *sequential* versus *concurrent*. Extensional sequential computation models are those whose *functions* cannot distinguish the internal structure of their *arguments*, here characterised by Church's λ-calculus [Chu36, CF58, Bar85]. Shifting along the first dimension from sequential to concurrent, ever since Milner et al. showed that their π-calculus generalises λ-calculus [Mil90, MPW92] (more precisely call-by-value or lazy λ-calculus), concurrency theorists expect process calculi to subsume sequential computation as represented by λ-calculus [Mil90, MPW92, CG98, Mil99]. Following from this, here extensional concurrent computation is characterised by process calculi that, at least, support (call-by-value) λ-calculus. In the second dimension shifting from extensional to intensional, Jay & Given-Wilson show that λ-calculus does not support all sequential computation [JGW11]. In particular, there are intensional Turing-computable [Tur36] functions, char-

acterised by *pattern-matching*, that can be represented within SF-calculus [JGW11] but not within λ-calculus [JGW11]. Of course λ-calculus can encode Turing computation, but this is a weaker claim. Further, Given-Wilson, Gorla & Jay present a *concurrent pattern calculus* (CPC) that not only generalises intensional pattern-matching from sequential computation to *pattern-unification* in a process calculus, but also increases the *symmetry* of interaction [GGJ10].

These four calculi form the corners of a *computation square*



where the left side is merely extensional and the right side also intensional; the top edge is sequential and the bottom edge concurrent. All the arrows preserve reduction. The horizontal arrows are *homomorphisms* in the sense that they also preserve *application* or *parallel composition*. The vertical arrows are *parallel encodings* in that they map application to a parallel composition (with some machinery). Thus each arrow represents increased expressive power with CPC completing the square.

As part of its greater expressive power, CPC provides a natural language to describe interactions involving information *exchange* [GGJ10]. Augmenting the pattern-matching language **bondi** [bon11] to implement CPC yields a Concurrent **bondi** [Con11] that is able to support web services that exploit

both sequential and concurrent intensionality [GWJ11].

The rest of this introduction considers the computation square and in doing so provides an overview of the dissertation's contents.

1.1 Extensional Sequential Computation

The top left corner of the square is concerned with extensional sequential computation, here characterised by Church's λ-calculus [Chu36, Bar85] or equivalently Schönfinkel's combinatory logic [Sch24, CF58]. Both models support the traditional concept of computation in that they are able to represent all the Turing-computable [Tur36] functions on natural numbers [Kle35]. Both have reduction rules based on the application of a function to one or more arguments. In particular, these models are extensional, that is a function does *not* have direct access to the internal structure of its arguments. Thus functions that are extensionally equal are indistinguishable within either λ-calculus or Schönfinkel's combinatory logic.

There are several variations of λ-calculus distinguished by their operational semantics. To exploit the results of Milner et al. in formalising the computation square requires choosing either call-by-value λ-calculus, denoted λ_v-calculus, or lazy λ-calculus, denoted λ_l-calculus [Bar85]. The choice here is to use λ_v-calculus, although all the results can be reproduced for λ_l-calculus as well.

There is a homomorphism from λ_v-calculus into any combinatory logic

that supports the combinators S and K [CF58, CHS72, Bar85, HS86]. Indeed, this is a central result of combinatory logic stating that any combinatory logic that represents S and K is *combinatorially complete*, that is, can represent λ-calculus [CF58]. Conversely there is a trivial homomorphism from Schönfinkel's combinatory logic to λ-calculus [CF58, CHS72, Bar85, HS86].

There are now two choices about which direction to take out of the top left corner of the square: across the top edge remains in sequential calculi; and down the left side remains in extensional calculi. This dissertation will follow extensionality down the left side, as it is more familiar.

1.2 Process Calculi

The bottom left corner of the square is concerned with extensional concurrent computation, here defined to be a process calculus that supports an encoding of some λ-calculus [Mil90]. Ever since Milner et al. showed this for π-calculus [Mil90, MPW92] process calculi have been expected to support sequential computation [Mil90, BB90, MPW92, CG98, Mil99, PV98]. Although this means many calculi can be the exemplar of extensional concurrent computation, the π-calculus is the obvious choice as it is: the first process calculus shown to generalise λ-calculus; has a widely recognised syntax and semantics; and is well related to many other process calculi [Mil90, GA97, CG98, PV98]. Formally the relation from λ_v-calculus to π-calculus is through a *parallel en-*

coding that preserves reduction and maps application to parallel composition with some additional machinery. In particular, parallel encodings are defined to support modular encoding of terms to processes and to exploit the potential for independent parallel reduction, thus taking advantage of concurrency.

This still leaves the problem of relating process calculi to each other and comparing their relative expressive power, a topic that has been the focus of much recent work [Nes06, Par08, Gor08a, Gor08b]. Relating process calculi is complicated by both the large variety of calculi, and the lack of any benchmark analogous to Turing-computability for sequential computation [Mil99, PV98]. One general approach to relating process calculi, taken by Gorla, is through *valid encodings* that preserve reduction (amongst other properties) [Gor08b, Gor08a]. These valid encodings provide a solid basis for relating process calculi and support the homomorphisms required for the horizontal arrow at the bottom of the computation square.

In his work on valid encodings Gorla presents a hierarchy of sets of process calculi related by their relative expressive power [Gor08b, Gor08a]. The π-calculus is classified in this hierarchy, however it is not within one of four equally-expressive sets that sit at the top of the hierarchy. As this dissertation is exploring expressive power, it is natural to address one of these four equally expressive sets of calculi. Gelernter's Linda [Gel85, YFY96, PMR99] is the most interesting choice to represent these sets as it has several differences to π-calculus and is also implemented as a programming language, as is CPC.

Considering expressive power, there are some other representative process

calculi that exploit intensionality and symmetry.

The Spi calculus of Gordon et al. [GA97] has a rich class of terms that introduce structure and intensional reductions upon terms. In particular terms can be tested for equality or structure, albeit only within a process and not as part of communication. This step towards intensional concurrent computation is of interest conceptually and in later formalisation of expressive power.

Another concept in concurrency is the shift away from the sequential asymmetry of function and argument to a *symmetric* parallel composition where both processes can evolve equally. This was supported by Milner who considered reductions in asynchronous calculi to be a symmetric interaction between a process and the environment [Mil99]. Parrow & Victor take this concept even further in their fusion calculus that shifts away from traditional uni-directional information flow to exchange of information during communication [PV98, BBM04]. They observe that in fusion calculus the terminology "input" and "output" are kept for familiarity, when "action" and "coaction" might be more appropriate [PV98, p. 177]. Again this is a step towards a different style of interaction that is conceptually similar to the work presented in this dissertation.

1.3 Sequential Pattern-Matching

A return to the top of the square follows the introduction of intensionality to sequential computation. Recent work by Jay and others develops a *pure pattern calculus* that bases computation on *pattern-matching* [Jay04, JK06, GW07, JK09, Jay09]. Although pure pattern calculus supports intensional functions that examine the internal structure of their arguments, these are limited to *data structures*. Intuitively, the intensionality of pure pattern calculus is more expressive than purely extensional models such as λ-calculus. However, while there is a trivial homomorphism from λ-calculus into pure pattern calculus, the lack of an inverse has yet to be investigated.

The presentation of pure pattern calculus in this dissertation is to illustrate pattern-matching concepts and to indicate related background material. There are several calculi that base reduction upon pattern-matching in the literature including; *compound calculus*, *static pattern calculus*, and earlier variations of pure/*dynamic pattern calculus* [Jay04, JK06, GW07, JK09, Jay09]. The CPC presented in this dissertation follows conceptually from this family of calculi, although no formal results about the direct relations of sequential pattern calculi to CPC will be presented. Some discussion of indirect relations and the complexities involved appear when detailing pure pattern calculus and SF-calculus, as well as in the conclusions.

1.4 Intensional Sequential Computation

New work begins in the top right corner of the square that formalises intensional sequential computation through combinatory logic. Indeed, there are intensional Turing-computable functions on combinators that cannot be represented within λ-calculus or Schönfinkel's combinatory logic [JGW11]. Specifically, combinators that are stable under reduction can be *factorised* into their components. Exploiting factorisation in a combinatory logic leads to *SF-calculus* that supports intensional sequential computation [JGW11].

The arrow across the top is formalised by showing a homomorphism from Schönfinkel's combinatory logic into SF-calculus. In the reverse direction, the factorisation of SF-calculus cannot be defined within λ-calculus [JGW11].

Just as extensionality can be characterised by β-reduction, intensionality can be characterised by pattern-matching in the style of pure pattern calculus. Similarly, as combinatorial completeness [CF58] captures the notion of extensional computation, a new *structure completeness* can be defined based upon pattern-matching for intensional computation [JGW11]. Structure complete combinatory logics support all Turing computable functions on their normal forms, for example the definition of a combinator that can test equality of normal forms. Indeed, there are a collection of structure complete combinatory logics that support factorisation and thus intensional sequential computation [JGW11].

1.5 Concurrent Pattern Calculus

The bottom right corner of the square is populated by CPC [GGJ10] that adapts intensionality, as captured by structure completeness, into concurrent computation. Generalising from structure complete style pattern-matching to symmetric *pattern-unification* provides a new basis for interaction in a concurrency model. Such interactions extend both intensionality and symmetry by performing both in an atomic manner.

The symmetry of process calculi and the properties of pattern-unification make CPC a natural choice to specify trade. Unification of patterns can be exploited to *discover* compatible trade partners before any exchange of information. An example of traders is developed in CPC to illustrate the concepts and as a precursor to later applications.

The support for matching on arbitrary number of symbols and arbitrary structures also makes CPC well suited to modelling reduction systems. By encoding the syntax or symbols into a pattern the reduction rules can then be encoded into a process that operates on this pattern. This provides an elegant encoding of reduction systems into CPC that can be highly modular.

1.6 Completing the Square

All that remains now to complete the square is to formalise the arrows down the right side and across the bottom.

Down the right side there is a parallel encoding from SF-calculus into

CPC. The same approach can also be used to translate Schönfinkel's combinatory logic, and thus λ-calculus, to form a diagonal parallel encoding from top left to bottom right.

Across the bottom there is a homomorphism, preserving reduction and parallel composition, from π-calculus into CPC that does not rely on any particular behavioural theory. The converse separation result can be proved in two ways that exploit multiple name matching and symmetry. This completes the square.

1.7 Behavioural Theory

Although the relations that form the computation square are formalised, further development is required to complete CPC and formalise relations with other process calculi. In particular a behavioural theory is needed to define what it means for two CPC processes to be equivalent, that is behave the same way. This can be done adapting the standard approach [MPW92, MS92, HY95, BGZZ98, WG04] taking into account the subtleties of CPC interaction.

A *barbed congruence* relation is defined for CPC in the usual manner, the only complexity being in the nature of the *barbs*. As unification of CPC patterns may involve many *names*, the barbs are parametrised by a set of names that appear in a pattern, subject to some limitations.

An alternative semantics for CPC is developed in the form of a *labelled*

transition system (LTS) that induces the same operational semantics as the reductions of CPC.

In developing the *bisimulation* relation the flexibility of pattern-unification needs to be accounted for. In particular, some patterns are more general than others and so a *compatibility* relation is defined that is a partial order on patterns. This is then used to define the bisimulation.

Properties of the compatibility relation are considered, in particular the motivation behind the ordering on patterns. Soundness of the bisimulation is shown by proving that it is a barbed congruence. Completeness of the bisimulation is shown by proving that the barbed congruence is a bisimulation. This requires the development of *contexts* that can enforce behaviours that fit the compatibility relation.

With the coincidence of the two semantics formalised, equivalence of processes can be proved. In particular, a general result that shows compatible patterns can be subsumed, with appropriate replications.

1.8 Relations to Other Process Calculi

With CPC's behavioural theory completed, the relationship to other process calculi can be formalised.

As Gorla demonstrates that Linda is more expressive than π-calculus it is natural to compare Linda and CPC. There is a straightforward homomorphism from Linda into CPC. The converse separation result can be proved

in two different ways using symmetry or intensionality.

As Spi calculus introduces some intensionality in a process calculus, the relationship to CPC is also of interest. The intensionality of Spi calculus can be homomorphically encoded into CPC. The symmetry of CPC cannot be rendered in Spi calculus, thus ensuring there is no converse encoding.

As the separation results can all be proved exploiting symmetry, the relationship of fusion calculus to CPC is of particular interest. It turns out that the peculiarities of name fusion prevent a valid encoding of fusion calculus into CPC. Conversely, fusion calculus is unable to render the multiple name matching or symmetry of CPC. Consequently, fusion calculus and CPC turn out to be unrelated.

1.9 Applications

The symmetry and information exchange of CPC provide a natural language to express trade. Traders can discover each other using common interest represented by some information in a pattern. They can then exchange information in a single interaction to complete a trade. The interplay of sequential and concurrent intensionality can be exploited to develop trading applications within Concurrent **bondi**. Combining all these elements allows programming of web services where trade partners discover each other and exchange information in an open collaborative environment [GWJ11].

1.10 How to Read This Dissertation

This dissertation covers a variety of areas that may appeal to different readers. Consequently there are several paths through the text that focus on different topics.

The path outlined in the introduction covers all topics and is as follows. The top left corner of the square is introduced in Chapter 2 using λ-calculus and Schönfinkel's combinatory logic. The arrow down the left side is followed in Chapter 3 which introduces several process calculi that support λ-calculus, and presents machinery for relating process calculi. Intensionality in sequential computation is first presented conceptually in Chapter 4, and then formally in Chapter 5 with the arrow across the top. The bottom right corner of the square on CPC is presented in Chapter 6 and the arrows completing the square in Chapter 7. A behavioural theory is developed for CPC in Chapter 8 as a foundation for relating CPC to other process calculi in Chapter 9. The expressive power of CPC is then demonstrated by applications in Chapter 10 that exploit sequential and concurrent intensionality.

Another path focuses upon the foundations of sequential computation and travels across the top of the square. This begins in Chapter 2 with extensional sequential computation and the relations between λ-calculus and Schönfinkel's combinatory logic. Intensionality is introduced conceptually by considering computation based on pattern-matching in Chapter 4 through recent work on pure pattern calculus. The completion of this path is in

Chapter 5 that develops SF-logic, formalises that SF-logic is more expressive than λ-calculus and Schönfinkel's combinatory logic, and presents a new structural completeness.

A similar path focuses upon concurrent computation and travels across the bottom of the square. This begins in Chapter 3 that revisits several popular process calculi and introduces formalism to relate them. CPC is developed in Chapter 6 and the arrow across the bottom of the square completed in Chapter 7. A supporting behavioural theory for CPC is presented in Chapter 8 and then exploited in Chapter 9 that relates CPC to other process calculi from Chapter 3.

A more practical path is driven by examples and applications supported by the theory. An extensive example is developed in Section 6.3 (using CPC as specification language) to motivate the theory and as a precursor to larger applications. The bulk of this path is in Chapter 10 that revisits earlier examples and concepts by developing programs in Concurrent **bondi**. This includes applications exploiting both sequential and concurrent computation, as well as the interplay between them.

Two chapters do not appear in the paths above: Chapter 1 provides an overview of the dissertation as a whole; and Chapter 11 draws conclusions and summarises the main results.

Chapter 2

Extensional Sequential Computation

This chapter considers the top left corner of the computation square; extensional sequential computation as supported by two models: Church's λ-calculus [Chu36, Bar85, HS86] and Schönfinkel's combinatory logic [Sch24, CF58]. Both models are recalled while introducing general concepts and terminology for use in later chapters. Details of the homomorphisms between them are presented to show equivalence and for later exploitation.

Both models base reduction rules upon the application of a function to one or more arguments. Functions in both models are extensional in nature, that is a function does not have direct access to the internal structure of its arguments. Thus, functions that are extensionally equal are indistinguishable within either Church's λ-calculus or Schönfinkel's combinatory logic even though they may have different normal forms.

The relationship between Church's λ-calculus and Schönfinkel's combinatory logic is closer than sharing application-based reduction and extensionality. There is a homomorphism from call-by-value λ_v-calculus into any combinatory logic that supports the combinators S and K [CF58, CHS72, Bar85, HS86]. There is also a homomorphism from Schönfinkel's combinatory logic to a λ-calculus with more generous operational semantics [CF58, CHS72, Bar85, HS86].

2.1 Abstraction

This section provides a skeletal introduction to Church's λ-calculus. Some general concepts and terminology are also presented along the way for later use.

The *term* syntax of the λ-calculus is given by

$$t ::= x \mid t\ t \mid \lambda x.t\ .$$

The *variable* x is a place holder for a term and is available for substitution. The *application* $s\ t$ applies the *function* s to the *argument* t. *Abstractions* $\lambda x.t$ have a binding variable x and a body t.

The *free variables* of a term t, denoted $\mathsf{fv}(t)$, are defined by

$$\begin{aligned} \mathsf{fv}(x) &= \{x\} \\ \mathsf{fv}(s\ t) &= \mathsf{fv}(s) \cup \mathsf{fv}(t) \\ \mathsf{fv}(\lambda x.t) &= \mathsf{fv}(t) \backslash \{x\}\ . \end{aligned}$$

A variable is free in itself. The free variables of an application are the union of the free variables of the function and argument. The free variables of an abstraction are the free variables of the body excluding the binding variable. A term t is *closed* if it has no free variables, that is, $\mathsf{fv}(t) = \{\}$.

A *substitution* σ is defined as a partial function from variables to terms. The *domain* of σ is denoted $\mathsf{dom}(\sigma)$; the free variables of σ, written $\mathsf{fv}(\sigma)$, is given by the union of the sets $\mathsf{fv}(\sigma x)$ where $x \in \mathsf{dom}(\sigma)$. The *variables* of σ, written $\mathsf{vars}(\sigma)$, are $\mathsf{dom}(\sigma) \cup \mathsf{fv}(\sigma)$. A substitution σ *avoids* a variable x (or collection of variables μ) if $x \notin \mathsf{vars}(\sigma)$ (respectively $\mu \cap \mathsf{vars}(\sigma) = \{\}$). Substitution *composition* is denoted $\sigma_2 \circ \sigma_1$ and indicates that $\sigma_2 \circ \sigma_1(t) = \sigma_2(\sigma_1 t)$. Note that all substitutions considered in this dissertation have finite domain.

For later convenience define an *identity substitution* id_X to map every variable in X to itself. That is, $\mathsf{id}_X = \{x_i/x_i\}$ for every $x_i \in X$.

The application of a substitution σ to a term t is defined as follows

$$\begin{array}{rcll} \sigma x & = & u & \text{if } \sigma \text{ maps } x \text{ to } u \\ \sigma x & = & x & \text{if } x \notin \mathsf{dom}(\sigma) \\ \sigma(s\ t) & = & (\sigma s)\ (\sigma t) & \\ \sigma(\lambda x.t) & = & \lambda x.(\sigma t) & \text{if } \sigma \text{ avoids } x \quad . \end{array}$$

If a variable is in the domain of the substitution then apply the mapping of the substitution, otherwise do nothing. Apply a substitution to the function and argument of an application. A substitution is applied to the body of an abstraction as long as the variable being bound in the abstraction is avoided. The side conditions on abstraction prevent accidental clash or capture of variables.

Problems with variable clash and capture are resolved by renaming of variables, or α-conversion $=_\alpha$ defined by

$$\lambda x.t =_\alpha \lambda y.\{y/x\}t$$

where y is not in the free variables of t.

Operational Semantics

There are several variations of the λ-calculus with different operational semantics. For construction of the computation square by exploiting the results of Milner et al. [Mil90], it is necessary to choose an operation semantics,

such as *call-by-value* λ_v-calculus or *lazy* λ_l-calculus. The choice here is to use call-by-value λ_v-calculus, although all the results can be reproduced for lazy λ_l-calculus as well. In addition a more generous operation semantics for λ-calculus will be presented for later discussion and relations.

To formalise the reduction of call-by-value λ_v-calculus requires a notion of *value* v. Here, these are defined in the usual way, by

$$v \quad ::= \quad x \mid \lambda x.t$$

consisting of variables and λ-abstractions.

Computation in the λ_v-calculus is through the β_v-reduction rule

$$(\lambda x.t)v \longrightarrow_v \{v/x\}t \ .$$

When an abstraction $\lambda x.t$ is applied to a value v then substitute v for x in the body t.

The *reduction relation* (also denoted $\longrightarrow_v$) is the smallest that satisfies

the following rules

$$\beta_v : \quad \frac{}{(\lambda x.t)v \longrightarrow_v \{v/x\}t}$$

$$\mathsf{appl} : \quad \frac{s \longrightarrow_v s'}{s\ t \longrightarrow_v s'\ t}$$

$$\mathsf{appr} : \quad \frac{t \longrightarrow_v t'}{s\ t \longrightarrow_v s\ t'} \ .$$

The transitive closure of the reduction relation is denoted $\longrightarrow_v^*$ though the star may be elided if it is obvious from the context.

The more generous operational semantics for the λ-calculus allows any term to be the argument when defining β-reduction. Thus the more generous β-reduction rule is

$$(\lambda x.s)t \longrightarrow \{t/x\}s$$

where t is any term of the λ-calculus.

The reduction relation (also denoted $\longrightarrow$) is the least relation satisfying

the following rules

$$\beta: \quad \frac{}{(\lambda x.s)t \longrightarrow \{t/x\}s}$$

$$\mathsf{appl}: \quad \frac{s \longrightarrow s'}{s\ t \longrightarrow s'\ t}$$

$$\mathsf{appr}: \quad \frac{t \longrightarrow t'}{s\ t \longrightarrow s\ t'}\ .$$

The transitive closure of the reduction relation is denoted $\longrightarrow^*$ is as for λ_v-calculus.

Observe that any reduction $\longrightarrow_v$ of λ_v-calculus is also a reduction $\longrightarrow$ of λ-calculus.

Lemma 2.1.1. *Every reduction $\longrightarrow_v$ of λ_v-calculus is also a reduction $\longrightarrow$ of λ-calculus.*

Proof: Trivial. □

As reduction invokes implicit substitution, a single β_v-reduction or β-reduction may result in renaming many variable occurrences. There are a number of approaches to managing the substitutions and variable scoping [ACCL91, Fis93, dB72, GP02], however one of the oldest and cleanest is to simply remove the variables from the picture.

2.2 Combination

This section provides an overview of combinatory logics or calculi. The presentation in this chapter is done using Schönfinkel's combinators S and K while describing general concepts and terminology for later use in Chapter 5. As the focus within this dissertation is on computation rather than logical paradoxes, the emphasis will be on calculi over logics, with rules for reduction rather than equations.

A *combinatory calculus* is given by a finite collection $\mathcal{O}$ of *operators* (meta-variable O) that are used to define the $\mathcal{O}$-*combinators* (meta-variables M, N, P, Q, X, Y, Z) built from these by application

$$M, N ::= O \mid MN \ .$$

Syntactic equality of combinators will be denoted by $\equiv$. The $\mathcal{O}$-*combinatory calculus* or $\mathcal{O}$-*calculus* is given by the combinators plus their reduction rules. A *homomorphism* of sequential calculi is a mapping from one calculus to another that preserves reduction and application.

Schönfinkel's combinatory logic can be represented by two combinators S and K [Sch24, CF58] so the equivalent SK-calculus has *reduction rules*

$$\begin{aligned} SMNX &\longrightarrow MX(NX) \\ KXY &\longrightarrow X \ . \end{aligned}$$

The combinator $SMNX$ duplicates X as the argument to both M and N.

The combinator KXY eliminates Y and returns X.

The rules are instantiated by replacing each meta-variable M, N, X or Y by a particular combinator. The *reduction relation* $\longrightarrow$ and reflexive transitive closure $\longrightarrow^*$ are as for λ-calculus.

The relation $\longrightarrow^*$ also induces an equivalence relation $=$ on the combinators, their *equality*. The $\mathcal{O}$*-combinatory logic* or $\mathcal{O}$*-logic* is the system of equivalence classes of combinators from $\mathcal{O}$-combinatory calculus.

Although this is sufficient to provide a direct account of functions in the style of λ-calculus, an alternative is to consider the representation of arbitrary computable functions that act upon combinators.

A *symbolic function* is defined to be an n-ary partial function $\mathcal{G}$ of some combinatory logic, i.e. a function of the combinators that preserves their equality, as determined by the reduction rules. That is, if $X_i = Y_i$ for $1 \leq i \leq n$ then $\mathcal{G}(X_1, X_2, \ldots, X_n) = \mathcal{G}(Y_1, Y_2, \ldots, Y_n)$ if both sides are defined. A symbolic function is *restricted* to a set of combinators, e.g. the normal forms, if its domain is within the given set.

A combinator G in a calculus *represents* $\mathcal{G}$ if

$$GX_1 \ldots X_n = \mathcal{G}(X_1, \ldots, X_n)$$

whenever the right-hand side is defined. For example, the symbolic functions

$$\begin{aligned} \mathcal{S}(X_1, X_2, X_3) &= X_1 X_3 (X_2 X_3) \\ \mathcal{K}(X_1, X_2) &= X_1 \end{aligned}$$

are represented by S and K, respectively, in SK-calculus. Consider the symbolic function

$$\mathcal{I}(X) = X \ .$$

In SKI-calculus where I has the rule

$$IY \longrightarrow Y$$

then $\mathcal{I}$ is represented by I. In both SKI-calculus and SK-calculus, $\mathcal{I}$ is represented by any combinator of the form SKX since

$$SKXY = KY(XY) = Y \ .$$

For convenience define the *identity combinator* I in SK-calculus to be SKK.

For later use, define some familiar logical constructs. Define the condition `if` P `then` M `else` N by PMN. It follows that truth is given by K and falsehood by KI since $KMN \longrightarrow M$ and $KIMN \longrightarrow IN \longrightarrow N$. The usual boolean operations are defined in the obvious way; write `not` M for *negation*; M `and` N for the *conjunction* of M and N; M `or` N for their *disjunction*; and M `implies` N for *implication*. Similarly, there is a fixpoint combinator

`fix` with the property that `fix` $M \longrightarrow^* M(\texttt{fix}\ M)$.

2.3 Relations

This section recalls the relations between various λ-calculi and SK-calculus. In particular the homomorphism from λ_v-calculus to SK-calculus that preserves reduction and application. When considering a homomorphism in the other direction, the more generous reduction relation of SK-calculus requires that the homomorphism be to λ-calculus, i.e. without the limitation of call-by-value reduction.

One of the goals of combinatory logic is to give an equational account of variable binding and substitution, particularly as it appears in λ-calculus.

In order to represent λ-abstraction, it is necessary to have some variables to work with. Given $\mathcal{O}$ as before, define the *$\mathcal{O}$-terms* by

$$M, N ::= x \mid O \mid MN$$

where x is as in λ-calculus. Free variables and the substitution $\{N/x\}M$ of the term N for the variable x in the term M are defined in the obvious manner, since the term calculus does not have any binding constructions built in. The *$\mathcal{O}$-term calculus* has reduction defined by the same rules as the $\mathcal{O}$-calculus, noting that instantiation may introduce variables. Symbolic computation and representation can be defined for terms just as for combinators.

Given a variable x and term M define a symbolic function $\mathcal{G}$ on terms by

$$\mathcal{G}(X) = \{X/x\}M \ .$$

Note that if M has no free variables other than x then $\mathcal{G}$ is also a symbolic computation of the combinatory logic. If every such function $\mathcal{G}$ on $\mathcal{O}$-combinators is representable then the $\mathcal{O}$-combinatory logic is *combinatorially complete* in the sense of Curry [CF58, p. 5].

Given S and K then $\mathcal{G}$ above can be represented by a term $\lambda^* x.M$ given by

$$\begin{aligned} \lambda^* x.x &= I \\ \lambda^* x.y &= Ky \quad \text{if } y \neq x \\ \lambda^* x.O &= KO \\ \lambda^* x.MN &= S(\lambda^* x.M)(\lambda^* x.N) \ . \end{aligned}$$

The following lemma is both central to the theorem that λ-calculus can be represented in combinatory logic, and exploited later in Chapter 5.

Lemma 2.3.1. *For all terms M and N and variables x there is a reduction*

$$(\lambda^* x.M)\ N \longrightarrow^* \{N/x\}M \ .$$

Proof: The proof is by induction on the structure of the combinator M.

- If M is x then $(\lambda^* x.M)N \equiv IN \longrightarrow N \equiv \{N/x\}M$.

- If M is any other variable or an operator then $(\lambda^* x.M)N \equiv KMN \longrightarrow M \equiv \{N/x\}M$.

- Finally, if M is of the form $M_1 M_2$ then

$$\begin{aligned}(\lambda^* x.M)N &\equiv S(\lambda^* x.M_1)(\lambda^* x.M_2)N \\ &\longrightarrow (\lambda^* x.M_1)N((\lambda^* x.M_2)N) \\ &\longrightarrow (\{N/x\}M_1)(\{N/x\}M_2) \\ &\equiv \{N/x\}M\end{aligned}$$

  by two applications of induction.

□

The following theorem is a central result of combinatory logic [CF58] and sufficient to show there is a homomorphism from λ_v-calculus to any combinatory calculus that represents S and K.

Theorem 2.3.2. *Any combinatory calculus that is able to represent S and K is combinatorially complete.*

Proof: Given $\mathcal{G}(X) = \{X/x\}M$ as above define G to be $\lambda^* x.M$ and apply Lemma 2.3.1. □

Indeed, this result can be used to show that there is a homomorphism from λ-calculus to SK-calculus. These homomorphisms will be exploited later when traversing the top edge of the computation square. During this traversal the converse directions will also consider the relation to SK-calculus, so

it is useful to formalise the converse to the homomorphisms from λ-calculi to SK-calculus.

The SK-calculus can be translated to λ-calculus as follows [CF58, CHS72, Bar85, HS86]:

$$\begin{aligned} [\![S]\!] &= \lambda g.\lambda f.\lambda x.g\ x\ (f\ x) \\ [\![K]\!] &= \lambda x.\lambda y.x \\ [\![MN]\!] &= [\![M]\!]\ [\![N]\!]\ . \end{aligned}$$

For example

$$\begin{aligned} [\![SKX]\!] &= (\lambda g.\lambda f.\lambda x.g\ x\ (f\ x))\ (\lambda x.\lambda y.x)\ [\![X]\!] \\ &\longrightarrow (\lambda f.\lambda x.(\lambda x.\lambda y.x)\ x\ (f\ x))\ [\![X]\!] \\ &\longrightarrow \lambda x.(\lambda x.\lambda y.x)\ x\ ([\![X]\!]\ x) \\ &\longrightarrow \lambda x.(\lambda y.x)\ ([\![X]\!]\ x) \\ &\longrightarrow \lambda x.x \end{aligned}$$

for any combinator X. Indeed in SK-calculus any combinator of the form SKX is an identity function and is equivalent to $\lambda x.x$ in λ-calculus.

The following theorem has been proved many times before [CF58, CHS72, Bar85, HS86] and shows that the translation is a homomorphism.

Theorem 2.3.3. *Translation from SK-calculus to λ-calculus preserves the reduction relation.*

Of course there is also a trivial homomorphism from λ_v-calculus into λ-calculus.

Theorem 2.3.4. *There is a homomorphism from λ_v-calculus to λ-calculus.*

Proof: Trivial by Lemma 2.1.1. □

Although the top left corner of the computation square is populated by λ_v-calculus, the arrows out allow for either λ_v-calculus or SK-calculus to be used. Indeed, the homomorphisms in both directions between λ-calculus and SK-calculus allow these two calculi to be considered equivalent. This completes the top left corner of the computation square with either λ_v-calculus or SK-calculus available when formalising the arrows out. Now there are two choices about which arrow to follow first: across the top follows sequentiality; and down the side follows extensionality. The path chosen here is to following extensionality as intensionality leads to new work and new results. Thus the next chapter introduces concurrency, discusses the left hand side of the square, and remains focused on the literature. If the reader prefers to remain on sequential computation then next step is in Chapter 4.

Chapter 3

Process Calculi

The bottom left corner of the computation square considers extensional concurrent computation, here defined to be process calculi that subsume λ-calculus. Milner identified the complexity faced by concurrency theorists [Mil99]:

> Building communicating systems is not a well-established science, or even a stable craft; we do not have an agreed repertoire of constructions for building and expressing interactive systems, in the way that we (more-or-less) have for building sequential computer programs.

Since Milner et al. [Mil90] showed that their π-calculus [MPW92] generalises λ-calculus [Chu36, Bar85], process calculi have been expected to support sequential computation represented by the λ-calculus [Mil90, BB90, MPW92, CG98, Mil99]. Even limiting consideration to process calculi that subsume λ-

calculus, there are many that could be the exemplar for the bottom left corner of the square. This is complicated by the lack of a concurrency benchmark analogous to Turing-computability for sequential computation [Mil99, PV98]. The most obvious choice is the π-calculus being: the first process calculus shown to subsume λ-calculus; having widely recognised syntax and semantics; and being widely related to other process calculi [Mil90, GA97, CG98, PV98]. Indeed, the encoding of Milner et al. for λ_v-calculus supports parallel reduction and translation of terms to processes, the key properties of *parallel encodings*.

Arising from the large variety of concurrency models and lack of benchmark is a recent focus on relative expressive power and relating different calculi to each other [Nes06, Par08, Gor08a, Gor08b]. Approaches to relating process calculi are usually done piecemeal due to the lack of a benchmark to start from and so many models to take into account. However, a general approach is taken by Gorla who uses *valid encodings* to relate process calculi and formalise their relative expressive power [Gor08a, Gor08b, GGJ10]. These valid encodings preserve many important properties and can be easily used to support the stronger homomorphisms required for the computation square.

In Gorla's work on valid encodings he presents a hierarchy of process calculi structured according to relative expressive power [Gor08b, Gor08a]. The nodes in the hierarchy consist of sets of calculi classified according to properties of communication [Gor08b, Gor08a]. The π-calculus already fits within

one of these sets, however it is not in one of the four equally-expressive sets that sit at the top of the hierarchy. As this dissertation is exploring expressive power, it is natural to address one of these sets of calculi. Gelernter's Linda [Gel85, YFY96, PMR99] makes a good choice as it is from one of the most expressive sets of calculi and has the least similarity to π-calculus, according to Gorla's classification system [Gor08b, Gor08a].

Considering relative expressive power, there are other process calculi that have conceptual similarities to later work but do not fit within Gorla's hierarchy.

One such representative calculus is Gordon et al.'s Spi calculus that introduces structured *terms* and intensional reductions upon them [GA97]. The structures of terms include *pairing*, *encryption* and natural numbers, with process forms and reduction axioms that rely upon intensionality upon the structure of terms. Although all these reduction axioms are within a process and not part of communication, this step towards intensional communication is of interest. The Spi calculus has been related to π-calculus before via translations that preserve some properties [BPV03, BPV05], however these do not meet the criteria for valid encodings.

Another concept in concurrency is the shift away from the sequential asymmetry of function and argument to a *symmetric* parallel composition where both processes can evolve equally. This was supported by Milner who considered reductions in asynchronous calculi to be a symmetric interaction between a process and the environment [Mil99]. Parrow & Victor take this

concept even further in their fusion calculus that shifts away from traditional uni-directional information flow to *exchange* of information during communication [PV98, BBM04]. They observe that in fusion calculus the terminology "input" and "output" are kept for familiarity, with "action" and "coaction" being more appropriate [PV98, p. 177]. Parrow & Victor show that π-calculus can be translated into fusion calculus [PV98], however this does not meet the criteria for valid encodings. Although relating fusion calculus to other calculi proves problematic, the symmetry of interaction is conceptually similar to later work.

The rest of this chapter recalls several process calculi and valid encodings. Note that all results in this chapter are formalised in the literature and so shall be restated rather than recalled in detail.

3.1 π-calculus

The π-calculus [MPW92, Mil93] holds a pivotal rôle amongst process calculi for both popularity and being the most compact that subsumes extensional sequential computation as represented by the λ-calculus [Mil90]. This section overviews the π-calculus while introducing concepts and definitions for later use. The culmination of the section is recalling Milner's encoding of λ_v-calculus in π-calculus.

The processes for the π-calculus are given as follows and exploit a class

of names (denoted $m, n, x, y, z, \ldots$ similar to variables in the λ-calculus):

$$P \quad ::= \quad \mathbf{0} \;\mid\; P \,|\, P \;\mid\; !P \;\mid\; (\nu a)P \;\mid\; a(b).P \;\mid\; \overline{a}\langle b\rangle.P \;\mid\; \surd\,.$$

The *null process* $\mathbf{0}$ is the process that can perform no communication or reduction. The *parallel composition* $P \mid Q$ are the two processes P and Q in parallel. The *replication* $!P$ is the unbounded repetition of the process P. The *restriction* $(\nu a)P$ limits the scope of the name a in the process P. The *input* $a(b).P$ coordinates on the channel a and binds an input to the name b in the process P. The *output* $\overline{a}\langle b\rangle.P$ coordinates on the channel a and outputs the name b before continuing with the process P. The *success process* $\surd$ is used to signal reaching a specific state and is added here to support valid encodings later.

The names of the π-calculus are used for channels of communication and for information being communicated. The *free names* of a process $\mathsf{fn}(P)$ are

given by

$$\begin{aligned}
\mathsf{fn}(\mathbf{0}) &= \{\} \\
\mathsf{fn}(P|Q) &= \mathsf{fn}(P) \cup \mathsf{fn}(Q) \\
\mathsf{fn}(!P) &= \mathsf{fn}(P) \\
\mathsf{fn}((\nu a)P) &= \mathsf{fn}(P) \backslash \{a\} \\
\mathsf{fn}(a(b).P) &= (\mathsf{fn}(P) \backslash \{b\}) \cup \{a\} \\
\mathsf{fn}(\overline{a}\langle b \rangle.P) &= \mathsf{fn}(P) \cup \{a, b\} \\
\mathsf{fn}(\surd) &= \{\} \; .
\end{aligned}$$

The null process has no free names. The free names of two processes in parallel is the union of their sets of free names. The free names of a replication are the free names of the process being replicated. The free names of a restriction are the free names of the process under restriction except the name being restricted. The free names of an input are the free names of the body excluding the name being bound, union the channel name. The free names of an output are the free names of the body, union the channel name and output name. The success process has no free names.

Substitutions in the π-calculus are partial functions that map names to names, with domain, range, free names, names, and avoidance, all straightforward adaptions from substitutions of the λ-calculus in Section 2.1. The

application of a substitution σ to name is given by

$$\begin{aligned} \sigma x &= y && \text{if } \sigma \text{ maps } x \text{ to } y \\ \sigma x &= x && \text{if } x \notin \mathsf{dom}(\sigma). \end{aligned}$$

The application of a substitution σ to a process is defined by

$$\begin{aligned} \sigma(\mathbf{0}) &= \mathbf{0} \\ \sigma(P|Q) &= (\sigma P) \mid (\sigma Q) \\ \sigma(!P) &= !(\sigma P) \\ \sigma((\nu a)P) &= (\nu a)(\sigma P) && \text{if } \sigma \text{ avoids } a \\ \sigma(a(b).P) &= (\sigma a)(b).(\sigma P) && \text{if } \sigma \text{ avoids b} \\ \sigma(\overline{a}\langle b\rangle.P) &= \overline{(\sigma a)}\langle \sigma b\rangle.(\sigma P) \\ \sigma(\surd) &= \surd . \end{aligned}$$

The substitution has no effect on the null process. A substitution applied to a parallel composition is applied to the processes in parallel. A substitution is applied under a replication. As restriction limits the scope of a name only apply the substitution if there is no clash of names. Similarly for input, apply the substitution to the channel name and body as long as the binding name b is avoided. Substitutions apply to all of: the channel name, output name, and body of an output process. A substitution has no effect on the success process.

Issues where substitutions must avoid restricted or input names are handled by α-conversion $=_\alpha$ that is the congruence relation generated by the

following axioms

$$\begin{array}{lll} (\nu a)P & =_\alpha & (\nu b)(\{b/a\}P) \qquad b \notin \mathsf{fn}(P) \\ a(b).P & =_\alpha & a(c).(\{c/b\}P) \qquad c \notin \mathsf{fn}(P)\ . \end{array}$$

The renaming replaces the name that limits scope by another that does not appear in the process whose scope is limited.

The general *structural equivalence relation* $\equiv$ is defined in the usual way: it includes $=_\alpha$ and its defining axioms are below.

$$P \mid \mathbf{0} \equiv P \qquad\qquad P \mid Q \equiv Q \mid P$$

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad\qquad (\nu n)\mathbf{0} \equiv \mathbf{0}$$

$$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \qquad\qquad !P \equiv P \mid !P$$

$$P \mid (\nu n)Q \equiv (\nu n)(P \mid Q) \quad \text{if } n \notin \mathsf{fn}(P)$$

The π-calculus has one *reduction rule* given by

$$a(b).P \mid \overline{a}\langle c\rangle.Q \qquad \longmapsto \qquad \{c/b\}P \mid Q\ .$$

It states that if an input and output in parallel composition share the same channel name a then they interact as follows. The input binds b to c in the body P which is then run in parallel composition with the body Q. The reduction rule is then closed under parallel composition, restriction and

structural equivalence to yield the reduction relation $\longmapsto$ as follows:

$$\frac{P \longmapsto P'}{P \mid Q \longmapsto P' \mid Q}$$

$$\frac{P \longmapsto P'}{(\nu n)P \longmapsto (\nu n)P'}$$

$$\frac{P \equiv Q \quad Q \longmapsto Q' \quad Q' \equiv P'}{P \longmapsto P'} \ .$$

The reflexive, transitive closure of $\longmapsto$ is denoted $\Longmapsto$.

Relations to the π-calculus

This section develops machinery and results for later use and formalises the relations from λ-calculus into π-calculus. The presentation here is to provide a sketch of results in the literature and to set up definitions for later use.

To support relations to the π-calculus requires a behavioural theory that defines what it means for two processes to be equivalent, that is behave the same way. The standard approach to defining behavioural equivalence [MPW92, MS92, HY95, BGZZ98, WG04] is summarised below. Note that the following summary is skeletal as a detailed development of behavioural theory is performed in Chapter 8 for CPC.

The first step is to define *barbs* that characterise the interactions a process can exhibit. For the π-calculus barbs $P \downarrow_n$ can be defined to mean that the

process P can perform an input or output on the channel n. That is, P has a form $(\nu\widetilde{m})(n(x).P_1 \mid P_2)$ or $(\nu\widetilde{m})(\overline{n}\langle x\rangle.P_1 \mid P_2)$ where $n \notin \widetilde{m}$. Also define *contexts* $\mathcal{C}(\cdot)$ for π-calculus to be processes where one instance of the null process is replaced by the dot. Exploiting these, a *barbed congruence* $\simeq$ can be defined as the least, symmetric, barb preserving, reduction and context closed binary relation on processes. That is, two processes are barbed congruent if: any barb of one process is also a barb of the other; every reduction of one process can be mimicked by the other; and the process remain barbed congruent in any context.

The next step is to define a *labelled transition system* (LTS) that provides an alternative semantics. The transitions are of the form $P \xrightarrow{\mu} P'$ where μ is a *label* that describes the action P takes to evolve to P'. Typically such actions are either internal actions τ that represent the process reducing, or external actions (e.g. $\overline{n}\langle x\rangle$) that represent an interaction with another process. For example, the process $\overline{n}\langle x\rangle.P$ has a transition $\overline{n}\langle x\rangle.P \xrightarrow{\overline{n}\langle x\rangle} P$ where the label is the output of x on the channel n and the process evolves to P.

A *bisimulation* relation $\sim$ is then defined to equate processes that cannot be distinguished by any sequence of transitions and that are closed under substitution. That is, two processes are bisimilar if: every internal transition of one process is mimicked by an internal transition of the other; every external transition of one process is mimicked by an external transition of the other; and the processes remain bisimilar under any substitution. Note that

while the bisimulation for π-calculus requires identical external transitions, this is not always the case.

To conclude, soundness and completeness are shown. That is; the bisimulation is shown to be a barbed congruence, and the barbed congruence is shown to be a bisimulation.

This has been developed before in detail for the π-calculus and the results from the literature shall be assumed [Mil89, MPW92, MS92, HY95].

Now that the π-calculus is recalled it remains to revisit the encoding of λ-calculus by Milner [Mil90]. In "Functions as Processes" Milner presents two translations of the form $[\![\cdot]\!]_c$ from λ-calculus into π-calculus that depend upon the operational semantics: one is *lazy* and the other is *call-by-value*. Although both show subsumption of some λ-calculus by π-calculus, the call-by-value encoding is chosen to meet the requirements of a parallel encoding.

Definition 3.1.1. *A translation $[\![\cdot]\!]_c$ from a language into process calculus, parametrised by a name c, is a* parallel encoding *if the following two properties hold.*

1. Reduction Preservation: *For all reductions $M \longrightarrow M'$ then there is a sequence of reductions $[\![M]\!]_c \quad \Longmapsto \simeq \quad [\![M']\!]_c$.*

2. Parallelisation: *The translation of the application MN is of the form*

$$[\![MN]\!]_c \quad \stackrel{\text{def}}{=} \quad (\nu\widetilde{n})(R \mid [\![M]\!]_{n1} \mid [\![N]\!]_{n2})$$

where $n1, n2 \in \widetilde{n}$ for some $\widetilde{n}$ and R that depend only upon the translation.

Reduction preservation is a natural requirement considering the focus is on models of computation. An encoding that does not preserve reduction, or whose reductions map behaviourally distinct terms to equivalent processes, would not support the same computations.

Parallelisation is a step towards compositionality (see valid encodings 3.2.1 in the next section) of encodings; that the encoding of components can be independent of the encoding of application. Further, this approach allows independent reduction of the components of an application, as is supported by λ_v-calculus, λ-calculus and SK-calculus (though notably not λ_l-calculus). As the shift from sequential to concurrent computation can exploit this to support parallel reductions, the definition of parallel encoding encourages a more flexible reduction strategy.

Observe that combining reduction preservation and parallelisation has the following consequence. For all applications MN such that $M \longrightarrow M'$ and $N \longrightarrow N'$ then the translation $[\![MN]\!]_c \stackrel{\text{def}}{=} (\nu\widetilde{n})(R \mid [\![M]\!]_{n1} \mid [\![N]\!]_{n2})$ has reductions

$$\begin{array}{lll} & (\nu\widetilde{n})(R \mid [\![M]\!]_{n1} \mid [\![N]\!]_{n2}) & \Longmapsto \simeq \quad (\nu\widetilde{n})(R \mid [\![M']\!]_{n1} \mid [\![N]\!]_{n2}) \\ \text{and} & (\nu\widetilde{n})(R \mid [\![M]\!]_{n1} \mid [\![N]\!]_{n2}) & \Longmapsto \simeq \quad (\nu\widetilde{n})(R \mid [\![M]\!]_{n1} \mid [\![N']\!]_{n2}) \ . \end{array}$$

That is, if the components of an application can each reduce according to the source reduction strategy, then these reductions can be performed in parallel in the translation.

As the β_v-reduction rule depends upon the argument being a value the translation into π-calculus must be able to recognise values. Thus, Milner defines the following

$$\begin{aligned} [\![y := \lambda x.t]\!] &\stackrel{\text{def}}{=} !y(w).w(x).w(p).[\![t]\!]_c \\ [\![y := x]\!] &\stackrel{\text{def}}{=} !y(w).\overline{x}\langle w\rangle \ . \end{aligned}$$

Also the following translation of λ_v-terms

$$\begin{aligned} [\![v]\!]_c &\stackrel{\text{def}}{=} (\nu y)\overline{c}\langle y\rangle.[\![y := v]\!] && y \text{ not free in } v \\ [\![s\ t]\!]_c &\stackrel{\text{def}}{=} (\nu q)(\nu r)(\mathsf{ap}(c,q,r) \mid [\![s]\!]_q \mid [\![t]\!]_r) \\ \mathsf{ap}(p,q,r) &\stackrel{\text{def}}{=} q(y).(\nu v)\overline{y}\langle v\rangle.r(z).\overline{z}\langle p\rangle \ . \end{aligned}$$

Observe that application is translated to parallel composition, with some additional side processes, and so meets the requirements for a parallel encoding.

That the translation preserves reduction is proved first by showing that $[\![\, t \,]\!]_c$ is weakly determinate and then in turn there is a call-by-value precongruence relation as done in Theorem 7.7 of Milner [Mil90].

Corollary 3.1.2. *The translation $[\![\cdot]\!]_c$ is a parallel encoding from λ_v-calculus to π-calculus.*

Proof: Straightforward by Definition 3.1.1 and Milner's Theorem 7.7 [Mil90]. □

The lack of a parallel encoding in the reverse direction is immediate, due to the lack of a parallel composition operator in λ_v-calculus. However, an

attempt to define some analogue could be as follows:

$$[\![P \mid Q]\!] \stackrel{\text{def}}{=} R[\![P]\!][\![Q]\!] \,.$$

Here the parallel composition is mapped to application, with some term R that acts as the machinery as in the translation $[\![\cdot]\!]_c$.

However, the difficulties of how to exploit such a structural encoding from π-calculus into λ_v-calculus can be avoided since there are straightforward results to show that λ_v-calculus reduction cannot capture concurrency. This is a corollary of Theorem 14.4.12 of Barendregt [Bar85, Ch 14], showing that λ-calculus is unable to render concurrency or support concurrent computations. A similar result is shown by Lemma B of Abramsky [Abr90] who proves that λ-calculus cannot solve the *parallel-or* problem. Thus, even without a structural requirement for an encoding of π-calculus into λ_v-calculus, the lack of support for π-calculus reductions is sufficient to show that no reasonable encoding can exist.

Corollary 3.1.3. *There is no encoding of π-calculus into λ-calculus that preserves concurrency.*

Proof:[Sketch] By Theorem 14.4.12 of Barendregt [Bar85], alternatively by Lemma B of Abramsky [Abr90]. Both of these theorems can be used to show that λ-calculus cannot represent a *parallel-or* function. The parallel-or

function is a function f that satisfies the following three rules

$$\begin{aligned} f\ \bot\ \bot &\longrightarrow^* \bot \\ f\ \mathtt{T}\ \bot &\longrightarrow^* \mathtt{T} \\ f\ \bot\ \mathtt{T} &\longrightarrow^* \mathtt{T} \end{aligned}$$

where $\bot$ represents non-termination and $\mathtt{T}$ represents true. Of course such a function is trivial to encode in π-calculus by

$$F \;=\; n_1(x).\overline{m}\langle x\rangle.\mathbf{0} \mid n_2(x).\overline{m}\langle x\rangle.\mathbf{0}\ .$$

Observe that F in parallel composition with two processes P_1 and P_2 that output true on n_1 and n_2, respectively, if they reduce to true will report true on m if either P_1 or P_2 output true. Since π-calculus can represent the parallel-or function and Theorem 14.4.12 of Barendregt [Bar85]and Lemma B of Abramsky [Abr90] show that λ-calculus cannot, the conclusion follows. □

3.2 Valid Encodings

Next is to formally compare the expressive power or process calculi while respecting certain properties. The basic idea is that given an encoding from a language $\mathcal{L}_1$ to a language $\mathcal{L}_2$ it then follows that $\mathcal{L}_2$ has equal or greater expressive power than $\mathcal{L}_1$. Although the idea is straight forward, there is some delicacy in the properties that the encoding must respect. This section: formalises what it is to be a language; recalls Gorla's *valid encodings* and

their properties; and restates some proof techniques for showing that valid encodings cannot exist.

A *language* $\mathcal{L}_i$ is here defined to be a process calculus that has certain properties as follows. A language must support a process that represents no action denoted $\mathbf{0}$, for example the null process of π-calculus. To support encoding a language must be able to represent a *k-ary context* $\mathcal{C}(\cdot_1; \ldots; \cdot_k)$ that is a process where k occurrences of $\mathbf{0}$ are linearly replaced by the holes $\{\cdot_1; \ldots; \cdot_k\}$ and where linearity ensures each hole occurs only once. As process calculi widely exploit names, for any process P of a language the free names, denoted $\mathsf{fn}(P)$ must be defined. As languages here must support computation it follows that each language $\mathcal{L}_i$ supports a notion of reduction, denoted $\longmapsto_i$, and the reflexive transitive closure of reduction denoted $\longmapsto\!\!\!\!\!\Rightarrow_i$. Denote an infinite sequence of reductions for a language by $\longmapsto_i^\omega$. Further, each language requires a reference behavioural equivalence $\simeq_i$ to equate behaviourally indistinguishable processes. Lastly, let $P \Downarrow_i$ mean that there exists P' such that $P \longmapsto\!\!\!\!\!\Rightarrow_i P'$ and $P' \equiv P'' \mid \surd$, for some P'' where $\surd$ is a process signalling success.

Gorla defines an *encoding* of a *source* language $\mathcal{L}_1$ into a *target* language $\mathcal{L}_2$ as a pair $(\llbracket \cdot \rrbracket, \varphi_{\llbracket\rrbracket})$ consisting of a *translation* $\llbracket \cdot \rrbracket$ and a *renaming policy* $\varphi_{\llbracket\rrbracket}$. The translation converts every source process into a target process. The renaming policy maps every source name to some construct of the target language containing k names for some k greater than zero. The translation may also fix some names to play a precise rôle or may encode a single name

into a target language construct containing several names. This fixing of names can be obtained by exploiting the renaming policy [Gor08b]. Finally, to simplify reading, let S range over processes of the source language (viz., $\mathcal{L}_1$) and T range over processes of the target language (viz., $\mathcal{L}_2$). Such a definition of encodings is very general and allows many encodings that do not preserve any behaviour.

Now consider encodings that satisfy the following properties that are reasoned about at length by Gorla [Gor08b]. Some discussion of these properties follows the definition of valid encodings, including specifics related to this dissertation.

Definition 3.2.1 (Valid Encoding [Gor08b])**.** *An encoding* $(\llbracket \cdot \rrbracket, \varphi_{\llbracket\rrbracket})$ *is* valid *if it satisfies the following five properties:*

1. Compositionality: *for every k-ary operator* op *of* $\mathcal{L}_1$ *and for every subset of names N, there exists a k-ary context* $\mathcal{C}^N_{\mathsf{op}}(\cdot_1; \ldots; \cdot_k)$ *such that, for all* $S_1, \ldots, S_k$ *with* $\mathsf{fn}(S_i) \subseteq N$ *for* $1 \leq i \leq k$*, it holds that* $\llbracket\, \mathsf{op}(S_1, \ldots, S_k) \,\rrbracket \stackrel{\text{def}}{=} \mathcal{C}^N_{\mathsf{op}}(\llbracket\, S_1 \,\rrbracket; \ldots; \llbracket\, S_k \,\rrbracket)$.

2. Name invariance: *for every S and substitution σ, it holds that*

$$\llbracket\, \sigma S \,\rrbracket \begin{cases} = \ \sigma' \llbracket\, S \,\rrbracket & \text{if } \sigma \text{ is injective} \\ \simeq_2 \ \sigma' \llbracket\, S \,\rrbracket & \text{otherwise} \end{cases}$$

   where σ' is such that $\varphi_{\llbracket\rrbracket}(\sigma(a)) = \sigma'(\varphi_{\llbracket\rrbracket}(a))$ for every name a.

3. Operational correspondence:

- *for all* $S \Longmapsto_1 S'$*, it holds that* $[\![S]\!] \Longmapsto_2 \simeq_2 [\![S']\!]$*;*
- *for all* $[\![S]\!] \Longmapsto_2 T$*, there exists* S' *such that* $S \Longmapsto_1 S'$ *and* $T \Longmapsto_2 \simeq_2 [\![S']\!]$*.*

4. Divergence reflection: *for every* S *such that* $[\![S]\!] \longmapsto_2^\omega$*, it holds that* $S \longmapsto_1^\omega$*.*

5. Success sensitiveness: *for every* S*, it holds that* $S \Downarrow_1$ *if and only if* $[\![S]\!] \Downarrow_2$*.*

Compositionality is a natural property that ensures the encoding of an operator upon processes is a context upon the encodings of the same processes. That is; the encoding of a process $\mathsf{op}(S_1, \ldots, S_k)$ must be defined by plugging in the encodings of its sub-processes $[\![S_i]\!]$ into a context the depends only upon the operator under translation $\mathcal{C}^N_{\mathsf{op}}([\![S_1]\!], \ldots, [\![S_k]\!])$ and the set of names N. Observe that this definition of compositionality accounts for all operators in the source language. Compositionality with respect to some operator has been assumed before to prove separation results [CCAV08, CCP07]. Indeed, for proving separation the most widely accepted criterion is a homomorphism of parallel composition [CM03, HMP08, Pal03, PSVV06, PV04, PV06], however this has been criticised and there are encodings that do not translate parallel composition homomorphically [BPV03, BPV05, BG07, Nes00]. As all process calculi in this dissertation share a common parallel composition operation, all valid encodings can be assumed *semi-homomorphic* [GGJ10]. That is; the encoding of parallel composition is through a context of the form

$(\nu\widetilde{n})(R \mid \cdot_1 \mid \cdot_2)$ for some restricted names $\widetilde{n}$ and process R that only depend on the free names of the translated processes.

Name invariance is required as the process calculi considered in this dissertation are name passing, so encodings cannot depend upon the names of a process being encoded. Thus, a substitution σ upon a process S should not change the behaviour of the corresponding substitution σ' applied to the encoding of that process $[\![S]\!]$ (taking into account the renaming policy $\varphi_{[\![]\!]}$).

As the focus of encodings is on the reduction or computational capabilities of the languages it follows that operational correspondence requires the source and target languages have the same reductions. This is captured by two aspects: *(i)* every reduction of the source can be mimicked by its translation *(ii)* every reduction of a translation corresponds to some reduction of its source. That is; translation does not prevent or introduce behaviours. The requirement that the target language only be behaviourally equivalent $\simeq_2$ allows for additional processes that are introduced by translation to be ignored as long as they do not introduce any new behaviour.

Divergence reflection ensures any translation does not introduce infinite reductions or computations, an important semantic issue [CM03, dBP94, Her91, Nes00].

As the success process $\surd$ is used for recognising the successful result of some reductions it is necessary that this is preserved by a valid encoding. Hence, success sensitiveness is necessary to ensure the encoding does not prevent recognition of reaching certain states.

Some results concerning valid encodings can be found in "Towards a unified approach to encodability and separation results for process calculi" [Gor08b]. In particular, proof techniques for showing separation results, i.e. for proving that no valid encoding can exist between a pair of languages $\mathcal{L}_1$ and $\mathcal{L}_2$ satisfying certain conditions. The following proposition and theorems are of particular use later in Chapters 7 & 9 and so restated here.

Below is one result of operational correspondence: if the source process has no reductions then its translation must have no reductions.

Proposition 3.2.2 (Gorla [Gor08b]). *Let $[\![\cdot]\!]$ be a valid encoding; then, $S \not\mapsto_1$ implies that $[\![S]\!] \not\mapsto_2$.*

The following theorem asserts that if an $\mathcal{L}_1$ process S does not reduce alone but reports success in parallel with itself, then there can be no valid encoding into a language $\mathcal{L}_2$ when no $\mathcal{L}_2$ process that does not reduce alone reduces in parallel with itself.

Theorem 3.2.3 (Gorla [Gor08b], updated by Given-Wilson, Gorla & Jay [GGJ10]). *Assume that there exists S such that $S \not\mapsto_1$ and $S \not\Downarrow_1$ and $S \mid S \Downarrow_1$; moreover, assume that every T that does not reduce is such that $T \mid T \not\mapsto_2$. Then, there cannot exist any semi-homomorphic valid encoding of $\mathcal{L}_1$ into $\mathcal{L}_2$.*

Proof:[Sketch] The proof is by contradiction. Assume there is such a process $[\![S]\!]$. Then by Proposition 3.2.2 and success sensitiveness $[\![S]\!] \not\mapsto_2$ and $[\![S]\!] \not\Downarrow_2$, respectively. With some further work it can be shown that if

$[\![S \mid S]\!] \Downarrow_2$ then $[\![S]\!] \mid [\![S]\!] \Downarrow_2$ which in turn implies $[\![S]\!] \mid [\![S]\!] \longmapsto_2$. Yet this contradicts the properties of all processes T of the target language. $\square$

The following proof technique exploits the number of names that can be tested for equality in a single interaction. Define the *matching degree* of a language $\text{MD}(\mathcal{L})$ to be the least upper bound on the number of names that must be matched to yield a reduction. It follows that a language $\mathcal{L}_1$ cannot be encoded into a language $\mathcal{L}_2$ if the matching degree of $\mathcal{L}_1$ is greater than that of $\mathcal{L}_2$, i.e. $\text{MD}(\mathcal{L}_1) > \text{MD}(\mathcal{L}_2)$.

Theorem 3.2.4 (from [Gor08b])**.** *If* $\text{MD}(\mathcal{L}_1) > \text{MD}(\mathcal{L}_2)$*, then there exists no valid encoding of* $\mathcal{L}_1$ *into* $\mathcal{L}_2$*.*

3.3 Linda

In his work on valid encodings Gorla presents a hierarchy of sets of process calculi relating their expressive power [Gor08a, Gor08b]. Four sets of equally-expressive process calculi sit at the top of the hierarchy. As this dissertation is focused upon expressive power it is natural to address these sets of calculi. Gelernter's Linda [Gel85, YFY96, PMR99] is an example of one such calculus that sits within a set with many differences to π-calculus. Thus Linda is both an exemplar from Gorla's work and distinct from other calculi presented here.

An instance of Linda can be defined by:

$$\begin{array}{rcl} t & ::= & \lambda x \mid \ulcorner n \urcorner \\ P & ::= & \mathbf{0} \mid P \,|\, P \mid\, !P \mid (\nu a)P \mid (t_1, \ldots, t_k).P \mid \langle n_1, \ldots, n_k \rangle \mid \surd \end{array}$$

exploiting names as in π-calculus. The *templates* t are either a *binding name* λx used for input or a *protected name* $\ulcorner n \urcorner$ used to test name equality. The null process, parallel composition, replication and restriction are as in π-calculus. The input is a k-tuple of templates and a body process P. As Linda is asynchronous the output is simply a k-tuple of names without any body process. Again the success process $\surd$ is added for use in encodings.

A *well formed* input is linear with respect to binding names. This rules out $(\lambda x, \lambda x)$ but accepts $(\lambda x, \ulcorner n \urcorner, \ulcorner n \urcorner)$. Only well formed inputs will be considered.

The free names of a template are given by

$$\begin{array}{rcl} \mathsf{fn}(\lambda x) & = & \{\} \\ \mathsf{fn}(\ulcorner n \urcorner) & = & \{n\} \end{array}$$

and the *binding names* by

$$\begin{array}{rcl} \mathsf{bn}(\lambda x) & = & \{x\} \\ \mathsf{bn}(\ulcorner n \urcorner) & = & \{\} \,. \end{array}$$

The free names and binding names of a tuple are as expected: the union of the free and binding names of their components, respectively. The free names of a process are the same for π-calculus with the following expansions for input and output

$$\begin{aligned} \mathsf{fn}((t_1,\ldots,t_k).P) &= (\mathsf{fn}(P)\backslash\mathsf{bn}(t_1,\ldots,t_k)) \cup \mathsf{fn}(t_1,\ldots,t_k) \\ \mathsf{fn}(\langle n_1,\ldots,n_k\rangle) &= \mathsf{fn}(\langle n_1,\ldots,n_k\rangle)\ . \end{aligned}$$

Substitutions are as in the π-calculus, i.e. a partial function from names to names. The application of a substitution to a name or process is as expected with the following rules for templates

$$\begin{aligned} \sigma(\lambda x) &= \lambda x \\ \sigma(\ulcorner x \urcorner) &= \ulcorner n \urcorner \qquad \text{if } \sigma \text{ maps } x \text{ to } n \\ \sigma(\ulcorner x \urcorner) &= \ulcorner x \urcorner \qquad \text{if } x \notin \mathsf{dom}(\sigma). \end{aligned}$$

Templates are used to implement Linda's pattern matching. The match of a template $\widetilde{t}$ against an tuple of names $\widetilde{n}$ is defined as follows

$$\textsc{Match}(\ ;\ ) = \{\} \qquad \textsc{Match}(\lambda x; n) = \{n/x\} \qquad \textsc{Match}(\ulcorner n \urcorner; n) = \{\}$$

$$\frac{\textsc{Match}(t; n) = \sigma_1 \qquad \textsc{Match}(\widetilde{t}; \widetilde{n}) = \sigma_2}{\textsc{Match}(t, \widetilde{t}\,;\ n, \widetilde{n}) = \sigma_1 \uplus \sigma_2}$$

where $\widetilde{e}$ denotes a (possibly empty) sequence of entities of kind e (names or template fields in this case). The match of empty tuples is the empty

substitution. The match of a binding name λx against a name n creates a substitution that maps x to n. The match of a protected name $\ulcorner n \urcorner$ against the name m is the empty substitution when $n = m$ and undefined otherwise. Here $\uplus$ denotes the union of partial functions with disjoin domain.

Observe that the match seeks a substitution whose domain is the binding names of the template and that also ensures equality of protected names in the template with names in the output. Also note that the match is only defined when the arities of the template and output are equal and all protected names are matched against equal names.

Linda's sole *reduction rule* is given by

$$(\widetilde{t}).P \mid \langle \widetilde{n} \rangle \;\longmapsto\; \sigma P \qquad \text{if } \textsc{Match}(\widetilde{t}; \widetilde{n}) = \sigma \ .$$

It states that if the match of the template $\widetilde{t}$ against the tuple $\widetilde{n}$ is defined and generates a substitution σ then apply the substitution to the body P.

The α-conversion, structural equivalence relation and reduction relation are straight forward in the style of the π-calculus. The barbed congruence for Linda is similar, except that the barbs are defined with a tuple of names $P \downarrow_{(\widetilde{n})}$ rather than a single name [BGZZ98]. This is to account for the multiple names that can be tested for equality by Match in reduction.

The relationship between π-calculus and Linda has been formalised by Gorla, thus the following are corollaries to his results [Gor08a].

Corollary 3.3.1. *There is a homomorphism from π-calculus into Linda.*

Proof:[Sketch] Exploit the translations of Gorla [Gor08a] that all meet the criteria for homomorphisms here. □

Corollary 3.3.2. *There is no homomorphism from Linda into π-calculus.*

Proof:[Sketch] Again exploit the results of Gorla [Gor08a] to show there cannot be a valid encoding or a homomorphism, for example the matching degree of π-calculus is one while the matching degree of Linda is infinite. □

Further, the homomorphism from π-calculus into Linda can be exploited to show that there is a parallel encoding from λ_v-calculus into Linda.

Corollary 3.3.3. *There is a parallel encoding from λ_v-calculus into Linda.*

Proof:[Sketch] Exploit the parallel encoding from λ_v-calculus into π-calculus and then the homomorphism from π-calculus into Linda, Corollaries 3.1.2 & 3.3.1 respectively. □

3.4 Intensionality in Spi Calculus

As this dissertation explores intensionality in computation it is natural to consider process calculi that already support arbitrary structures. Gordon et al.'s Spi calculus supports a variety of structured *terms* and intensional reductions based upon these structures [GA97]. However, in Spi calculus these reductions are limited to within a process and not part of communication. The rest of this section overviews the Spi calculus as a process calculus supporting intensionality, albeit in a limited manner.

The Spi calculus is distinct from the other calculi presented in this chapter as the names are now generalised to *terms* of the form

$$M, N \quad ::= \quad n \;\mid\; x \;\mid\; (M, N) \;\mid\; 0 \;\mid\; suc(M) \;\mid\; \{M\}_N \; .$$

The names n are as in π-calculus and Linda, and are intended for labelling channels. The variables x are familiar from λ-calculus. The pair (M, N) support the combination of any two component terms into a single term form. The zero 0 and successor $suc(M)$ are used to construct the natural numbers. The encryption $\{M\}_N$ encrypts the term M using the term N as the key.

Although the names and variables are separated for conceptual reasons in the original work, there is no impact on intensionality so the differences will be elided here.

Of particular structural interest are the pair and encryption terms that can be bound to a single name (or variable) and have internal structure. Although the pairing may appear similar to the tupling of polyadic calculi they are distinct as pairs may be bound to a single name while tuples cannot. The structural difference is clearer for the encryption that has no apparent analogue.

The processes of the Spi calculus are

$$\begin{aligned} P \ ::= \quad & 0 \mid P \,|\, P \mid !P \mid (\nu m)P \mid \surd \mid M(x).P \mid \overline{M}\langle N\rangle.P \\ & \mid [M \ is \ N]P \mid let \ (x,y) = M \ in \ P \\ & \mid case \ M \ of \ \{x\}_N : P \mid case \ M \ of \ 0 : P \ suc(x) : P \ . \end{aligned}$$

The null process, parallel composition, replication, restriction and success process are all familiar from the π-calculus and Linda. The input $M(x).P$ and output $\overline{M}\langle N\rangle.P$ are generalised from their π-calculus analogues to allow terms in the place of channel names and output. The *(Spi) match* $[M \ is \ N]P$ determines structural equality of M and N. The *pair splitting let* $(x, y) = M \ in \ P$ decomposes pairs and binds the components to x and y respectively in P. The *(Spi decryption) case case* $M \ of \ \{x\}_N : P$ decrypts M binding the encrypted message to x. The *integer case case* $M \ of \ 0 : P \ suc(x) : Q$ branches according to the number M.

The free names of terms and processes are expected: unions of components for all the term forms; and with the names being bound excluded from the restriction, input, pair splitting, decryption case and integer case.

Substitutions in the Spi calculus are partial functions from names to terms and otherwise as expected.

Concerning the operational semantics, consider the general form where

communication is given by the rule

$$M(x).P \mid \overline{M}\langle N\rangle.Q \quad \longmapsto \quad \{N/x\}P \mid Q$$

where M is any term of the Spi calculus. Further, the Spi calculus has several axioms that occur within a process and it is here that intensionality appears:

$$\begin{aligned}
[M \ is \ M]P &\longmapsto P \\
let \ (x,y) = (M,N) \ in \ P &\longmapsto \{M/x, N/y\}P \\
case \ \{M\}_N \ of \ \{x\}_N : P &\longmapsto \{M/x\}P \\
case \ 0 \ of \ 0 : P \ suc(x) : Q &\longmapsto P \\
case \ (suc(N)) \ of \ 0 : P \ suc(x) : Q &\longmapsto \{N/x\}Q \ .
\end{aligned}$$

The match process $[M \ is \ N]P$ reduces when the two terms M and N are the same, naturally this requires that they have the same internal structure. The pair splitting process $let \ (x,y) = M \ in \ P$ reduces when M is a pair and so is intensional with respect to M. Similarly the decryption case process form $case \ M \ of \ \{x\}_N : P$ reduces when M is some message M encrypted with the key N. Here intensionality can be used to recognise the encrypted structure of M before decrypting with N. The integer case process $case \ M \ of \ 0 : P \ suc(x) : Q$ has two reduction rules according to the structure of M. If M is zero then reduce to P. If M is of the form $suc(N)$ then bind x to N in Q. Again intensionality is used to examine the structure of the term to determine how reduction should proceed.

The α-conversion, structural equivalence relation and reduction relation are straight forward in the usual manner as for π-calculus or Linda. The barbed congruence for Spi calculus is similar to π-calculus [AG97].

Clearly the Spi calculus allows the binding of an arbitrary term to a name and has axioms that are intensional. Despite this there are translations of the Spi calculus into process calculi captured here, such as the π-calculus [BPV03, BPV05]. However, these translations do not meet the criteria for valid encodings, specifically the translation of a process may reduce when the original process does not thus violating Proposition 3.2.2.

Showing there is a parallel encoding from λ_v-calculus into Spi calculus is trivial as all the primitives required in π-calculus are in Spi calculus.

Theorem 3.4.1. *There is a parallel encoding from λ_v-calculus into Spi calculus.*

Proof: Exploit the parallel encoding from λ_v-calculus into π-calculus as in Corollary 3.1.2. □

The homomorphism from π-calculus into Spi calculus is similarly trivial.

Theorem 3.4.2. *There is a homomorphism from π-calculus into Spi calculus.*

Proof: There is a trivial translation that maps π-calculus to Spi calculus and is homomorphic upon all syntax. This meets the criteria for valid encoding and thus homomorphism. □

3.5 Exchange in Fusion Calculus

All of the process calculi above have explicit input and output actions, however process calculi can support information *exchange* where both input and output can be performed by the same primitive in an atomic reduction. Most process calculi have explicit input and output similar to the explicit function (input) and argument (output) of sequential computation. However, in concurrency there is much more symmetry between processes. This ranges from Milner's observation [Mil99] that even an asynchronous reduction is an interaction with the environment (an application of Newton's third law [New87]), to Parrow & Victor's fusion calculus where interaction applies the resulting substitution to both processes (and more) [PV98, BBM04]. This section considers information exchange by recalling Parrow & Victor's fusion calculus.

The processes of fusion calculus are

$$P \ ::= \ \mathbf{0} \ \mid \ P \,|\, P \ \mid \ (\nu x)P \ \mid \ !P \ \mid \ u(\widetilde{x}).P \ \mid \ \overline{u}\langle\widetilde{x}\rangle.P \ \mid \ \surd$$

that exploit a class of names as in π-calculus. The null process, parallel composition, restriction, replication, input, output and success process are all as for the (polyadic) π-calculus.

The free names, substitutions, α-conversion and structural equivalence relation are all the same as for π-calculus accounting for polyadicity.

The difference appears in the reduction axiom that is given by

$$u(x_1, \ldots, x_k).P \mid \overline{u}\langle y_1, \ldots, y_k\rangle.Q \quad \overset{\{x_1=y_1,\ldots,x_k=y_k\}}{\Leftrightarrow\!\!\!\!\rightarrow} \quad P \mid Q \ .$$

Here the input and output primitives *fuse* their names and continue with the processes P and Q. Interestingly, this fusing of names applies to the entire environment and may effect other processes. This is more clearly captured by the notation of Wischik and Gardner [WG05]

$$(\nu\widetilde{u})(u(\widetilde{x}).P \mid \overline{u}\langle\widetilde{y}\rangle.Q \mid R) \longmapsto \sigma P \mid \sigma Q \mid \sigma R \qquad \begin{array}{l} \mathsf{dom}(\sigma) \cup \mathsf{ran}(\sigma) \subseteq \{\widetilde{x}, \widetilde{y}\} \\ \text{and } \widetilde{u} = \mathsf{dom}(\sigma) \setminus \mathsf{ran}(\sigma) \\ \text{and } \sigma(v) = \sigma(w) \\ \text{iff } (v, w) \in E(\widetilde{x} = \widetilde{y}) \end{array}$$

where $E(\widetilde{x} = \widetilde{y})$ is the least equivalence relation on names generated by the equalities $\widetilde{x} = \widetilde{y}$ that is defined whenever the tuples have the same arity ($|\widetilde{x}| = |\widetilde{y}|$). Expressed in this manner the fusion calculus reduction relation is obtained by closing under parallel composition, restriction and the structural equivalence as per usual.

The reduction relation then follows as usual with the reference behavioural equivalence available in the literature [WG04]; again the details shall not be revisited here.

As the names are fused and the result applicable to both processes the interaction is conceptually symmetric. Indeed Parrow & Victor noted that

"input" and "output" are chosen for familiarity while "action" and "coaction" may be more appropriate [PV98, p. 177]. In addition to this fusing of names, the fusion calculus is also synchronous and polyadic. Thus both processes gain all the information, input or output, of processes they communicate with.

The peculiarity of name fusion is such that relating fusion calculus to other process calculi is non-trivial. Due to this, the formal relationship of fusion calculus to other process calculi shall be withheld until Chapter 9.

Regarding the subsumption of λ-calculus by fusion calculus, Parrow & Victor formalise both a generalisation of π-calculus and an encoding of strong lazy λ-calculus in their paper "The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes (extended abstract)" [PV98]. Thus, fusion calculus subsumes λ_v-calculus indirectly via the parallel encoding into π-calculus and other relations are straightforward.

This completes the background material introducing concurrent computation. The next chapter returns to the top of the computation square to consider intensionality.

Chapter 4

Sequential Pattern-Matching

Returning to the top of the computation square considers the introduction of intensionality to sequential computation. Recent work by Jay and others on *pure pattern calculus* [Jay04, JK06, GW07, JK09, Jay09] bases reduction upon pattern-matching. In addition to the extensional functions of λ-calculus, pure pattern calculus supports *intensional* functions that examine the internal structure of their arguments [JGW11]. The intensionality of pattern calculus is limited to *data structures*, a subset of terms that are headed by a *matchable symbol*. Intuitively, the intensionality of pure pattern calculus is more expressive than purely extensional models such as λ-calculus, however the relationship is not so clear. Although there is a trivial homomorphism from λ-calculus into pure pattern calculus, the inability to define pure pattern calculus within λ-calculus has yet to be formalised.

This chapter introduces pattern-matching as a basis for defining intensionality in sequential computation using pure pattern calculus.

4.1 Syntax

The terms of the pure pattern calculus are

$$t ::= x \mid \hat{x} \mid t\ t \mid [\theta]t \to t\ .$$

The *variable symbols* x are similar to the variables of λ-calculus. The *matchable symbols* $\hat{x}$ serve to build data structures and as binding symbols in patterns. Application is familiar from λ-calculus and SK-calculus in Chapter 2. Cases $[\theta]p \to s$ have a sequence of *binding symbols* θ, a *pattern* p and *body* s. The binding symbols θ are used to denote which matchable symbols in the pattern are data structures and which are binding (bound in pattern matching, detailed later in this chapter). A case is *well formed* if each binding symbol appears only once in θ. All cases appearing in the rest of this dissertation are assumed well formed.

The *free variable symbols* (or free variables) of a term $\mathsf{fv}(t)$ are given by

$$\begin{aligned} \mathsf{fv}(x) &= \{x\} \\ \mathsf{fv}(\hat{x}) &= \{\} \\ \mathsf{fv}(s\ t) &= \mathsf{fv}(s) \cup \mathsf{fv}(t) \\ \mathsf{fv}([\theta]p \to s) &= \mathsf{fv}(p) \cup (\mathsf{fv}(s) \backslash \theta)\ . \end{aligned}$$

A variable symbol is free in itself. A matchable symbol is not free. Free variable symbols of applications are the union of the free variable symbols of the function and argument. The free variable symbols of a case are the

free variable symbols of the body excluding those bound in θ, union the free variable symbols of the pattern. Note that the binding symbols of a case bind the body only and not the pattern.

The *free matchable symbols* of a term $\mathsf{fm}(t)$ are

$$\begin{aligned} \mathsf{fm}(x) &= \{\} \\ \mathsf{fm}(\hat{x}) &= \{x\} \\ \mathsf{fm}(s\ t) &= \mathsf{fm}(s) \cup \mathsf{fm}(t) \\ \mathsf{fm}([\theta]p \to s) &= \mathsf{fm}(s) \cup (\mathsf{fm}(p) \backslash \theta)\ . \end{aligned}$$

The free matchable symbols of a variable symbol is empty. The free matchable symbols of a matchable symbol are itself. The free matchables of an application is the union of the free matchables of the function and argument. The free matchable symbols of a case are the free matchables of the pattern excluding the binding symbols, union the free matchables of the body.

A substitution is a partial function from variable symbols to terms as for the λ-calculus. The application of a substitution to a term is defined by

$$\begin{aligned} \sigma x &= u && \text{if } \sigma \text{ maps } x \text{ to } u \\ \sigma x &= x && \text{if } x \notin \mathsf{dom}(\sigma) \\ \sigma \hat{x} &= \hat{x} \\ \sigma(s\ t) &= (\sigma s)\ (\sigma t) \\ \sigma([\theta]p \to t) &= [\theta](\sigma p) \to (\sigma t) && \text{if } \sigma \text{ avoids } \theta\ . \end{aligned}$$

If a variable symbol is in the domain of the substitution then apply the map-

ping of the substitution, otherwise do nothing. Substitutions have no effect on matchable symbols. Apply a substitution to the function and argument of applications. A substitution is applied to the pattern and body of a case as long as the substitution avoids the binding symbols θ.

The action $\hat{\sigma}$ of a substitution σ on matchable symbols can be defined as follows

$$\begin{aligned}
\hat{\sigma}x &= x \\
\hat{\sigma}\hat{x} &= u && \text{if } \hat{\sigma} \text{ maps } x \text{ to } u \\
\hat{\sigma}\hat{x} &= \hat{x} && \text{if } x \notin \mathsf{dom}(\hat{\sigma}) \\
\hat{\sigma}(s\ t) &= (\hat{\sigma}s)\ (\hat{\sigma}t) \\
\hat{\sigma}([\theta]p \to t) &= [\theta](\hat{\sigma}p) \to (\hat{\sigma}t) && \text{if } \hat{\sigma} \text{ avoids } \theta\ .
\end{aligned}$$

The behaviour is the same as normal substitution except operating on matchable symbols rather than variable symbols. When σ is of the form $\{u_i/x_i\}$ then $\{u_i/\hat{x}_i\}$ may be used to denote $\hat{\sigma}$.

One of the foci of the pure pattern calculus is the identification of *data structures*. Data structures are terms headed by a matchable symbol defined as follows

$$d ::= \hat{x} \mid d\ t\ .$$

Pattern-matching is based on a notion of *matchable form*, a subset of the terms that are stable under substitution and reduction. In pure pattern calculus these are given by the data structures and cases

$$m ::= d \mid [\theta]t \to t\ .$$

A *compound* is a matchable form that is also an application; all other matchable forms are *atoms*, i.e. matchable symbols and cases.

4.2 Pattern-Matching

The *match* of a pattern p against an argument u with respect to binding symbols θ is either a *successful match* some of a substitution σ or *match failure* none. The disjoint union $\uplus$ of two matches is: undefined if either match is undefined; the disjoint union of the substitutions if both are some; and none otherwise. The algorithm for matching $\{u/\!\!/[\theta]p\}$ is given by

$$
\begin{array}{rcll}
\{u/\!\!/[\theta]\hat{x}\} & = & \mathsf{some}\ \{u/x\} & x \in \theta \\
\{\hat{x}/\!\!/[\theta]\hat{x}\} & = & \mathsf{some}\ \{\} & x \notin \theta \\
\{u\ v/\!\!/[\theta]p\ q\} & = & \{u/\!\!/[\theta]p\} \uplus \{v/\!\!/[\theta]q\} & \text{if } p\ q \text{ and } u\ v \text{ are matchable forms} \\
\{u/\!\!/[\theta]p\} & = & \mathsf{none}\ \text{otherwise} & \text{if } u, p \text{ are matchable forms} \\
\{u/\!\!/[\theta]p\} & = & \text{undefined} & \text{otherwise.}
\end{array}
$$

The match of a binding symbol $\hat{x}$ (with $x \in \theta$) against any argument u is some of the substitution $\{u/x\}$. The match of a matchable symbol $\hat{x}$ (with $x \notin \theta$) against itself is some of the empty substitution $\{\}$. Matches of applications are done component wise when both pattern and argument are matchable forms, with the disjoint union of the results. Otherwise if the argument is a matchable form then the result is match failure none. If none of these apply then the argument is not a matchable form and so suspend the matching

until later.

As reduction of a pattern may eliminate binding symbols it is necessary to ensure that successful matches account for all the binding symbols of the pattern. The *check* of a defined match μ on a set of binding symbols θ is μ if μ is $\mathsf{some}\ \sigma$ when the domain of σ is exactly θ and none otherwise.

The *matching* $\{u/[\theta]p\}$ of a pattern p against an argument u with respect to binding names θ is the check of the match $\{u/\!\!/[\theta]p\}$. The application of the result of matching to a term is given by

$$\begin{aligned} \mathsf{some}\ \sigma\ t &\longrightarrow \sigma t \\ \mathsf{none}\ t &\longrightarrow [x]\hat{x} \to x\ . \end{aligned}$$

If the result of a matching is some substitution then apply the substitution to the term, otherwise reduce to the identity function. The choice of identity function when matching fails is to support *extensions* (pattern-matching functions that have many cases, discussed below and in Section 5.4) and typing (details in "Pattern Calculus: Computing with Functions and Data Structures" [Jay09]).

4.3 Operational Semantics

As substitutions and reductions may require renaming of symbols as in λ-calculus, renaming through α-conversion is defined by

$$[\theta]p \to s =_\alpha [\{y/x\}\theta]\{\hat{y}/\hat{x}\}p \to \{y/x\}s \quad y \notin \mathsf{fm}(p) \cup \mathsf{fv}(s) \ .$$

Where $\{y/x\}\theta$ acts as expected by replacing x with y in θ. Note that the symbol being renamed is a matchable symbol in the pattern p, and a variable symbol in both the bindings θ and body s.

Reduction in the pure pattern calculus is by the *match rule*

$$([\theta]p \to s)\ u \longrightarrow \{u/[\theta]p\}s$$

whenever the matching is defined. Observe that the match rule is similar to the β-reduction of λ-calculus except that there may be patterns or arguments that never reduce to a matchable form. Despite this, pure pattern calculus is progressive and every closed irreducible term is a matchable form [Jay09, p. 50].

The reduction relation (also denoted $\longrightarrow$) is the relation obtained by applying an instantiation of a reduction rule to some sub-expression. The transitive closure of the reduction relation is denoted $\longrightarrow^*$.

There is a translation from λ-calculus to pure pattern calculus defined as

follows

$$\begin{aligned} [\![x]\!] &= x \\ [\![s\ t]\!] &= [\![s]\!]\ [\![t]\!] \\ [\![\lambda x.t]\!] &= [x]\hat{x} \to [\![t]\!]\ . \end{aligned}$$

The only interesting translation is the λ-abstraction $\lambda x.t$ which becomes a case of the form $[x]\hat{x} \to [\![t]\!]$.

Theorem 4.3.1. *There translation from λ-calculus to pure pattern calculus preserves reduction.*

Proof: The only interesting translation is of abstractions as the variables and applications are direct. Consider the translation of an abstraction

$$[\![\lambda x.t]\!] = [x]\hat{x} \to [\![t]\!]\ .$$

The pattern $\hat{x}$ with binding symbol $[x]$ will always match and always pass the check. Thus $([x]\hat{x} \to [\![t]\!])[\![s]\!]$ will always reduce by $\{[\![s]\!]/x\}[\![t]\!]$ and the rest is straightforward. Thus reduction is preserved. □

Corollary 4.3.2. *There is a homomorphism from λ-calculus to pure pattern calculus*

Proof: Application is preserved by the translation and reduction by Theorem 4.3.1. □

Given a sequence of cases $[\theta_i]p_i \to s_i$ the *pattern-matching function*

$$\begin{aligned} & [\theta_1]p_1 \to s_1 \\ \mid\ & [\theta_2]p_2 \to s_2 \\ & \ldots \\ \mid\ & [\theta_n]p_n \to s_n \end{aligned}$$

is specified as follows. When applied to some argument, it reduces to the first case $[\theta_i]p_i \to s_i$ where matching succeeds, or to match failure if none succeed. These can be represented as cases using *extensions* [Jay04, Jay09].

Using extensions the function

$$\begin{aligned} & [x, y]\hat{x}\ \hat{y} \to \texttt{True} \\ \mid\ & [z]\hat{z} \to \texttt{False} \end{aligned}$$

reduces to `True` when applied to a compound and `False` when applied to anything else. Indeed such a function is intensional with respect to its argument and can distinguish $\hat{c}\ s$ from $\hat{c}$ or $[\theta]p \to t$. Indeed, these can be exploited to develop generic queries that traverse arbitrary data structures [Jay09]. However, the intensionality is limited to data structures and patterns cannot be used to match or decompose cases.

Intuitively pure pattern calculus supports intensionality that cannot be represented within λ-calculus. However, the limitations of intensionality on data structures and the subtleties of managing both variable and matchable

symbols in translations make formalising this intuition difficult. A more direct approach is to consider intensionality within the combinatory logic setting.

Notes. A detailed account of pure pattern calculus, including standard properties, relation to other pattern calculi, and typing, can be found in "Pattern Calculus: Computing with Functions and Data Structures" [Jay09].

Chapter 5

Intensional Sequential Computation

Intuitively intensional functions are more expressive than merely extensional functions, however populating the top right corner of the computation square requires more formality than intuition. As the relations between λ-calculus and pure pattern calculus are complicated by the subtleties of symbols (variable and matchable), a cleaner account is by considering combinatory logic.

Even in SK-calculus there are Turing-computable functions defined upon the combinators that cannot be represented within SK-calculus. For example, consider the function that reduces any combinator of the form SKX to X. Such a function cannot be represented in SK-calculus, or λ-calculus, as all combinators of the form SKX represent the identity function. However, such a function is Turing-computable and definable upon the combinators. This is an example of a more general problem of *factorising* combinators that

are applications and stable under reduction.

Exploiting this factorisation is SF-logic (or SF-calculus) [JGW11] that is able to support intensional functions on combinators including a structural equality of normal forms. Thus SF-calculus sits at the top right hand corner of the computation square. The arrow across the top of the square is formalised by showing a homomorphism from SK-calculus into SF-calculus. The lack of a converse is proven by showing that the intensionality of SF-calculus cannot be represented within SK-calculus, or λ-calculus.

Although this completes the top edge of the computation square, there are also stylistic links to the pattern-matching of pure pattern calculus, and more general results about combinatory logics. The intensionality supported by factorisation can be characterised by pattern-matching in the style of pure pattern calculus and used to define a new property, namely *structure completeness* [JGW11]. This completeness subsumes combinatorial completeness and captures a notion of symbolic computation on normal forms [JGW11].

This chapter introduces symbolic functions on combinators and proves there are intensional Turing-computable functions that cannot be represented within λ-calculus. A full development of SF-calculus is presented to illustrate the concepts before other factorisation logics are discussed. Pattern-matching in the combinatory setting is presented as a basis for structure completeness.

5.1 Symbolic Functions

Symbolic functions, as presented in Section 2.2, need not be merely extensional, indeed it is possible to define symbolic functions that consider the structure of their arguments. Define a symbolic function $\mathcal{R}$ by

$$\begin{aligned} \mathcal{R}(O, M, N) &= M \\ \mathcal{R}(PQ, M, N) &= NPQ\,. \end{aligned}$$

Here $\mathcal{R}$ branches according to its first argument. Of course $\mathcal{R}$ does not respect equality since applications may reduce to operators.

One approach to resolving this would be to modify the reduction relation, as in Kearns' system of *discriminators* [Kea69, Kea73] which includes a discriminator similar to $\mathcal{R}$. Although this preserves a weakened motion of reduction, the equivalence relation is not an equality in the sense of Leibnitz, which allows the substitution of equals for equals.

A better approach is to restrict $\mathcal{R}$ to partially applied operators. Each operator O has an *arity* given by the minimum number of arguments it requires to instantiate a rule. Thus, K has arity 2 while S has arity 3. A *partially applied operator* is a combinator of the form $OX_1 \ldots X_k$ where k is less than the arity of O. An operator with a positive arity is an *atom* (meta-variable A). A partially applied operator that is an application is a *compound*. Hence, the partially applied operators of SK-calculus are the atoms S and K, and the compounds SM, SMN and KM for any M and

N.

Now define a *factorisation function* $\mathcal{F}$ on combinators by

$$\begin{array}{rcll} \mathcal{F}(A, M, N) & \longrightarrow & M & \text{if } A \text{ is an atom} \\ \mathcal{F}(PQ, M, N) & \longrightarrow & NPQ & \text{if } PQ \text{ is a compound.} \end{array}$$

Lemma 5.1.1. *If reduction is confluent then factorisation is a symbolic computation.*

Proof: To prove that $\mathcal{F}$ is a symbolic function, it suffices to prove that $\mathcal{F}(X, M, N) = \mathcal{F}(X', M', N')$ whenever both sides are defined and $X = X'$ and $M = M'$ and $N = N'$ are three pairs of combinators. If X is a compound PQ then, by confluence, X' must also be a compound $P'Q'$ such that $P = P'$ and $Q = Q'$. Thus, $\mathcal{F}(X, M, N) = NPQ = N'P'Q' = \mathcal{F}(X', M', N')$. Similarly, if X is an atom A then by confluence X' must also be A so that $\mathcal{F}(X, M, N) = M = M' = \mathcal{F}(X', M', N')$. Finally, $\mathcal{F}$ is computable by leftmost reduction of its first argument to a partially applied operator. □

Theorem 5.1.2. *Factorisation of SK-combinators is a symbolic computation that is not representable within SK-calculus.*

Proof: Suppose that there is an SK-combinator F that represents $\mathcal{F}$. Then, for any combinator X it follows that

$$F(SKX)S(KI) \longrightarrow KI(SK)X \longrightarrow X \ .$$

Translating this to λ-calculus as in Lemma 2.3.1 yields $[\![F(SKX)S(KI)]\!] \longrightarrow$

$[\![X]\!]$ and also

$$[\![F(SKX)S(KI)]\!] = [\![F]\!]\ [\![(SKX)]\!]\ [\![S]\!]\ [\![KI]\!] \longrightarrow [\![F]\!]\ (\lambda x.x)\ [\![S]\!]\ [\![KI]\!]\ .$$

Hence, by confluence of reduction in λ-calculus, all $[\![X]\!]$ share a reduct with $[\![F]\!]\ (\lambda x.x)\ [\![S]\!]\ [\![KI]\!]$ but this is impossible since $[\![S]\!]$ and $[\![K]\!]$ are distinct normal forms. Hence $\mathcal{F}$ cannot be represented by an SK-combinator. □

This result appears at odds with the traditional understanding of computation. It is straight forward to encode factorisation using a Turing machine, yet λ-calculus, SK-logic and Turing machines all compute the same things. The difficulty is resolved by observing that the classical theorems all concern numerical rather than symbolic computations. For example, Kleene [Kle52] states Church's thesis as:

> THESIS 1: Every effectively calculable function (effectively decidable predicate) is general recursive.

As general recursive functions are numerical by definition, the restriction to natural numbers, and things encoded as numbers through Gödelisation, is implicit.

A natural objection is that the proofs of encodings between various models are symbolic and so can be generalised. For example, that an SK-combinator can be encoded on the tape of a Turing machine that supports factorisation and this machine then expressed in SK-calculus. However, the catch is that the expression of factorisation does not imply its representation as a

combinator, indeed the encoding of the combinators onto the tape has already performed the factorisation.

For example, consider the factorisation of SKK. Using Polish notation, it can be encoded on a tape by $aaSKK$ where a represents application. Then expressing the tape as an SK combinator yields a list $[a, a, S, K, K]$ whose factorisation is a routine list operation. However, to produce this list within the calculus would require support for factorisation that SK-calculus does not possess (Theorem 5.1.2). In other words, the metamathematical process of encoding a combinator onto the tape of a Turing machine is not representable by an SK-combinator. Note that Gödelisation is not relevant here, being solely concerned with the representation of $[a, a, S, K, K]$ by a number.

5.2 SF-calculus

When considering intensionality in a combinatory logic it is tempting to specify a factorisation combinator F as a representative for $\mathcal{F}$. However, $\mathcal{F}$ is defined using partially applied operators, which cannot be known until all reduction rules are given, including those for F. This circularity of definition is broken by beginning with a syntactic characterisation of the combinators that are to be factorable; similar to the matchable forms of pure pattern calculus.

The SF-calculus [JGW11] has *factorable forms* given by

$$S \mid SM \mid SMN \mid F \mid FM \mid FMN$$

and *reduction rules*

$$\begin{array}{rcll} SMNX & \longrightarrow & MX(NX) & \\ FOMN & \longrightarrow & M & \text{if } O \text{ is } S \text{ or } F \\ F(PQ)MN & \longrightarrow & NPQ & \text{if } PQ \text{ is a factorable form.} \end{array}$$

Observe that these rules can also be expressed with each of the factorable forms as the first argument to F

$$\begin{array}{rcl} SMNX & \longrightarrow & MX(NX) \\ FSMN & \longrightarrow & M \\ F(SX)MN & \longrightarrow & NSX \\ F(SXY)MN & \longrightarrow & N(SX)Y \\ FFMN & \longrightarrow & M \\ F(FX)MN & \longrightarrow & NFX \\ F(FXY)MN & \longrightarrow & N(FX)Y \ . \end{array}$$

Lemma 5.2.1. *The partially applied operators of SF-calculus are its factorable forms. Hence F represents $\mathcal{F}$.*

Proof: Trivial. □

Theorem 5.2.2. *Reduction of SF-calculus is confluent.*

Proof: It is enough to observe that the reduction rules are orthogonal [Ros73, Hue80], since partially applied operators are stable under reduction. □

The expressive power of SF-calculus subsumes that of SK-calculus since K is here defined to be FF and I is defined to be SKK as before.

Theorem 5.2.3. *There is a homomorphism from SK-calculus into SF-calculus.*

Proof: Define K to be FF and the rest is trivial. □

Corollary 5.2.4. *There is a homomorphism from λ_v-calculus to SF-calculus.*

Proof: SF-calculus represents S and K and so there is a homomorphism from λ_v-calculus into SF-calculus by Theorem 2.3.2. □

Corollary 5.2.5. *There is a homomorphism from λ-calculus to SF-calculus.*

Proof: SF-calculus represents S and K and so there is a homomorphism from λ-calculus into SF-calculus by Theorem 2.3.2. □

Theorem 5.2.6. *There is no homomorphism from SF-calculus to λ-calculus.*

Proof: Straightforward by exploiting the ability of F to retrieve X from identity combinators of the form SKX and by Theorem 5.1.2. □

This completes the top edge of the computation square by showing that SF-calculus subsumes λ_v-calculus and that the subsumption is irreversible. Indeed, these results hold for SK-calculus and λ-calculus as well.

The rest of this chapter explores intensional sequential computation culminating in a new completeness result for combinatory logics.

SF-calculus' further power is illustrated by defining a combinator for structural equality of normal forms. In brief, the algorithm is as follows: if both normal forms are compounds then compare their corresponding components: if both are atoms then compare them directly using a combinator `eqatom`; otherwise the normal forms are not equal.

Clearly F is able to distinguish the atoms from the compounds. The combinator $\texttt{isComp} = \lambda^* x.Fx(KI)(K(KK))$ tests for being a compound since

$$\begin{aligned}
\texttt{isComp}\ O &\longrightarrow FO(KI)(K(KK)) \\
&\longrightarrow KI \\
\texttt{isComp}(PQ) &\longrightarrow F(PQ)(KI)(K(KK)) \\
&\longrightarrow K(KK)PQ \\
&\longrightarrow KKQ \\
&\longrightarrow K \qquad \text{if } PQ \text{ is a compound.}
\end{aligned}$$

Further, the first and second components of a factorable application can be

recovered by

$$\begin{aligned} \texttt{car} &= \lambda^* x.FxIK \\ \texttt{cdr} &= \lambda^* x.FxI(KI) \end{aligned}$$

whose names are taken from the corresponding Lisp [McC60] operators. Note that they map operators to I so it is normal to check for being a compound first.

Now all that remains is to separate the operators S and F. Define $\mathsf{is}(F)$ by

$$\mathsf{is}(F) = \lambda^* x.x(KI)(K(KI))K \ .$$

It maps F to K and S to KI as desired since

$$\begin{aligned} &\mathsf{is}(F)\ F \longrightarrow F(KI)(K(KI))K \longrightarrow KKI \longrightarrow K \\ &\mathsf{is}(F)\ S \longrightarrow S(KI)(K(KI))K \longrightarrow KIK(K(KI)K) \longrightarrow KI \ . \end{aligned}$$

Further, define equality of operators by

$$\texttt{eqatom} = \lambda^* x.\lambda^* y.\texttt{if}\ \mathsf{is}(F)x\ \texttt{then}\ \mathsf{is}(F)y\ \texttt{else}\ (\texttt{not}\ \mathsf{is}(F)y) \ .$$

Structural equality of normal forms can now be given by

$$
\begin{array}{l}
\texttt{equal} = \texttt{fix}(\lambda^* e.\lambda^* x.\lambda^* y. \\
\quad \texttt{if isComp } x \\
\quad \texttt{then if isComp } y \\
\qquad \texttt{then } (e\ (\texttt{car } x)\ (\texttt{car } y))\ \texttt{and}\ (e\ (\texttt{cdr } x)\ (\texttt{cdr } y)) \\
\qquad \texttt{else } KI \\
\quad \texttt{else if isComp } y \\
\qquad \texttt{then } KI \\
\qquad \texttt{else eqatom } x\ y\ .
\end{array}
$$

Theorem 5.2.7. *Let M and N be SF-combinators in normal form. If $M = N$ then* `equal` $M\ N \longrightarrow K$ *else* `equal` $M\ N \longrightarrow KI$.

Proof: The proof is by straightforward induction on the structure of M. □

Thus, the combinator `equal` represents the symbolic computation whose domain is given by pairs of combinators that have a normal form. Of course, `equal` also detects inequality of, say, an operator O and a factorable form PQ even if Q does not have a normal form.

The development above illustrates a more general result.

Theorem 5.2.8. *Any symbolic computation restricted to normal forms of SF-calculus is representable.*

Proof: The encoding of the normal form of the function argument can be revealed by first factorising into operators and then using `eqatom` to identify

the operator values. That done, the computation can be represented by a combination of S and K in the traditional manner. □

Head Normal Forms

In extensional combinatory calculi such as SK-calculus, the head normal forms are the partially applied operators, but this is not true of intensional calculi such as SF-calculus. This section defines head normal forms in this new setting, and shows that the definition of SF-calculus cannot be modified to force the factorable forms to be the head normal forms without making the logic unsound.

The *head normal forms* are defined by induction on their structure. An operator is head normal if it is irreducible. An application PQ is head normal if P is head normal and no reduct of PQ instantiates a reduction rule.

For SK-calculus the head normal forms are exactly the partially applied operators, but this is not true of SF-calculus as the combinator $\Omega = (SII)(SII)$ does not reduce to a factorable form, and so $F\Omega FF$ is a head normal form but not a partially applied operator.

Theorem 5.2.9. *Consider a combinatory calculus with two operators S and H. Given a collection of* headable forms *define the reduction rules as follows*

$$\begin{array}{rcll} SMNX & \longrightarrow & MX(NX) & \\ HOMN & \longrightarrow & M & O \text{ is } S \text{ or } H \\ H(PQ)MN & \longrightarrow & NPQ & PQ \text{ is headable.} \end{array}$$

Let $K = HH$ and $I = SKK$ as usual. If the head normal forms of this calculus are the headable forms then the resulting logic is unsound, i.e. $K = KI$.

Proof: The proof proceeds by using the decidability of head normality to obtain decidability of normality, which yields a contradiction. Consider the combinator

$$\mathtt{isn} = \mathtt{fix}(\lambda^* i.\lambda^* x.H(HxH(\lambda^* y.\lambda^* z.H(i\ y)(i\ z)I))\ S\ (K(K(SS)))) .$$

For all combinators X, the combinator $\mathtt{isn}\,X$ reduces to S if X is normalisable and to SS otherwise. The proof is by induction on the structure of X.

First consider the situation when X does not reduce to a head normal form or headable form. Then $(HXH(\lambda^* y.\lambda^* z.H(\mathtt{isn}\ y)(\mathtt{isn}\ z)I))$ is a head normal form, i.e. a headable form, and so

$$\begin{aligned} \mathtt{isn}\,X &\longrightarrow H(HXH(\lambda^* y.\lambda^* z.H(\mathtt{isn}\,y)(\mathtt{isn}\,z)I))\ S\ (K(K(SS)))) \\ &\longrightarrow K(K(SS))(HXH)(\lambda^* y.\lambda^* z.H(\mathtt{isn}\,y)(\mathtt{isn}\,z)I) \\ &\longrightarrow SS \end{aligned}$$

as required. If X is an atom A then

$$\begin{aligned} \mathtt{isn}\,A &\longrightarrow H(HAH(\lambda^* y.\lambda^* z.H(\mathtt{isn}\ y)(\mathtt{isn}\ z)I))\ S\ (K(K(SS)))) \\ &\longrightarrow HHS(K(K(SS))) \\ &\longrightarrow S\ . \end{aligned}$$

If X is a headable form PQ then

$$\begin{aligned}\mathtt{isn}(PQ) &\longrightarrow H(H(PQ)H(\lambda^*y.\lambda^*z.H(\mathtt{isn}\ y)(\mathtt{isn}\ z)I))\ S\ (K(K(SS))))\\ &\longrightarrow H((\lambda^*y.\lambda^*z.H(\mathtt{isn}\ y)(\mathtt{isn}\ z)I)PQ)\ S\ (K(K(SS))))\\ &\longrightarrow H(H(\mathtt{isn}\,P)(\mathtt{isn}\,Q)I)\ S\ (K(K(SS)))) \ .\end{aligned}$$

If P is not normalisable then

$$\begin{aligned}\mathtt{isn}(PQ) &\longrightarrow H(H(SS)(\mathtt{isn}\,Q)I)\ S\ (K(K(SS))))\\ &\longrightarrow H(ISS)\ S\ (K(K(SS))))\\ &\longrightarrow SS\end{aligned}$$

as required. If P is normalisable then

$$\begin{aligned}\mathtt{isn}(PQ) &\longrightarrow H(HS(\mathtt{isn}\,Q)I)\ S\ (K(K(SS))))\\ &\longrightarrow H(\mathtt{isn}\,Q)\ S\ (K(K(SS)))) \ .\end{aligned}$$

If Q is normalisable then this reduces to $HSS(K(K(SS))) \longrightarrow S$ while if Q is not normalisable then it reduces to $K(K(SS))SS \longrightarrow SS$, all as required.

This completes the proof of the properties of `isn`. Now it is routine to show that

$$\mathtt{isnormal} = \lambda^*x.H(\mathtt{isn}\ x)K(K(K(KI)))$$

decides whether its argument has a normal form. Finally, consider the para-

doxical combinator

$$\begin{aligned} \texttt{paradox} &= \texttt{fix}(\lambda^* f.\texttt{if isnormal } f \texttt{ then } \Omega \texttt{ else } K) \\ &= \texttt{if isnormal paradox then } \Omega \texttt{ else } K\ . \end{aligned}$$

If $\texttt{isnormal paradox} = KI$ then $\texttt{paradox} = K$ and so

$$KI = \texttt{isnormal paradox} = \texttt{isnormal } K = K$$

in which case the logic is unsound. Alternatively, if $\texttt{isnormal paradox} = K$ then $\texttt{paradox} = \Omega$ so that $K = \texttt{isnormal paradox} = \texttt{isnormal } \Omega = KI$. □

Related Combinatory Calculi

Now consider how factorisation can be exploited in the presence of different operators that define additional calculi. Each calculus to be developed includes a formal description of its factorable forms. It is then trivial to show that these are the partially applied operators and that reduction is confluent in the style of the corresponding proofs for SF-calculus. It is easy to confirm in each case that structural equality of normal forms is definable, once equality of atoms is supported, and that the analogue of Theorem 5.2.8 holds. Rather than do the proofs here, these results will follow from Corollaries 5.4.4 and 5.4.5.

Perhaps the closest calculus to SF-calculus is SKF-calculus where S, K and F take their usual meanings and the factorable forms are S, SM, SMN,

K, KM, F, FM and FMN.

Define:

$$\begin{aligned}
\mathsf{is}(K) &= \lambda x.F(xFF)K(K(K(KI))) \\
\mathsf{is}(F) &= \lambda x.x(KI)(K(KI))K \\
\texttt{eqatom} &= \lambda x.\lambda y.\texttt{if } \mathsf{is}(K)x \texttt{ then } \mathsf{is}(K)y \\
&\qquad \texttt{else if } \mathsf{is}(F)x \texttt{ then } ((\texttt{not } \mathsf{is}(K)y) \texttt{ and } \mathsf{is}(F)y) \\
&\qquad\qquad \texttt{else not } (\mathsf{is}(K)y \texttt{ or } \mathsf{is}(F)y) \ .
\end{aligned}$$

There is a trivial homomorphism of SF-calculus into SKF-calculus that maps S to S and F to F but it is not clear if there is a homomorphism in the opposite direction. The natural approach would be to map S to S and K to FF, but then F cannot be mapped to F since $FKMN$ reduces to M in SKF-calculus but to NFF in SF-calculus, so this will not do. Such problems will arise whenever an atom is translated to a compound, as when Schönfinkel's original combinators are represented as SK-combinators. Hence, it is not yet clear if there is a "best" combinatory logic, much less that SF-calculus is best.

Another way of extending SF-calculus is to add constructors in the style of the matchable symbols of pure pattern calculus. A *constructor* is an atom that does not appear at the head of any reduction rule, so that its arity is infinite. Typical examples are `Pair` for building pairs, or `Nil` for the empty list. Let $\mathcal{C}$ be a finite collection of constructors (meta-variable C).

Also required is an operator `eqatom` for deciding equality of atoms, since constructors are extensionally equal.

The constructors are used to build data structures as in pure pattern calculus:

$$d ::= C \mid d\ M\ .$$

The $SF\mathcal{C}$-calculus is then defined with factorable forms:

$$d \mid S \mid SM \mid SMN \mid F \mid FM \mid FMN \mid \texttt{eqatom} \mid \texttt{eqatom}\ M$$

which now include all the data structures. The reduction rules for S and F are as usual. The reduction rules for `eqatom` are

$$\begin{aligned} \texttt{eqatom}\ O\ O &\longrightarrow K \\ \texttt{eqatom}\ P\ Q &\longrightarrow KI \text{ otherwise, if } P \text{ and } Q \text{ are factorable.} \end{aligned}$$

Observe that any countable collection of constructors can be encoded as data structures built from a single, universal constructor C, as CC, $C(CC)$, $C(CCC)$ etc; exploiting this yields the SFC-calculus. There is no need for `eqatom` to be an operator, since it can be defined as follows. The combinator

$$\mathsf{is}(C) = \lambda^* x.F(x(FK)IK)(KI)(K(KK))$$

maps C to K and maps S and F to KI. Hence, `eqatom` can be defined by using $\mathsf{is}(C)$ first and then separating S and F as before.

Finally, data structures can be represented in a variant of SF-calculus in which $\Omega = (SII)(SII)$ plays the role of constructor. Define $\{S, F, \Omega\}$-calculus to have *data structures*

$$d ::= \Omega \mid d\ M$$

and *factorable forms*

$$d \mid S \mid SM \mid SMN \mid F \mid FM \mid FMN$$

and reduction rules

$$\begin{array}{rcll} SMNX & \longrightarrow & MX(NX) & \\ FOMN & \longrightarrow & M & \text{if } O \text{ is } S \text{ or } F \\ F(PQ)MN & \longrightarrow & NPQ & \text{if } PQ \text{ is factorable .} \end{array}$$

Confluence is established as for SF-calculus since the only reduct of Ω that is a factorable form is Ω itself. Again `eqatom` can be defined to distinguish S and F as before. In a sense, Ω is playing the role of an constructor, even though it is not a operator, or even a partially applied operator. Obviously, this approach can be generalised to include other non-normalising combinators as "constructors."

5.3 Combinatory Pattern-Matching

As intensionality in the combinatory setting is inspired from pattern calculi it is natural to define pattern-matching in the combinatory setting. Consider a confluent combinatory calculus whose *patterns* (meta-variable P) are its terms that are in normal form. From now on, limit attention to *linear patterns* in which no variable occurs twice, similar to the bindings in pure pattern calculus. While this may seem a little artificial, non-linear patterns describe structures that come with side-conditions about the equality of substructures; it is simpler and more natural to replace these side conditions with an explicit equality test.

A *case* $\mathcal{G}$ is given by an equation of the form

$$\mathcal{G}(P) = M$$

where P is a pattern and M is an arbitrary term. Such a case yields a symbolic function on terms given by pattern-matching, which is defined as follows.

When $\mathcal{G}$ is applied to some term U the pattern P will be matched against U to try and determine the values of the free variables in P so that these can be substituted into M. That is, matching seeks a substitution σ such that $\sigma P = U$. However, the presence or absence of such a substitution is not an infallible guide to evaluation.

If the equation $\sigma P = U$ is that of the logic then there can be many

substitutions. For example, consider that P is $x\ y$ and U is S. A naive interpretation would consider that matching must fail, however as $SKSS = S = SKKS$ it would be acceptable to match x against either SKS or SKK. Rather than take this course, it is more natural to develop a syntactic procedure for matching. In turn, this must respect reduction, which requires a notion of partially applied operator in a term calculus.

The *matchable forms* upon which matching acts can be identified with the partially applied operators of the term calculus on the understanding that variables have arity 0. It follows that variables are neither atoms nor appear at the head of a compound, which is appropriate since substitution may trigger new reductions

Define a *match* to be either a *successful match*, $\mathsf{some}\ \sigma$ where σ is a substitution, or a *match failure* none. *Match equality* is defined using term equality. Two successful matches $\mathsf{some}\ \sigma_1$ and $\mathsf{some}\ \sigma_2$ are equal if σ_1 and σ_2 have the same domain and $\sigma_1 x = \sigma_2 x$ for each variable x in their domain. Also none equals none. Otherwise, matches are not equal. Disjoint unions $\uplus$ of matches are as for pure pattern calculus. The application of a match to a term is also defined as in pure pattern calculus

$$\begin{aligned} \mathsf{some}\ \sigma\ M &= \sigma M \\ \mathsf{none}\ M &= I\ . \end{aligned}$$

For definiteness, match failure must produce a combinator; the identity

proves to be a useful choice when defining extensions as before.

The *match* $\{U/\!\!/\!\!/P\}$ of a pattern P against a term U is defined by

$$\begin{array}{rcll} \{U/\!\!/\!\!/x\} & = & \mathsf{some}\ \{U/x\} & \\ \{A/\!\!/\!\!/A\} & = & \mathsf{some}\ \{\} & \text{if } A \text{ is an atom} \\ \{UV/\!\!/\!\!/PQ\} & = & \{U/\!\!/\!\!/P\} \uplus \{V/\!\!/\!\!/Q\} & \text{if } UV \text{ is a compound} \\ \{U/\!\!/\!\!/P\} & = & \mathsf{none}\ \text{otherwise} & \text{if } U \text{ is matchable} \\ \{U/\!\!/\!\!/P\} & = & \text{undefined} & \text{otherwise.} \end{array}$$

The restrictions to matchable forms in the above definition are necessary to ensure that matching is stable under reduction of U.

Now the case $\mathcal{G}$ introduced earlier becomes a partial function of combinators defined by

$$\mathcal{G}(U) = \{U/\!\!/\!\!/P\}M\ .$$

Lemma 5.3.1. *Cases are symbolic computations on confluent term calculi.*

Proof: For $\mathcal{G}$ above to be well-defined, it suffices to prove that if $U = U'$ and the matches $\{U/\!\!/\!\!/P\}$ and $\{U'/\!\!/\!\!/P\}$ are both defined then these matches are equal. The proof is by induction upon the structure of P. If P is a variable x then $\{U/\!\!/\!\!/x\} = \{U'/\!\!/\!\!/x\}$ since $U = U'$. Otherwise, U and U' must be matchable forms. By confluence, if U is an atom then U' must be the same atom, while if U is some compound U_1U_2 then U' must be some compound $U_1'U_2'$ such that $U_1 = U_1'$ and $U_2 = U_2'$. Thus the only possibility of interest is when P is an application P_1P_2 and U and U' are compounds as described above. Hence

$$\{U/\!\!/\!\!/P\} = \{U_1/\!\!/\!\!/P_1\} \uplus \{U_2/\!\!/\!\!/P_2\} = \{U_1'/\!\!/\!\!/P_1\} \uplus \{U_2'/\!\!/\!\!/P_2\} = \{U'/\!\!/\!\!/P\}$$

by two applications of induction. □

5.4 Completeness

A confluent combinatory calculus is *structure complete* if, for every normal term P and term M, the case $\mathcal{G}(P) = M$ is represented by some term $P \to M$.

Such a calculus is combinatorially complete since $\lambda x.M$ is given by $x \to M$. Hence, the calculus has combinators S and K with the usual behaviours. Given a sequence of cases $P_i \to M_i$ the *pattern-matching function*

$$\begin{aligned} & P_1 \to M_1 \\ \mid\ & P_2 \to M_2 \\ & \cdots \\ \mid\ & P_n \to M_n \end{aligned}$$

is defined as in pattern calculus. In the combinatory setting, the *extension* of a combinator N (the *default*) by a *special case* consisting of a pattern P and a body M is given by

$$P \to M \mid N = S(P \to KM)N \ .$$

When applied to some term U such that $\{U /\!\!/ P\} = \mathsf{some}\ \sigma$ for some substi-

tution σ then

$$\begin{aligned}(P \to M \mid N)U &= S(P \to KM)NU \\ &\longrightarrow (P \to KM)U(NU) \\ &\longrightarrow (\sigma KM)(NU) \\ &= K(\sigma M)(NU) = \sigma M\ .\end{aligned}$$

Alternatively, if $\{U/\!\!/P\}$ is none then

$$\begin{aligned}(P \to M \mid N)U &\longrightarrow (P \to KM)U(NU) \\ &\longrightarrow I(NU) \\ &\longrightarrow NU\ .\end{aligned}$$

For example, F is defined by

$$\begin{aligned}&x\ y \to (m \to n \to n\ x\ y) \\ \mid\ &x \to (m \to n \to m)\end{aligned}$$

where the arrow in the case is right-associative. Similarly, `eqatom` is defined

by

$$\begin{aligned}
&\texttt{eqatom} = \\
&\quad A_1 \to (A_1 \to K \mid y \to KI) \\
&\mid A_2 \to (A_2 \to K \mid y \to KI) \\
&\quad \ldots \\
&\mid A_n \to (A_n \to K \mid y \to KI) \\
&\mid x \to y \to KI \ .
\end{aligned}$$

where $A_1, \ldots, A_n$ is a listing of the finite collection of atoms.

A confluent combinatory calculus *supports duplication* (respectively, *elimination, factorisation, separation of atoms*) if it has a combinator S (respectively, $K, F, \texttt{eqatom}$) that represents $\mathcal{S}$ (respectively, $\mathcal{K}, \mathcal{F}$, equality of atoms). Less formally, it *supports* S (respectively, $K, F, \texttt{eqatom}$) if it supports the corresponding symbolic function.

Lemma 5.4.1. *For any three of S, K, F and* `eqatom` *there is a confluent combinatory calculus that supports them but does not support the fourth.*

Proof: For not supporting S, consider the F-calculus with factorable forms F, FM and FMN. Then define $K = FF$ and $\texttt{eqatom} = K(KK)$. As nothing can be duplicated, S is not representable.

For K, consider the S-calculus in which the usual rule for S is supplemented by $S \longrightarrow S$. Since there are no normal forms, there are no partially applied operators or atoms, so define $F = \texttt{eqatom} = S$. However, nothing can be eliminated by S so K is not definable.

For F, use SK-calculus with `eqatom` defined by extensionality.

For `eqatom`, consider SFT-calculus in which T satisfies the same rules as S. □

Theorem 5.4.2. *A confluent combinatory calculus is structure complete if and only if, it supports S, K, F and* `eqatom`.

Proof: The forward direction follows from the previous constructions. For the converse, suppose that suitable combinators S, K, F and `eqatom` exist.

Every case $\mathcal{G}$ defined by $\mathcal{G}(P) = M$ is represented by $P \to M$ which is defined by induction on the structure of P, employing a fresh variable x, as follows:

- If P is a variable y then $P \to M$ is $\lambda^* y.M$.
- If P is an atom A then $P \to M$ is $\lambda^* x.Fx(\texttt{eqatom}\, AxMI)(K(KI))$.
- If P is an application P_1P_2 then $P \to M$ is

$$\lambda^* x.FxI(S(P_1 \to K(P_2 \to M))(K(KI))) \; .$$

The proof that $P \to M$ represents $\mathcal{G}(P) = M$ is by induction on the structure of P. Let U be a combinator such that $\mathcal{G}(U)$ is defined and consider $(P \to M)U$. Without loss of generality, no free variable of P is free in U.

- If P is a variable x then $(P \to M)U$ is $(\lambda x.M)U$ which reduces to $\{U/x\}M$ by Lemma 2.3.1.
- If P is an atom A then $(P \to M)U \longrightarrow FU(\texttt{eqatom}\, AUMI)(K(KI))$. When U is A this reduces to M. If U is any other matchable form then $(P \to M)U$ reduces to I.

- If P is an application P_1P_2 then $(P \to M)U$ reduces to

$$FUI(S(P_1 \to K(P_2 \to M))(K(KI))) .$$

If U is an atom then this reduces to I. Alternatively, if U is a matchable form U_1U_2 then this reduces to

$$S(P_1 \to K(P_2 \to M))(K(KI))U_1U_2$$

which reduces to $(P_1 \to K(P_2 \to M))U_1(KI)U_2$. Now if $\{U_1/\!\!/P_1\}$ is $\mathsf{some}\ \sigma_1$ for some substitution σ_1 then the reduct becomes $(\sigma_1 K(P_2 \to M))(KI)U_2$ which is $(P_2 \to M\sigma_1)U_2$ since P_2 and P_1 do not share any free variables. In turn, if $\{U_2/\!\!/P_2\} = \mathsf{some}\ \sigma_2$ for some substitution σ_2 then the combinator reduces to $\sigma_2(\sigma_1 M)$. Now, free variables in the range of σ_1 are also free in U, and so not in the domain of σ_2. Thus, the result is $(\sigma_1 \uplus \sigma_2)M = \{U/\!\!/P\}M$. Alternatively, if $\{U_2/\!\!/P_2\} = \mathsf{none}$ then the result is I as required. Finally, if $\{U_1/\!\!/P_1\} = \mathsf{none}$ then the result is $I(KI)U_2 = I$ as required.

□

Corollary 5.4.3. *A confluent combinatory calculus that supports S, F and* `eqatom` *is structure complete if it has any normal forms.*

Proof: If the calculus has a normal form then it has an atom A so that K can be defined to be FA. □

Corollary 5.4.4. *The calculi with operators SF or SKF or $SF\mathcal{C}$ or SFC, and the $\{S, F, \Omega\}$-calculus are all structure complete.*

Corollary 5.4.5. *Any symbolic computation restricted to normal forms of a structure complete, confluent combinatory calculus is representable.*

Proof: The combinators S, K, F and `eqatom` are sufficient to redeploy the proof of Theorem 5.2.8. □

Pattern-matching functions of the sort described here have been used to define *path polymorphic functions* [Jay09] which traverse the internal structure of their arguments. This is achieved by recursively using the pattern $x\ y$ to represent an arbitrary compound. In the examples below, recursion is made implicit, and function arguments may be placed on the left-hand side of defining equations.

A familiar example is structural equality, which can be described by the pattern-matching function

$$
\begin{aligned}
&\texttt{equal} = \\
&\quad x_1\ x_2 \to (\ \ y_1\ y_2 \to (\texttt{equal}\ x_1\ y_1)\ \texttt{and}\ (\texttt{equal}\ x_2\ y_2) \\
&\qquad\qquad \mid y \to KI) \\
&\mid x \to (\ \ y_1\ y_2 \to KI \\
&\qquad\quad \mid \texttt{eqatom}\ x)\ .
\end{aligned}
$$

Theorem 5.4.6. *Let M and N be combinators in normal form. If $M = N$ then* `equal` $M\ N \longrightarrow K$ *else* `equal` $M\ N \longrightarrow KI$.

Proof: The proof is by straightforward induction on the structure of M. □

More generally, path polymorphism can be used to define generic queries that can select from, or update within, arbitrary structures. Since selecting requires a significant amount of list processing, the principles are better illustrated through updating. First, define `apply2all` by

$$\begin{array}{l}\texttt{apply2all}\ f\ x = \\ (\ \ y_1\ y_2 \to (\texttt{apply2all}\ f\ y_1)\ (\texttt{apply2all}\ f\ y_2) \\ \ |\ y \to y) \\ \ \ (f\ x)\ .\end{array}$$

The query `apply2all` $f\ x$ recursively applies itself to the components of the result of applying f to x as a whole. Building on this, define the `update` combinator by

$$\texttt{update}\ t\ f = \texttt{apply2all}\ (\lambda^* x.\ \texttt{if}\ t\ x\ \texttt{then}\ f\ x\ \texttt{else}\ x)\ .$$

The basic path polymorphism of `apply2all` is used, but the function f is only applied when a test t is passed. Once lists have been defined, then it is equally easy to define a query `select` that produces a list of components of a structure satisfying some property.

Notes. The pattern-matching described in this chapter is subtly different from that of pure pattern calculus (and other pattern calculi [Jay04, JK06, GW07, JK09, Jay09]). On one hand the patterns of pattern completeness do not allow for free variables in the manner of pure pattern calculus. On the

other hand, SF-calculus can factorise combinators, i.e. functions, that pure pattern calculus cannot. Formally, there are (unpublished) translations from pure pattern calculus into compound calculus, and from compound calculus into $SF\mathcal{C}$-calculus. In the reverse, none of compound calculus, static pattern calculus, or dynamic/pure pattern calculus can represent $\mathcal{F}$.

Further details on SF-calculus are available in "A combinatory account of internal structure" [JGW11] including additional commentary on the relation to classical theory and the typing of F using System **F** [GLT89].

Chapter 6

Concurrent Pattern Calculus

Intensionality in sequential computation yields greater expressive power so it is natural to consider intensional concurrent computation. This chapter introduces *concurrent pattern calculus* (CPC) [GGJ10] that populates the bottom right corner of the computation square. Intensionality in CPC is supported by generalising from structure complete style pattern-matching to symmetric *pattern-unification*. This in turn provides the basis for defining interaction, much as pattern-matching defines reduction for pure pattern calculus.

The symmetry and bi-directional information flow of pattern-unification in CPC provide a natural language to express trade. Trades can discover each other using common interest represented by some information in a pattern. They can then exchange information in an atomic interaction to complete a trade.

The ability to unify patterns based on multiple names and upon structure

allows CPC to support reduction and rewriting systems. There are several ways to approach encoding computation models in CPC that support different properties ranging from a simple reduction system to parallel rewriting systems.

This chapter develops CPC with interaction based on pattern-unification. Concepts are illustrated through examples of share trading and reduction systems which serve as a precursor to later applications and computation models, respectively.

6.1 Syntax

The *trade patterns* (meta-variables $p, p', p_1, q, q', q_1, \ldots$) are built using a class of *names* (familiar from π-calculus and Linda) and have the following forms

$$\begin{array}{llll} \textit{Patterns} & p ::= & \lambda x & \text{binding name} \\ & & x & \text{variable name} \\ & & \ulcorner x \urcorner & \text{protected name} \\ & & p \bullet p & \text{compound.} \end{array}$$

Binding names λx denote information sought by a trader and are similar to binding names in Linda templates. Variable names x represent information being offered and are familiar from π-calculus and Linda. Protected names $\ulcorner x \urcorner$ represent recognised information that cannot be traded, again similar to templates in Linda. A compound combines two patterns p and q, its

components, into a pattern $p \bullet q$, somewhat similar to the pairs of Spi calculus. Compounds are left associative similar to application in λ-calculus, pure pattern calculus, and combinatory logics. The *atoms* are patterns that are not compounds. The atoms x and $\ulcorner x \urcorner$ are defined to *know* x.

There is a correspondence between the trade patterns and the communication primitives of more familiar process calculi. Binding names correspond to inputs and variable names to outputs. The protected names are similar to the protected names in Linda templates and can also be used to represent channels as in the π-calculus. There is some subtlety in the relationship to variable names. As protected names specify a requirement or template similar to their rôle in Linda it is natural that they unify with the variable form of the name. Similarly, as protected names in CPC are used to support channel-based communication it is also natural that protected names unify with themselves.

The subtleties can be clarified by considering a simple trading example from the stock market that is trading shares. A stock jobber is offering to sell shares in company `ABC` for one dollar, as represented by a pattern of the form

$$\texttt{ABC} \bullet \texttt{\$1} \ .$$

A stock raider is offering to buy shares in `ABC` but does not want anyone to know this, unless they are offering to sell, as represented by the pattern

$$\ulcorner \texttt{ABC} \urcorner \bullet \lambda x \ .$$

Finally, a bottom feeder is interested in buying any cheap shares and so may use a pattern of the form

$$\lambda x \bullet \$1 \ .$$

Given a pattern p the sets of: *variables names*, denoted $\mathsf{vn}(p)$; *protected names*, denoted $\mathsf{pn}(p)$; and *binding names*, denoted $\mathsf{bn}(p)$, are as expected with the union being taken for compounds. The *free names* of a pattern p, written $\mathsf{fn}(p)$, is the union of the variable names and protected names of p. A pattern is *well formed* if each binding name appears only once, similar to Linda templates and structure complete patterns. All patterns appearing in the rest of this dissertation are assumed to be well formed.

As protected names are limited to recognition and binding names are being sought, neither should be communicable to another process. Thus, a pattern is *communicable*, able to be traded to another process, if it contains no protected or binding names.

Protection of a name can be extended to a communicable pattern p by defining

$$\ulcorner x \urcorner = \ulcorner x \urcorner \qquad\qquad \ulcorner p \bullet q \urcorner = \ulcorner p \urcorner \bullet \ulcorner q \urcorner \ .$$

A *substitution* σ (also denoted $\sigma, \sigma_1, \rho, \rho_1, \theta, \theta_1, \ldots$) is a partial function from free names to communicable patterns. Otherwise substitutions and their properties are familiar from Chapters 2, 3, 4 & 5. These are applied to

patterns as follows:

$$\begin{aligned}
\sigma x &= \begin{cases} p & \text{if } \sigma \text{ maps } x \text{ to } p \\ x & \text{if } x \notin \mathsf{dom}(\sigma) \end{cases} \\
\sigma\ulcorner x \urcorner &= \begin{cases} \ulcorner p \urcorner & \text{if } \sigma \text{ maps } x \text{ to } p \\ \ulcorner x \urcorner & \text{if } x \notin \mathsf{dom}(\sigma) \end{cases} \\
\sigma(\lambda x) &= \lambda x \\
\sigma(p \bullet q) &= (\sigma p) \bullet (\sigma q) \ .
\end{aligned}$$

Similar to pure pattern calculus, the action $\hat{\sigma}$ of a substitution σ can be defined by

$$\begin{aligned}
\hat{\sigma} x &= x \\
\hat{\sigma}\ulcorner x \urcorner &= \ulcorner x \urcorner \\
\hat{\sigma}(\lambda x) &= \begin{cases} p & \text{if } \sigma \text{ maps } x \text{ to } p \\ \lambda x & \text{if } x \notin \mathsf{dom}(\hat{\sigma}) \end{cases} \\
\hat{\sigma}(p \bullet q) &= (\hat{\sigma} p) \bullet (\hat{\sigma} q) \ .
\end{aligned}$$

The behaviour is the same as a normal substitution except operating on binding names rather than free names. When σ is of the form $\{p_i/x_i\}$ then $\{p_i/\lambda x_i\}$ may be used to denote $\hat{\sigma}$.

The *symmetric matching* or *unification* $\{p \| q\}$ of two patterns p and q attempts to unify p and q by generating substitutions upon their binding names. When defined, the result is some pair of substitutions whose domains

are the binding names of p and of q. The rules to generate the substitutions are:

$$\left.\begin{array}{r}\{x \| x\} \\ \{x \| \ulcorner x \urcorner\} \\ \{\ulcorner x \urcorner \| x\} \\ \{\ulcorner x \urcorner \| \ulcorner x \urcorner\}\end{array}\right\} = (\{\}, \{\})$$

$$\{\lambda x \| q\} = (\{q/x\}, \{\}) \quad \text{if } q \text{ is communicable}$$

$$\{p \| \lambda x\} = (\{\}, \{p/x\}) \quad \text{if } p \text{ is communicable}$$

$$\{p_1 \bullet p_2 \| q_1 \bullet q_2\} = ((\sigma_1 \cup \sigma_2), (\rho_1 \cup \rho_2)) \quad \begin{cases} \{p_1 \| q_1\} = (\sigma_1, \rho_1) \\ \{p_2 \| q_2\} = (\sigma_2, \rho_2) \end{cases}$$

$$\{p \| q\} = \text{undefined} \quad \text{otherwise.}$$

Two atoms unify if they know the same name. A name that seeks information, a binding name, unifies with any communicable pattern to produce a binding for its underlying name. Two compounds unify if their corresponding components do; the resulting substitutions are given by taking unions of those produced by unifying the components (necessarily disjoint as patterns are well-formed). Otherwise the patterns cannot be unified and the matching is undefined.

Observe that unlike in pure pattern calculus, there is no failure of pattern unification. The reason for this is discussed when defining interaction in the next section.

The processes of CPC are given by

Processes	P	::=	$\mathbf{0}$	null process
			$P \mid P$	parallel composition
			$!P$	replication
			$(\nu x)P$	restriction
			$p \to P$	case.

The null process, parallel composition, replication and restriction are the classical ones for process calculi: $\mathbf{0}$ is the inactive process; $P \mid Q$ is the parallel composition of processes P and Q, allowing the two processes to evolve independently by interacting; the replication $!P$ provides as many parallel copies of P as desired; $(\nu x)P$ declares a new name x, visible only within P and distinct from any other name. The traditional input and output primitives are replaced by the *case* $p \to P$ that has a pattern p and body P. A case with the null process as the body $p \to \mathbf{0}$ may also be written p when no ambiguity may occur. Note that replication, restriction and cases bind stronger than parallel composition.

The free names of processes, denoted $\mathsf{fn}(P)$, are defined as usual for all the traditional primitives and

$$\mathsf{fn}(p \to P) \quad = \quad \mathsf{fn}(p) \cup (\mathsf{fn}(P) \backslash \mathsf{bn}(p))$$

for the case. As expected the binding names of the pattern bind their free

occurrences in the body.

6.2 Operational Semantics

Renaming is handled through α-conversion, $=_\alpha$, that is the congruence relation generated by the following axioms

$$\begin{array}{lll} (\nu x)P & =_\alpha & (\nu y)(\{y/x\}P) \qquad\qquad y \notin \mathsf{fn}(P) \\ p \to P & =_\alpha & (\{\lambda y/\lambda x\}p) \to (\{y/x\}P) \qquad x \in \mathsf{bn}(p), y \notin \mathsf{fn}(P) \cup \mathsf{bn}(p)\ . \end{array}$$

Renaming of a restriction is as usual. The renaming of a binding name is also as expected with the usual restrictions.

The general *structural equivalence relation* $\equiv$ is defined just as in π-calculus [Mil93] and includes α-conversion. The defining axioms of structural equivalence are:

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

$$(\nu n)\mathbf{0} \equiv \mathbf{0} \qquad (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \qquad !P \equiv P \mid !P$$

$$P \mid (\nu n)Q \equiv (\nu n)(P \mid Q) \quad \text{if } n \notin \mathsf{fn}(P)\ .$$

It states that: $\mid$ is a commutative, associative, monoidal operator, with $\mathbf{0}$ acting as the identity; that restriction is useless when applied to the empty process; that the order of restricted names is immaterial; that replication can be freely unfolded; and that the scope of a restricted name can be freely

extended, provided that no name capture arises.

The application σP of a substitution σ to a process P is defined in the usual manner to avoid name capture. For cases this ensures that substitution avoids the binding names in the pattern:

$$\sigma(p \to P) \quad = \quad (\sigma p) \to (\sigma P) \qquad \text{if } \sigma \text{ avoids } \mathsf{bn}(p).$$

Proposition 6.2.1. *For every substitution σ and process P, there is an α-equivalent process P' such that $\sigma P'$ is defined. If P_1 and P_2 are α-equivalent terms, then $\mathsf{fn}(P_1) = \mathsf{fn}(P_2)$ and, if $Q_1 = \sigma P_1$ and $Q_2 = \sigma P_2$ are both defined, then $Q_1 =_\alpha Q_2$.*

Proof: The proof is by straightforward induction. □

CPC has one *interaction axiom* given by

$$(p \to P) \mid (q \to Q) \qquad \longmapsto \qquad (\sigma P) \mid (\rho Q) \qquad \text{if } \{p \| q\} = (\sigma, \rho)\ .$$

It states that if the unification of two patterns p and q is defined and generates (σ, ρ), then apply the substitutions σ and ρ to the bodies P and Q, respectively. If the matching of p and q is undefined then no interaction occurs.

Unlike in the sequential setting there is no need for capturing failure of unification. This is due to interaction being opportunistic between processes rather than being forced by application between terms. As such, failure to interact should not prevent interactions with other processes.

The interaction rule is then closed under parallel composition, restriction and structural equivalence in the usual manner (although not under a case as in pure pattern calculus)

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \frac{P \longrightarrow P'}{(\nu n)P \longrightarrow (\nu n)P'}$$

$$\frac{P \equiv Q \quad Q \longrightarrow Q' \quad Q' \equiv P'}{P \longrightarrow P'} \ .$$

The reflexive and transitive closure of $\longmapsto$ is denoted $\Longmapsto$.

For later convenience $\widetilde{n}$ shall denote a collection of names $n_1, \ldots, n_i$ or other entities as before in defining Linda in Section 3.2.

The examples and theorems developed later in the dissertation rely on control of interaction. The remainder of this section formalises some properties of interaction particularly to support the behaviour of channel-based communication.

Definition 6.2.2 (Interaction)**.** *Two processes P and Q interact if they have a reduct R such that $P \Longmapsto p \to P_1 \mid P_2$ and $Q \Longmapsto q \to Q_1 \mid Q_2$ and $\{p \| q\} = (\sigma, \rho)$ and $(\sigma P_1) \mid (\rho Q_1) \mid P_2 \mid Q_2 \Longmapsto R$.*

The rest of this section formalises that protected names can be used to ensure that interacting processes both know the same name. Similar results have shown that Linda style pattern-matching can be used to ensure a channel name is known [Gor08a, Gor08b]. Here the results are generalised

to account for pattern-unification and the larger class of patterns.

First is to show that if a case has a protected name in the pattern then it will only interact with other cases that know the same name, i.e. have the same name free in their pattern.

Lemma 6.2.3. *If the unification of patterns p and q is defined then any protected name of p is a free name of q.*

Proof: Without loss of generality q is a not a binding name λx as this implies p is communicable (and so without protected names). Now proceed by induction on the structure of p:

- If p is a name then it follows that p is some protected name $\ulcorner x \urcorner$. Then q must be either x or $\ulcorner x \urcorner$ for the matching to be defined and x is in the free names of q.

- If p is of the form $p_1 \bullet p_2$ then by unification q must be of the form $q_1 \bullet q_2$ and $\{p_i \| q_i\}$ is defined for $i \in \{1, 2\}$. Now $\mathsf{pn}(p) = \mathsf{pn}(p_1) \cup \mathsf{pn}(p_2)$ and $\mathsf{fn}(q) = \mathsf{fn}(q_1) \cup \mathsf{fn}(q_2)$ and by two applications of induction $\mathsf{pn}(p_i) \subseteq \mathsf{fn}(q_i)$ and conclude with $\mathsf{pn}(p) \subseteq \mathsf{fn}(q)$.

□

Lemma 6.2.4. *Given a process P and substitution σ, then* $\mathsf{fn}(\sigma P) \subseteq \mathsf{fn}(P) \cup \mathsf{fn}(\sigma)$.

Proof: Trivial by definition of the application of σ. □

Lemma 6.2.5. *Given a process P such that $P \Longmapsto P'$, then* $\mathsf{fn}(P') \subseteq \mathsf{fn}(P)$.

Proof: Trivial by properties of substitutions and Lemma 6.2.4. □

Lemma 6.2.6. *Suppose a process $p \to P$ interacts with a process Q. If x is a protected name in p then x must be a free name in Q.*

Proof: For Q to interact with $p \to P$ it must be that $Q \Longmapsto q \to Q_1 \mid Q_2$ such that $\{p \| q\}$ is defined. Then by Lemma 6.2.3 the free names of q include x and consequently x must be free in $q \to Q_1 \mid Q_2$ and it follows by Lemma 6.2.5 that x is free in Q. □

These results can be used to show that CPC supports channel-based communication with similar properties to π-calculus. The key idea is that a protected name requires both processes to know that name to interact. This shall be exploited in the following section's examples with a formal account of supporting channel-based communication in Chapters 7 & 9.

6.3 Trade in CPC

This section uses the example of share trading to explore the potential of CPC. The scenario is of two potential traders, a buyer and a seller, who wish to engage in trade. To complete a transaction the traders need to progress through two stages: *discovering* each other and *exchanging* information. Both traders begin with a pattern for their desired transaction. The discovery phase can be characterised as a pattern-unification problem where traders' patterns are used to find a compatible partner. The exchange phase occurs when a buyer and seller have agreed upon a transaction. Now each trader

wishes to exchange information in a single interaction, ensuring both traders are satisfied with a complete transaction.

The rest of this section develops a solution in three stages. The first stage demonstrates discovery, the second introduces a registrar to validate the traders, the third extends the second with protected names to ensure privacy. The development is done here in CPC to present the concepts and expressiveness of CPC. Later in Section 10.1 the development will be repeated and extended in Concurrent **bondi** a programming language that implements CPC [Con11].

Solution 1

Consider two traders, a buyer and a seller. The buyer Buy_1 with bank account b and desired shares s can be given by

$$\mathsf{Buy}_1 \quad = \quad s \bullet \lambda m \to m \bullet b \bullet \lambda x \to B(x) \; .$$

The first pattern $s \bullet \lambda m$ is used to match with a compatible seller using share information s, and to input a name m to be used to coordinate the exchange of bank account information b for share certificates bound to x. The transaction concludes successfully with $B(x)$.

The seller Sell_1 with share certificates c and desired share sale s is given by

$$\mathsf{Sell}_1 \quad = \quad (\nu n) s \bullet n \to n \bullet \lambda y \bullet c \to S(y) \; .$$

The seller creates a fresh name n and then attempts to find a buyer for the shares described in s, offering n to the buyer to continue the transaction. The fresh name n is then used to coordinate the exchange of billing information bound to y for the share certificates c. The seller then concludes with the successfully completed transaction as $S(y)$.

The discovery phase succeeds when the traders are placed in a parallel composition and discover each other by unifying on s as follows

$$\begin{aligned} & \mathsf{Buy}_1 \mid \mathsf{Sell}_1 \\ \equiv \quad & (\nu n)(s \bullet \lambda m \to m \bullet b \bullet \lambda x \to B(x) \mid s \bullet n \to n \bullet \lambda y \bullet c \to S(y)) \\ \longmapsto \quad & (\nu n)(n \bullet b \bullet \lambda x \to B(x) \mid n \bullet \lambda y \bullet c \to S(y)) \ . \end{aligned}$$

The next phase is to exchange billing information for share certificates

$$(\nu n)(n \bullet b \bullet \lambda x \to B(x) \mid n \bullet \lambda y \bullet c \to S(y)) \longmapsto (\nu n)(B(c) \mid S(b)) \ .$$

The transaction concludes with the buyer having the share certificates c and the seller having the billing account b.

This solution allows the traders to discover each other and exchange information atomically to complete a transaction. However, there is no way to determine if a process is a trustworthy trader.

Solution 2

Now add a registrar that keeps track of registered traders. Traders offer their identity to potential partners and the registrar confirms if the identity belongs to a valid trader. The buyer is now

$$\mathsf{Buy}_2 \;=\; s \bullet i_B \bullet \lambda j \to n_B \bullet j \bullet \lambda m \to m \bullet b \bullet \lambda x \to B(x)\ .$$

The first pattern now swaps the buyer's identity i_B for the seller's, bound to j. The buyer then consults the registrar using the identifier n_B to validate j; if valid, the exchange continues as before coordinating with a name provided by the registrar.

Now define the seller symmetrically by

$$\mathsf{Sell}_2 \;=\; s \bullet \lambda j \bullet i_S \to n_S \bullet j \bullet \lambda m \to m \bullet \lambda y \bullet c \to S(y)\ .$$

Also define the registrar Reg_2 with identifiers n_B and n_S to communicate with the buyer and seller, respectively, by

$$\mathsf{Reg}_2 \;=\; (\nu n)(n_B \bullet i_S \bullet n \mid n_S \bullet i_B \bullet n)\ .$$

The registrar creates a new name n to provide to traders who have been validated; then it makes the name available to known traders who attempt to validate another known trader. Although rather simple, the registrar can easily be extended to support a multitude of traders.

Running these processes in parallel yields the following interaction

$$
\begin{aligned}
& \mathsf{Buy}_2 \mid \mathsf{Sell}_2 \mid \mathsf{Reg}_2 \\
\equiv\ & (\nu n)(s \bullet i_B \bullet \lambda j \to n_B \bullet j \bullet \lambda m \to m \bullet b \bullet \lambda x \to B(x) \mid n_B \bullet i_S \bullet n \\
& \quad \mid s \bullet \lambda j \bullet i_S \to n_S \bullet j \bullet \lambda m \to m \bullet \lambda y \bullet c \to S(y) \mid n_S \bullet i_B \bullet n) \\
\longmapsto\ & (\nu n)(n_B \bullet i_S \bullet \lambda m \to m \bullet b \bullet \lambda x \to B(x) \mid n_B \bullet i_S \bullet n \\
& \quad \mid n_S \bullet i_B \bullet \lambda m \to m \bullet \lambda y \bullet c \to S(y) \mid n_S \bullet i_B \bullet n)\ .
\end{aligned}
$$

The share information s allows the buyer and seller to discover each other and swap identities i_B and i_S. The next two interactions involve the buyer and seller validating each other's identity and inputting the name to coordinate the exchange phase

$$
\begin{aligned}
& (\nu n)(n_B \bullet i_S \bullet \lambda m \to m \bullet b \bullet \lambda x \to B(x) \mid n_B \bullet i_S \bullet n \\
& \quad \mid n_S \bullet i_B \bullet \lambda m \to m \bullet \lambda y \bullet c \to S(y) \mid n_S \bullet i_B \bullet n) \\
\longmapsto\ & (\nu n)(n \bullet b \bullet \lambda x \to B(x) \\
& \quad \mid n_S \bullet i_B \bullet \lambda m \to m \bullet \lambda y \bullet c \to S(y) \mid n_S \bullet i_B \bullet n) \\
\longmapsto\ & (\nu n)(n \bullet b \bullet \lambda x \to B(x) \mid n \bullet \lambda y \bullet c \to S(y))\ .
\end{aligned}
$$

Now that the traders have validated each other, they can continue with the exchange step from before

$$
(\nu n)(n \bullet b \bullet \lambda x \to B(x) \mid n \bullet \lambda y \bullet c \to S(y)) \qquad \longmapsto \qquad (\nu n)(B(c) \mid S(b))\ .
$$

The traders exchange information and successfully complete with $B(c)$ and $S(b)$ as before.

Although this solution satisfies the desire to validate that traders are legitimate, the freedom of unification allows for malicious processes to interfere. Consider the promiscuous process Prom given by

$$\mathsf{Prom} \;=\; \lambda z_1 \bullet \lambda z_2 \bullet a \to P(z_1, z_2)\ .$$

This process is willing to match any other process that will swap two pieces of information for some arbitrary information a. Such a process could interfere with the traders trying to complete the exchange phase of a transaction. For example,

$$\begin{aligned} & (\nu n)(n \bullet b \bullet \lambda x \to B(x) \mid n \bullet \lambda y \bullet c \to S(y)) \\ & \quad \mid \lambda z_1 \bullet \lambda z_2 \bullet a \to P(z_1, z_2) \\ \longmapsto \;\; & (\nu n)(B(a) \mid n \bullet \lambda y \bullet c \to S(y) \mid P(n, b)) \end{aligned}$$

where the promiscuous process has stolen the identifier n and the bank account information b. The unfortunate buyer is left with some useless information a and the seller is waiting to complete the transaction.

Solution 3

The vulnerability of Solution 2 can be repaired by using protected names. The buyer, seller and registrar can be repaired to

$$\begin{aligned}
\mathsf{Buy}_3 &= s \bullet i_B \bullet \lambda j \to \ulcorner n_B \urcorner \bullet j \bullet \lambda m \to \ulcorner m \urcorner \bullet b \bullet \lambda x \to B(x) \\
\mathsf{Sell}_3 &= s \bullet \lambda j \bullet i_S \to \ulcorner n_S \urcorner \bullet j \bullet \lambda m \to \ulcorner m \urcorner \bullet \lambda y \bullet c \to S(y) \\
\mathsf{Reg}_3 &= (\nu n)(\ulcorner n_B \urcorner \bullet \ulcorner i_S \urcorner \bullet n \mid \ulcorner n_S \urcorner \bullet \ulcorner i_B \urcorner \bullet n) \ .
\end{aligned}$$

Now all communication between the buyer, seller and registrar use protected identifiers: $\ulcorner n_B \urcorner, \ulcorner n_S \urcorner$ and $\ulcorner m \urcorner$. Thus, all that remains is to ensure appropriate restrictions:

$$(\nu n_B)(\nu n_S)(\mathsf{Buy}_3 \mid \mathsf{Sell}_3 \mid \mathsf{Reg}_3) \ .$$

Therefore, other processes can only interact with the traders during the discovery phase, which will not lead to a successful transaction. The registrar will only interact with the traders because all the registrar's patterns have protected names known only to the registrar and a trader (Lemma 6.2.6).

6.4 Computing with CPC

This section develops a simple boolean reduction system and shows how this can be encoded in CPC. This serves to further introduce CPC syntax and interactions as well as a precursor to formalising CPC's expressive power

with respect to sequential computation models and rewriting in Chapter 7.

Consider a simple boolean reduction system $\mathcal{B}$ with *terms* given by

$$t \;::=\; \mathsf{true} \mid \mathsf{false} \mid \mathsf{cond}\ t\ t\ t\ .$$

The two boolean *primitives* true and false, and the typical *conditional* statement cond $r\ s\ t$ with arguments r, s and t.

As the development in this section will demonstrate some different approaches to encoding sequential reduction systems into CPC, three different collections of reduction rules are given for $\mathcal{B}$: core, optional and parallel.

There are three *core reduction rules*

$$\frac{}{\mathsf{cond}\ \mathsf{true}\ s\ t \longrightarrow s}$$

$$\frac{}{\mathsf{cond}\ \mathsf{false}\ s\ t \longrightarrow t}$$

$$\frac{r \longrightarrow r'}{\mathsf{cond}\ r\ s\ t \longrightarrow \mathsf{cond}\ r'\ s\ t}\ .$$

The first two rules switch according to the first argument to a conditional; with true reducing to the second argument and false reducing to the third. The third rule reduces the first argument to a conditional.

Although these suffice for $\mathcal{B}$'s core behaviour, additional rules can be defined that allow reduction of the second and third arguments to the con-

ditional. Two *optional reduction rules* to support reduction under the conditional are

$$\frac{s \longrightarrow s'}{\mathsf{cond}\ r\ s\ t \longrightarrow \mathsf{cond}\ r\ s'\ t} \qquad \frac{t \longrightarrow t'}{\mathsf{cond}\ r\ s\ t \longrightarrow \mathsf{cond}\ r\ s\ t'}$$

that reduce the second or third arguments, respectively. Observe that this is similar to reducing arguments to combinators in SK-calculus or SF-calculus.

The $\mathcal{B}$-terms can be easily encoded into patterns by defining the *construction* $(\!|\ \cdot\ |\!)$, exploiting reserved names true, false and cond, as follows

$$\begin{aligned} (\!|\mathsf{true}|\!) &\stackrel{\text{def}}{=} \mathsf{true} \\ (\!|\mathsf{false}|\!) &\stackrel{\text{def}}{=} \mathsf{false} \\ (\!|\mathsf{cond}\ r\ s\ t|\!) &\stackrel{\text{def}}{=} \mathsf{cond} \bullet (\!|r|\!) \bullet (\!|s|\!) \bullet (\!|t|\!)\ . \end{aligned}$$

Observe that all three map the operator, true, false or cond, to a name and use compounds to preserve the arguments to conditionals.

By representing $\mathcal{B}$-terms in the pattern of a CPC case, the reduction can then be driven by defining cases that recognise a reducible structure and perform the appropriate operations. This is similar to how a Turing machine operates on a tape, as is discussed further when encoding SF-calculus in Chapter 7 and in the conclusions in Chapter 11. The reduction for the first

two core rules can be captured by cases of the form

$$\mathsf{cond} \bullet \mathsf{true} \bullet \lambda s \bullet \lambda t \to s$$

$$\mathsf{cond} \bullet \mathsf{false} \bullet \lambda s \bullet \lambda t \to t$$

that will unify with the construction of terms of the form $\mathsf{cond}\ \mathsf{true}\ s\ t$ and $\mathsf{cond}\ \mathsf{false}\ s\ t$, respectively. The third core rule is a little more complex, requiring a pattern that will unify with the encoding of the form $\mathsf{cond}\ (\mathsf{cond}\ r_1\ s_1\ t_1)\ s_2\ t_2$. The first argument can then be reduced independently before being combined with the rest of the original pattern. Thus, define a process to perform these operations as follows

$$\begin{aligned}&\mathsf{cond} \bullet (\lambda a \bullet \lambda b \bullet \lambda c \bullet \lambda d) \bullet \lambda s \bullet \lambda t \\ &\qquad \to (a \bullet b \bullet c \bullet d \to \lambda z \to \mathsf{cond} \bullet z \bullet s \bullet t)\ .\end{aligned}$$

The first pattern unifies with an encoded term whose first argument can be reduced, binding the components of the first argument with $\lambda a \bullet \lambda b \bullet \lambda c \bullet \lambda d$. The second pattern is used to reduce $a \bullet b \bullet c \bullet d$ by interaction with another process to perform the reduction. The third pattern binds the result of reducing $a \bullet b \bullet c \bullet d$ with λz and then the components are recombined into $\mathsf{cond} \bullet z \bullet s \bullet t$.

These three processes that perform the core reductions can be combined

into a $\mathcal{B}$-*reducing process* R as follows

$$\begin{aligned} R \stackrel{\text{def}}{=} \quad & !\mathsf{cond} \bullet \mathsf{true} \bullet \lambda s \bullet \lambda t \to s \\ | \quad & !\mathsf{cond} \bullet \mathsf{false} \bullet \lambda s \bullet \lambda t \to t \\ | \quad & !\mathsf{cond} \bullet (\lambda a \bullet \lambda b \bullet \lambda c \bullet \lambda d) \bullet \lambda s \bullet \lambda t \\ & \qquad \to (a \bullet b \bullet c \bullet d \to \lambda z \to \mathsf{cond} \bullet z \bullet s \bullet t)\ . \end{aligned}$$

Observe that R is irreducible due to all cases having a binding name as their rightmost component.

Thus the encoding $[\![\cdot]\!]$ of $\mathcal{B}$ terms into CPC can then be easily defined by

$$[\![t]\!] \stackrel{\text{def}}{=} (\!|t|\!) \mid R$$

where the term is translated to the parallel composition of its construction and the $\mathcal{B}$-reducing process.

Theorem 6.4.1. *The encoding $[\![\cdot]\!]$ of $\mathcal{B}$-terms into CPC preserves the core reduction rules.*

Proof: Consider the $\mathcal{B}$ term t that has a reduction $t \longrightarrow t'$. The proof is by induction on the structure of t to show that $(\!|t|\!) \to P \mid R \Longmapsto (\!|t'|\!) \mid P \mid R$. Then taking $P = \mathbf{0}$ yields $[\![t]\!] \Longmapsto [\![t']\!]$.

- If t is of the form $\mathsf{cond}\ \mathsf{true}\ r\ s$ then the core reduction must be

$$\frac{}{\mathsf{cond}\ \mathsf{true}\ r\ s \longrightarrow r}$$

and $t' = r$. Now consider the following reduction sequence in CPC:

$$\begin{aligned} & \mathsf{cond} \bullet \mathsf{true} \bullet (\!|r|\!) \bullet (\!|s|\!) \to P \mid R \\ \equiv \quad & \mathsf{cond} \bullet \mathsf{true} \bullet (\!|r|\!) \bullet (\!|s|\!) \to P \mid R \\ & \mid \mathsf{cond} \bullet \mathsf{true} \bullet \lambda s \bullet \lambda t \to s \\ \longmapsto \quad & (\!|r|\!) \mid P \mid R \,. \end{aligned}$$

- If t is of the form $\mathsf{cond}\ \mathsf{false}\ r\ s$ then the core reduction must be

$$\frac{}{\mathsf{cond}\ \mathsf{false}\ r\ s \longrightarrow s}$$

and $t' = s$. Now consider the following reduction sequence in CPC:

$$\begin{aligned} & \mathsf{cond} \bullet \mathsf{false} \bullet (\!|r|\!) \bullet (\!|s|\!) \to P \mid R \\ \equiv \quad & \mathsf{cond} \bullet \mathsf{false} \bullet (\!|r|\!) \bullet (\!|s|\!) \to P \mid R \\ & \mid \mathsf{cond} \bullet \mathsf{false} \bullet \lambda s \bullet \lambda t \to t \\ \longmapsto \quad & (\!|r|\!) \mid P \mid R \,. \end{aligned}$$

- If t is of the form $\mathsf{cond}\ (\mathsf{cond}\ b_1\ r_1\ s_1)\ r\ s$ then the core reduction must be

$$\frac{\mathsf{cond}\ b_1\ r_1\ s_1 \longrightarrow c}{\mathsf{cond}\ (\mathsf{cond}\ b_1\ r_1\ s_1)\ r\ s \longrightarrow \mathsf{cond}\ c\ r\ s}$$

and $t' = \mathsf{cond}\ c\ r\ s$. Now consider the following reduction sequence in

CPC:

$$
\begin{aligned}
& \mathsf{cond} \bullet (\mathsf{cond} \bullet (\!|b_1|\!) \bullet (\!|r_1|\!) \bullet (\!|s_1|\!)) \bullet (\!|r|\!) \bullet (\!|s|\!) \to P \mid R \\
\equiv \quad & \mathsf{cond} \bullet (\mathsf{cond} \bullet (\!|b_1|\!) \bullet (\!|r_1|\!) \bullet (\!|s_1|\!)) \bullet (\!|r|\!) \bullet (\!|s|\!) \to P \mid R \\
& \mid \mathsf{cond} \bullet (\lambda a \bullet \lambda b \bullet \lambda c \bullet \lambda d) \bullet \lambda s \bullet \lambda t \\
& \qquad \to (a \bullet b \bullet c \bullet d \to \lambda z \to \mathsf{cond} \bullet z \bullet s \bullet t) \\
\longmapsto \quad & \mathsf{cond} \bullet (\!|b_1|\!) \bullet (\!|r_1|\!) \bullet (\!|s_1|\!) \to (\lambda z \to \mathsf{cond} \bullet z \bullet (\!|r|\!) \bullet (\!|s|\!)) \mid P \mid R \\
= \quad & (\!|\mathsf{cond}\ b_1\ r_1\ s_1|\!) \to (\lambda z \to \mathsf{cond} \bullet z \bullet (\!|r|\!) \bullet (\!|s|\!)) \mid P \mid R\,.
\end{aligned}
$$

Now by induction hypothesis, since $\mathsf{cond}\ b_1\ r_1\ s_1 \longrightarrow c$ there is a sequence of reductions $(\!|\mathsf{cond}\ b_1\ r_1\ s_1|\!) \to P_1 \mid R \Longmapsto (\!|c|\!) \mid P_1 \mid R$. Take $P_1 = \lambda z \to \mathsf{cond} \bullet z \bullet (\!|r|\!) \bullet (\!|s|\!)$ and proceed as follows

$$
\begin{aligned}
& (\!|\mathsf{cond}\ b_1\ r_1\ s_1|\!) \to (\lambda z \to \mathsf{cond} \bullet z \bullet (\!|r|\!) \bullet (\!|s|\!)) \mid P \mid R \\
= \quad & (\!|\mathsf{cond}\ b_1\ r_1\ s_1|\!) \to P_1 \mid P \mid R \\
\Longmapsto \quad & (\!|c|\!) \mid P_1 \mid P \mid R \\
= \quad & (\!|c|\!) \mid \lambda z \to \mathsf{cond} \bullet z \bullet (\!|r|\!) \bullet (\!|s|\!) \mid P \mid R \\
\longmapsto \quad & \mathsf{cond} \bullet (\!|c|\!) \bullet (\!|r|\!) \bullet (\!|s|\!) \mid P \mid R \\
= \quad & (\!|\mathsf{cond}\ c\ r\ s|\!) \mid P \mid R\,.
\end{aligned}
$$

□

The $\mathcal{B}$ optional reduction rules can also be encoded into CPC with the

following additions to R yielding R^{opt}:

$$\begin{aligned} R^{opt} \stackrel{\text{def}}{=} \quad & R \mid \ !\mathsf{cond} \bullet \lambda r \bullet (\lambda a \bullet \lambda b \bullet \lambda c \bullet \lambda d) \bullet \lambda t \\ & \qquad \to (a \bullet b \bullet c \bullet d \to \lambda z \to \mathsf{cond} \bullet r \bullet z \bullet t) \\ & \mid \ !\mathsf{cond} \bullet \lambda r \bullet \lambda s \bullet (\lambda a \bullet \lambda b \bullet \lambda c \bullet \lambda d) \\ & \qquad \to (a \bullet b \bullet c \bullet d \to \lambda z \to \mathsf{cond} \bullet r \bullet s \bullet z) \ . \end{aligned}$$

The encoding $[\![\cdot]\!]^{opt}$ of $\mathcal{B}$-terms with core and optional reduction rules is then

$$[\![t]\!]^{opt} \stackrel{\text{def}}{=} (\!|t|\!) \mid R^{opt}$$

as before with the modified $\mathcal{B}$ reducing process R^{opt}.

Theorem 6.4.2. *The encoding $[\![\cdot]\!]^{opt}$ of $\mathcal{B}$-terms into CPC preserves the core and optional reduction rules.*

Proof: Straightforward as in Theorem 6.4.1. □

Observe that the encoding $[\![\cdot]\!]^{opt}$ preserves reduction; however this does not exploit concurrency to support parallel reductions. Parallel reduction can be supported with some modifications to the $\mathcal{B}$ reducing process and the encoding. Observe that the only opportunity for parallel reduction is the multiple arguments to the conditional, and so the parallel reduction rules build upon the optional reduction rules. The *parallel reduction rules* support

the parallel reduction of arguments to a conditional, for example:

$$\frac{s \longrightarrow s' \qquad t \longrightarrow t'}{\mathsf{cond}\ r\ s\ t \longrightarrow \mathsf{cond}\ r\ s'\ t'}$$

where s and t can be reduced in parallel, similarly r and s or r and t.

A pattern that can detect the potential parallel reduction of the second and third arguments to a condition is

$$\mathsf{cond} \bullet \lambda r \bullet (\lambda a \bullet \lambda b \bullet \lambda c \bullet \lambda d) \bullet (\lambda e \bullet \lambda f \bullet \lambda g \bullet \lambda h)$$

where $\lambda a \bullet \lambda b \bullet \lambda c \bullet \lambda d$ detects a reducible second argument and similarly $\lambda e \bullet \lambda f \bullet \lambda g \bullet \lambda h$ for the third argument. The naive approach would be to follow the style of the optional rules and share both $a \bullet b \bullet c \bullet d$ and $e \bullet f \bullet g \bullet h$ to have them reduce with the (modified) $\mathcal{B}$ reducing process. However, if $a \bullet b \bullet c \bullet d = e \bullet f \bullet g \bullet h$ then these patterns could unify and this would not be appropriate.

The solution is to follow the style of Milner's encodings of λ-calculus into π-calculus [Mil90] and include an identifying name for each component. Of course this requires modification to the $\mathcal{B}$ reducing process and consequently the encoding of the terms.

The modification to the $\mathcal{B}$ reducing process is to prefix each case with an additional binding name. For example the first two core rules for the

conditional are modified as follows:

$$\mathsf{cond} \bullet \mathsf{true} \bullet \lambda s \bullet \lambda t \to s \quad \text{becomes} \quad \lambda n \bullet (\mathsf{cond} \bullet \mathsf{true} \bullet \lambda s \bullet \lambda t) \to n \bullet s$$

$$\mathsf{cond} \bullet \mathsf{false} \bullet \lambda s \bullet \lambda t \to t \quad \text{becomes} \quad \lambda n \bullet (\mathsf{cond} \bullet \mathsf{false} \bullet \lambda s \bullet \lambda t) \to n \bullet t \ .$$

The other rules are similarly modified, the only non-trivial modification is to create new restricted names for components that are being reduced, as in the rule below for parallel reduction of the second and third arguments to a conditional.

The rule for parallel reduction of s and t in the term $\mathsf{cond}\ r\ s\ t$ is now defined as follows:

$$\begin{aligned}
&\lambda m \bullet (\mathsf{cond} \bullet \lambda r \bullet (\lambda a \bullet \lambda b \bullet \lambda c \bullet \lambda d) \bullet (\lambda e \bullet \lambda f \bullet \lambda g \bullet \lambda h)) \\
&\quad \to (\nu n)(\nu o)(n \bullet (a \bullet b \bullet c \bullet d) \\
&\qquad\qquad \mid o \bullet (e \bullet f \bullet g \bullet h) \\
&\qquad\qquad\quad \to n \bullet \lambda x \to o \bullet \lambda y \\
&\qquad\qquad\qquad \to m \bullet (\mathsf{cond} \bullet r \bullet x \bullet y)) \ .
\end{aligned}$$

Observe that the two reducible components use restricted names n and o to prevent unification with each other. After the third argument $(e \bullet f \bullet g \bullet h)$ has been bound then the restricted names are used to rebuild the reduct $\mathsf{cond} \bullet r \bullet x \bullet y$. Note that although this supports parallel reduction of the second and third arguments, there is a possible reduction sequence that only

reduces the third argument.

Modifying R^{opt} to maintain identifying names and adding in the three parallel reduction rules yields the parallel $\mathcal{B}$ reducing process R^{par}. Now the encoding $[\![\cdot]\!]^m$ is modified to include an initial identifying name m and defined as follows:

$$[\![t]\!]^m \stackrel{\text{def}}{=} (\nu m)(m \bullet (\!|t|\!)) \mid R^{par} \ .$$

Theorem 6.4.3. *The encoding $[\![\cdot]\!]^m$ of $\mathcal{B}$ terms into CPC preserves the core, optional and parallel reduction rules.*

Proof: Straightforward as in Theorems 6.4.1 & 6.4.2. □

Observe that with the optional reduction rules, $\mathcal{B}$ can be considered as a rewriting system since any sub-term can be reduced. Although such reductions are already supported by $[\![\cdot]\!]^{opt}$ and $[\![\cdot]\!]^m$, the encoding always encodes the entire term into a pattern. However, Milner's encodings (and parallel encodings: Definition 3.1.1) encode terms to processes. Since the parallel $\mathcal{B}$ reducing process already uses names to identify sub-terms, an encoding $[\![\cdot]\!]_m$ can be easily developed to encode terms as processes as follows:

$$\begin{aligned}
[\![\mathsf{true}]\!]_m &\stackrel{\text{def}}{=} m \bullet \mathsf{true} \mid R^{par} \\
[\![\mathsf{false}]\!]_m &\stackrel{\text{def}}{=} m \bullet \mathsf{false} \mid R^{par} \\
[\![\mathsf{cond}\ r\ s\ t]\!]_m &\stackrel{\text{def}}{=} (\nu n)(\nu o)(\nu p)(\mathsf{co}(m,n,o,p) \mid [\![\, r \,]\!]_n \mid [\![\, s \,]\!]_o \mid [\![\, t \,]\!]_p) \\
\mathsf{co}(m,n,o,p) &= n \bullet \lambda x \to o \bullet \lambda y \to p \bullet \lambda z \to m \bullet (\mathsf{cond} \bullet x \bullet y \bullet z) \ .
\end{aligned}$$

The encoding is in the same style as Milner with the $\mathsf{co}(m, n, o, p)$ process combining components of complex terms. This approach to encoding both: simplifies reduction sequences for sub-terms, as they can be reduced before combining with other sub-terms via $\mathsf{co}(m, n, o, p)$; and inherently supports parallel reduction in the style of a parallel encoding.

Theorem 6.4.4. *The encoding $[\![\cdot]\!]_m$ of $\mathcal{B}$ terms into CPC preserves the core, optional and parallel rewriting rules.*

Proof: Straightforward as in Theorems 6.4.1, 6.4.2 & 6.4.3. □

The above development illustrates how reduction and rewriting systems can be encoded into CPC by exploiting the name matching and structure of pattern-unification. It will be used to show a parallel encoding of SF-calculus into CPC in Chapter 7.

Chapter 7

Completing the Square

Support for both intensionality and concurrency places CPC at the bottom right corner of the computation square. This chapter shows how SF-calculus and π-calculus can be subsumed by CPC, and thus completes the computation square.

Down the right side of the square there is a parallel encoding (Definition 3.1.1) from SF-calculus into CPC that also maps the combinators S and F to reserved names S and F, respectively. The lack of a converse encoding is due to the sequential nature of SF-calculus and is familiar from the relation between λ_v-calculus and π-calculus in Chapter 3. Interestingly, in contrast with the parallel encoding of λ-calculus into π-calculus, the parallel encoding of SF-calculus into CPC does *not* fix a reduction strategy for SF-calculus. This is achieved by exploiting the ability of CPC to match structure and multiple names to directly encode the reduction rules for SF-calculus into an SF-reducing process, or SF-machine. In turn, this process can then

operate on translated combinators and so support reduction and rewriting in the manner of Section 6.4. By further exploiting the techniques introduced in Section 6.4 a general approach to encoding $\mathcal{O}$-combinatory logics, or rewriting systems, is developed. Although this general approach simplifies the encoding of a combinatory logic into CPC and supports parallel reduction, it does not meet the criteria for parallel encodings. Both approaches to encodings support a direct translation of SK-calculus, and thus λ-calculus, into CPC. That is, there is a diagonal from the top left corner to the bottom right corner of the computation square.

Relating π-calculus to CPC is done through valid encodings (Definition 3.2.1) and ensuring that the encoding also meets the criteria for a homomorphism. A valid encoding suffices to show that behaviour is preserved, along with other important properties related to concurrent systems (as discussed in Chapter 3). However, there are two further details: valid encodings are defined with a notion of behavioural equivalence, and valid encodings do not imply homomorphisms. For relating π-calculus to CPC no particular behaviour theory is required, thus the development of behavioural equivalence for CPC is suspended until Chapter 8. Valid encodings do not require the mapping of parallel composition to parallel composition, however this is the case for the translation presented here. To support valid encodings, CPC is augmented with a success process $\surd$ as familiar from the process calculi in Chapter 3. There is a straightforward homomorphism from π-calculus into CPC that meets all the criteria for valid encodings. The lack of a valid

encoding or homomorphism from CPC into π-calculus can be proved in two ways that exploit CPC's multiple name matching and symmetry.

7.1 SF-calculus

This section develops a parallel encoding from SF-calculus into CPC and discusses some alternative encodings and implications. Similar to the encodings in Section 6.4, the reduction rules are supported by a single *SF-reducing process* $\mathcal{R}$ that performs all the reduction rules using reserved names S and F. Since parallel encodings require components of applications be separate processes, identifying names are required to track sub-terms as in the parallel $\mathcal{B}$ reducing process of Section 6.4. Details of the SF-reducing process $\mathcal{R}$ are shown in Figure 7.1.

Observe that $\mathcal{R}$ has a replicating process for each of the axioms and reduction rules. The first seven replications instantiate the seven possible reduction axioms of SF-calculus by considering each of the factorable forms as a separate axiom. The next replication supports reduction of the first argument to F by matching a reducible structure $(\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x)$ and performing reductions upon it before recombining with the original F and m and n. The last four replications identify when sub-structures can be reduced and performs that reduction before recombining into a single structure. Observe that each pattern is headed by a binding name used to track components of a combinator being reduced before reconstruction; due to this

$$
\begin{aligned}
& !\lambda c \bullet (S \bullet \lambda m \bullet \lambda n \bullet \lambda x) \to c \bullet (m \bullet x \bullet (n \bullet x)) \\
| \;\; & !\lambda c \bullet (F \bullet S \bullet \lambda m \bullet \lambda n) \to c \bullet m \\
| \;\; & !\lambda c \bullet (F \bullet F \bullet \lambda m \bullet \lambda n) \to c \bullet m \\
| \;\; & !\lambda c \bullet (F \bullet (S \bullet \lambda q) \bullet \lambda m \bullet \lambda n) \to c \bullet (n \bullet S \bullet q) \\
| \;\; & !\lambda c \bullet (F \bullet (F \bullet \lambda q) \bullet \lambda m \bullet \lambda n) \to c \bullet (n \bullet F \bullet q) \\
| \;\; & !\lambda c \bullet (F \bullet (S \bullet \lambda p \bullet \lambda q) \bullet \lambda m \bullet \lambda n) \to c \bullet (n \bullet (S \bullet p) \bullet q) \\
| \;\; & !\lambda c \bullet (F \bullet (F \bullet \lambda p \bullet \lambda q) \bullet \lambda m \bullet \lambda n) \to c \bullet (n \bullet (F \bullet p) \bullet q) \\
| \;\; & !\lambda c \bullet (F \bullet (\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x) \bullet \lambda m \bullet \lambda n) \\
& \quad \to (\nu d) d \bullet (u \bullet v \bullet w \bullet x) \to d \bullet \lambda z \to c \bullet (F \bullet z \bullet m \bullet n) \\
| \;\; & !\lambda c \bullet (\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x \bullet \lambda y) \\
& \quad \to (\nu d) d \bullet (u \bullet v \bullet w \bullet x) \to d \bullet \lambda z \to c \bullet (z \bullet y) \\
| \;\; & !\lambda c \bullet (\lambda m \bullet \lambda n \bullet \lambda o \bullet (\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x)) \\
& \quad \to (\nu d) d \bullet (u \bullet v \bullet w \bullet x) \to d \bullet \lambda z \to c \bullet (m \bullet n \bullet o \bullet z) \\
| \;\; & !\lambda c \bullet (\lambda m \bullet \lambda n \bullet (\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x) \bullet \lambda p) \\
& \quad \to (\nu d) d \bullet (u \bullet v \bullet w \bullet x) \to d \bullet \lambda z \to c \bullet (m \bullet n \bullet z \bullet p) \\
| \;\; & !\lambda c \bullet (\lambda m \bullet (\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x) \bullet \lambda o \bullet \lambda p) \\
& \quad \to (\nu d) d \bullet (u \bullet v \bullet w \bullet x) \to d \bullet \lambda z \to c \bullet (m \bullet z \bullet o \bullet p)
\end{aligned}
$$

Figure 7.1: The SF-reducing process $\mathcal{R}$.

the SF-reducing process is irreducible.

The translation $[\![\cdot]\!]_c$ from SF-combinators into CPC processes is here parametrised by a name c and combines application with a process $\mathsf{ap}(c, m, n)$. This is similar to Milner's encoding from λ_v-calculus into π-calculus, and to the parallel encoding of $\mathcal{B}$ into CPC in Section 6.4. This allows the parallel encoding to exploit compositional encoding of sub-terms as processes and thus parallel reduction, while preventing confusion of application.

The translation $[\![\cdot]\!]_c$ of SF-combinators into CPC, exploiting the SF-reducing process $\mathcal{R}$ and reserved names S and F, is defined as follows:

$$\begin{aligned}
[\![S]\!]_c &\stackrel{\text{def}}{=} c \bullet S \mid \mathcal{R} \\
[\![F]\!]_c &\stackrel{\text{def}}{=} c \bullet F \mid \mathcal{R} \\
[\![MN]\!]_c &\stackrel{\text{def}}{=} (\nu m)(\nu n)(\mathsf{ap}(c, m, n) \mid [\![M]\!]_m \mid [\![N]\!]_n) \\
\mathsf{ap}(c, m, n) &\stackrel{\text{def}}{=} m \bullet \lambda x \to n \bullet \lambda y \to c \bullet (x \bullet y) \mid \mathcal{R} \,.
\end{aligned}$$

Observe that operators are translated to processes that consist of the channel name c and the operator. As with Milner's encoding, the application is translated to a process of the form $[\![MN]\!]_c = (\nu m)(\nu n)(R \mid [\![M]\!]_m \mid [\![N]\!]_n)$ and thus meets the structural requirements for parallel encoding.

Now all that remains is to show that the translation $[\![\cdot]\!]_c$ of an SF-combinator M preserves reduction. The simplest solution is to prove the results in the same manner as for showing that encodings of $\mathcal{B}$ into CPC preserve reduction. This is simplified by first defining the *construction* $(\!|\cdot|\!)$

of combinators and showing that there is a reduction sequence from the translation $[\![\cdot]\!]_c$ to $c \bullet (\!| \cdot |\!) \mid \mathcal{R}$.

Define the *construction* $(\!| \cdot |\!)$ of an $\mathcal{O}$-combinator into CPC, exploiting reserved names O for each $O \in \mathcal{O}$, as follows:

$$\begin{array}{rcll} (\!|O|\!) & \stackrel{\text{def}}{=} & O & O \in \mathcal{O} \\ (\!|MN|\!) & \stackrel{\text{def}}{=} & (\!|M|\!) \bullet (\!|N|\!) \ . & \end{array}$$

That is, a mapping from operators to themselves and from applications to compounds.

The following proofs are simplified by observing that $\mathcal{R} \mid \mathcal{R} \equiv \mathcal{R}$ to remove redundant copies of $\mathcal{R}$. This is also formalised in Theorem 8.7.2 by exploiting bisimulation in Chapter 8.

Lemma 7.1.1. *Given an SF-combinator M the translation $[\![M]\!]_c$ has a reduction sequence to a process of the form $c \bullet (\!|M|\!) \mid \mathcal{R}$.*

Proof: The proof is by induction on the structure of M.

- If M is S or F then $[\![M]\!]_c = c \bullet M \mid \mathcal{R} = c \bullet (\!|M|\!) \mid \mathcal{R}$ and the result is immediate.

- If M is of the form $M_1 M_2$ then $[\![M]\!]_c$ is of the form

$$(\nu m)(\nu n)(m \bullet \lambda x \to n \bullet \lambda y \to c \bullet (x \bullet y) \mid \mathcal{R} \mid [\![M_1]\!]_m \mid [\![M_2]\!]_n) \ .$$

  By two applications of induction, $[\![M_1]\!]_m \Longmapsto m \bullet (\!|M_1|\!) \mid \mathcal{R}$ and also

$[\![M_2]\!]_n \Longmapsto n \bullet (\!|M_2|\!) \mid \mathcal{R}$. Now there are reductions as follows:

$$\begin{aligned}
[\![M]\!]_c &\Longmapsto (\nu m)(\nu n)(m \bullet \lambda x \to n \bullet \lambda y \to c \bullet (x \bullet y) \mid \mathcal{R} \\
&\qquad\qquad \mid m \bullet (\!|M_1|\!) \mid n \bullet (\!|M_2|\!)) \\
&\longmapsto (\nu m)(\nu n)(n \bullet \lambda y \to c \bullet ((\!|M_1|\!) \bullet y) \mid \mathcal{R} \mid n \bullet (\!|M_2|\!)) \\
&\longmapsto (\nu m)(\nu n)(c \bullet ((\!|M_1|\!) \bullet (\!|M_2|\!)) \mid \mathcal{R}) \\
&= c \bullet (\!|M_1 M_2|\!) \mid \mathcal{R}
\end{aligned}$$

yielding the required result.

□

Theorem 7.1.2. *Given an SF-combinator M the translation $[\![M]\!]_c$ preserves reduction.*

Proof: The proof is routine, if tedious, by considering each reduction rule, as in Theorem 6.4.1 and Lemma 7.1.1. □

Corollary 7.1.3. *The translation $[\![\cdot]\!]_c$ is a parallel encoding from SF-calculus into CPC.*

Proof: Straightforward by Definition 3.1.1 and Theorem 7.1.2. □

The lack of a converse encoding from CPC into SF-calculus is as for π-calculus into λ-calculus.

Theorem 7.1.4. *There is no support for concurrency in SF-calculus.*

Proof: As in Corollary 3.1.3, exploit Theorem 14.4.12 of Barendregt [Bar85], alternatively by Lemma B of Abramsky [Abr90]. As before this exploits the ability for a process calculus to easily encode a parallel-or function, while sequential calculi cannot. □

This completes the arrow down the right side of the computation square. The rest of this section discusses some properties of translations and the diagonal from the top left to the bottom right corner of the square.

Observe that the parallel encoding from SF-calculus into CPC does not require the choice of a reduction strategy, unlike Milner's encodings from λ-calculus into π-calculus. The structure of patterns and peculiarities of pattern-unification allow the reduction relation to be directly rendered by CPC. In some sense this is similar to encoding the SF-combinators onto the tape of a Turing machine, the pattern $(\!| \cdot |\!)$, and providing another process to be the state that reads the tape and performs operations upon it, the SF-reducing process $\mathcal{R}$.

The translation from SF-calculus to CPC presented here is designed to map application to parallel composition (with some restriction and R so as to meet the criteria for parallel encoding), however the construction $(\!| \cdot |\!)$ can be used to provide a cleaner translation as for the parallel encoding of $\mathcal{B}$. Consider an alternative translation $[\![\cdot]\!]^c$ parametrised by a name c as usual and defined by

$$[\![M]\!]^c \stackrel{\text{def}}{=} (\nu c)(c \bullet (\!|M|\!)) \mid \mathcal{R} \,.$$

Although such a translation does not allow reduction of combinators before combining into a single pattern, it does preserve reduction (again by Theorem 7.1.2).

This second approach could still support parallel reduction of components by extending the SF-reducing process in the same manner as for $\mathcal{B}$. That is, additional cases to detect when two components could reduce independently and allow parallel reductions. For example, consider $FFFF(FFFF)$ that can reduce either copy of $FFFF$ in parallel. An addition to $\mathcal{R}$ to support this behaviour could be:

$$\begin{aligned}
&!\lambda c \bullet (\lambda m \bullet \lambda n \bullet \lambda o \bullet \lambda p \bullet (\lambda u \bullet \lambda v \bullet \lambda w \bullet \lambda x)) \\
&\quad \to (\nu d)(\nu d)(d \bullet (m \bullet n \bullet o \bullet p) \\
&\qquad\qquad \mid e \bullet (u \bullet v \bullet w \bullet x) \to d \bullet \lambda y \to e \bullet \lambda z \to c \bullet (y \bullet z)) \ .
\end{aligned}$$

Such a modified SF-reducing process would allow the more elegant translation to support parallel reduction of the translation. Observe that by placing the process to reconstruct the components $d \bullet \lambda y \to e \bullet \lambda z \to c \bullet (y \bullet z)$ as the body of a case, this ensures that there are no infinite sequences of reduction introduced by the translation.

Indeed, both translation styles can easily be adapted to translate SK-calculus into CPC, with an appropriately defined SK-reducing process. It follows that there is a parallel encoding from SK-calculus into CPC without requiring a specific reduction strategy (lazy or call-by-value), and thus a di-

agonal from the top left corner to the bottom right corner of the computation square.

7.2 π-calculus

Across the bottom of the computation square there is a homomorphism from π-calculus into CPC. The converse separation result can be proved in multiple ways that exploit the matching degree or symmetry of CPC.

The translation $[\![\cdot]\!]$ from π-calculus into CPC is homomorphic on all process forms except for the input and output which are translated as follows:

$$\begin{aligned} [\![a(b).P]\!] &\stackrel{\text{def}}{=} a \bullet \lambda b \bullet \mathsf{in} \to [\![P]\!] \\ [\![\overline{a}\langle b\rangle.P]\!] &\stackrel{\text{def}}{=} a \bullet b \bullet \lambda x \to [\![P]\!] \qquad x \text{ not free in } P. \end{aligned}$$

Here in is any name, a symbolic name is used for clarity but no result relies upon this choice. The fresh name x is used to prevent the introduction of new reductions due to CPC's symmetric matching. Observe that a naive translation may be $[\![\overline{a}\langle b\rangle.P]\!] \stackrel{\text{def}}{=} a \bullet b \to [\![P]\!]$, however two copies of this process would reduce as the pattern $a \bullet b$ unifies with itself. Thus the fresh binding name λx ensures that outputs cannot unify with themselves in translation.

Now to show that this translation meets the criteria for a valid encoding.

Lemma 7.2.1. *Two processes P and Q are structurally equivalent if and only if their translations are structurally equivalent. That is: $P \equiv Q$ if and only if $[\![\, P \,]\!] \equiv [\![\, Q \,]\!]$.*

Proof: Trivial, from the fact that $\equiv$ acts only on operators that $[\![\cdot]\!]$ translates homomorphically. $\square$

Theorem 7.2.2. *The translation $[\![\cdot]\!]$ from π-calculus into CPC preserves reduction and does not introduce new reductions. That is:*

- *If $P \longmapsto P'$ then $[\![P]\!] \longmapsto [\![P']\!]$;*
- *if $[\![P]\!] \longmapsto Q$ then $Q = [\![P']\!]$ for some P' such that $P \longmapsto P'$.*

Proof: Both parts can be easily proved by a straightforward induction on the judgements $P \longmapsto P'$ and $[\![P]\!] \longmapsto Q$, respectively. In both cases, the base step is the most interesting.

- For $P \longmapsto P'$ consider $a(b).P_1 \mid \overline{a}\langle c\rangle.P_2 \longmapsto \{c/b\}P_1 \mid P_2$. The translation is $[\![P]\!] = a \bullet \lambda b \bullet \mathsf{in} \to [\![P_1]\!] \mid a \bullet c \bullet \lambda x \to [\![P_2]\!]$ where x is fresh. Then there is a reduction $a \bullet \lambda b \bullet \mathsf{in} \to [\![P_1]\!] \mid a \bullet c \bullet \lambda x \to [\![P_2]\!] \longmapsto \{c/b\}[\![P_1]\!] \mid \{\mathsf{in}/x\}[\![P_2]\!]$ that is equivalent to $\{c/b\}[\![P_1]\!] \mid [\![P_2]\!]$ by freshness of x.
- For $[\![P]\!] \longmapsto Q$ the reduction must be of the form $a \bullet \lambda b \bullet \mathsf{in} \to [\![P_1]\!] \mid a \bullet c \bullet \lambda x \to [\![P_2]\!]) \longmapsto \{c/b\}[\![P_1]\!] \mid \{\mathsf{in}/x\}[\![P_2]\!]$. That is, the translation of an input and an output in parallel with each other. Further, as x must be fresh from the translation then $Q = \{c/b\}[\![P_1]\!] \mid [\![P_2]\!]$. Since the reduction is of a translated input and a translated output, then P must be of the form $a(b).P_1 \mid \overline{a}\langle c\rangle.P_2$ and there is a reduction $a(b).P_1 \mid \overline{a}\langle c\rangle.P_2 \longmapsto \{c/b\}P_1 \mid P_2$ and $P' = \{c/b\}P_1 \mid P_2$ and so $Q = [\![P']\!]$ as required.

For the inductive cases where the last judgement is structural then apply Lemma 7.2.1. □

Corollary 7.2.3. *The encoding of π-calculus into CPC is valid.*

Proof: Compositionality and name invariance hold by construction. Operation correspondence and divergence reflection follow from Theorem 7.2.2. Success sensitiveness is proved as follows: $P \Downarrow$ means there exists P' and $k \geq 0$ such that $P \longmapsto^k P' \equiv P'' \mid \surd$ and by exploiting Theorem 7.2.2 k times and Lemma 7.2.1 obtain that $[\![P]\!] \longmapsto^k [\![P']\!] \equiv [\![P'']\!] \mid \surd$, that is that $[\![P]\!] \Downarrow$. The converse implication can be proved similarly. □

Corollary 7.2.4. *There is a homomorphism from π-calculus into CPC.*

Proof: Trivial by the definition of translation $[\![\cdot]\!]$ and Corollary 7.2.3. □

Thus the translation provided above is a homomorphism from π-calculus into CPC. Now consider the converse separation result. It can be proved in two ways.

The first exploits the matching degree $\textsc{Md}(\cdot)$ and is an application of Theorem 3.2.4.

Theorem 7.2.5. *There is no valid encoding of CPC into π-calculus.*

Proof:[by matching degree] The matching degree of π-calculus $\textsc{Md}(\pi)$ is one while the matching degree of CPC $\textsc{Md}(CPC)$ is infinite. Now apply Theorem 3.2.4. □

An alternative proof of Theorem 7.2.5 exploits CPC's symmetry by defining a *self matching* process $x \to \surd$ and then using Theorem 3.2.3.

Proof:[by symmetry] The self-matching CPC process $S = x \to \surd$ is such that $S \not\longmapsto$ and $S \not\Downarrow$, however $S \mid S \longmapsto$ and $S \mid S \Downarrow$. Every π-calculus process T is such that if $T \mid T \longmapsto$ then $T \longmapsto$. Hence by Theorem 3.2.3 there is no process $T = [\![S]\!]$. □

These results ensure that π-calculus cannot validly encode the behaviour of CPC. Indeed, they both show that the reduction of CPC cannot be properly rendered in π-calculus. Thus they support the lack of a homomorphism from CPC into π-calculus.

Theorem 7.2.6. *There is no homomorphism from CPC into π-calculus.*

Proof: By Theorem 7.2.5. □

This completes the arrow across the bottom of the computation square and thus the square as a whole.

Observe that the results here, despite using valid encodings, do not rely on any particular behavioural theory for CPC. However, relations to other process calculi do require such a behavioural theory. The next chapter formalises barbed congruence and bisimulation for CPC as a precursor to relating CPC to other process calculi and implementing CPC in a programming language.

Chapter 8

Behavioural Theory

A behavioural theory is needed for CPC to define what it means for two processes to be equivalent, that is, to behave the same way. This can be done by adapting standard approaches [MPW92, MS92, HY95, BGZZ98, WG04] to take into account the subtleties of CPC interaction. The development here is done in five steps as follows.

The first step in Section 8.1 is to characterise the interactions a CPC process can participate in. This is done by defining *barbs* that capture information about how a process can interact with another process. Commonly, the barb is parametrised by the channel name when an interaction is channel based [MPW92, ACS98, SW01, WG04], however in calculi that can test many names for equality in a single interaction the barb is parametrised by a collection of names [BGZZ98]. As CPC can test many names for equality in a single interaction, it follows that the barbs of CPC are parametrised by a set of names. Once the barbs of CPC are defined the *barbed congruence*

will be defined in the usual manner.

The second step in Section 8.2 is to define an alternative operational semantics for CPC in the form of a *labelled transition system* (LTS). Here *transitions* of the form $P \xrightarrow{\mu} P'$ indicate that the process P performs some *action* described by the *label* μ and evolves to the process P'. Such labels can be either: internal actions that indicate reduction within the process P; or external actions that indicate interaction with another process. The main property that must hold is that the internal actions induce the same operational semantics as the reduction relation.

The third step in Section 8.3 is to define a *bisimulation* relation that equates processes with the same interactional behaviour as represented by the labels of the LTS. The complexity here is that labels include patterns and some patterns are more general than others. For example, a transition of the form $P \xrightarrow{\ulcorner n \urcorner} P'$ performs the external action $\ulcorner n \urcorner$, however a similar external action of another process could be n and the transition $Q \xrightarrow{n} Q'$. If P is behaviourally equivalent to Q, and also P' to Q', then this should be accepted by the bisimulation as it would be by the barbed congruence. Thus a *compatibility* relation is defined on patterns that is then used to formalise acceptable actions for the definition of bisimulation.

A diversion in Section 8.4 formalises several properties of the compatibility relation for later exploitation and to illustrate relations between patterns in general. In particular, that compatibility preserves the information used for defining barbs is, stable under substitution, reflexive, and transitive.

The fourth step in Section 8.5 is to show that the bisimulation relation is sound, i.e. a barbed congruence. That is, the bisimulation is barb preserving, reduction closed, and context closed.

The fifth step in Section 8.6 is to show that the bisimulation relation is complete, i.e. the barbed congruence is a bisimulation. This requires the development of reply contexts that can yield reductions according to the limitations on transitions defined by the bisimulation.

The chapter concludes with some examples of behaviourally equivalent processes and some general results.

8.1 Barbed Congruence

The first step is to characterise the interactions a CPC process can participate in. This is done by defining *barbs* that have information about a particular interaction that can occur. In process calculi that use channels for interaction the barbs are commonly parametrised by the channel name that is tested for equality in an interaction, such as in π-calculus [MPW92, MS92, HY95], Spi calculus [AG97] and fusion calculus [WG04]. When interactions can test many names for equality the barbs usually have information about all these names, such as in Linda [BGZZ98].

As protected names in CPC are used to test for equality, a naive attempt to define the barbs would be to use these names. However, as variable names can also be tested for equality in an interaction these also need to be ac-

counted for by the definition of barbs. Thus in CPC the barbs $P \downarrow_{\widetilde{m}}$ indicate that there exists a process Q of the form $q \to Q'$ such that the protected names of q are the names of the barb $\widetilde{m}$ and that P and Q interact. More formally, barbs are defined as follows.

Definition 8.1.1 (Barb)**.** *Let* $P \downarrow_{\widetilde{m}}$ *mean that*

$$P \equiv (\nu\widetilde{n})(p \to P' \mid P'') \qquad \begin{array}{l} \textit{for some } \widetilde{n} \textit{ and } p \textit{ and } P' \textit{ and } P'' \\ \textit{such that } \mathsf{pn}(p) \cap \widetilde{n} = \{\} \textit{ and } \widetilde{m} = \mathsf{fn}(p) \backslash \widetilde{n}. \end{array}$$

To explain the conditions within the definition of barbs, consider some naive attempts and their limitations. As a barb is a potential interaction, a simplistic barb $P \downarrow$ could be

$$\begin{array}{l} \text{Naive Bad Barb Attempt: } P \downarrow \text{ if} \\ \quad P \equiv p \to P' \mid P'' \qquad \text{for some } p \text{ and } P' \text{ and } P''. \end{array} \tag{8.1}$$

However, this attempted definition is too strong; there are processes that can interact with an external process but do not exhibit a barb by (8.1). For example, the process $(\nu n)(n \to P)$ does not exhibit a barb, but can interact with another process of the form $\lambda y \to Q$. Thus, a first attempt to improve might be

$$\begin{array}{l} \text{Unprotected Bad Barb Attempt: } P \downarrow \text{ if} \\ P \equiv (\nu\widetilde{n})(p \to P' \mid P'') \qquad \text{for some } \widetilde{n} \text{ and } p \text{ and } P' \text{ and } P''. \end{array} \tag{8.2}$$

Now this attempt is too weak; there are processes that exhibit a barb by this definition but cannot interact with an external process. For example, the process $(\nu n)(\ulcorner n \urcorner \to P)$, that cannot interact with any other process by Lemma 6.2.6. A further refinement of (8.2) could be

$$\begin{array}{l} \text{Protected Bad Barb Attempt: } P \downarrow \text{ if} \\ P \equiv (\nu \widetilde{n})(p \to P' \mid P'') \qquad \text{for some } \widetilde{n} \text{ and } p \text{ and } P' \text{ and } P'' \\ \qquad\qquad\qquad\qquad\quad \text{such that } \mathsf{pn}(p) \cap \widetilde{n} = \{\}. \end{array} \tag{8.3}$$

Although this solves the problem of (8.2), having a single kind of barb is insufficient. As is usual in name passing calculi, including all those considered in this dissertation, the barb accounts for the names that must be known for interaction. As CPC also supports matching of many names in a single interaction, the definition of barbs must account for these names. Thus, the definition of barbs must be parametrised by the names of the pattern being considered. Although a first attempt might be to simply take the free names of the pattern p, this is insufficient as a name may appear both free and protected while also being restricted, e.g. $(\nu n) n \bullet \ulcorner n \urcorner \to P'$. So refining (8.3) to account for all this yields the barbs as in Definition 8.1.1.

Thus the original definition of barbs is sufficient to characterise potential interactions and begin defining process equivalence. For later use, let $P \Downarrow_{\widetilde{m}}$ denote that there exists a reduction sequence $P \Longmapsto P'$ such that $P' \downarrow_{\widetilde{m}}$.

Using the definition of barbs for CPC, a barbed congruence can be defined in the standard manner by requiring three properties [MS92, HY95, BGZZ98,

WG04]. Let $\Re$ denote a binary relation on CPC processes, and let a *context* $\mathcal{C}(\cdot)$ be a CPC process with the hole $\cdot$ replacing one instance of the null process.

Definition 8.1.2 (Barb preservation)**.** $\Re$ *is barb preserving if and only if, for every pair of processes* $(P, Q) \in \Re$*, it holds that* $P \downarrow_{\widetilde{m}}$ *implies* $Q \downarrow_{\widetilde{m}}$*.*

Definition 8.1.3 (Reduction closure)**.** $\Re$ *is reduction closed if and only if, for every pair of processes* $(P, Q) \in \Re$*, it holds that* $P \longmapsto P'$ *implies* $Q \longmapsto Q'$ *for some* Q' *such that* $(P', Q') \in \Re$*.*

Definition 8.1.4 (Context closure)**.** $\Re$ *is context closed if and only if, for every pair of processes* $(P, Q) \in \Re$ *and for every CPC context* $\mathcal{C}(\cdot)$*, it holds that* $(\mathcal{C}(P), \mathcal{C}(Q)) \in \Re$*.*

Definition 8.1.5 (Barbed congruence)**.** *Barbed congruence* $\simeq$ *is the largest, symmetric, barb preserving, reduction and context closed binary relation on CPC processes.*

This is the usual barbed congruence that equates processes with the same interactional behaviour as characterised by barbs. That is: any barb of one process must also be a barb of the other; any reduction of one process must be able to be mimicked by the other; and context closure ensures equivalence under any circumstance.

This defines the *strong* version of barbed congruence, the *weak* counterpart consists of replacing the predicate $\longmapsto$ with $\Longrightarrow$ in Definition 8.1.3, and $\downarrow_{\widetilde{m}}$ with $\Downarrow_{\widetilde{m}}$ in Definition 8.1.2 in the usual manner [MPW92, MS92].

The following proofs are simplified by working in the strong setting, however everything can be rephrased in the weak setting, as usual in concurrency.

Many later results will exploit contexts that perform some actions and report success and do not report failure. The name w is used with a barb $\downarrow_w$ indicating *success* and $\Downarrow_w$ for a reduction sequence that eventually indicates success. The name f is used similarly for *failure*. A process P *succeeds* (alternatively *succeeds by* w *and* f) if it reports success and does not report failure, i.e. $P \Downarrow_w$ and $P \not\Downarrow_f$.

The next two lemmas suffice to show that the barbed congruence is closed under any substitution.

Lemma 8.1.6. *Consider two processes P and Q and a name w which does not appear free in P or Q. If P in parallel with $\ulcorner w \urcorner$ is barbed congruent to Q in parallel with $\ulcorner w \urcorner$ then P is barbed congruent to Q. That is: if $P \mid \ulcorner w \urcorner \simeq Q \mid \ulcorner w \urcorner$ then $P \simeq Q$.*

Proof: Define a context $\mathcal{C}(\cdot) \stackrel{\text{def}}{=} (\ulcorner w \urcorner \mid \cdot\ )$. By context closure $\mathcal{C}(P \mid \ulcorner w \urcorner) \simeq \mathcal{C}(Q \mid \ulcorner w \urcorner)$. There is a reduction $\mathcal{C}(P \mid \ulcorner w \urcorner) \longmapsto P$ and by reduction closure there is a reduction $\mathcal{C}(Q \mid \ulcorner w \urcorner) \longmapsto Q'$ such that $P \simeq Q'$. Now consider Q': if $Q' \equiv Q'' \mid \ulcorner w \urcorner$ then $Q' \downarrow_w$ however this contradicts $P \not\downarrow_w$. Therefore Q' must be Q and hence $P \simeq Q$. □

Lemma 8.1.7. *Given two barbed congruent processes P and Q and a substitution σ, then P and Q remain barbed congruent when σ is applied to them. That is: if $P \simeq Q$ then $\sigma P \simeq \sigma Q$.*

Proof: Since σ has finite domain, there are patterns p and q such that $\{p \| q\} = (\sigma, \{\})$. Given a fresh name w define a context $\mathcal{C}(\cdot) \stackrel{\text{def}}{=} (p \to (\cdot \mid \ulcorner w \urcorner) \mid q)$. For any process R there is a reduction from $\mathcal{C}(R)$ to $\sigma R \mid \ulcorner w \urcorner$. It follows with reduction closure that $\mathcal{C}(P) \longmapsto \sigma P \mid \ulcorner w \urcorner$ and $\mathcal{C}(Q) \longmapsto \sigma Q \mid \ulcorner w \urcorner$. Conclude by Lemma 8.1.6 that $\sigma P \simeq \sigma Q$. □

The complexity in proving (strong or weak) barbed congruence is in its closure under any context. The typical way of solving this problem is by giving a coinductive (bisimulation-based) characterisation that yields an easier-to-handle proof technique. In turn, this requires an alternative operational semantics, by means of an LTS, on top of which the bisimulation equivalence can be defined.

8.2 Labelled Transition System

The second step in developing CPC's behavioural theory is to define an alternative operational semantics in the form of a *labelled transition system* (LTS). This section details a straightforward adaption of the standard π-calculus *late* LTS [MPW92]. The rest of the section defines the labels for CPC, the judgements for inferring transitions, and then proves the operational semantics for internal transitions is the same as CPC reductions.

The basic idea is to describe the behaviour of CPC by *transitions* between processes. The transitions describe interactions which the process can participate in which can be within the process or with some external pro-

cess. Specific information about the interaction is carried by a *label* on the transition.

The labels μ for CPC's LTS are defined as follows:

$$\mu \ ::= \ \tau \quad | \quad (\nu\widetilde{n})p \ .$$

Here τ denotes an *internal action*; that is an interaction entirely within the process, as is standard. The *external action* $(\nu\widetilde{n})p$ denotes an interaction between the process and some other process. The external actions carry information about the restricted names $(\nu\widetilde{n})$ and pattern p; a standard adaptation of the usual input or output actions of π-calculus [MPW92].

A *transition* is a judgement of the form $P \xrightarrow{\mu} P'$ where P transitions to P' via the label μ. The label denotes whether the actions is internal or external, consider the following examples.

An example of an internal action would be a process P of the form $p_1 \to P_1 \mid p_2 \to P_2$ where $\{p_1 \| p_2\}$ is defined and yields (σ_1, σ_2). Then a transition for P is $P \xrightarrow{\tau} P'$ where the τ represents the internal actions and $P' = \sigma_1 P_1 \mid \sigma_2 P_2$.

An example of an external action would be a process P of the form $p \to P'$. Here P has a transition p to P', denoted $P \xrightarrow{p} P'$. Observe that external actions do not assume any substitution upon their binding names, so the binding names in p are free in P'. It follows that some care must be taken when considering the binding names of labels.

Define the variable names, protected names, binding names and free names of a label μ to be those of the pattern when μ is of the form $(\nu\widetilde{n})p$ and to be the empty set otherwise. Also the restricted names of a label μ are those that are restricted when μ is of the form $(\nu\widetilde{n})p$, i.e. $\widetilde{n}$, and the empty set when μ is τ. Lastly, define the *names* $\mathsf{names}(\mu)$ of a label μ to be all the names (restricted, free, or binding) of μ.

$$\mathsf{parint}: \quad \frac{P \xrightarrow{\tau} P'}{P \mid Q \xrightarrow{\tau} P' \mid Q}$$

$$\mathsf{parext}: \quad \frac{P \xrightarrow{(\nu\widetilde{n})p} P'}{P \mid Q \xrightarrow{(\nu\widetilde{n})p} P' \mid Q} \quad (\widetilde{n} \cup \mathsf{bn}(p)) \cap \mathsf{fn}(Q) = \{\}$$

$$\mathsf{rep}: \quad \frac{!P \mid P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'}$$

$$\mathsf{resnon}: \quad \frac{P \xrightarrow{\mu} P'}{(\nu n)P \xrightarrow{\mu} (\nu n)P'} \quad n \notin \mathsf{names}(\mu)$$

$$\mathsf{resin}: \quad \frac{P \xrightarrow{(\nu\widetilde{n})p} P'}{(\nu m)P \xrightarrow{(\nu\widetilde{n},m)p} P'} \quad m \in \mathsf{vn}(p) \setminus (\widetilde{n} \cup \mathsf{pn}(p) \cup \mathsf{bn}(p))$$

$$\mathsf{case}: \quad (p \to P) \xrightarrow{p} P$$

$$\mathsf{match}: \quad \frac{P \xrightarrow{(\nu\widetilde{m})p} P' \quad Q \xrightarrow{(\nu\widetilde{n})q} Q'}{P \mid Q \xrightarrow{\tau} (\nu\widetilde{m},\widetilde{n})(\sigma P' \mid \rho Q')} \quad \begin{array}{l} \{p \| q\} = (\sigma, \rho) \\ \widetilde{m} \cap \mathsf{fn}(Q) = \widetilde{n} \cap \mathsf{fn}(P) = \{\} \\ \widetilde{m} \cap \widetilde{n} = \{\} \end{array}$$

Figure 8.1: Labelled Transition System judgements for CPC.

The rules for inferring transitions of the form $P \xrightarrow{\mu} P'$ are defined in Figure 8.1. The first rule parint states that if either process in parallel composition can transition by an internal action then the whole process can transition by an internal action. The second rule parext is for when one of the processes in parallel has an external action, then both processes have the same external action as long as the restricted or binding names of the label do not appear free in the parallel process. The third rule rep unfolds replication to infer the action. The fourth rule resnon is for when a restricted name does not appear in the names of the label. Simply maintain the restriction on the process after the transition. The fifth rule resin is for when a restricted name is shared by the label. As the restricted name has been shared and may now be used by other processes, the restriction is removed. The sixth rule case states that a case's pattern can be used to interact with external processes. The seventh rule match defines when processes interact with each other to perform an internal action. This can occur whenever process in parallel have labels that have unifiable patterns and there is no possibility of clash or capture due to restricted names. Note that α-conversion is always assumed to satisfy the side conditions when required and that symmetric instances of parint and parext are assumed, i.e. transitions on Q.

The main result that must be proved is that the LTS induces the same operation semantics as the reductions of CPC. As CPC reductions only involve interaction between processes and not the external actions of the LTS, it is sufficient to show that any internal action of the LTS is mimicked by a

reduction in CPC, and visa versa.

Lemma 8.2.1. *If there is a transition* $P \xrightarrow{(\nu\widetilde{m})p} P'$ *then* P *has a form* $(\nu\widetilde{m})(\nu\widetilde{n})(p \to Q_1 \mid Q_2)$ *for some* $\widetilde{n}$ *and* Q_1 *and* Q_2 *with* $\widetilde{n} \cap \mathsf{names}((\nu\widetilde{m})p) = \{\}$ *and* $\mathsf{bn}(p) \cap \mathsf{fn}(Q_2) = \{\}$.

Proof: The proof is by induction on the structure of the inference for $P \xrightarrow{(\nu\widetilde{m})p} P'$, considering the last rule in the derivation.

- If the last rule is parext then P has a form $P_1 \mid P_2$ and $P_1 \xrightarrow{(\nu\widetilde{m})p} P_1'$ and $\mathsf{fn}(P_2) \cap (\widetilde{m} \cup \mathsf{bn}(p)) = \{\}$. By induction hypothesis on $P_1 \xrightarrow{(\nu\widetilde{m})p} P_1'$ it follows that P_1 has a form $(\nu\widetilde{m})(\nu\widetilde{n}')(p \to Q_1' \mid Q_2')$ for some $\widetilde{n}' \cap \mathsf{names}((\nu\widetilde{m})p) = \{\}$ and $(\widetilde{m} \cup \mathsf{bn}(p)) \cap \mathsf{fn}(Q_2) = \{\}$. As $\mathsf{bn}(p) \cap \mathsf{fn}(P_2)$ already holds, conclude with $\widetilde{n}'$ and Q_1' and $Q_2' \mid P_2$.

- If the last rule is rep then P has a form $!Q \xrightarrow{(\nu\widetilde{m})p} P'$ where there is a transition $Q \mid !Q \xrightarrow{(\nu\widetilde{m})p} P'$. Conclude by induction and the fact that $P \equiv Q \mid !Q$.

- If the last rule is resnon then P has a form $(\nu o)P_1 \xrightarrow{(\nu\widetilde{m})p} (\nu o)P_1'$ where $o \notin \mathsf{names}((\nu\widetilde{m})p)$. By induction on $P_1 \xrightarrow{(\nu\widetilde{m})p} P_1'$ then P_1 has a form $(\nu\widetilde{m})(\nu\widetilde{n}')(p \to Q_1' \mid Q_2')$ for some $\widetilde{n}' \cap \mathsf{names}((\nu\widetilde{m})p) = \{\}$ and $\mathsf{bn}(p) \cap \mathsf{fn}(Q_2) = \{\}$. As $o \notin \mathsf{names}((\nu\widetilde{m})p)$ and by α-conversion $o \notin \widetilde{n}'$ conclude with $\widetilde{n}', o$ and Q_1' and Q_2'.

- If the last rule is resin then P has a form $(\nu o)P_1 \xrightarrow{(\nu\widetilde{m}', o)p} P_1'$ and $o \in \mathsf{vn}(p) \backslash (\widetilde{m}' \cup \mathsf{pn}(p) \cup \mathsf{bn}(p))$ and $P_1 \xrightarrow{(\nu\widetilde{m}')p} P_1'$ and $\widetilde{m} = \widetilde{m}', o$. Then by induction hypothesis upon $P_1 \xrightarrow{(\nu\widetilde{m}')p} P_1'$ it follows that P_1 has a

form $(\nu\widetilde{m}')(\nu\widetilde{n}')(p \to Q_1' \mid Q_2')$ for some $\widetilde{n}' \cap \mathsf{names}((\nu\widetilde{m}')p) = \{\}$ and $\mathsf{bn}(p) \cap \mathsf{fn}(Q_2) = \{\}$. Conclude with $\widetilde{n}'$ and Q_1' and Q_2'.

- The base case is when the last rule is case and P has a form $(p \to P_1) \xrightarrow{p} P_1$. Conclude with $\{\}$ and P_1 and $\mathbf{0}$.

□

Proposition 8.2.2. *If $P \xrightarrow{\tau} P'$ then $P \longmapsto P''$ such that $P' \equiv P''$. Conversely, if $P \longmapsto P'$ then $P \xrightarrow{\tau} P''$ such that $P' \equiv P''$.*

Proof: The first claim is proved by induction on the structure of the inference for $P \xrightarrow{\tau} P'$.

- If the last rule is parint then P has a form $P_1 \mid P_2$ for $P_1 \xrightarrow{\tau} P_1'$ and $P' = P_1' \mid P_2$. Apply induction to the transition $P_1 \xrightarrow{\tau} P_1'$ to show that $P_1 \longmapsto P_1'$.

- If the last rule is rep then P has a form $!P_1$, for $P_1 \mid !P_1 \xrightarrow{\tau} P'$. By induction $P_1 \mid !P_1 \longmapsto P'$ and easily conclude, since $P \equiv P_1 \mid !P_1$.

- If the last rule is resnon then P has a form $(\nu n)P_1$, for $P_1 \xrightarrow{\tau} P_1'$ and $P' = (\nu n)P_1'$. Again proceed by induction on the transition $P_1 \xrightarrow{\tau} P_1'$ since (νn) preserves reduction.

- If the rule is match then P has a form $P_1 \mid Q_1$ where $P_1 \xrightarrow{(\nu\widetilde{m})p} P_1''$ and $Q_1 \xrightarrow{(\nu\widetilde{n})q} Q_1''$ and $P' = (\nu\widetilde{m}, \widetilde{n})(\sigma P_1'' \mid \rho Q_1'')$ and $\{p \| q\} = (\sigma, \rho)$ and $\widetilde{m} \cap \mathsf{fn}(Q_1) = \widetilde{n} \cap \mathsf{fn}(P_1) = \{\}$ and $\widetilde{m} \cap \widetilde{n} = \{\}$. Then by two applications of Lemma 8.2.1 it follows that P_1 has a form $(\nu\widetilde{m})(\nu\widetilde{o})(p \to P_1' \mid P_2')$ such that $\widetilde{o} \cap \mathsf{names}((\nu\widetilde{n})p) = \{\}$ and $\mathsf{bn}(p) \cap \mathsf{fn}(P_2') = \{\}$ and Q_1 has

a form $(\nu\widetilde{n})(\nu\widetilde{r})(q \to Q_1' \mid Q_2')$ such that $\widetilde{r} \cap \mathsf{names}((\nu\widetilde{n})q) = \{\}$ and $\mathsf{bn}(q) \cap \mathsf{fn}(Q_2') = \{\}$. Since $\widetilde{o}, \widetilde{r} \cap (\mathsf{names}((\nu\widetilde{m})p) \cup \mathsf{names}((\nu\widetilde{n})q)) = \{\}$ and exploiting α-conversion on the name in $\widetilde{o}, \widetilde{n}$ it follows that

$$\begin{aligned} P_1 \mid Q_1 \qquad & (\nu\widetilde{m}, \widetilde{n})((\nu\widetilde{o})(p \to P_1' \mid P_2') \mid (\nu\widetilde{r})(q \to Q_1' \mid Q_2')) \\ \equiv \; & (\nu\widetilde{m}, \widetilde{n})(\nu\widetilde{o}, \widetilde{r})(p \to P_1' \mid P_2' \mid q \to Q_1' \mid Q_2') \; . \end{aligned}$$

Now there is a reduction

$$\begin{aligned} P_1 \mid Q_1 \quad \equiv \quad & (\nu\widetilde{m}, \widetilde{n})(\nu\widetilde{o}, \widetilde{r})(p \to P_1' \mid P_2' \mid q \to Q_1' \mid Q_2') \\ \longmapsto \quad & (\nu\widetilde{m}, \widetilde{n})(\nu\widetilde{o}, \widetilde{r})(\sigma P_1' \mid P_2' \mid \rho Q_1' \mid Q_2') \; . \end{aligned}$$

Since σ avoids $\widetilde{o}$ and $\mathsf{dom}(\sigma) \cap \mathsf{fn}(P_2') = \{\}$ and ρ avoids $\widetilde{r}$ and $\mathsf{dom}(\rho) \cap \mathsf{fn}(Q_2') = \{\}$ and $\widetilde{o} \cap \mathsf{fn}(Q_1' \mid Q_2') = \{\}$ and $\widetilde{r} \cap \mathsf{fn}(P_1' \mid P_2') = \{\}$, conclude with

$$\begin{aligned} & (\nu\widetilde{m}, \widetilde{n})(\nu\widetilde{o}, \widetilde{r})(\sigma P_1' \mid P_2' \mid \rho Q_1' \mid Q_2') \\ \equiv \; & (\nu\widetilde{m}, \widetilde{n})(\sigma((\nu\widetilde{o})(P_1' \mid P_2')) \mid \rho((\nu\widetilde{r})(Q_1' \mid Q_2'))) \\ \equiv \; & (\nu\widetilde{m}, \widetilde{n})(\sigma P_1'' \mid \rho Q_1'') \; . \end{aligned}$$

The second claim is by induction on the inference for $P \longmapsto P'$.

- The base case is when P is of the form $p \to P_1' \mid q \to Q_1'$ and $P' = \sigma P_1' \mid \rho Q_1'$, for $\{p \| q\} = (\sigma, \rho)$. Then by the match rule in the LTS

$$\frac{(p \to P_1') \xrightarrow{p} P_1' \quad (q \to Q_1') \xrightarrow{q} Q_1'}{p \to P_1' \mid q \to Q_1' \quad \xrightarrow{\tau} \quad \sigma P_1' \mid \rho Q_1'} \{p \| q\} = (\sigma, \rho)$$

and the result is immediate.

- If the reduction is due to $P = P_1 \mid P_2$, where $P_1 \longmapsto P_1'$ and $P' = P_1' \mid P_2$, then exploit the parint rule for

$$\frac{P_1 \xrightarrow{\tau} P_1'}{P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2}$$

proceed by induction on $P_1 \longmapsto P_1'$.

- If the reduction is $P = (\nu\widetilde{n})P_1$, where $P_1 \longmapsto P_1'$ and $P' = (\nu\widetilde{n})P_1'$, then for each $n \in \widetilde{n}$ exploit the resnon rule

$$\frac{P_1 \xrightarrow{\tau} P_1'}{(\nu n)P_1 \xrightarrow{\tau} (\nu n)P_1'}$$

as $n \notin \mathsf{names}(\tau)$. Then apply induction on $P_1 \longmapsto P_1'$.

- Otherwise, it must be that $P \equiv Q \longmapsto Q' \equiv P'$. Now proceed by an induction on the inference of the judgement $P \equiv Q$. The following are the two most interesting base cases as the other cases are easier, including the inductive case.

  - If P is of the form $!R$ and $P \equiv R \mid !R = Q$ then by the first induction hypothesis (the length of the inference for the reduction) it follows that $R \mid !R \xrightarrow{\tau} Q''$ for $Q'' \equiv Q'$. By the rep rule of the LTS $P \xrightarrow{\tau} Q''$ and conclude with $Q'' \equiv Q' \equiv P'$.
  - If P is of the form $(\nu n)P_1 \mid \ P_2$ and $P \equiv (\nu n)(P_1 \mid \ P_2) = Q$ since $n \notin \mathsf{fn}(P_2)$ then by the first inductive hypothesis it follows that $(\nu n)(P_1 \mid \ P_2) \xrightarrow{\tau} Q''$ for $Q'' \equiv Q'$. Now by definition of the

LTS the last rule used in this inference must be resnon and so $P_1 \mid \ P_2 \xrightarrow{\tau} Q'''$ and $Q'' = (\nu n)Q'''$. There are now three possible ways to generate the latter τ transition.

* If $P_1 \xrightarrow{\tau} P_1'$ and $Q''' = P_1' \mid P_2$ then it follows that

$$\frac{\dfrac{P_1 \xrightarrow{\tau} P_1'}{(\nu n)P_1 \xrightarrow{\tau} (\nu n)P_1'}}{P = (\nu n)P_1 \mid P_2 \xrightarrow{\tau} (\nu n)P_1' \mid P_2}$$

  and conclude by observing that $(\nu n)P_1' \mid P_2 \equiv (\nu n)(P_1' \mid P_2) = Q'' \equiv Q' \equiv P'$.

* If $P_2 \xrightarrow{\tau} P_2'$ and $Q''' = P_1 \mid P_2'$ then this case is similar to the previous one, but simpler.

* If $P_1 \xrightarrow{(\nu \widetilde{m})p} P_1'$ and $P_2 \xrightarrow{(\nu \widetilde{n})q} P_2'$ and $Q''' = (\nu \widetilde{m}, \widetilde{n})(\sigma P_1' \mid \rho P_2')$ where $\{p \| q\} = (\sigma, \rho)$ and $\widetilde{m} \cap \mathsf{fn}(P_2') = \widetilde{n} \cap \mathsf{fn}(P_1') = \{\}$ and $\widetilde{m} \cap \widetilde{n} = \{\}$ then this case is similar to the base case of the first claim of this Proposition and, essentially, relies on Lemma 8.2.1. The details are left to the interested reader.

□

This suffices to show that the LTS of internal actions has the same operational semantics as the reductions. The next step is to define the bisimulation relation based on the LTS.

8.3 Bisimulation

The third step is to develop a *bisimulation* relation for CPC that equates processes with the same interactional behaviour as captured by the labels of the LTS. The complexity for CPC is that the labels for external actions contain patterns, and some patterns are more general than others. For example, a transition $P \xrightarrow{\ulcorner n \urcorner} P'$ performs the action $\ulcorner n \urcorner$, however a similar external action of another process could be the variable name n and the transition $Q \xrightarrow{n} Q'$. Both transitions have the same barb, that is $P \downarrow_n$ and $Q \downarrow_n$, however their labels are not identical. Thus, a *compatibility* relation is defined on patterns that can be used to develop the bisimulation. The rest of this section discusses the development of compatibility and concludes with the definition of bisimulation for CPC.

Similarity of processes two processes P and Q can be captured by a challenge-reply game based upon the actions the processes can take. One process, say P, issues a *challenge* and evolves to a new state P'. Now Q must perform an action that is a *proper reply* and evolve to a state Q'. If Q cannot perform a proper reply then the challenge issued by P can distinguish P and Q, that is, show they are not equivalent. If Q can properly reply then the game continues with the processes P' and Q'. Two processes are bisimilar (or equivalent) if the game can always continue, or neither process can perform any actions.

The main complexity in defining a bisimulation to capture this challenge-

reply game is the choice of actions, i.e. challenges and replies. For a barbed congruence, the barbs provide the definition of actions. For an LTS, the labels describe the actions that a process can perform. Although this is straightforward for the barbs of CPC, the relationship between the challenge and the reply in the LTS requires some delicacy to formulate.

In most process calculi, a challenge is answered with an identical action as the proper reply [Mil89, MPW92]. However, there are situations in which an exact reply would make the bisimulation equivalence too fine for characterising barbed congruence [ACS98, DGP07]. This is due to impossibility for the language contexts to force barbed congruent processes to execute the same action; in such calculi more liberal replies must be allowed. That CPC lies in this second group of calculi as demonstrated by the following two examples.

Example 1 Consider the processes

$$\begin{aligned} P &= \lambda x \bullet \lambda y \to x \bullet y \to \mathbf{0} \\ Q &= \lambda z \to z \to \mathbf{0} \end{aligned}$$

where P exhibits the challenge $P \xrightarrow{\lambda x \bullet \lambda y} (x \bullet y \to \mathbf{0})$. One may think that a possible context $\mathcal{C}_{\lambda x \bullet \lambda y}(\cdot)$ to enforce a proper reply could be $\cdot \mid w \bullet w \to \ulcorner w \urcorner \to \mathbf{0}$, for w fresh. Indeed, $\mathcal{C}_{\lambda x \bullet \lambda y}(P) \longmapsto w \bullet w \to \mathbf{0} \mid \ulcorner w \urcorner \to \mathbf{0}$ and the latter process exhibits a barb over w. However, the exhibition of action $\lambda x \bullet \lambda y$ is *not* necessary for the production of such a barb: as $\mathcal{C}_{\lambda x \bullet \lambda y}(Q) \longmapsto w \bullet w \to \mathbf{0} \mid \ulcorner w \urcorner \to \mathbf{0}$, but in doing so Q performs λz instead of $\lambda x \bullet \lambda y$.

Example 2 Consider the processes

$$\begin{aligned} P &= \ulcorner n \urcorner \to \mathbf{0} \\ Q &= n \to \mathbf{0} \end{aligned}$$

and with the possible context $\mathcal{C}_{\ulcorner n \urcorner}(\cdot) \longmapsto \mathbf{0} \mid \ulcorner w \urcorner \to \mathbf{0}$, for w fresh. Although $\mathcal{C}_{\ulcorner n \urcorner}(P) \longmapsto \mathbf{0} \mid \ulcorner w \urcorner \to \mathbf{0}$ exhibits a barb over w, the exhibition of action $\ulcorner n \urcorner$ is *not* necessary for the production of such a barb as $\mathcal{C}_{\ulcorner n \urcorner}(Q) \longmapsto \mathbf{0} \mid \ulcorner w \urcorner \to \mathbf{0}$ also exhibits a barb on w but in doing so Q performs n instead of $\ulcorner n \urcorner$.

Example 1 shows that CPC pattern-unification allows binding names to be contractive; it is not necessary to fully decompose a pattern to bind it. Thus a compound pattern may be bound to a single name or to more than one name in unification.

Example 2 illustrates that CPC pattern-unification on protected names only requires the other pattern know the name, thus a protected or variable name is sufficient.

These two observations make it clear that some patterns are more discerning than others, i.e. match fewer patterns than others. This leads to the following definitions.

Definition 8.3.1. *Let p and q be two patterns. That p is* harmonious *with*

q, written $p \lll q$, is defined as follows.

$$\begin{array}{rcll} n & \lll & n & \\ \ulcorner n \urcorner & \lll & \ulcorner n \urcorner & \\ \ulcorner n \urcorner & \lll & n & \\ p_1 \bullet p_2 & \lll & q_1 \bullet q_2 & p_i \lll q_i \text{ for } i \in \{1,2\}. \end{array}$$

Definition 8.3.2. *Let p and q be patterns each with a* linked *substitution, σ and ρ respectively, such that the binding names of the pattern equal the domain of the linked substitution, i.e. $\mathsf{bn}(p) = \mathsf{dom}(\sigma)$ and $\mathsf{bn}(q) = \mathsf{dom}(\rho)$. Define that p is* compatible *with q by σ and ρ, denoted $p, \sigma \ll q, \rho$ by induction as follows.*

$$\begin{array}{rcll} p, \sigma & \ll & \lambda y, \{\hat{\sigma} p / y\} & \mathsf{fn}(p) = \{\} \\ n, \{\} & \ll & n, \{\} & \\ \ulcorner n \urcorner, \{\} & \ll & \ulcorner n \urcorner, \{\} & \\ \ulcorner n \urcorner, \{\} & \ll & n, \{\} & \\ p_1 \bullet p_2, \sigma_1 \cup \sigma_2 & \ll & q_1 \bullet q_2, \rho_1 \cup \rho_2 & p_i, \sigma_i \ll q_i, \rho_i, \text{ for } i \in \{1,2\}\ . \end{array}$$

Observe that as all patterns are well formed, the union of substitutions is disjoint.

Lemma 8.3.3. *If two patterns p and q are compatible by the substitutions σ and ρ, then p and q can be made harmonious by applying the respective substitutions to the binding names of the patterns. That is: if $p, \sigma \ll q, \rho$ then $\hat{\sigma} p \lll \hat{\rho} q$.*

Proof: Trivial by induction on the structure of p. □

Note that the converse does not hold: consider the patterns λx and n and the substitutions $\{n/x\}$ and $\{\}$. Then $\widehat{\{n/x\}}\lambda x = n$ and $\{\}n = n$ and so $n \lll n$, however $\lambda x, \{n/x\} \not\ll n, \{\}$.

Definition 8.3.4. *Two patterns p and q are* comparable*, denoted $p \ll q$, if these exists substitutions σ and ρ such that $p, \sigma \ll q, \rho$.*

The idea behind these definitions is that a pattern p is comparable with another pattern q if and only if every other pattern r that unifies with p by some substitutions (θ_1, σ) also unifies with q with some substitutions (θ_2, ρ) such that $\theta_1 = \theta_2$ and $p, \sigma \ll q, \rho$. That is, the sets of patterns r that match against p are a subset of the patterns that match against q. This will be proved later in Proposition 8.4.8.

A naive approach to understanding these definitions would be to avoid the details of compatibility and define comparability via harmony. The problem with this approach is that the choice of substitution in comparability could unify a free name with a binding name, which would violate the barbed congruence relation. For example, the patterns n and λx could be made harmonious by the substitutions $\{\}$ and $\{n/x\}$, but they are not compatible.

The comparability relation on patterns provides the concept behind the proper replies of the challenge-reply game. However, as bisimulation also requires delicate control of substitutions, compatibility is used in the definition of bisimulation below.

Definition 8.3.5 (Bisimulation)**.** *A symmetric binary relation $\Re$ on CPC*

processes is a bisimulation if, for every $(P, Q) \in \Re$ *and* $P \xrightarrow{\mu} P'$*, it holds that:*

- *if* $\mu = \tau$*, then* $Q \xrightarrow{\tau} Q'$*, for some* Q' *such that* $(P', Q') \in \Re$*;*
- *if* $\mu = (\nu \widetilde{n})p$ *then, for all* σ *with* $\mathsf{dom}(\sigma) = \mathsf{bn}(p)$ *and* $\mathsf{fn}(\sigma) \cap \widetilde{n} = \{\}$ *and* $(\mathsf{bn}(p) \cup \widetilde{n}) \cap \mathsf{fn}(Q) = \{\}$*, there exist* q *and* Q' *and* ρ *such that* $Q \xrightarrow{(\nu \widetilde{n})q} Q'$ *and* $p, \sigma \ll q, \rho$ *and* $(\sigma P', \rho Q') \in \Re$*.*

Denote with $\sim$ *the largest bisimulation closed under any substitution.*

The definition is inspired by the early bisimulation congruence for the π-calculus [MPW92]: for every possible instantiation σ of the binding names, there exists a proper reply from Q. Of course, σ cannot be chosen arbitrarily: it cannot use in its range names that were restricted in P. Also the action μ cannot be arbitrary; its restricted and binding names cannot occur free in Q. Differently from the π-calculus, however, the reply from Q can be different from the challenge from P: this is due to the fact that contexts in CPC are not powerful enough to enforce an identical reply (as evidence by Examples 1 and 2). Indeed, this notion of bisimulation allows a challenge p to be replied to by some different q, provided that σ is properly adapted (yielding ρ and making p compatible with q) before being applied to Q'.

At first this may appear slightly odd, as the bisimulation relation is symmetric, yet the relation between the challenges and replies, i.e. compatibility, is not. This is due to the inability to force exact replies in CPC as illustrated by Examples 1 & 2. A simple demonstration of this is to consider the two

processes

$$\begin{aligned} P &= \ulcorner n \urcorner \mid !n \\ Q &= !n \ . \end{aligned}$$

Similar to Example 2, if P issues the challenge $\ulcorner n \urcorner$ then Q can properly reply with n; as $\ulcorner n \urcorner, \{\} \ll n, \{\}$. If Q issues the challenge n then P can also reply with n. Thus P and Q can be shown bisimilar, even though they may not always take the same action. Such similarities will be detailed in Section 8.7 once the coincidence of the two semantics has been proved.

Now that the bisimulation has been formalised, the challenges and replies can also be formalised for later convenience. Observe that although the challenges are based on the label of the transition as expected, the choice of substitution in the bisimulation relation is also significant as reflected in the following definitions. A *challenge by* P is a label, substitution and process (μ, σ, P') where the binding names of μ equals the domain of σ. A *reply by* Q is a label, substitution and process (μ', ρ, Q') where the binding names of μ' equals the domain of ρ. Given bisimilar processes P and Q then given a challenge by P (μ, σ, P'), a *proper reply by* Q of the form (μ', ρ, Q') implies that $P \xrightarrow{\mu} \sigma P'$ and $Q \xrightarrow{\mu'} \rho Q'$ such that $\sigma P'$ is bisimilar to $\rho Q'$. Observe that when μ is of the form $(\nu \widetilde{n})p$ then the μ' of any proper reply must be of the form $(\nu \widetilde{n})q$ where $p, \sigma \ll q, \rho$. To simplify later results, a *challenge* is a label and a substitution (μ, σ) such that the binding names of μ equals the

domain of σ and a *proper reply* is a label and substitution $(\mu'\rho)$ such that the following holds. If μ is τ then μ' is also τ. If μ is of the form $(\nu\widetilde{n})p$ then μ' is of the form $(\nu\widetilde{n})q$ such that $p, \sigma \ll q, \rho$.

8.4 Properties of Patterns

This section considers some properties of the relations on patterns, particularly compatibility and comparability. Most are formalised for later exploitation and to illustrate the relations between patterns in general. In particular, that compatibility preserves information used for barbs, is stable under substitution, is reflexive and transitive. Also that comparability captures the intuition behind the definitions in the first place. Although developed to formalise the coincidence of the two semantics, some properties are interesting observations in their own right and may also be of use in sequential pattern calculi.

The first property is that compatible and comparable patterns have the same free names.

Proposition 8.4.1. *If the patterns p and q are compatible by substitutions σ and ρ, then p and q share the same free names. That is: if $p, \sigma \ll q, \rho$ then $\mathsf{fn}(p) = \mathsf{fn}(q)$.*

Proof: Trivial by definition of compatibility. □

Corollary 8.4.2. *If the patterns p and q are comparable then they share the same free names.*

Proof: Trivial by Proposition 8.4.1. □

The next three results ensure that compatibility is stable under substitution.

Proposition 8.4.3. *If the patterns p and q are compatible by substitutions σ and ρ, then they remain compatible when any substitution θ is applied to (the free names of) p and q. More precisely, if $p, \sigma \ll q, \rho$ then $\theta p, \sigma \ll \theta q, \rho$.*

Proof: Straightforward by definition of compatibility and induction on the structure of p. □

Proposition 8.4.4. *If the patterns p and q are compatible by the substitutions σ and ρ, then the free names of σ are the same as those of ρ. That is: if $p, \sigma \ll q, \rho$ then $\mathsf{fn}(\sigma) = \mathsf{fn}(\rho)$.*

Proof: Straightforward by definition of compatibility. □

Definition 8.4.5. *Given two substitutions σ and θ, define $\theta[\sigma]$ to be $\{\theta p_i / x_i\}$ for $x_i \in \mathsf{dom}(\sigma)$ and $p_i = \sigma x_i$.*

Lemma 8.4.6. *If the patterns p and q are compatible by substitutions σ and ρ then for all substitutions θ it follows that p and q are compatible by $\theta[\sigma]$ and $\theta[\rho]$. That is: if $p, \sigma \ll q, \rho$ then $p, \theta[\sigma] \ll q, \theta[\rho]$.*

Proof: Straightforward by induction on the structure of p and exploiting Definition 8.4.5 and Proposition 8.4.4. □

The compatibility relationship is reflexive.

Proposition 8.4.7 (Compatibility is reflexive)**.** *For all patterns p and substitutions σ whose domain is the binding names of p it holds that $p, \sigma \ll p, \sigma$.*

Proof: Trivial by definition of compatibility. □

The next result captures the idea behind the definitions leading to comparability. A pattern p is comparable with a pattern q if and only if, for every pattern r that unifies with p by some substitutions (θ_1, σ) then r also unifies with q with some substitutions (θ_2, ρ) such that $\theta_1 = \theta_2$ and $p, \sigma \ll q, \rho$. That is, the patterns matched by p are a subset of the patterns matched by q.

Proposition 8.4.8. *The patterns p and q are comparable if and only if, for all patterns r such that $\{r \| p\}$ exists and yields (θ_1, σ) then r unifies with q by $\{r \| q\} = (\theta_2, \rho)$ and $\theta_1 = \theta_2$ and $p, \sigma \ll q, \rho$.*

Proof: In the forward direction proceed by induction on the comparability relation for p.

- If $\mathsf{fn}(p) = \{\}$ and q is of the form λy then for the unification of r and p to be defined it follows that $\{r \| p\} = (\{\}, \sigma)$ such that $\sigma = \{r_i / x_i\}$ for $x_i \in \mathsf{bn}(p)$ and each r_i is communicable and $\hat{\sigma} p = r$. It follows that $\{r \| q\} = (\{\}, \{r/y\}) = (\{\}, \{\hat{\sigma} p / y\})$ and so $\theta_1 = \{\} = \theta_2$ and $p, \sigma \ll \lambda y, \{\hat{\sigma} p / y\}$.

- If p is a variable name n then by comparability $q = n$. Now consider r.

  - If r is a binding name λz then $\{r \| p\} = (\{n/z\}, \{\}) = \{r \| q\}$ and $n, \{\} \ll n, \{\}$.

  - If r is n or $\ulcorner n \urcorner$ then $\{r \| p\} = (\{\}, \{\}) = \{r \| q\}$ and $n, \{\} \ll n, \{\}$.

- If p is a protected name $\ulcorner n \urcorner$ then by comparability q is n or $\ulcorner n \urcorner$. For r to unify with p it follows that r is n or $\ulcorner n \urcorner$. In either case $\{r \| p\} = (\{\}, \{\}) = \{r \| q\}$ and $\ulcorner n \urcorner, \{\} \ll q, \{\}$ as required.

- Otherwise, if p is of the form $p_1 \bullet p_2$ then by comparability q is of the form $q_1 \bullet q_2$. Now consider r.

  - If r is a binding name λz then for r to unify with p it follows that p is communicable and by definition of comparability that $q = p$. Hence $\{r \| p\} = (\{p/z\}, \{\}) = (\{q/z\}, \{\}) = \{r \| q\}$ and $p, \{\} \ll q, \{\}$ by Lemma 8.4.7.
  - Otherwise for r to unify with p then r must be of the form $r_1 \bullet r_2$ and proceed by two applications of induction.

In the reverse direction examine that $\{r \| p\} = (\theta_1, \sigma)$ implies $\{r \| q\} = (\theta_2, \rho)$ such that $\theta_1 = \theta_2$ and $p, \sigma \ll q, \rho$. Consider p.

- If $\mathsf{fn}(p) = \{\}$, then r must be of the form $\widehat{\{r_i/x_i\}}p$ for $x_i \in \mathsf{bn}(p)$, where each r_i can be any communicable pattern and $\theta_1 = \{\}$. As each r_i can be chosen to avoid any free names of q, it follows that, for $\{r \| q\}$ to be defined, $\mathsf{fn}(q) = \{\}$. Now consider the structure of p:

  - If p is λx then by choosing $r = n$ it must be that q is of the form λy and $\{n \| \lambda y\} = (\{\}, \{n/y\}) = (\theta_1, \{\hat{\sigma} p/y\})$ and $p, \sigma \ll q, \rho$ follows and thus $p \ll q$.
  - If p is of the form $p_1 \bullet p_2$ and q is of the form λy then $\{r \| \lambda y\} = (\{\}, \{r/y\}) = (\theta_1, \{\hat{\sigma} p/y\})$ and $p, \sigma \ll q, \rho$ follows and thus $p \ll q$.

  - Otherwise, if p is of the form $p_1 \bullet p_2$ then q must be of the form $q_1 \bullet q_2$ and proceed by induction.

- If p is n then r can be n or $\ulcorner n \urcorner$ or λz. For $\{r \| q\}$ to always be defined, it must be that $q = n$, so now consider r.

  - If r is λz then $\{r \| p\} = (\{n/z\}, \{\})$ and $\{r \| q\} = (\{n/z\}, \{\})$. It follows that $\theta_1 = \theta_2$ and $p, \sigma \ll q, \rho$ and thus $p \ll q$.

  - If r is n or $\ulcorner n \urcorner$ then $\{r \| p\} = (\{\}, \{\})$ and $\{r \| q\} = (\{\}, \{\})$. Again $\theta_1 = \theta_2$ and $p, \sigma \ll q, \rho$ and thus $p \ll q$.

- If p is $\ulcorner n \urcorner$ then r can be n or $\ulcorner n \urcorner$. For q to unify with $\ulcorner n \urcorner$ it follows that q may be n or $\ulcorner n \urcorner$. In both cases $\{r \| p\} = (\{\}, \{\})$ and $\{r \| q\} = (\{\}, \{\})$ and it follows that $\theta_1 = \theta_2$ and $p, \sigma \ll q, \rho$ and thus $p \ll q$.

- Otherwise, if p is of the form $p_1 \bullet p_2$ and $\mathsf{fn}(p) \neq \{\}$. It follows that there must be at least one free name n in p_1 or p_2 and so chooser r to have a protected name $\ulcorner n \urcorner$ that unifies with that free name. Now for $\{r \| q\}$ to be defined it follows that q is of the form $q_1 \bullet q_2$. Proceed by two applications of induction.

□

The above result can be exploited to show that comparability is transitive. Given p comparable with q, then every pattern that unifies with p also unifies with q. That is, the patterns that unify with p are a subset of the patterns that unify with q. It follows that if q is comparable with r, then p must be comparable with r.

Proposition 8.4.9 (Comparability is transitive)**.** *Given p comparable with q and q comparable with r, then p is comparable with r. That is: if $p \ll q$ and $q \ll r$ implies $p \ll r$.*

Proof: For any pattern p_1 that unifies with p, it follows by Proposition 8.4.8 that p_1 unifies with q and then again by Proposition 8.4.8 that p_1 unifies with r. □

The following lemma is a variation on Proposition 8.4.8 that uses compatibility and fixes the substitutions σ and ρ in advance. This is done to simplify later proofs and flexibility in the substitutions can be recovered by exploiting Lemma 8.4.6.

Lemma 8.4.10. *Given p is compatible with q by substitutions σ and ρ, then for all r such that r unifies with p to yield (θ, σ) it follows that r unifies with q by substitutions (θ, ρ). That is: $p, \sigma \ll q, \rho$ and $\{r \| p\} = (\theta, \sigma)$ implies $\{r \| q\} = (\theta, \rho)$.*

Proof: Straightforward by induction in the manner of Proposition 8.4.8. □

The following lemma shows that given p comparable to q and a substitution σ whose domain is the binding names of p, by exploiting pattern unification a substitution ρ can be found that makes p and q compatible by σ and ρ. Although of interest by itself, this result is exploited to show that compatibility is also transitive. That is, if p is compatible with q by σ_1 and ρ_1 and q is compatible with r by ρ_2 and θ_2 then there is a substitution θ_3 such that p is compatible with r by σ_1 and θ_3. Observe that as this shall later

be used to show that the bisimulation relation is transitive, the approach fixes the substitution on the left side of the compatibility relation to find the substitution on the right side, as in the definition of bisimulation.

Lemma 8.4.11. *Given p comparable with q and a substitution σ whose domain is the binding names of p, then $\{\hat{\sigma}p \| q\} = (\{\}, \rho)$ such that p is compatible with q by σ and ρ. That is: if $p \ll q$ and $\mathsf{dom}(\sigma) = \mathsf{bn}(p)$ then $\{\hat{\sigma}p \| q\} = (\{\}, \rho)$ and $p, \sigma \ll q, \rho$.*

Proof: The proof is by induction on the structure of q.

- If q is of the form λy then by comparability $\mathsf{fn}(p) = \{\}$. Now $\hat{\sigma}p$ is a communicable pattern and so $\{\hat{\sigma}p \| q\} = (\{\}, \{\hat{\sigma}p/y\})$ and $p, \sigma \ll \lambda y, \{\hat{\sigma}p/y\}$.

- If q is a variable name n then p is either n or $\ulcorner n \urcorner$ and so $\sigma = \{\}$. It follows that $\{p \| n\} = (\{\}, \{\})$ and thus $p, \{\} \ll n, \{\}$.

- If q is $\ulcorner n \urcorner$ then by comparability p is also $\ulcorner n \urcorner$ and so $\sigma = \{\}$. It follows that $\{\ulcorner n \urcorner \| \ulcorner n \urcorner\} = (\{\}, \{\})$ and thus $\ulcorner n \urcorner, \{\} \ll \ulcorner n \urcorner, \{\}$.

- If q is of the form $q_1 \bullet q_2$ then by comparability p is of the form $p_1 \bullet p_2$ and also there must be σ_1 and σ_2 such that $\sigma_1 \cup \sigma_2 = \sigma$ and $\mathsf{dom}(\sigma_1) = \mathsf{bn}(p_1)$ and $\mathsf{dom}(\sigma_2) = \mathsf{bn}(p_2)$. Proceed by two applications of induction.

□

Proposition 8.4.12 (Compatibility is transitive)**.** *If p is compatible with q by σ_1 and ρ_1, and q is compatible with r by ρ_2 and θ_2, then p is compatible with*

r by substitutions σ_1 and some θ_3. That is: if $p, \sigma_1 \ll q, \rho_1$ and $q, \rho_2 \ll r, \theta_2$ implies $p, \sigma_1 \ll r, \theta_3$.

Proof: When $\rho_1 = \rho_2$ then $\theta_3 = \theta_2$ and the result is immediate. Otherwise by Proposition 8.4.9 p is comparable with r and so exploit Lemma 8.4.11 with p and r and σ_1 to yield θ_3. □

As compatibility and comparability are orderings upon patterns it is interesting to observe that for every pattern p there is a unique maximal pattern with respect to $\ll$.

Proposition 8.4.13. *For every pattern p there exists a maximal pattern q with respect to $\ll$ that is unique up-to α-conversion of binding names.*

Proof: The proof is by induction on the structure of p:

- If $\mathsf{fn}(p) = \{\}$ then $q = \lambda y$ for some fresh y.
- If p is n or $\ulcorner n \urcorner$ then q is n.
- If $p = p_1 \bullet p_2$ then proceed by induction on p_1 and p_2.

The only arbitrary choice is the y used in the first item, that can be α-converted to any other fresh name. □

To conclude the properties of the compatibility and comparability relations, it is worth remarking they do not yield a lattice: there is no supremum for the two patterns λx and n.

Notes. The development of compatibility of patterns in CPC raises questions about a similar relation on patterns for pure pattern calculus (and

other pattern calculi [Jay09]). Such a relation has been suggested before as a way to indicate when extensions are overshadowed by previous cases, or in optimising evaluation of pattern-matching.

8.5 Soundness of the Bisimulation

The fourth step is to show that the bisimulation relation is a barbed congruence. That is, the bisimulation relation is barb preserving, reduction closed, and context closed.

Barb preserving and reduction closed are ensured by the following two lemmas.

Lemma 8.5.1. *The bisimulation relation $\sim$ is barb preserving.*

Proof: Straightforward by definition of bisimulation and Proposition 8.4.1. □

Lemma 8.5.2. *The bisimulation relation $\sim$ is reduction closed.*

Proof: Trivial by Proposition 8.2.2. □

Closure under any context is less easy to prove. The next lemma serves to show that bisimilarity is closed under cases. Then the following lemma shows that the bisimulation is closed under name restriction and parallel composition (as in π-calculus it is necessary to handle these simultaneously due to name extrusions).

Lemma 8.5.3. *Bisimilarity is closed under cases, that is, if $P \sim Q$ then $p \to P \sim p \to Q$.*

Proof: It is necessary to prove that $p \to P$ is bisimilar to $p \to Q$. The only possible transition of $p \to P$ is $p \to P \xrightarrow{p} P$ such that $\mathsf{bn}(p) \cap \mathsf{fn}(Q) = \{\}$. Also the only possible transition of $p \to Q$ is $p \to Q \xrightarrow{p} Q$ such that $\mathsf{bn}(p) \cap \mathsf{fn}(P) = \{\}$. Thus for either process the only possible challenge is (p, σ) for any substitution σ such that $\mathsf{dom}(\sigma) = \mathsf{bn}(p)$. The proper reply is also (p, σ) and conclude with $\sigma P \sim \sigma Q$ by bisimilarity of P and Q and closure under substitution of the bisimulation. □

Lemma 8.5.4. *Bisimulation is closed under name restriction and parallel composition. Given two bisimilar processes P and Q then they remain bisimilar when placed under the same restricted names $\widetilde{n}$ and in parallel composition with some other process R. That is: if $P \sim Q$ then $(\nu\widetilde{n})(P \mid R) \sim (\nu\widetilde{n})(Q \mid R)$.*

Proof: It is necessary to prove that the relation

$$\Re = \{((\nu\widetilde{n})(P \mid R), (\nu\widetilde{n})(Q \mid R)) \ : \ P \sim Q\}$$

is a bisimulation. If there are no transitions then the result is immediate. Otherwise these must be a transition of the form $(\nu\widetilde{n})(P \mid R) \xrightarrow{\mu} \hat{P}$ and it suffices to show that for any μ there is a transition $(\nu\widetilde{n})(Q \mid R) \xrightarrow{\mu'} \hat{Q}$ such that μ' is a proper reply and $\hat{P} \sim \hat{Q}$. Now proceed by induction on the structure of the inference for $(\nu\widetilde{n})(P \mid R) \xrightarrow{\mu} \hat{P}$.

- If the last rule is parint then $(\nu\widetilde{n})(P \mid R)$ is $P \mid R$ and $\mu = \tau$, now there are two possibilities.

– If the transition is

$$\frac{P \xrightarrow{\tau} P'}{P \mid R \xrightarrow{\tau} P' \mid R}$$

then by $P \sim Q$ there exists $Q \xrightarrow{\tau} Q'$ and conclude with $\hat{Q} = Q' \mid R$.

– If the transition is

$$\frac{R \xrightarrow{\tau} R'}{P \mid R \xrightarrow{\tau} P \mid R'}$$

then take the same transition and $\hat{Q} = Q \mid R'$ and conclude.

- If the last rule is parext then $(\nu\widetilde{n})(P \mid R)$ is $P \mid R$ and there are two possibilities.

– If the transition is

$$\frac{P \xrightarrow{(\nu\widetilde{n})p} P'}{P \mid R \xrightarrow{(\nu\widetilde{n})p} P' \mid R} \quad (\widetilde{n} \cup \mathsf{bn}(p)) \cap \mathsf{fn}(R) = \{\}$$

then define a substitution $\sigma = \mathsf{id}_{\mathsf{bn}(p)}$. Now by $P \sim Q$ there exists q and Q' and ρ such that $Q \xrightarrow{(\nu\widetilde{n})q} Q'$ and $(\widetilde{n} \cup \mathsf{bn}(q)) \cap \mathsf{fn}(R) = \{\}$ (by α-conversion on binding names of q as required) and $p, \sigma \ll q, \rho$ and $\sigma P' \sim \rho Q'$. As σ only maps names to themselves it follows that $\sigma P' = P'$, and then by $P' \sim \rho Q'$ take $\hat{Q} = \rho Q' \mid R$ and conclude.

– If the transition is

$$\frac{R \xrightarrow{(\nu\widetilde{n})r} R'}{P \mid R \xrightarrow{(\nu\widetilde{n})r} P \mid R'} \quad (\widetilde{n} \cup \mathsf{bn}(r)) \cap \mathsf{fn}(P) = \{\}$$

then take the same transition

$$\frac{R \xrightarrow{(\nu\widetilde{n})r} R'}{Q \mid R \xrightarrow{(\nu\widetilde{n})r} Q \mid R'} \quad (\widetilde{n} \cup \mathsf{bn}(r)) \cap \mathsf{fn}(Q) = \{\}$$

exploiting α-conversion as required to avoid the free names of Q. Conclude by taking $\hat{Q} = Q \mid R'$.

- If the last rule is rep then there are two possibilities.
  - If the transition is
  $$\frac{P \xrightarrow{\mu} P'}{P \mid R \xrightarrow{\mu} P' \mid R}$$
  then by $P \sim Q$ there exists $Q \xrightarrow{\mu'} Q'$ such that μ' is a proper reply to μ and $P' \sim Q'$ and conclude with $\hat{Q} = Q' \mid R$.
  - If the transition is
  $$\frac{R \xrightarrow{\mu} R'}{P \mid R \xrightarrow{\mu} P \mid R'}$$
  then take the same transition and $\hat{Q} = Q \mid R'$ and conclude.

- If the last rule is resnon then

$$\frac{(P \mid R) \xrightarrow{\mu} P'}{(\nu n)(P \mid R) \xrightarrow{\mu} (\nu n)P'} \quad n \notin \mathsf{names}(\mu)\ .$$

Now consider the transition μ.

  - If μ is an internal action τ then μ' is also an internal action and $n \notin \mathsf{names}(\mu')$. Now by the induction hypothesis there is a transition $(Q \mid R) \xrightarrow{\mu'} Q'$ and so conclude with $\hat{Q} = (\nu n)Q'$.

– If μ is of the form $(\nu\widetilde{n})p$ then define a substitution $\sigma = \mathsf{id}_{\mathsf{bn}(p)}$. Now by induction on $(P \mid R) \xrightarrow{\mu} P'$ there exists q and Q' and ρ such that $(Q \mid R) \xrightarrow{(\nu\widetilde{n})q} Q'$ and $p, \sigma \ll q, \rho$ and $\sigma P' \sim \rho Q'$. Also $n \notin \widetilde{n}$ and by Proposition 8.4.1 $n \notin \mathsf{fn}(q)$ and by α-conversion $n \notin \mathsf{bn}(q)$, it follows that $n \notin \mathsf{names}(\mu')$. As σ maps names to themselves it follows that $\sigma P' = P'$ and thus conclude with $\hat{Q} = (\nu n)(\rho Q')$.

- If the last rule is resin then

$$\frac{(P \mid R) \xrightarrow{(\nu\widetilde{n})p} P'}{(\nu m)(P \mid R) \xrightarrow{(\nu\widetilde{n},m)p} P'} \quad m \in \mathsf{vn}(p) \backslash (\widetilde{n} \cup \mathsf{pn}(p) \cup \mathsf{bn}(p)) \ .$$

Define a substitution $\sigma = \mathsf{id}_{\mathsf{bn}(p)}$. Now by induction hypothesis there is a transition $(Q \mid R) \xrightarrow{(\nu\widetilde{n})q} Q'$ for some q and Q' and ρ such that $p, \sigma \ll q, \rho$ and $\sigma P' \sim \rho Q'$. It remains to show that the side conditions hold in the following transition

$$\frac{(Q \mid R) \xrightarrow{(\nu\widetilde{n})q} Q'}{(\nu m)(Q \mid R) \xrightarrow{(\nu\widetilde{n},m)q} Q'} \quad m \in \mathsf{vn}(q) \backslash (\widetilde{n} \cup \mathsf{pn}(q) \cup \mathsf{bn}(q)) \ .$$

That $m \notin \widetilde{n}$ follows from the original transition. By definition of compatibility of p and q it follows that $m \in \mathsf{vn}(q)$ and $m \notin \mathsf{pn}(q)$. That $m \notin \mathsf{bn}(q)$ is handled by exploiting α-conversion. Thus conclude by observing that as σ maps names to themselves $\sigma P' = P'$ and so $P' \sim \rho Q'$ and finally take $\hat{Q} = \rho Q'$.

- If the last rule is case then there are two possibilities; either P or R is the null process.

- If P is the null process then by bisimilarity Q must also have no transitions. That $R \sim R$ is immediate and the conclusion follows.
- If R is the null process then there is a transition $(p \to P') \xrightarrow{p} P'$. Define a substitution $\sigma = \mathsf{id}_{\mathsf{bn}(p)}$. By $\sim$ there exists q and Q' and ρ such that $Q \xrightarrow{q} Q'$ and $p, \sigma \ll q, \rho$ and $\sigma P' \sim \rho Q'$. As $\sigma P' = P'$ take $\hat{Q} = \rho Q'$ and conclude.

- If the last rule is match then

$$\frac{P \xrightarrow{(\nu \widetilde{m})p} P' \quad R \xrightarrow{(\nu \widetilde{o})r} R'}{P \mid R \xrightarrow{\tau} (\nu \widetilde{m}, \widetilde{o})(\sigma P' \mid \theta R')}$$

with $\{p \| r\} = (\sigma, \theta)$ and $\widetilde{m} \cap \mathsf{fn}(R) = \widetilde{o} \cap \mathsf{fn}(P) = \{\}$ and $\mathsf{dom}(\sigma) = \mathsf{bn}(p)$ and $\mathsf{dom}(\theta) = \mathsf{bn}(r)$ and $\widetilde{m} \cap \widetilde{n} = \{\}$. Then $\hat{P} = (\nu \widetilde{m}, \widetilde{o})(\sigma P' \mid \theta R')$. Now by $\sim$ there exists q and Q' and ρ such that $Q \xrightarrow{(\nu \widetilde{m})q} Q'$ and $p, \sigma \ll q, \rho$ and $\sigma P' \sim \rho Q'$. By Lemma 8.4.10 $\{q \| r\} = (\rho, \theta)$ and so

$$\frac{Q \xrightarrow{(\nu \widetilde{m})q} Q' \quad R \xrightarrow{(\nu \widetilde{o})r} R'}{Q \mid R \xrightarrow{\tau} (\nu \widetilde{m}, \widetilde{o})(\rho Q' \mid \theta R')}$$

with $\widetilde{o} \cap \mathsf{fn}(Q)$ by α-conversion and the other side conditions already hold. Thus $\hat{Q} = (\nu \widetilde{m}, \widetilde{o})(\rho Q' \mid \theta R')$.

□

Continuing with context closure of the bisimulation, the next two lemmas and proposition ensures that bisimilar processes remain so under replication.

Lemma 8.5.5 (Bisimulation is transitive)**.** *If P is bisimilar to Q and Q is bisimilar to R, then P is bisimilar to R. That is: if $P \sim Q$ and $Q \sim R$ then $P \sim R$.*

Proof: Straightforward by Proposition 8.4.12. □

Definition 8.5.6. *An LTS is* structurally image finite *if, for every P and μ, it holds that $\{P' : P \xrightarrow{\mu} P'\}/_{\equiv}$ contains finitely many elements.*

Proposition 8.5.7. *The LTS defined for CPC in Figure 8.1 is structurally image finite.*

Proof: It suffices to show that for a given P and μ there are finitely many structurally equivalent P' when $P \xrightarrow{\mu} P'$ is defined. The proof by induction on the inference for $P \xrightarrow{\mu} P'$. The base case is when the last rule is case and there is just one equivalence class wiz. $[P']_{\equiv}$. All the inductive steps are straightforward except for when the last rule is rep. When the last rule is rep then the rule must be of the form

$$\frac{!P_1 \mid P_1 \xrightarrow{\mu} P''}{!P_1 \xrightarrow{\mu} P''}\,.$$

Now consider the inference for the transition $!P_1 \mid P_1 \xrightarrow{\mu} P''$.

- If the last rule is parint or parext then P'' must be of the form $Q_1 \mid Q_2$ and there are two possibilities.
  - If the rule is of the form

$$\frac{!P_1 \xrightarrow{\mu} Q_1}{!P_1 \mid P_1 \xrightarrow{\mu} Q_1 \mid Q_2}$$

then Q_2 must be P_1. Now by induction $!P_1 \xrightarrow{\mu} Q_1$ there are finitely many Q_1 up to structural equivalence. Further, it is straightforward to show that all such Q_1 are structurally equivalent to $!P_1 \mid Q'$ for some Q'. Thus conclude with $Q_1 \mid Q_2 = !P_1 \mid Q' \mid P_1 \equiv !P_1 \mid Q' = Q_1$.

– If the rule is of the form

$$\frac{P_1 \xrightarrow{\mu} Q_2}{!P_1 \mid P_1 \xrightarrow{\mu} Q_1 \mid Q_2}$$

then the conclusion is straightforward.

- If the last rule is match then the conclusion is straightforward.

□

Lemma 8.5.8. *If two processes P and Q are bisimilar, then their respective replications are also bisimilar. That is: if $P \sim Q$ then $!P \sim !Q$.*

Proof: This proof rephrases the similar one by Sangiorgi & Walker [SW01]; here the main steps shall be revisited. First, define the n-th approximation

of the bisimulation:

$$
\begin{aligned}
\sim_0 &= Proc \times Proc \\
\sim_{n+1} &= \{(P,Q) \ : \\
&\qquad \text{For all transitions } P \xrightarrow{\mu} P' \text{ the following hold. If } \mu = \tau \text{ there} \\
&\qquad \text{exists a transition } Q \xrightarrow{\tau} Q' \text{ such that } (P',Q') \in \sim_n. \\
&\qquad \text{Otherwise if } \mu = (\nu \widetilde{n})p \text{ then for all substitutions } \sigma \text{ such that} \\
&\qquad \mathsf{dom}(\sigma) = \mathsf{bn}(p) \text{ and } \mathsf{fn}(\sigma) \cap \widetilde{n} = \{\} \text{ and } (\mathsf{bn}(p) \cup \widetilde{n}) \cap \mathsf{fn}(Q) \\
&\qquad = \{\} \text{ then there exists } q \text{ and } Q' \text{ and } \rho \text{ such that } Q \xrightarrow{(\nu \widetilde{n})q} Q' \\
&\qquad \text{and } p, \sigma \ll q, \rho \text{ and } (\sigma P', \rho Q') \in \sim_n. \\
&\qquad \text{Also the same for transitions of } Q.
\end{aligned}
$$

Trivially, $\sim_0 \supseteq \sim_1 \supseteq \sim_2 \supseteq \ \ldots$

Since the LTS is structurally image finite by Proposition 8.5.7 it follows that

$$\sim \ = \ \bigcap_{n \geq 0} \sim_n \ . \tag{8.4}$$

One inclusion is trivial: by induction on n, it is straightforward to prove that $\sim \subseteq \sim_n$ for every n and so $\sim \subseteq \ \bigcap_{n \geq 0} \sim_n$. For the converse, fix $P \xrightarrow{\mu} P'$. Consider the case for $\mu = (\nu \widetilde{n})p$, since the case for $\mu = \tau$ can be proved as in π-calculus. For every n, there exist q_n, Q_n and ρ_n such that $Q \xrightarrow{(\nu \widetilde{m})q_n} Q_n$ and $p, \sigma \ll q_n, \rho_n$ and $\sigma P' \sim_n \rho_n Q_n$. By Proposition 8.4.13, there are finitely many (up-to α-equivalence) such q_n's and thus there must exist (at least) one q_k that leads to infinitely many Q_n's. However, by Proposition 8.5.7 such Q_n's cannot be all different up to structural equivalence. Thus, there must exist (at least) one Q_h such that $Q \xrightarrow{(\nu \widetilde{m})q_k} Q_h$ and there are infinitely

many Q_n's such that $Q \xrightarrow{(\nu\widetilde{m})q_k} Q_n$ and $Q_n \equiv Q_h$. It suffices to prove that $\sigma P' \sim_n \rho_h Q_h$, for every n. This is trivial whenever $n \leq h$ and it follows that $\sim_n \supseteq \sim_h$. So consider when $n > h$. If $Q_n \equiv Q_h$, conclude since $\equiv$ is closed under substitutions (as $\rho_n = \rho_h$ since $q_n = q_h = q_k$) and $\equiv \subseteq \sim_n$, for every n. Otherwise, there must exist $m > n$ such that $Q_m \equiv Q_h$ (otherwise there would not be infinitely many Q_n's structurally equivalent to Q_h). Thus, $\sigma P' \sim_m \rho_h Q_h$ implies $\sigma P' \sim_n \rho_h Q_h$, since $m > n$.

Following on from this, $!P \sim !Q$ if and only if $!P \sim_n !Q$, for all n. Let P^n denote the parallel composition of n copies of the process P (and similarly for Q). Now, it can be proved that

$$!P \sim_n P^{2n} \quad \text{and} \quad !Q \sim_n Q^{2n} \,. \tag{8.5}$$

By repeatedly exploiting Lemma 8.5.4, for all n it follows that $P^{2n} \sim Q^{2n}$ and so by (8.4)

$$P^{2n} \sim_n Q^{2n} \,. \tag{8.6}$$

Now by (8.6) it follows that $P \sim Q$ implies for all n that $P^{2n} \sim_n Q^{2n}$. Then by (8.5) and Lemma 8.5.5 (that also holds with $\sim_n$ in place of $\sim$) for all n it follows that $!P \sim_n !Q$. Finally, conclude by (8.4) to show that $!P \sim !Q$. □

Context closure can now be proved by exploiting the above lemmas for restriction, parallel composition and replication.

Lemma 8.5.9. *The bisimulation relation $\sim$ is contextual.*

Proof: Given two bisimilar processes P and Q, it is necessary to show that for any context $\mathcal{C}(\cdot)$ it holds that $\mathcal{C}(P) \sim \mathcal{C}(Q)$. The proof is by induction

on the structure of the context.

- If $\mathcal{C}(\cdot) \stackrel{\text{def}}{=} \cdot$ then the result is immediate.
- If $\mathcal{C}(\cdot) \stackrel{\text{def}}{=} \mathcal{C}'(\cdot) \mid R$ or $\mathcal{C}(\cdot) \stackrel{\text{def}}{=} (\nu n)\mathcal{C}'(\cdot)$, then $\mathcal{C}'(P) \sim \mathcal{C}'(Q)$ by induction, and conclude by Lemma 8.5.4.
- If $\mathcal{C}(\cdot) \stackrel{\text{def}}{=} !\mathcal{C}'(\cdot)$, then $\mathcal{C}'(P) \sim \mathcal{C}'(Q)$ by induction, and conclude by Lemma 8.5.8.
- If $\mathcal{C}(\cdot) \stackrel{\text{def}}{=} p \to \mathcal{C}'(\cdot)$, then $\mathcal{C}'(P) \sim \mathcal{C}'(Q)$ by induction, and conclude by Lemma 8.5.3.

□

Thus the soundness of the bisimulation follows.

Theorem 8.5.10 (Soundness of the bisimulation)**.** *Bisimilar processes are barbed congruent. That is:* $\sim \subseteq \simeq$.

Proof: By Definition 8.1.5 is suffices to show that the bisimulation relation is: barb preserving by Lemma 8.5.1, reduction closed by Lemma 8.5.2, and context closed by Lemma 8.5.9. □

8.6 Completeness of the Bisimulation

The fifth step is by showing that the barbed congruence relation is a bisimulation. This section develops the machinery required to construct reply contexts for any challenge and then exploits these to prove the barbed congruence is a bisimulation. The more complex case arises when a process P

has a challenge is of the form $((\nu\widetilde{n})p, \sigma)$. This is addressed by defining a reply context that succeeds if and only if supplied with a process Q that can properly reply by $((\nu\widetilde{n})q, \rho')$ to the challenge $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$. The reply context is of the form $(\nu\widetilde{o})(p' \to \mathsf{tests}) \mid \cdot$ where unification of p' with q is necessary for comparability. However, further testing is required to ensure that $p \ll q$ is satisfied. Constructing the reply context begins with a *specification* developed from p and N (to be thought of as the free names of P and Q) that defines the *complementary pattern* p' and ingredients for building the tests. In particular p' is defined so that unification with q yields (θ, ρ') where θ is required for further testing and ρ' is used to make p and q compatible. Tests are then defined that succeed if and only if the variable names of q can be made equal to those of p. That is: when a variable name n is free in $(\nu\widetilde{n})p$ then n is free in $(\nu\widetilde{n})q$; and when a variable name n is restricted in $(\nu\widetilde{n})p$ then (perhaps exploiting renaming) n is restricted in $(\nu\widetilde{n})q$. The complementary pattern and tests are then combined into a *characteristic process* that is in turn used to define a reply context $\mathcal{C}_p^N(\cdot)$ for the challenge $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$. From here it is straightforward to generalise to challenges $((\nu\widetilde{n})p, \sigma)$ with any substitution σ and so show the barbed congruence is a bisimulation.

Specification

Given a challenge $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$ the specification of the pattern p provides the ingredients required to construct the reply context. In particular, the specification yields a complementary pattern p' that will unify with any q

such that $p \ll q$ while also binding the components of q that require further testing to ensure that $p \ll q$. One complexity is in distinguishing the free variable names from the restricted variable names in p and q. To resolve this, a set of names N disjoint from $\widetilde{n}$ is used that will later be defined to be the free names of the processes being considered.

Definition 8.6.1. *The* specification $\mathsf{spec}^N(p)$ *of a pattern* p *with respect to a finite set of names* N *is defined follows:*

$$
\begin{array}{rcll}
\mathsf{spec}^N(\lambda x) & = & x, \{\}, \{\} & \\
\mathsf{spec}^N(n) & = & \lambda x, \{(x,n)\}, \{\} & n \in N \text{ and } x \text{ is fresh} \\
\mathsf{spec}^N(n) & = & \lambda x, \{\}, \{(x,n)\} & n \notin N \text{ and } x \text{ is fresh} \\
\mathsf{spec}^N(\ulcorner n \urcorner) & = & \ulcorner n \urcorner, \{\}, \{\} & \\
\mathsf{spec}^N(p \bullet q) & = & p' \bullet q', F_p \uplus F_q, R_p \uplus R_q & \begin{cases} \mathsf{spec}^N(p) = p', F_p, R_p \\ \mathsf{spec}^N(q) = q', F_q, R_q \end{cases}
\end{array}
$$

where $F_p \uplus F_q$ *means the disjoint union of the pairs of names.*

Given a pattern p, the specification $\mathsf{spec}^N(p) = p', F, R$ of p with respect to a set of names N has three components: the *complementary pattern* p'. a collection F of pairs (x, n) of: a binding name in p', and the expected (free) name it will be bound to n; and a collection R of pairs (x, n) of: a binding name in p' and the expected (restricted) name it will be bound to n. Observe that p' is well formed as all binding names are fresh.

The specification is straightforward for binding names, protected names and compounds. When p is a variable name then p' is a fresh binding names

λx and the intended binding of x to n is recorded in F or R according to whether n is free or restricted, respectively.

Proposition 8.6.2. *Given a pattern p and a finite set of names N, define $\mathsf{spec}^N(p) = p', F, R$. The unification of p and p' is defined and yields the substitutions σ and θ such that σ is the identity substitution on the binding names of p and for every pair (x, n) in $F \cup R$ then θ maps x to n.*

Proof: By straightforward induction on the structure of p. □

For later convenience define the *first projection* $\mathsf{fst}(F)$ and *second projection* $\mathsf{snd}(F)$ of a set of pairs, that is, $\mathsf{fst}(\{(x, m), (y, n)\}) = \{x, y\}$ and $\mathsf{snd}(\{(x, m), (y, n)\}) = \{m, n\}$, respectively.

Testing

Tests are used that succeed when the binding names of p' have bound to appropriate patterns in unification with q. When a binding name of p' is intended to bind to a free variable name n of the challenge, i.e. $(x, n) \in F$, then the test should succeed if and only if x can unify with n. Alternatively, when a binding name of p' is intended to bind to a restricted name n of the challenge, i.e. $(x, n) \in R$, then a test should succeed if and only if x is not bound to a compound or any name in N. Although this ensures the binding of x when $(x, n) \in R$ is a restricted name, some renaming may be required to achieve compatibility. To ensure that renaming is possible a further test is required to indicate that renaming is safe. To illustrate the difficulties of

such renaming, consider the following three transitions and specification:

$$\begin{aligned}
\mu_1 &= (\nu m_1)(\nu m_2) m_1 \bullet m_1 \bullet m_2 \\
\mu_2 &= (\nu n_1)(\nu n_2) n_1 \bullet n_1 \bullet n_2 \\
\mu_3 &= (\nu o_1)(\nu o_2) o_1 \bullet o_2 \bullet o_2 \\
\mathsf{spec}^{\{\}}(m_1 \bullet m_1 \bullet m_2) &= \lambda x \bullet \lambda y \bullet \lambda z, \{\}, \{(x, m_1), (y, m_1), (z, m_2)\} \; .
\end{aligned}$$

Observe that the patterns of all three transitions unifies with $\lambda x \bullet \lambda y \bullet \lambda z$ and that all of the names bound to x, y and z will be restricted. Now consider a challenge $(\mu_1, \{\})$ and the three transitions above. Obviously $(\mu_1, \{\})$ is a proper reply. Subject to renaming then μ_2 can also be a proper reply by taking $n_1 = m_1$ and $n_2 = m_2$. No renaming allows μ_3 to be a proper reply. The ability to rename μ_2 is due to the structure of $n_1 \bullet n_1 \bullet n_2$, or when unified with $\lambda x \bullet \lambda y \bullet \lambda z$ then $x = y$ and $x \neq z$ and $y \neq z$. Observe that the relations that make renaming safe can be recovered from the information in R. Thus, a further test is defined that succeeds when these relations hold and renaming is safe.

To simplify the test definitions, define the *production* $\prod_{x \in S} \mathcal{P}(x)$ of a set S to be the parallel composition of processes $\mathcal{P}(x)$ for each x in S. That is, if $\mathcal{P}(x) = x \rightarrow \mathbf{0}$ and $S = \{x_1, \ldots, x_i\}$ then

$$\prod_{x \in S} \mathcal{P}(x) \equiv x_1 \rightarrow \mathbf{0} \mid \ldots \mid x_i \rightarrow \mathbf{0} \; .$$

The tests are also simplified by defining a *check* $\mathsf{check}(x, m, y, n, w)$ to ensure equality or inequality of names. That is, when $m = n$ then $x = y$ and when $m \neq n$ then $x \neq y$.

$$\begin{array}{lcll} \mathsf{check}(x, m, y, n, w) & = & (\nu z)(\ulcorner z \urcorner \bullet \ulcorner x \urcorner \mid \ulcorner z \urcorner \bullet \ulcorner y \urcorner \rightarrow \ulcorner w \urcorner) & m = n \\ \mathsf{check}(x, m, y, n, w) & = & \ulcorner w \urcorner \mid (\nu z)(\ulcorner z \urcorner \bullet \ulcorner x \urcorner \mid \ulcorner z \urcorner \bullet \ulcorner y \urcorner \rightarrow \ulcorner f \urcorner \bullet \lambda z) & m \neq n \end{array}$$

Observe that a check succeeds if and only if m and n share the same equality or inequality as x and y.

Definition 8.6.3 (Tests)**.** *Tests for ensuring compatibility.*

$$\mathsf{free}(x, n, w) = (\nu m)(\ulcorner m \urcorner \bullet \ulcorner n \urcorner \rightarrow \ulcorner w \urcorner \mid \ulcorner m \urcorner \bullet \ulcorner x \urcorner)$$

$$\begin{array}{lcl} \mathsf{rest}^N(x, w) & = & \ulcorner w \urcorner \mid (\nu m)(\nu z)(\\ & & \ulcorner m \urcorner \bullet x \bullet z \\ & & \mid \ulcorner m \urcorner \bullet (\lambda y_1 \bullet \lambda y_2) \bullet \lambda z \rightarrow \ulcorner f \urcorner \bullet \lambda z \quad \textit{fail on compound} \\ & & \mid \prod_{n \in N} \ulcorner m \urcorner \bullet \ulcorner n \urcorner \bullet \lambda z \rightarrow \ulcorner f \urcorner \bullet \lambda z \quad \textit{fail on } n \in N \\ & &) \end{array}$$

$$\begin{array}{lcl} \mathsf{equality}^R(x, m, w) & = & (\nu \widetilde{w_y})(\\ & & \ulcorner w_{y1} \urcorner \rightarrow \ldots \rightarrow \ulcorner w_{yi} \urcorner \rightarrow \ulcorner w \urcorner \\ & & \mid \prod_{(y,n) \in R} \mathsf{check}(x, m, y, n, w_y) \\ & &) \\ & & \textit{where } \widetilde{y} = \mathsf{fst}(R). \end{array}$$

A *free test* $\mathsf{free}(x, n, w)$ for x is a process that succeeds by reporting success

with $\ulcorner w \urcorner$ if and only if x is equal to n. A *restricted test* $\mathsf{rest}^N(x, w)$ for x is a process that succeeds by: immediately reporting success with $\ulcorner w \urcorner$; and not reporting failure with $\ulcorner f \urcorner \bullet \lambda z$ when x is a compound or any name in N. An *equality test* $\mathsf{equality}^R(x, m, w)$ for x is a process that succeeds by ensuring every check on pairs (y, n) in R reports success and does not report failure.

Lemma 8.6.4. *A free test* $\mathsf{free}(x, n, w)$ *for* x *succeeds if and only if* x *equals* n*.*

Proof: Trivial. □

Lemma 8.6.5. *A restricted test* $\mathsf{rest}^N(x, w)$ *for* x *succeeds if and only if* x *is not a compound and* x *is not any name in* N*.*

Proof: Straightforward. □

Lemma 8.6.6. *An* equality test $\mathsf{equality}^R(x, m, w)$ *for* x *succeeds if and only if, for every pair* (y, n) *in* R *then:*

- *if* $m = n$ *then* $x = y$
- *if* $m \neq n$ *then* $x \neq y$*.*

Proof: In order for $\mathsf{equality}^R(x, m, w)$ to succeed by exhibiting a barb $\ulcorner w \urcorner$, each check $\mathsf{check}(x, m, y, n, w_y)$ must succeed by producing $\ulcorner w_y \urcorner$. The rest of the proof is straightforward. □

For each test that succeeds there is an exact number of reductions required to reduce to a form barbed congruent to $\ulcorner w \urcorner$.

Lemma 8.6.7. *For each test T that succeeds, there is exactly k reductions to a form barbed congruent to $\ulcorner w \urcorner$ where k depends only on the structure of the test.*

Proof: Trivial for free and restricted tests. For equality tests it suffices to observe that each check has an exact number of reductions to succeed and then there is a reduction to consume the success barb of each check. □

Reply Context

Given a pattern p and a finite set of names N (thought of as the free names of P and Q), the *characteristic process* can be defined by exploiting the specification and tests. In turn the characteristic process can then by used to construct a reply context that succeeds if and only if supplied supplied with a process Q that can properly reply to the challenge $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$ where $\widetilde{n}$ is disjoint from N.

Definition 8.6.8. *The* characteristic process $\mathsf{char}^N(p)$ *of a pattern p with respect to a finite set of names N is* $\mathsf{char}^N(p) = p' \rightarrow \mathsf{tests}^N(p)$ *where*

$$
\begin{aligned}
\mathsf{tests}^N(p) \stackrel{\text{def}}{=}\ & (\nu\widetilde{w_x})(\nu\widetilde{w_y})(\\
& \quad \ulcorner w_{x1} \urcorner \rightarrow \ldots \rightarrow \ulcorner w_{xi} \urcorner \rightarrow \ulcorner w_{y1} \urcorner \rightarrow \ldots \rightarrow \ulcorner w_{yj} \urcorner \rightarrow \ulcorner w \urcorner \\
& \quad \mid \textstyle\prod_{(x,n)\in R} \mathsf{equality}^R(x, n, w_x) \\
& \quad \mid \textstyle\prod_{(x,n)\in F} \mathsf{free}(x, n, w_y) \\
& \quad \mid \textstyle\prod_{(x,n)\in R} \mathsf{rest}^N(x, w_y) \\
& \quad)
\end{aligned}
$$

and $\mathsf{spec}^N(p) = p', F, R$ *and* $\{x1, \ldots, xi\} = \mathsf{fst}(R)$ *and* $\{y1, \ldots, yj\} = \mathsf{fst}(F) \cup \mathsf{fst}(R)$. *For later convenience the denote* $\mathsf{tests}^N(p)$ *as the* tests of p *and* N.

Proposition 8.6.9. *The characteristic process of a pattern* p *with respect to a set of names* N *does not reduce.*

Proof: It is sufficient to observe that $\mathsf{char}^N(p)$ is a case for every p and N. □

Lemma 8.6.10. *Given a characteristic process* $\mathsf{char}^N(p)$ *and any substitution* θ *such that* $\theta(\mathsf{tests}^N(p))$ *succeeds, then there are exactly* k *reduction steps* $\theta(\mathsf{tests}^N(p)) \longmapsto^k \simeq \ulcorner w \urcorner$ *where* k *depends only on* p *and* N.

Proof: Straightforward by induction on the number of tests and repeated applications of Lemma 8.6.7. □

A reply context for a challenge $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$ with a finite set of names N can be defined by exploiting the characteristic process.

Definition 8.6.11. *A* reply context $\mathcal{C}_p^N(\cdot)$ *for the challenge* $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$ *with a finite set of names* N *such that* $\widetilde{n}$ *is disjoint from* N *is defined as follows:*

$$\mathcal{C}_p^N(\cdot) \stackrel{\text{def}}{=} \mathsf{char}^N(p) \mid \cdot \ .$$

Proposition 8.6.12. *Given a reply context* $\mathcal{C}_p^N(\cdot)$ *then there is a minimum number of reductions required for* $\mathcal{C}_p^N(\cdot)$ *to succeed when supplied with a process. That is, the minimum number of reductions required for* $\mathcal{C}_p^N(\cdot)$ *to succeed is the number of reduction steps* k' *for* $\mathsf{tests}^N(p)$ *to succeed plus 1.*

Proof: By Lemma 8.6.10 then for any substitution θ such that $\mathsf{tests}^N(p)$ succeeds then $\mathsf{tests}^N(p) \longmapsto^{k'} \simeq \ulcorner w \urcorner$ where k' depends only on p and N. Thus the minimum number of reductions is when the dot is replaced with a process of the form $(\nu\widetilde{m})(q \to Q_1 \mid Q_2)$ for some $\widetilde{m}$ and q and Q_1 and Q_2 such that $\{p' \| q\} = (\theta, \rho)$ and $\theta S \longmapsto^{k'} \simeq \ulcorner w \urcorner$. Conclude with $k = k' + 1$ depending only upon p and N. □

Thus, the minimum number of reductions required for a reply context $\mathcal{C}_p^N(\cdot)$ to succeed is equal to the number of reductions for $\mathsf{test}^N(p)$ to succeed plus 1. Denote this number of reductions by Proposition 8.6.12 as $\textsc{Mr}(N, p)$.

The next theorem serves to show that a reply context $\mathcal{C}_p^N(\cdot)$ succeeds in exactly $\textsc{Mr}(N, p)$ reduction steps when supplied with a process Q that has a transition that yields a proper reply to $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$.

Theorem 8.6.13. *Suppose given a challenge $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$ and a finite set of names N and a process Q and fresh names w and f such that restricted names of the challenge union w and f are disjoint from N, and the free names of the challenge union the free names of Q are contained within N. If Q has a transition of the form $Q \xrightarrow{(\nu\widetilde{n})q} Q'$ and there is a substitution ρ such that $p, \mathsf{id}_{\mathsf{bn}(p)} \ll q, \rho$ then $\mathcal{C}_p^N(Q)$ succeeds and has a reduction sequence $\mathcal{C}_p^N(Q) \longmapsto^k \simeq \rho Q' \mid \ulcorner w \urcorner$ where $k = \textsc{Mr}(N, p)$.*

Proof: To simplify the proof, α-conversion can be used to ensure that binding names of p are fresh, in particular do not appear in Q. Observe that by Lemma 8.6.10 then the number of reductions for $\mathsf{tests}^N(p)$ to succeed k' is fixed and by Proposition 8.6.12 $k' = k - 1$. By Proposition 8.6.2 $\{p \| p'\} =$

(σ, θ) where $\sigma = \mathsf{id}_{\mathsf{bn}(p)}$ and θ is such that $\mathsf{dom}(\theta) = \mathsf{bn}(p') = \mathsf{fst}(F) \cup \mathsf{fst}(R)$ and for every $(x, n) \in F \cup R$ then θ maps x to n. By Lemma 8.4.10 $\{q \| p'\} = (\rho, \theta)$. Thus $\mathcal{C}_p^N(Q) \longmapsto \rho Q' \mid \theta(\mathsf{tests}^N(p))$. Since w and f do not appear in Q or p', it suffices to show that $\theta(\mathsf{tests}^N(p))$ succeeds and the proof is by contradiction.

- Assume that $\theta(\mathsf{tests}^N(p)) \not\Downarrow_w$, then there are two possibilities.
  - If there is a $\ulcorner w_{xi} \urcorner$ that cannot be consumed then consider the equality test $\mathsf{equality}^R(x, m, w_{xi})$ of the form $(\nu \widetilde{w_z})(\ulcorner w_{z1} \urcorner \to \ldots \to \ulcorner w_{zk} \urcorner \to \ulcorner w_{xi} \urcorner \mid \prod_{(y,n) \in R} \mathsf{check}(x, m, y, n, w_z))$. Now consider the $\ulcorner w_{zk} \urcorner$ that cannot be consumed. There are two possibilities depending on the check that does not report success.
    - If the check is of the form $\mathsf{check}(x, m, y, n, w_{zk})$ where $m = n$ then it must be that $x \neq y$, however by compatibility it must be that $x = y$.
    - If the check is of the form $\mathsf{check}(x, m, y, n, w_{zk})$ where $m \neq n$ then contradiction is immediate as these checks immediately exhibit a barb on w_{zk}.
  - If there is a $\ulcorner w_{yi} \urcorner$ that cannot be consumed then consider the test that fails to yield $\ulcorner w_{yi} \urcorner$:
    - If $yi \in \mathsf{fst}(F)$ then there is some component of S of the form $\ulcorner m \urcorner \bullet \ulcorner q_1 \urcorner \to \ulcorner w_{yi} \urcorner \mid \ulcorner m \urcorner \bullet \ulcorner n \urcorner$ and $q_1 \neq n$. However, by compatibility it must be that $q_1 = n$.
    - If $yi \in \mathsf{fst}(R)$ then contradiction is immediate, since restricted tests immediately exhibit a barb on w_{yi}.

- Assume that $\theta(\mathsf{tests}^N(p)) \Downarrow_f$ then there are two possibilities depending on the test that failed.

  - If the failure is due to an equality test then it must be that there is some $(x, m) \in R$ and some other $(y, n) \in R$ such that $m = n$ and $x \neq y$. However, this contradicts Lemma 8.6.6.
  - If the failure is due to a restricted test then consider the pattern bound by $y \in \mathsf{fst}(R)$.
    - If y is bound to a compound then this contradicts compatibility.
    - If y is bound to n and $n \in N$ this yields contradiction as for $n \in R$ it must be that $n \notin N$.

□

The difficulty in ensuring a reply context succeeds is to account for renaming of restricted names in determining a proper reply. The next two lemmas deal with renaming of restricted names by showing when patterns can be renamed to be compatible and then considering the requirements for a label of the form $(\nu\widetilde{m})q$ to be part of a proper reply to a challenge $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$. These are in turn used to show that if a reply context succeeds in exactly $\textsc{Mr}(N, p)$ reduction steps then there is a transition that meets the criteria for a proper reply.

Lemma 8.6.14. *Suppose given patterns p_1 and p_2 and q_1 and q_2 and substitutions θ_1 and θ_2 such that $p_1 \ll \theta_1 q_1$ and $p_2 \ll \theta_2 q_2$. If $\mathsf{dom}(\theta_2) \cap$*

$(\mathsf{vn}(q_1)\backslash\mathsf{dom}(\theta_1))$ *and* $\mathsf{dom}(\theta_1) \cap (\mathsf{vn}(q_2)\backslash\mathsf{dom}(\theta_2))$ *are both empty and for all* $x \in \mathsf{dom}(\theta_1)\cap\mathsf{dom}(\theta_2)$ *it follows that* $\theta_1 x = \theta_2 x$ *then* $p_1 \bullet p_2 \ll (\theta_1 \cup \theta_2)(q_1 \bullet q_2)$.

Proof: Straightforward by induction on the structure of $q_1 \bullet q_2$. □

Lemma 8.6.15. *Given a challenge* $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$ *and a finite set of names* N *and a label of the form* $(\nu\widetilde{m})q$ *and fresh names* w *and* f *such that* $\widetilde{n} \cap N = \{\}$ *and* $s, f \notin N \cup \mathsf{fn}((\nu\widetilde{m})q)$ *and* $\widetilde{m} = \mathsf{vn}(q)\backslash N$ *then* $\mathcal{C}_p^N((\nu\widetilde{m})q)$ *succeeds if and only if there exists a pattern* q' *and substitutions* θ *and* ρ *such that* θ *maps names to names and* $\mathsf{dom}(\theta) = \widetilde{m}$ *and* $\mathsf{fn}(\theta) = \widetilde{n}$ *and* $|\widetilde{n}| = |\widetilde{m}|$ *and* $q' = \theta q$ *and* $p, \mathsf{id}_{\mathsf{bn}(p)} \ll q', \rho$. *That is:* $\mathcal{C}_p^N((\nu\widetilde{m})q)$ *succeeds if and only if* $(\nu\widetilde{m})q =_\alpha (\nu\widetilde{n})q'$ *and* $p, \mathsf{id}_{\mathsf{bn}(p)} \ll q', \rho$.

Proof: To simplify the proof, α-conversion can be used to ensure that binding names of p are fresh, in particular do not appear in $(\nu\widetilde{m})q$, and that restricted names of $(\nu\widetilde{m})q$ do not appear in p.

For the reverse direction, obtain the label μ by α-converting the names $\widetilde{m}$ to $\widetilde{n}$ according to θ, i.e. $\mu = (\nu\widetilde{n})\theta q$. Then define the process $Q = \mu \rightarrow \mathbf{0}$ and by existence of ρ such that $p, \mathsf{id}_{\mathsf{bn}(p)} \ll \theta q, \rho$ conclude via Theorem 8.6.13.

In the forward direction observe that $\mathcal{C}_p^N((\nu\widetilde{m})q)$ is of the form $(\nu\widetilde{m})(p' \rightarrow \mathsf{tests}^N(p) \mid q)$ where $\mathsf{spec}^N(p) = p', F, R$. By freshness of w and f for $\mathcal{C}_p^N((\nu\widetilde{m})q)$ to succeed there must be a reduction $(\nu\widetilde{m})(p' \rightarrow \mathsf{tests}^N(p) \mid q) \longmapsto (\nu\widetilde{m})\theta(\mathsf{tests}^N(p))$ where $\{p' \| q\} = (\theta, \rho)$. Now proceed by induction on the structure of q to show that $|\widetilde{n}| = |\widetilde{m}|$ and $p, \mathsf{id}_{\mathsf{bn}(p)} \ll \theta q, \rho$.

- If q is of the form λy then it follows by unification of p' and q that p' is a communicable pattern and that $\rho = \{p'/y\}$. By definition of the

specification it must be that $\mathsf{fn}(p) = \{\}$ and thus $p' = (\widehat{\mathsf{id}_{\mathsf{bn}(p)}})p$. Thus $\theta = \{\}$ and $\widetilde{n} = \{\} = \widetilde{m}$ and conclude with $p, \mathsf{id}_{\mathsf{bn}(p)} \ll q, \rho$.

- If q is n and $n \in N$ then by freshness of binding names of p and unification of p' with q there are two possibilities for p'.
  - If p' is λz then by definition of the specification there is a pair $(z, m) \in F \cup R$. If $(z, m) \in R$ then by Lemma 8.6.5 the reply context would not succeed and this would contradict the premise. Therefore, $(z, m) \in F$ and for the reply context to succeed and by Lemma 8.6.4 it follows that $n = m$. Conclude with $\theta = \{\}$ and $\widetilde{n} = \{\} = \widetilde{m}$ and $n, \{\} \ll n, \{\}$.
  - If p' is $\ulcorner n \urcorner$ then by definition of the specification p is also $\ulcorner n \urcorner$. Conclude with $\theta = \{\}$ and $\widetilde{n} = \{\} = \widetilde{m}$ and $\ulcorner n \urcorner, \{\} \ll n, \{\}$.
- If q is m and $m \notin N$ then it follows that $\widetilde{m} = \{m\}$ and p' must be of the form λz. It follows that there is a pair $(z, n) \in F \cup R$. However, if $(z, n) \in F$ then $n \in N$ and by Lemma 8.6.4 then $m \in N$ which yields contradiction. Therefore, $(z, n) \in R$ and $\widetilde{n} = \{n\}$ and $|\widetilde{m}| = |\widetilde{n}|$ and conclude with $\theta = \{n/m\}$ and $n, \{\} \ll \{n/m\}m, \{\}$.
- If q is $\ulcorner n \urcorner$ then by unification p' is n or $\ulcorner n \urcorner$. By definition of the specification and freshness of the binding names of p it follows that p' is $\ulcorner n \urcorner$ and thus p is $\ulcorner n \urcorner$. Conclude with $\theta = \{\}$ and $\widetilde{n} = \{\} = \widetilde{m}$ and $\ulcorner n \urcorner, \{\} \ll \ulcorner n \urcorner, \{\}$.
- If q is of the form $q_1 \bullet q_2$ then consider p'. If p' is a binding name λz then it follows that $z \in \mathsf{fst}(F) \cup \mathsf{fst}(R)$. However, if $z \in \mathsf{fst}(F)$ then

this contradicts Lemma 8.6.4 and if $z \in \mathsf{fst}(R)$ then this contradicts Lemma 8.6.5. Therefore p' must be of the form $p'_1 \bullet p'_2$ and by definition of the specification p must be of the form $p_1 \bullet p_2$.

Now by two applications of the induction hypothesis there are substitution θ_1 and θ_2 and also $\widetilde{n}_1$ and $\widetilde{n}_2$ and $\widetilde{m}_1$ and $\widetilde{m}_2$ such that for $i \in \{1, 2\}$ then $p_i, \mathsf{id}_{\mathsf{bn}(p_i)} \ll \theta_i q_i, \rho_i$ and $|\widetilde{n}_i| = |\widetilde{m}_i|$. Since $\mathsf{dom}(\theta_i) = \mathsf{vn}(q_i) \cap \widetilde{m}$ it follows that $\mathsf{dom}(\theta_2) \cap (\mathsf{vn}(q_1) \backslash \mathsf{dom}(\theta_1)) = \{\}$ and $\mathsf{dom}(\theta_1) \cap (\mathsf{vn}(q_2) \backslash \mathsf{dom}(\theta_2)) = \{\}$. To apply Lemma 8.6.14 it remains to show that every $x \in \mathsf{dom}(\theta_1 \cap \theta_2)$ then $\theta_1 x = \theta_2 x$. Also to show that $|\widetilde{n}_1 \cup \widetilde{n}_2| = |\widetilde{m}_1 \cup \widetilde{m}_2|$ it suffices to show that for every $x \in \mathsf{dom}(\theta_1)$ and $y \in \mathsf{dom}(\theta_2)$ if $x = y$ then $\theta_1 x = \theta_2 y$ and if $x \neq y$ then $\theta_1 x \neq \theta_2 y$. These can both be resolved at once by observing that for every $x \in \mathsf{dom}(\theta_1)$ and $y \in \mathsf{dom}(\theta_2)$ there is a check of the form $\mathsf{check}(x, \theta_1 x, y, \theta_2 y, w_z)$. Since the reply context succeeds it follows by Lemma 8.6.6 that every check succeeds. Thus conclude with $\widetilde{n} = \widetilde{n}_1 \cup \widetilde{n}_2$ and $\widetilde{m} = \widetilde{m}_1 \cup \widetilde{m}_2$ and $\theta = \theta_1 \cup \theta_2$ and $\rho = \rho_1 \cup \rho_2$ and thus $p_1 \bullet p_2, \mathsf{id}_{\mathsf{bn}(p)} \ll \theta(q_1 \bullet q_2), \rho$.

□

Theorem 8.6.16. *Suppose given a challenge* $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$ *and a finite set of names* N *and a process* Q *and fresh names* w *and* f *such that restricted names of the challenge union* w *and* f *are disjoint from* N*, and the free names of the challenge union the free names of* Q *are contained within* N*. If* $\mathcal{C}_p^N(Q)$ *succeeds in* k *reduction steps where* $k = \textsc{Mr}(N, p)$ *then there exists* q *and* Q' *and* ρ *such that* $Q \xrightarrow{(\nu\widetilde{n})q} Q'$ *and* $p, \mathsf{id}_{\mathsf{bn}(p)} \ll q, \rho$.

Proof: By Proposition 8.6.9 and $w, f \notin \mathsf{fn}(Q)$ and for $\mathcal{C}_p^N(Q) \Downarrow_w$ there must a reduction $Q \longmapsto^j Q''$ such that $\mathcal{C}_p^N(Q'') \longmapsto \theta(\mathsf{tests}^N(p)) \mid \rho Q'$ $\theta(\mathsf{tests}^N(p)) \longmapsto^{k'} \simeq \ulcorner w \urcorner$ and $Q'' \xrightarrow{(\nu \widetilde{m})q'} Q'$ and $\{p' \| q\} = (\theta, \rho)$. Observe that by Lemma 8.6.10 then the number of reductions for $\theta(\mathsf{tests}^N(p))$ to succeed k' is fixed and by Proposition 8.6.12 $k' = k - 1$ and so $j = 0$ and Q'' is Q.

Now by Lemma 8.6.15 $|\widetilde{n}| = |\widetilde{m}|$ and there exists a substitution θ' and $q = \theta' q'$ such that $p, \mathsf{id}_{\mathsf{bn}(p)} \ll q, \rho$ and thus $Q \xrightarrow{(\nu \widetilde{n})q} Q'$ by exploiting α-conversion. □

Thus a reply context succeeds if and only if supplied with a process that can properly reply to a challenge of the form $((\nu \widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$. From here it is straightforward to generalise to any substitution as part of the challenge and thus show the barbed congruence is a bisimulation.

Theorem 8.6.17 (Completeness of the bisimulation)**.** *The barbed congruence is a bisimulation. That is:* $\simeq \subseteq \sim$.

Proof: It is sufficient to prove that for every pair of processes P and Q such that $P \simeq Q$ and every challenge by P of the form (μ, θ, P') there exists a proper reply by Q.

When the challenge by P is $(\tau, \{\}, P')$ then the result follows by reduction closure and Proposition 8.2.2.

When the challenge by P is of the form $((\nu \widetilde{n})p, \sigma, P')$ consider the reply context $\mathcal{C}_p^N(\cdot)$ where $N = \mathsf{fn}(P) \cup \mathsf{fn}(Q)$.

By Theorem 8.6.13 and Proposition 8.4.7 then $\mathcal{C}_p^N(P)$ succeeds in k reduction steps where $k = \textsc{Mr}(N, p)$. It follows by barbed congruence that $\mathcal{C}_p^N(Q)$

succeeds in k reduction steps, and Theorem 8.6.16 implies that $Q \xrightarrow{(\nu\widetilde{n})q} Q'$ for some q and Q' and ρ' such that $p, \mathsf{id}_{\mathsf{bn}(p)} \ll q, \rho'$.

By two applications of Theorem 8.6.13 it follows that $\mathcal{C}_p^N(P) \longmapsto^k \simeq P' \mid \ulcorner w \urcorner$ and $\mathcal{C}_p^N(Q) \longmapsto^k \simeq \rho' Q' \mid \ulcorner w \urcorner$. Further, by barbed congruence and Lemma 8.1.6 it follows that $P' \simeq \rho' Q'$.

Now by Lemma 8.1.7 obtain $\sigma P' \simeq \sigma(\rho' Q')$. Use Lemma 8.4.6 to prove that $p, \mathsf{id}_{\mathsf{bn}(p)} \ll q, \rho'$ implies $p, \sigma[\mathsf{id}_{\mathsf{bn}(p)}] \ll q, \sigma[\rho']$. Now conclude by defining $\rho = \sigma[\rho']$ to obtain $p, \sigma \ll q, \rho$ and finally the proper reply by Q of $((\nu\widetilde{n})q, \rho, Q')$.

This suffices to show that barbed congruence is a bisimulation. □

Thus the barbed congruence and bisimulation relations capture the same semantics for CPC. Indeed barbed congruent, or bisimilar, processes are behaviourally equivalent and indistinguishable in any context.

8.7 Equational Reasoning

This section considers some examples where bisimulation can be used to show equivalence of processes. Since CPC does not have an operator for non-deterministic choice, it is predictable that such bisimilarities involve replicated processes. This section presents two examples where bisimulation can be used to show equivalence, a general result that subsumes both examples, and some equivalences for later use.

The first example exploits the unification of protected names with both variable and protected names. Observe that the processes $\ulcorner n \urcorner \to P \mid !n \to P$

can be subsumed by the more compact process $!n \to P$, that is

$$\ulcorner n \urcorner \to P \mid !n \to P \;\; \sim \;\; !n \to P \; .$$

Any challenge of the left hand processes can be properly responded to by the right hand process and visa versa.

The second example considers the contractive nature of binding names in CPC. A case with the pattern $\lambda x \bullet \lambda y$ can be subsumed by a replicated case with the pattern λz as long as some conditions are met. For example:

$$\lambda x \bullet \lambda y \to P \mid !\lambda z \to Q \;\; \sim \;\; !\lambda z \to Q \qquad \text{if } P \sim \{x \bullet y/z\}Q \; .$$

The same structure as the previous example exploits the replication. The side condition requires that the bodies of the cases are bisimilar when a substitution is applied to Q that preserves the structure of any pattern bound by $\lambda x \bullet \lambda y$.

These examples both arise from pattern-unification and also appear in the compatibility relation. Indeed, the examples above are instances of a general result:

$$p \to P \mid !q \to Q \;\; \sim \;\; !q \to Q \qquad \text{when } p, \mathsf{id}_{\mathsf{bn}(p)} \ll q, \rho \text{ and } P \sim \rho Q \; .$$

Theorem 8.7.1. *Given processes $P = p \to P' \mid !q \to Q'$ and $Q = !q \to Q'$ and $\sigma = \mathsf{id}_{\mathsf{bn}(p)}$ and there exists a substitution ρ such that $p, \sigma \ll q, \rho$ and $\sigma P' \sim \rho Q'$ it follows that $P \sim Q$.*

Proof: It suffices to show that any challenge by P of the form $(\mu, \sigma, \hat{P})$ there exists a proper reply by Q of the form $(\mu', \rho, \hat{Q})$ and $\hat{P} \sim \hat{Q}$.

Now proceed by induction on the structure of the inference for $P \xrightarrow{\mu} \hat{P}$.

- If the last rule is parint then $\mu = \tau$ the transition must be due to

$$\frac{Q \xrightarrow{\tau} Q''}{(p \to P' \mid Q) \xrightarrow{\tau} (p \to P') \mid Q''}$$

  and Q'' must be of the form $Q \mid Q'''$. There is the same transition $Q \xrightarrow{\tau} Q''$ and $Q'' \equiv Q \mid Q'''$ and proceed by induction on $p \to P' \mid Q \mid Q''' \sim Q \mid Q'''$.

- If the last rule is parext then there are two possibilities.

  - If the transition is as follows

$$\frac{(p \to P') \xrightarrow{p} P'}{(p \to P' \mid Q) \xrightarrow{p} P' \mid Q} \quad \mathsf{bn}(p) \cap \mathsf{fn}(Q) = \{\}$$

    due to the case rule for $p \to P'$. Now take $Q_1 = q \to Q' \mid Q \equiv Q$ and there is a transition $Q_1 \xrightarrow{q} (Q' \mid Q)$ and $\mathsf{bn}(q) \cap \mathsf{fn}(Q)$ by definition. As $p, \sigma \ll q, \rho$ and $\sigma P' = P'$ then $P' \sim \rho Q'$ and conclude with $\hat{P} = P' \mid Q$ and $\hat{Q} = \rho Q' \mid Q$.

  - If the transition is

$$\frac{Q \xrightarrow{(\nu \widetilde{n}) q'} Q''}{(p \to P' \mid Q) \xrightarrow{(\nu \widetilde{n}) q'} (p \to P' \mid Q'')}$$

    where $(\widetilde{n} \cup \mathsf{bn}(q')) \cap \mathsf{fn}(p \to P') = \{\}$. It follows that Q'' must be of

the form $Q \mid Q'''$. Then there is the same transition $Q \xrightarrow{(\nu\widetilde{n})q'} Q''$ and proceed by induction on $p \to P' \mid Q \mid Q''' \;\sim\; Q \mid Q'''$.

- If the last rule is rep then it must be due to

$$\frac{Q \xrightarrow{\mu} Q''}{(p \to P' \mid Q) \xrightarrow{\mu} (p \to P') \mid Q''}$$

and Q'' must be of the form $Q \mid Q'''$. There is the same transition $Q \xrightarrow{\mu} Q''$ and $Q'' \equiv Q \mid Q'''$ and proceed by induction on $p \to P' \mid Q \mid Q''' \;\sim\; Q \mid Q'''$.

- If the last rule is resnon then it must be of the form

$$\frac{P \xrightarrow{\mu} P''}{(\nu n)P \xrightarrow{\mu} (\nu n)P''} \quad n \notin \mathsf{names}(\mu)\ .$$

Now consider the label μ.

  - If μ is an internal action τ then by induction on $P \xrightarrow{\tau} P''$ there exists a transition $Q \xrightarrow{\tau} Q''$. As $n \notin \mathsf{names}(\tau)$ it follows that $(\nu n)Q \xrightarrow{\tau} (\nu n)Q''$ and conclude.
  - If μ is $(\nu\widetilde{n})p'$ then by induction on $P \xrightarrow{(\nu\widetilde{n})p'} P''$ and for all σ' such that $(\mathsf{bn}(p') \cup \widetilde{n}) \cap \mathsf{fn}(Q) = \{\}$ there exists q' and Q'' and ρ' such that $Q \xrightarrow{(\nu\widetilde{n})q'} Q''$ and $p', \sigma' \ll q', \rho'$ and $\sigma' P'' \sim \rho' Q''$. As $n \notin \widetilde{n}$ and by Lemma 8.4.1 $n \notin \mathsf{fn}(q')$ and by α-conversion $n \notin \mathsf{bn}(q')$, it follows that $n \notin \mathsf{names}(\mu')$. Conclude by taking $\sigma' = \mathsf{id}_{\mathsf{bn}(p')}$ such that $\sigma' P'' = P''$ and then $(\nu n)Q \xrightarrow{(\nu\widetilde{n})q'} (\nu n)\rho' Q''$.

- If the last rule is resin then it must be of the form

$$\frac{P \xrightarrow{(\nu\widetilde{n})p'} P''}{(\nu m)P \xrightarrow{(\nu\widetilde{n},m)p'} P''} \quad m \in \mathsf{vn}(p')\backslash(\widetilde{n} \cup \mathsf{pn}(p') \cup \mathsf{bn}(p'))\ .$$

  By induction hypothesis on the transition $P \xrightarrow{(\nu\widetilde{n})p'} P''$ for all σ' such that $(\mathsf{bn}(p') \cup \widetilde{n}) \cap \mathsf{fn}(Q) = \{\}$ there exists q' and Q'' and ρ' such that $Q \xrightarrow{(\nu\widetilde{n})q'} Q''$ and $p', \sigma' \ll q', \rho'$ and $\sigma' P'' \sim \rho' Q''$. By compatibility it follows that $m \in \mathsf{vn}(q')$ and $m \notin \mathsf{pn}(q')$. That $m \notin \widetilde{n}$ is by the original transition. That $m \notin \mathsf{bn}(q')$ is handled by α-conversion. Thus the side conditions hold and so by taking $\sigma' = \mathsf{id}_{\mathsf{bn}(p')}$ then $\sigma' P'' = P''$ and conclude with the transition $(\nu m)Q \xrightarrow{(\nu\widetilde{n},m)q'} \rho' Q''$.

- If the last rule is match then

$$\frac{(p \to P') \xrightarrow{p} P' \quad Q \xrightarrow{(\nu\widetilde{n})r} Q''}{(p \to P' \mid Q) \xrightarrow{\tau} (\nu\widetilde{n})(\sigma' P' \mid \theta' Q'')}$$

  such that $\{p \| r\} = (\sigma', \theta')$ and $\widetilde{n} \cap \mathsf{fn}(p \to P') = \{\}$. Take $Q_1 = q \to Q' \mid Q \equiv Q$. Since σ maps names to themselves it follows that $\sigma'[\sigma] = \sigma'$ and by Lemma 8.4.6 take $\rho' = \sigma'[\rho]$ and $p, \sigma' \ll q, \rho'$. By exploiting renaming as required $\widetilde{n} \cap \mathsf{fn}(q \to Q') = \{\}$ and by Lemma 8.4.10 $\{q \| r\} = (\rho', \theta')$ and there is a transition

$$\frac{(q \to Q') \xrightarrow{q} Q' \quad Q \xrightarrow{(\nu\widetilde{n})r} Q''}{(q \to Q' \mid Q) \xrightarrow{\tau} (\nu\widetilde{n})(\rho' Q' \mid \theta' Q'')}\ .$$

  As $\sigma' P' \sim \rho' Q'$ the conclusion follows.

□

Observe that the substitution linked to p in the compatibility relation cannot be generalised. Consider $\lambda x \to P' \mid Q$ where $P' = (\nu n)(\ulcorner n \urcorner \bullet x \mid \ulcorner n \urcorner \bullet m \to \ulcorner w \urcorner)$ where $x \neq m$ and $Q = \lambda x \to Q'$ and $Q' = (\nu n)(\ulcorner n \urcorner \mid \ulcorner n \urcorner \to \ulcorner w \urcorner)$. By taking a substitution $\sigma = \{m/x\}$ it follows that $\lambda x, \sigma \ll \lambda x, \sigma$ and that $\sigma P' \sim \sigma Q'$. However, P is not bisimilar to Q as the context $\mathcal{C}(\cdot) = \cdot \mid k$ for $k \neq m$ yields different behaviours.

The following results are of minor interest to previous results and optimisations in Appendix 10. The first has been referenced (although not required) previously in Lemma 7.1.1. The latter two are exploited in Section A.3.

Theorem 8.7.2. *Given a process P, then $!P \mid !P \quad \sim \quad !P$.*

Proof: Straightforward by induction on the inference for the transitions $!P \mid !P \xrightarrow{\mu} P'$. □

Theorem 8.7.3. *Given two processes P and Q, then $!(P \mid Q) \quad \sim \quad !P \mid !Q$.*

Proof: Straightforward by induction on the inference for the transitions $!(P|Q) \xrightarrow{\mu} P'$. □

Theorem 8.7.4. *Given a process P, then $!P \quad \sim \quad !!P$.*

Proof: Straightforward by induction on the inference for the transitions $!P \xrightarrow{\mu} P'$. □

These results conclude the discussion of bisimulation for CPC. Now that the semantics of CPC have been formalised, the next chapter exploits these to relate to CPC to other process calculi.

Chapter 9

Relations to Other Process Calculi

Now that the behavioural theory for CPC has been defined, valid encodings (Definition 3.2.1) can be used to formalise the relations between CPC and the process calculi in Chapter 3. Since the relation between π-calculus and CPC has already been formalised in Chapter 7, this chapter relates CPC to Linda, Spi calculus and fusion calculus. Although a behavioural theory is part of the definition of valid encodings, CPC's behavioural equivalence is only required for the relation to Spi calculus.

The pattern-matching of Linda can be rendered in CPC and it follows that there is a homomorphism from Linda into CPC. The converse separation result can be proved by exploiting CPC's symmetry or intensionality.

The intensionality of Spi calculus can also be homomorphically encoded into CPC up to behavioural equivalence. The lack of a valid encoding of

CPC into Spi calculus is proved by exploiting symmetry.

As the separation results between CPC and π-calculus, Linda and Spi calculus can all be proved by exploiting symmetry, the relationship between fusion calculus and CPC is of particular interest. The peculiarities of name fusion prevent a valid encoding, and thus a homomorphism, of fusion calculus into CPC. Conversely, that fusion calculus cannot validly encode CPC is shown by exploiting multiple name matching or symmetry. Consequently fusion calculus and CPC are unrelated.

9.1 Linda

In Gorla's work on valid encodings he presents a hierarchy of sets of process calculi structured according to relative expressive power [Gor08b, Gor08a]. The π-calculus is contained within one of these sets, however there are several sets that are more expressive according to the hierarchy. As this dissertation focuses upon expressive power, this section formalises the relationship between CPC and Linda, a process calculus that is within one of the four most expressive sets of Gorla's hierarchy. This relationship can be summarised as follows. There is a straightforward homomorphism from Linda into CPC. The converse separation result can be proved in two different ways using symmetry or intensionality. The rest of this section formalises these results.

The first step is to show a valid encoding of Linda into CPC. Similar to π-calculus, the encoding $[\![\cdot]\!]$ is homomorphic with respect to all operators

except for input and output which are encoded as follows:

$$\begin{aligned} [\![\, (\widetilde{t}).P \,]\!] &\stackrel{\text{def}}{=} \mathsf{pat\text{–}t}(\widetilde{t}) \to [\![\, P \,]\!] \\ [\![\, \langle \widetilde{b} \rangle \,]\!] &\stackrel{\text{def}}{=} \mathsf{pat\text{–}d}(\widetilde{b}) \to \mathbf{0} \ . \end{aligned}$$

The functions $\mathsf{pat\text{–}t}(\cdot)$ and $\mathsf{pat\text{–}d}(\cdot)$ are used to translate templates and data, respectively, into CPC patterns. The functions are defined as follows:

$$\begin{aligned} \mathsf{pat\text{–}t}(\ ) &\stackrel{\text{def}}{=} \lambda x \bullet \mathsf{in} && \text{for } x \text{ a fresh name} \\ \mathsf{pat\text{–}t}(t, \widetilde{t}) &\stackrel{\text{def}}{=} t \bullet \mathsf{in} \bullet \mathsf{pat\text{–}t}(\widetilde{t}) \\ \mathsf{pat\text{–}d}(\ ) &\stackrel{\text{def}}{=} \mathsf{in} \bullet \lambda x \\ \mathsf{pat\text{–}d}(b, \widetilde{b}) &\stackrel{\text{def}}{=} b \bullet \lambda x \bullet \mathsf{pat\text{–}d}(\widetilde{b}) && \text{for } x \text{ a fresh name} \end{aligned}$$

where in is a symbolic name as in the encoding for π-calculus. Moreover, the function $\mathsf{pat\text{–}d}(\cdot)$ associates a bound variable to every name in the sequence; this fact ensures that a pattern that translates a template and a pattern that translates a datum match only if they have the same length (this is a feature of Linda's pattern matching but not of CPC's). It is worth noting that the simpler translation $[\![\, \langle b_1, \ldots, b_n \rangle \,]\!] \stackrel{\text{def}}{=} b_1 \bullet \ldots \bullet b_n \to \mathbf{0}$ would not work: the Linda process $\langle b \rangle \mid \langle b \rangle$ does not reduce, whereas such an encoding $(b \to \mathbf{0} \mid b \to \mathbf{0})$ does. This fact would contradict Proposition 3.2.2.

Next is to prove that this encoding is valid. This is an easy corollary of the following lemma, stating a strict correspondence between Linda's pattern matching and CPC's (on patterns arising from the translation).

Lemma 9.1.1. $\textsc{Match}(\widetilde{t};\widetilde{b}) = \sigma$ *if and only if* $\{\mathsf{pat\text{–}t}(\widetilde{t})\|\mathsf{pat\text{–}d}(\widetilde{b})\} = (\sigma \cup \{\mathsf{in}/x\}, \{\mathsf{in}/x_0, \ldots, \mathsf{in}/x_n\})$, *where* $\{x_0, \ldots, x_n\} = \mathsf{bn}(\mathsf{pat\text{–}d}(\widetilde{b}))$ *and* $\mathsf{dom}(\sigma) \uplus \{x\} = \mathsf{bn}(\mathsf{pat\text{–}t}(\widetilde{t}))$ *and* σ *maps names to names.*

Proof: In both directions the proof is by induction on the length of $\widetilde{t}$. The forward direction is as follows.

- The base case is when $\widetilde{t}$ is the empty sequence of template fields; thus, $\mathsf{pat\text{–}t}(\widetilde{t}) = \lambda x \bullet \mathsf{in}$. Then by definition of pattern-unification it must be that $\widetilde{b}$ is the empty sequence and that σ is the empty substitution. Thus, $\mathsf{pat\text{–}d}(\widetilde{b}) = \mathsf{in} \bullet \lambda x$ and the thesis easily follows.

- For the inductive step $\widetilde{t} = t, \widetilde{t'}$ and $\mathsf{pat\text{–}t}(\widetilde{t}) = t \bullet \mathsf{in} \bullet \mathsf{pat\text{–}t}(\widetilde{t'})$. Then by pattern-unification it must be that $\widetilde{b} = b, \widetilde{b'}$ and $\textsc{Match}(t, b) = \sigma_1$ and $\textsc{Match}(\widetilde{t'}, \widetilde{b'}) = \sigma_2$ and $\sigma = \sigma_1 \uplus \sigma_2$. By the induction hypothesis, $\{\mathsf{pat\text{–}t}(\widetilde{t'})\|\mathsf{pat\text{–}d}(\widetilde{b'})\} = (\sigma_2 \cup \{\mathsf{in}/x\}; \{\mathsf{in}/x_1, \ldots, \mathsf{in}/x_n\})$, where $\{x_1, \ldots, x_n\} = \mathsf{bn}(\mathsf{pat\text{–}d}(\widetilde{b'}))$ and $\mathsf{dom}(\sigma_2) \uplus \{x\} = \mathsf{bn}(\mathsf{pat\text{–}t}(\widetilde{t'}))$. There are now two sub-cases to consider according to the kind of template field t.

  - If $t = \ulcorner b \urcorner$ then $\sigma_1 = \{\}$ and it follows that $\sigma = \sigma_2$ and $\{\mathsf{pat\text{–}t}(\widetilde{t})\|\mathsf{pat\text{–}d}(\widetilde{b})\} = (\sigma \cup \{\mathsf{in}/x\}, \{\mathsf{in}/x_0, \ldots, \mathsf{in}/x_n\})$.
  - If $t = \lambda y$ then $\sigma_1 = \{b/y\}$ and $y \notin \mathsf{dom}(\sigma_2)$. Thus, $\mathsf{pat\text{–}t}(\widetilde{t})$ is a pattern in CPC and it follows that $\{\mathsf{pat\text{–}t}(\widetilde{t})\|\mathsf{pat\text{–}d}(\widetilde{b})\} = (\sigma_1 \cup \sigma_2 \cup \{\mathsf{in}/x\}, \{\mathsf{in}/x_0, \ldots, \mathsf{in}/x_n\}) = (\sigma \cup \{\mathsf{in}/x\}, \{\mathsf{in}/x_0, \ldots, \mathsf{in}/x_n\})$.

The reverse direction is as follows.

- The base case is when $\widetilde{t}$ is the empty sequence of template fields; thus, $\mathsf{pat\text{–}t}(\widetilde{t}) = \lambda x \bullet \mathsf{in}$. Now proceed by contradiction. Assume that $\widetilde{b}$ is not the empty sequence. In this case, $\mathsf{pat\text{–}d}(\widetilde{b}) = b_0 \bullet \lambda x_0 \bullet (b_1 \bullet \lambda x_1 \bullet (\ldots (b_n \bullet \lambda x_n \bullet (\mathsf{in} \bullet \lambda x_{n+1}))\ldots)$, for some $n > 0$. By definition of pattern matching in CPC, $\mathsf{pat\text{–}d}(\widetilde{b})$ and $\mathsf{pat\text{–}t}(\widetilde{t})$ cannot match, and this would contradict the hypothesis. Thus, it must be that $\widetilde{b}$ is the empty sequence and easily conclude.

- The inductive case is when $\widetilde{t} = t, \widetilde{t'}$ and thus, $\mathsf{pat\text{–}t}(\widetilde{t}) = t \bullet \mathsf{in} \bullet \mathsf{pat\text{–}t}(\widetilde{t'})$. If $\widetilde{b}$ was the empty sequence, then $\mathsf{pat\text{–}d}(\widetilde{b}) = \mathsf{in} \bullet \lambda x$ and it would not match against $\mathsf{pat\text{–}t}(\widetilde{t})$. Hence, $\widetilde{b} = b, \widetilde{b'}$ and so $\mathsf{pat\text{–}d}(\widetilde{b}) = b \bullet \lambda x \bullet \mathsf{pat\text{–}d}(\widetilde{b'})$. By definition of pattern-unification in CPC it follows that $\{\mathsf{pat\text{–}t}(\widetilde{t}) \| \mathsf{pat\text{–}d}(\widetilde{b})\} = (\sigma_1 \cup \sigma_2 \cup \{\mathsf{in}/x\}, \{\mathsf{in}/x_0, \ldots, \mathsf{in}/x_n\})$, where $\{t \| b\} = (\sigma_1, \{\})$ and $\{\mathsf{pat\text{–}t}(\widetilde{t'}) \| \mathsf{pat\text{–}d}(\widetilde{b'})\} = (\sigma_2 \cup \{\mathsf{in}/x\}, \{\mathsf{in}/x_1, \ldots, \mathsf{in}/x_n\})$ and $\sigma = \sigma_1 \cup \sigma_2$. Now consider the two sub-cases according to the kind of the template field t.

  - If $t = \ulcorner b \urcorner$ then $\sigma_1 = \{\}$ and so $\sigma_2 = \sigma$. By induction hypothesis, $\textsc{Match}(\widetilde{t'}; \widetilde{b'}) = \sigma$, and so $\textsc{Match}(\widetilde{t}; \widetilde{b}) = \sigma$.
  - If $t = \lambda y$ then $\sigma_1 = \{b/y\}$ and $\sigma_2 = \{n_i/y_i\}$ for $y_i \in \mathsf{dom}(\sigma) \backslash \{y\}$ and $n_i = \sigma y_i$. Thus, $y \notin \mathsf{dom}(\sigma_2)$ and so $\sigma = \sigma_1 \uplus \sigma_2$. So by the induction hypothesis, $\textsc{Match}(\widetilde{t'}; \widetilde{b'}) = \sigma_2$ and, by definition of $\textsc{Match}$, $\textsc{Match}(t; b) = \sigma_1$. Thus, $\textsc{Match}(\widetilde{t}; \widetilde{b}) = \sigma$.

□

Lemma 9.1.2. *Two processes P and Q are structurally equivalent if and only if their translations are structurally equivalent. That is: $P \equiv Q$ if and*

only if $[\![P]\!] \equiv [\![Q]\!]$.

Proof: Trivial, from the fact that $\equiv$ acts only on operators that $[\![\cdot]\!]$ translates homomorphically. □

Theorem 9.1.3. *The translation* $[\![\cdot]\!]$ *from Linda into CPC preserves reduction and does not introduce new reductions. That is:*

- *If* $P \longmapsto P'$ *then* $[\![P]\!] \longmapsto [\![P']\!]$*;*
- *if* $[\![P]\!] \longmapsto Q$ *then* $Q = [\![P']\!]$ *for some* P' *such that* $P \longmapsto P'$*.*

Proof: Both parts can be easily proved by a straightforward induction on judgements $P \longmapsto P'$ and $[\![P]\!] \longmapsto Q$, respectively. In both cases, the base step is the most interesting one and it trivially follows from Lemma 9.1.1; the inductive cases where the last rule used is the structural one rely on Lemma 9.1.2. □

Corollary 9.1.4. *The encoding of Linda into CPC is valid.*

Proof: Reuse the proof for Corollary 7.2.3, that is as follows. Compositionality and name invariance hold by construction. Operational correspondence and divergence reflection easily follow from Theorem 9.1.3. Success sensitiveness can be proved as follows: $P \Downarrow$ means that there exists P' and $k \geq 0$ such that $P \longmapsto^k P' \equiv P'' \mid \surd$; by exploiting Theorem 9.1.3 k times and Lemma 9.1.2, obtain that $[\![P]\!] \longmapsto^k [\![P']\!] \equiv [\![P'']\!] \mid \surd$, i.e. that $[\![P]\!] \Downarrow$. The converse implication can be proved similarly. □

Corollary 9.1.5. *There is a homomorphism from Linda into CPC.*

Proof: Trivial by definition of $[\![\cdot]\!]$ and Corollary 9.1.4. □

Thus CPC subsumes Linda via a homomorphism. The converse separation result can be proved in two different ways.

The first proof exploits CPC's symmetry and the self matching process as in the proof by symmetry of Theorem 7.2.5 for the π-calculus.

Theorem 9.1.6. *There is no valid encoding of CPC into Linda.*

Proof:[by symmetry] Reuse the proof by symmetry for Theorem 7.2.5 as every Linda process T is such that if $T \mid T \longmapsto$ then $T \longmapsto$. □

The second proof exploits CPC's support for arbitrarily complex patterns that can be bound to a single name or matched explicitly. The proof is by showing that any Linda encoding that supports matching of both a single binding name λx and an arbitrary number of protected names $\ulcorner n_1 \urcorner \bullet \ldots \bullet \ulcorner n_k \urcorner$ leads to contradiction.

Proof:[by intensionality] Suppose there exists a valid encoding $[\![\cdot]\!]$ from CPC into Linda. Consider the process $S = \lambda x \rightarrow \surd$ and it's encoding $T = [\![S]\!]$. It follows that T interacts with other encoded CPC processes by an input or output with some arity k. Now consider the following CPC processes

$$\begin{aligned} S_1 &= n_1 \bullet \ldots \bullet n_k \bullet n_{k+1} \rightarrow \mathbf{0} \\ S_2 &= \ulcorner m_1 \urcorner \bullet \ldots \bullet \ulcorner m_k \urcorner \bullet \ulcorner m_{k+1} \urcorner \rightarrow \surd \, . \end{aligned}$$

It follows that $S \mid S_1 \longmapsto$ and $S \mid S_1 \Downarrow$ and so for the encoding to be valid $T \mid [\![S_1]\!] \longmapsto$ and $T \mid [\![S_1]\!] \Downarrow$. Also if $n_i = m_i$ for all i then $S_1 \mid S_2 \longmapsto$ and

$S_1 \mid S_2 \Downarrow$ and so $[\![S_1]\!] \mid [\![S_2]\!] \longmapsto$ and $[\![S_1]\!] \mid [\![S_2]\!] \Downarrow$ for the encoding to be valid.

Now consider the arity of the interaction between $[\![S_1]\!]$ and $[\![S_2]\!]$.

- If the arity is less than or equal to k then at least one n_i and m_i are not being tested for equality in the interaction. Thus, there exists some $S_3 = \ulcorner o_1 \urcorner \bullet \ldots \bullet \ulcorner o_i \urcorner$ such that at least one $n_i \neq o_i$ and thus $S_1 \mid S_3 \not\longmapsto$, however $[\![S_1]\!] \mid [\![S_3]\!] \longmapsto$ and this contradicts Proposition 3.2.2.

- If the arity is greater than k then $[\![S_1]\!]$ must interact with arity k and arity greater than k. Clearly there is contradiction if $[\![S_1]\!]$ has a single input or output as this would require $k > k$, thus it must be that $[\![S_1]\!]$ is of the form $(\nu\widetilde{o})(T_1 \mid T_2)$ where T_1 has an input or output of arity k and T_2 has an input or output of arity greater than k. Also it follows that $T \mid (\nu\widetilde{o})T_1 \longmapsto$ and $[\![S_2]\!] \mid (\nu\widetilde{o})T_2 \longmapsto$. Therefore $T \mid (\nu\widetilde{o})(T_1 \mid T_2) \mid [\![S_2]\!] \longmapsto^2$, however $S \mid S_1 \mid S_2 \not\longmapsto^2$ and this contradicts Proposition 3.2.2.

□

Observe that the proof by intensionality technique could be adapted to use $k + 1$ binding names to show that CPC's intensionality cannot be rendered into a polyadic calculus by rendering compounds as polyadic structures. Here the use of name matching simplifies the proof and illustrates that the technique holds even when both calculi, e.g. Linda and CPC, have infinite matching degree.

Corollary 9.1.7. *There is no homomorphism from CPC into Linda.*

Proof: By Theorem 9.1.6. □

Thus there is no homomorphism, or valid encoding, of CPC into Linda and hence CPC is more expressive than the sets of calculi captured by Gorla's hierarchy.

9.2 Spi Calculus

As Spi calculus introduces some intensionality in a process calculus, the relationship to CPC is also of interest. The intensionality of Spi calculus can be homomorphically encoded into CPC. The symmetry of CPC cannot be rendered in Spi calculus, thus ensuring there is no converse encoding. This section takes some space to cover the rich syntax and reduction rules of Spi calculus and also to discuss peculiarities of the encoding.

The first step is to show that there is an encoding of Spi calculus into CPC. The terms can be encoded as patterns using the reserved names pair, encr, 0 and suc by

$$\begin{aligned}
[\![n]\!] &\stackrel{\text{def}}{=} n \\
[\![x]\!] &\stackrel{\text{def}}{=} x \\
[\![0]\!] &\stackrel{\text{def}}{=} 0 \\
[\![suc(M)]\!] &\stackrel{\text{def}}{=} \mathsf{suc} \bullet [\![M]\!] \\
[\![(M, N)]\!] &\stackrel{\text{def}}{=} \mathsf{pair} \bullet [\![M]\!] \bullet [\![N]\!] \\
[\![\{M\}_N]\!] &\stackrel{\text{def}}{=} \mathsf{encr} \bullet [\![M]\!] \bullet [\![N]\!] .
\end{aligned}$$

The tagging is used for safety, as otherwise there are potential pathologies in the translation: without tags, the representation of a natural number could be confused with a pair or an encryption.

The encoding of the familiar process forms are homomorphic as expected. The input and output both encode as cases as in π-calculus:

$$\begin{array}{lcll} [\![\, M(x).P \,]\!] & \stackrel{\text{def}}{=} & [\![\, M \,]\!] \bullet \lambda x \bullet \mathsf{in} \rightarrow [\![\, P \,]\!] & \\ [\![\, \overline{M}\langle N \rangle.P \,]\!] & \stackrel{\text{def}}{=} & [\![\, M \,]\!] \bullet ([\![\, N \,]\!]) \bullet \lambda x \rightarrow [\![\, P \,]\!] & x \text{ is a fresh name.} \end{array}$$

The symbolic name in (input) and fresh name x (output) are used to ensure that encoded inputs will only match with encoded outputs as in π-calculus and Linda.

The four remaining process forms all require pattern matching and so translate to cases in parallel. In each encoding a fresh name n is used to prevent interaction with other processes, see Lemma 6.2.6. As in the Spi calculus, the encodings will reduce only after a successful matching and will

be stuck otherwise. The encodings are

$$
\begin{aligned}
[\![\, [M \ is \ N]P \,]\!] &\stackrel{\text{def}}{=} (\nu n)(\ulcorner n \urcorner \bullet [\![\, M \,]\!] \to [\![\, P \,]\!] \mid \ulcorner n \urcorner \bullet [\![\, N \,]\!]) \\
[\![\, let \ (x,y) = M \ in \ P \,]\!] &\stackrel{\text{def}}{=} (\nu n)(\ulcorner n \urcorner \bullet (\ulcorner \mathsf{pair} \urcorner \bullet \lambda x \bullet \lambda y) \to [\![\, P \,]\!] \\
&\qquad\qquad |\ulcorner n \urcorner \bullet [\![\, M \,]\!]) \\
[\![\, case \ M \ of \ \{x\}_N : P \,]\!] &\stackrel{\text{def}}{=} (\nu n)(\ulcorner n \urcorner \bullet (\ulcorner \mathsf{encr} \urcorner \bullet \lambda x \bullet [\![\, N \,]\!]) \to [\![\, P \,]\!] \\
&\qquad\qquad |\ulcorner n \urcorner \bullet [\![\, M \,]\!]) \\
[\![\, case \ M \ of \ 0 : P \ suc(x) : Q \,]\!] &\stackrel{\text{def}}{=} (\nu n)(\ulcorner n \urcorner \bullet \ulcorner 0 \urcorner \to [\![\, P \,]\!] \\
&\qquad\qquad |\ulcorner n \urcorner \bullet (\ulcorner \mathsf{suc} \urcorner \bullet \lambda x) \to [\![\, Q \,]\!] \\
&\qquad\qquad |\ulcorner n \urcorner \bullet [\![\, M \,]\!]) \ .
\end{aligned}
$$

The match $[M \ is \ N]P$ only reduces to P if $M = N$, thus the encoding creates two patterns using $[\![\, M \,]\!]$ and $[\![\, N \,]\!]$ with one reducing to $[\![\, P \,]\!]$. The pair splitting $let \ (x,y) = M \ in \ P$ encoding creates a case with a pattern that matches a tagged pair and binds the components to x and y in $[\![\, P \,]\!]$. This is put in parallel with another case that has $[\![\, M \,]\!]$ in the pattern. The decryption case $case \ M \ of \ \{x\}_N : P$ checks whether M is a message encoded with key $[\![\, N \,]\!]$ and retrieves the value encrypted by binding it to x in the continuation. Lastly the integer case $case \ M \ of \ 0 : P \ suc(x) : Q$ translation creates a case for each of the zero and the successor possibilities. These cases match the tag and the reserved names 0, reducing to $[\![\, P \,]\!]$, or suc and binding x in $[\![\, Q \,]\!]$. The term to be compared M is as in the others.

Lemma 9.2.1. *Two processes P and Q are structurally equivalent if and only if their translations are structurally equivalent. That is: $P \equiv Q$ if and only if $[\![P]\!] \equiv [\![Q]\!]$.*

Proof: Trivial, from the fact that $\equiv$ acts only on operators that $[\![\cdot]\!]$ translates homomorphically. □

Theorem 9.2.2. *The translation $[\![\cdot]\!]$ from Spi calculus into CPC preserves reduction and does not introduce new reductions. That is:*

- *If $P \longmapsto P'$ then $[\![P]\!] \longmapsto \simeq_2 [\![P']\!]$;*
- *if $[\![P]\!] \longmapsto Q$ then $Q \simeq_2 [\![P']\!]$ for some P' such that $P \longmapsto P'$*

where $\simeq_2$ is barbed congruence for CPC.

Proof: The first claim can be easily proved by a straightforward induction on judgement $P \longmapsto P'$. The base case is proved by reasoning on the Spi axiom used to infer the reduction. Although all the cases are straightforward, a reduction rule for integers is shown for illustration. Consider the reduction for a successor as the reduction for zero is simpler. In this case, $P = case\ suc(M)\ of\ 0 : P_1\ suc(x) : P_2$ and $P' = \{M/x\}P_2$. Then,

$$\begin{aligned} [\![P]\!] \stackrel{\text{def}}{=}\ & (\nu n)(\ulcorner n \urcorner \bullet (\mathsf{num} \bullet \ulcorner 0 \urcorner) \rightarrow [\![P_1]\!] \\ & \quad \mid \ulcorner n \urcorner \bullet (\mathsf{num} \bullet (\ulcorner \mathsf{suc} \urcorner \bullet \lambda x)) \rightarrow [\![P_2]\!] \\ & \quad \mid \ulcorner n \urcorner \bullet (\mathsf{num} \bullet (\mathsf{suc} \bullet [\![M]\!])) \rightarrow \mathbf{0}) \ . \end{aligned}$$

and it can only reduce to

$$\{[\![M]\!]/x\}[\![P_2]\!] \mid (\nu n)\ulcorner n \urcorner \bullet (\mathsf{num} \bullet \ulcorner 0 \urcorner) \rightarrow [\![P_1]\!] \ .$$

By a straightforward induction on the structure of P_2 it is easy to prove that $\{[\![M]\!]/x\}[\![P_2]\!] = [\![\{M/x\}P_2]\!]$. Thus, $[\![P]\!] \longmapsto [\![\{M/x\}P_2]\!] \mid (\nu n)\ulcorner n \urcorner \bullet (\mathsf{num} \bullet \ulcorner 0 \urcorner) \rightarrow [\![P_1]\!] \simeq_2 [\![P']\!]$, where the last equivalence follows from Lemma 6.2.6. The inductive case is straightforward, with the structural case relying on Lemma 9.2.1.

The second part can be proved by induction on judgement $[\![P]\!] \longmapsto Q$. There is just one base case, i.e. when $[\![P]\!] = p \rightarrow Q_1 \mid q \rightarrow Q_2$ and $Q = \sigma Q_1 \mid \rho Q_2$ and $\{p \| q\} = (\sigma, \rho)$. By definition of the encoding, it can only be that $p = [\![M]\!] \bullet \lambda x \bullet \mathsf{in}$ and $Q_1 = [\![P_1]\!]$ and $q = [\![M]\!] \bullet ([\![N]\!]) \bullet \lambda x$ and $Q_2 = [\![P_2]\!]$ for some P_1, P_2, M and N. This means that $P = M(x).P_1 \mid \overline{M}\langle N \rangle.P_2$ and that $Q = \{[\![N]\!]/x\}[\![P_1]\!] \mid [\![P_2]\!] = [\![\{N/x\}P_1 \mid P_2]\!]$. To conclude, it suffices to take $P' = \{N/x\}P_1 \mid P_2$. For the inductive case there are two possibilities.

- The inference of $[\![P]\!] \longmapsto Q$ ends with an application of the rule for parallel composition or for structural equivalence: this case can be proved by a straightforward induction.

- The inference of $[\![P]\!] \longmapsto Q$ ends with an application of the rule for restriction; thus, $[\![P]\!] = (\nu n)Q'$, with $Q' \longmapsto Q''$ and $Q = (\nu n)Q''$. If $Q' = [\![P'']\!]$, for some P'', apply a straightforward induction. Otherwise, there are the following four possibilities.

  - $Q' = \ulcorner n \urcorner \bullet \ulcorner [\![M]\!] \urcorner \rightarrow [\![P_1]\!] \mid \ulcorner n \urcorner \bullet \ulcorner [\![N]\!] \urcorner$ and, hence, $Q'' = [\![P_1]\!]$. By definition of the encoding, $P = [M \ is \ N]P_1$. Notice that the reduction $Q' \longmapsto Q''$ can happen only if $[\![M]\!]$ and $[\![N]\!]$ match; by construction of the encoding of Spi-terms, this can happen only

if $M = N$ and, hence, $P \longmapsto P_1$. The thesis follows by letting $P' = P_1$, since n is a fresh name and so $Q = (\nu n)[\![P_1]\!] \equiv [\![P_1]\!]$.

– $Q' = \ulcorner n \urcorner \bullet (\ulcorner \mathsf{pair} \urcorner \bullet (\lambda x \bullet \lambda y)) \rightarrow [\![P_1]\!] \mid \ulcorner n \urcorner \bullet (\mathsf{pair} \bullet ([\![M]\!] \bullet [\![N]\!]))$ and, hence, $Q'' = \{[\![M]\!]/x, [\![N]\!]/y\}[\![P_1]\!]$. This case is similar to the previous one, by letting P be *let* $(x, y) = (M, N)$ *in* P_1.

– $Q' = \ulcorner n \urcorner \bullet (\ulcorner \mathsf{encr} \urcorner \bullet (\lambda x \bullet [\![N]\!])) \rightarrow [\![P_1]\!] \mid \ulcorner n \urcorner \bullet (\mathsf{encr} \bullet ([\![M]\!] \bullet [\![N]\!]))$ and, hence, $Q'' = \{[\![M]\!]/x\}[\![P_1]\!]$. This case is similar to the previous one, by letting P be *case* $\{M\}_N$ *of* $\{x\}_N : P_1$.

– $Q' = \ulcorner n \urcorner \bullet (\mathsf{num} \bullet \ulcorner 0 \urcorner) \rightarrow [\![P_1]\!] \mid \ulcorner n \urcorner \bullet (\mathsf{num} \bullet (\ulcorner \mathsf{suc} \urcorner \bullet \lambda x)) \rightarrow [\![P_2]\!] \mid \ulcorner n \urcorner \bullet [\![M]\!]$. Hence, $P =$ *case* M *of* $0 : P_1$ $suc(x) : P_2$. According to the kind of $[\![M]\!]$, there are two sub-cases (notice that, since $Q' \longmapsto Q''$, no other possibility is allowed for $[\![M]\!]$).

  * $[\![M]\!] = \mathsf{num} \bullet 0$: in this case, $Q'' = [\![P_1]\!] \mid \ulcorner n \urcorner \bullet (\mathsf{num} \bullet (\ulcorner \mathsf{suc} \urcorner \bullet \lambda x)) \rightarrow [\![P_2]\!]$ and so $Q = (\nu n)Q'' \equiv [\![P_1]\!] \mid (\nu n)\ulcorner n \urcorner \bullet (\mathsf{num} \bullet (\ulcorner \mathsf{suc} \urcorner \bullet \lambda x)) \rightarrow [\![P_2]\!] \simeq_2 [\![P_1]\!]$. In this case, $M = 0$ and so $P \longmapsto P_1$; to conclude, it suffices to let P' be P_1.

  * $[\![M]\!] = \mathsf{num} \bullet (\mathsf{suc} \bullet [\![M']\!])$, for some M': in this case, $Q'' = \{[\![M']\!]/x\}[\![P_2]\!] \mid \ulcorner n \urcorner \bullet (\mathsf{num} \bullet \ulcorner 0 \urcorner) \rightarrow [\![P_1]\!]$ and so $Q = (\nu n)Q'' \equiv [\![\{M'/x\}P_2]\!] \mid (\nu n)\ulcorner n \urcorner \bullet (\mathsf{num} \bullet 0) \rightarrow [\![P_1]\!] \simeq_2 [\![\{M'/x\}P_2]\!]$. In this case, $M = suc(M')$ and so $P \longmapsto \{M'/x\}P_2$; to conclude, it suffices to let P' be $\{M'/x\}P_2$.

□

Corollary 9.2.3. *The encoding of Spi calculus into CPC is valid.*

Proof: Reuse the proof for Corollaries 7.2.3 and 9.1.4, that is as follows. Compositionality and name invariance hold by construction. Operational correspondence and divergence reflection easily follow from Theorem 9.2.2. Success sensitiveness can be proved as follows: $P \Downarrow$ means that there exists P' and $k \geq 0$ such that $P \longmapsto^k P' \equiv P'' \mid \surd$; by exploiting Theorem 9.2.2 k times and Lemma 9.2.1, obtain that $[\![P]\!] \longmapsto^k [\![P']\!] \equiv [\![P'']\!] \mid \surd$, i.e. that $[\![P]\!] \Downarrow$. The converse implication can be proved similarly. □

Corollary 9.2.4. *There is a homomorphism from Spi calculus into CPC.*

Proof: By definition of $[\![\cdot]\!]$ and Corollary 9.2.3. □

Thus CPC is able to render the intensionality of Spi calculus and there is a homomorphism from Spi calculus into CPC. The converse separation result exploits the symmetry of CPC.

Theorem 9.2.5. *There is no valid encoding of CPC into Spi calculus.*

Proof: Exploit the self-matching CPC process $x \rightarrow \surd$ as in the proof by symmetry in Theorems 7.2.5 & 9.1.6. □

Corollary 9.2.6. *There is no homomorphism from CPC into Spi calculus.*

Proof: By Theorem 9.2.5. □

Now that the relations between Spi calculus and CPC have been formalised, the rest of this section considers some artefacts of the encoding of Spi calculus into CPC.

Notice that the criteria for a valid encoding does not imply full abstraction of the encoding (actually, they were defined as an alternative to full abstraction [Gor08a, Gor08b]). This means that the encodings of equivalent Spi calculus processes can be distinguished by contexts in CPC that do not result from the encoding of any Spi calculus context. Indeed, while this encoding allows Spi calculus to be modelled in CPC, it does *not* entail that cryptography can be properly rendered. Consider the pattern $\mathsf{encr} \bullet \lambda x \bullet \lambda y$ that could match the encoding of an encrypted term to bind the message and key, so that CPC can break any encryption! Indeed this is an artefact of the straightforward approach to encoding taken here.

An alternative approach to encryption is to exploit CPC's matching degree. A restricted name can be created for the ciphertext c and then a process created that offers the plaintext p to any process that knows the ciphertext and the key k. For example

$$(\nu c)(!\ulcorner k \urcorner \bullet \ulcorner c \urcorner \bullet p \mid P)$$

where P is the process that transmits the ciphertext. Here any other process must know both the key and ciphertext to obtain the plaintext, by Lemma 6.2.6. Note that the replication is used to allow any number of copies of the plaintext to be accessed.

Indeed, such an approach can modify the process offering the plaintext to yield interesting effects. For example, by removing the replication the

process encrypting the message can ensure that only one recipient can access the plaintext:

$$(\nu c)(\ulcorner k \urcorner \bullet \ulcorner c \urcorner \bullet p \mid P) \ .$$

Of course this recipient may then make the plaintext available after obtaining it, however the plaintext can only accessed once from the original encryption.

The behaviour of the process that offers the plaintext can be taken advantage of in other ways. By modifying the process slightly it is possible to detect each time the plaintext is accessed. Consider the following process

$$(\nu c)(!\ulcorner k \urcorner \bullet \ulcorner c \urcorner \bullet p \bullet \lambda x \rightarrow R \mid P)$$

where R reports back to P that the plaintext has been accessed. A minor modification here is to add a binding name λx to the process offering the plaintext as otherwise the replication would cause an unbounded number of interactions and false reports of access via R. Of course these mechanisms could be combined to limit and detect how often the plaintext is accessed.

Another effect of this approach is that encryption hides information about the message from third parties that may only transport ciphertext. In Spi calculus the encryption of the same plaintext with the same key can be tested for equality

$$[c_1 \ is \ c_2]P \ \longmapsto \ P \qquad \text{if } c_1 = \{M\}_N \text{ and } c_2 = \{M\}_N \ .$$

So even though the process above is not able to access the plaintext M encrypted by N and bound to c_1 and c_2, testing for equality reveals that the message and keys are the same. This does not hold when restricted names are used for the ciphertext as α-conversion ensures they can never be equal.

9.3 Fusion Calculus

As the separation results for CPC and the other process calculi presented so far can all be proved via symmetry, the relationship between fusion calculus and CPC is of particular interest. It turns out that the peculiarities of name fusion prevent a valid encoding of fusion calculus into CPC. Conversely, that fusion calculus cannot validly encode CPC can be shown by exploiting matching degree or even symmetry. Consequently, fusion calculus and CPC turn out to be unrelated. The rest of this section formalises these results.

The lack of a valid encoding of fusion calculus into CPC is ensured by the following theorem. The proof is by contradiction and can be summarised as follows. Define two fusion calculus processes in parallel composition such that one is an input and the other an output on the same channel name. Now place these processes under a restriction so that they can perform a reduction and report success. Thus, the reduction arises from a ternary cooperation between the two processes and the restriction. Since interactions are binary in CPC, it can be shown that any reduction of the encoding yields a contradiction.

Theorem 9.3.1. *There exists no valid encoding of Fusion calculus into CPC.*

Proof: By contradiction, assume that there exists a valid encoding $[\![\cdot]\!]$ of Fusion into CPC. Consider the Fusion process $P \stackrel{\text{def}}{=} (\nu x)(\overline{u}\langle x\rangle \mid u(y).\surd)$, for x, y and u pairwise distinct. By success sensitiveness, $P \Downarrow$ entails that $[\![P]\!] \Downarrow$; first consider that the latter fact can only happen after a reduction of $[\![P]\!]$, i.e. that every occurrence of $\surd$ in $[\![P]\!]$ falls underneath some prefix. By compositionality, $[\![P]\!] \stackrel{\text{def}}{=} \mathcal{C}^{\{u,x,y\}}_{(\nu x)}(\mathcal{C}^{\{u,x,y\}}_{|}([\![\overline{u}\langle x\rangle]\!]; [\![u(y).\surd]\!]))$. If $[\![P]\!]$ had a top-level unguarded occurrence of $\surd$, then such an occurrence could be in $\mathcal{C}^{\{u,x,y\}}_{(\nu x)}(\_)$, in $\mathcal{C}^{\{u,x,y\}}_{|}(\_1;\_2)$, in $[\![\overline{u}\langle x\rangle]\!]$ or in $[\![u(y).\surd]\!]$; in any case, it would also follow that at least one of $[\![(\nu x)(\overline{u}\langle x\rangle \mid y(u).\surd)]\!]$ or $[\![(\nu x)(\overline{x}\langle u\rangle \mid u(y).\surd)]\!]$ would report success, whereas both $(\nu x)(\overline{u}\langle x\rangle \mid y(u).\surd) \not\Downarrow$ and $(\nu x)(\overline{x}\langle u\rangle \mid u(y).\surd) \not\Downarrow$, against success sensitiveness of $[\![\cdot]\!]$. Thus, the only possibility for $[\![P]\!]$ to report success is to perform some reduction steps (at least one) and then exhibit a top-level unguarded occurrence of $\surd$.

Thus it is clear that $[\![P]\!]$ must reduce. Now prove to that every possible reduction leads to contradiction of validity of $[\![\cdot]\!]$; this suffices to conclude. There are five possibilities for any reduction $[\![P]\!] \longmapsto$.

1. Either $\mathcal{C}^{\{u,x,y\}}_{(\nu x)} \longmapsto$, or $\mathcal{C}^{\{u,x,y\}}_{|} \longmapsto$, or $[\![\overline{u}\langle x\rangle]\!] \longmapsto$ or $[\![u(y).\surd]\!] \longmapsto$. In any of these cases, at least one out of $[\![(\nu x)(\overline{u}\langle x\rangle \mid y(u).\surd)]\!]$ or $[\![(\nu x)(\overline{x}\langle u\rangle \mid u(y).\surd)]\!]$ would reduce, however $(\nu x)(\overline{u}\langle x\rangle \mid y(u).\surd) \not\longmapsto$ and $(\nu x)(\overline{x}\langle u\rangle \mid u(y).\surd) \not\longmapsto$, against Proposition 3.2.2 (that must hold whenever $[\![\cdot]\!]$ is valid).

2. Reduction is generated by interaction between $\mathcal{C}^{\{u,x,y\}}_{(\nu x)}$ and $\mathcal{C}^{\{u,x,y\}}_{|}$. As before, $[\![(\nu x)(\overline{u}\langle x\rangle \mid y(u).\surd)]\!] \longmapsto$ whereas $(\nu x)(\overline{u}\langle x\rangle \mid y(u).\surd) \not\longmapsto$,

against Proposition 3.2.2.

3. Reduction is generated by interaction between $\mathcal{C}_{\mathsf{op}}^{\{u,x,y\}}$ and $[\![\, \overline{u}\langle x\rangle \,]\!]$, for $\mathsf{op} \in \{(\nu x), \mid\}$. Like case 2.

4. Reduction is generated by interaction between $\mathcal{C}_{\mathsf{op}}^{\{u,x,y\}}$ and $[\![\, u(y).\surd \,]\!]$, for $\mathsf{op} \in \{(\nu x), \mid\}$. As before it follows that $[\![\, (\nu x)(\overline{x}\langle u\rangle \mid u(y).\surd) \,]\!] \longmapsto$ whereas $(\nu x)(\overline{x}\langle u\rangle \mid u(y).\surd) \not\longmapsto$, against Proposition 3.2.2.

5. The reduction is generated by an interaction between the processes $[\![\, \overline{u}\langle x\rangle \,]\!]$ and $[\![\, u(y).\surd \,]\!]$. In this case, it follows that $[\![\, \overline{u}\langle x\rangle \mid u(y).\surd \,]\!] \longmapsto$ whereas $\overline{u}\langle x\rangle \mid u(y).\surd \not\longmapsto$: indeed, the interaction rule of Fusion imposes that at least one between x and y must be restricted to yield the interaction.

□

Corollary 9.3.2. *There is no homomorphism from fusion calculus into CPC.*

Proof: By Theorem 9.3.1 □

The converse separation result can be proved in two ways. The first exploits the matching degree as in the proof by matching degree of Theorem 7.2.5 for the π-calculus.

Theorem 9.3.3. *There is no valid encoding of CPC into fusion calculus.*

Proof:[by matching degree] Observe that the matching degree of fusion calculus $\textsc{Md}(fusion)$ is one while the matching degree of CPC $\textsc{Md}(CPC)$ is infinite. Now apply Theorem 3.2.4. □

The alternative proof exploits CPC's symmetry and the self matching process as in the proof by symmetry of Theorems 7.2.5, 9.1.6 & 9.2.5.

Proof:[by symmetry] Reuse the proof by symmetry for Theorems 7.2.5, 9.1.6 & 9.2.5 as every fusion calculus process T is such that if $T \mid T \longmapsto$ then $T \longmapsto$. □

Corollary 9.3.4. *There is no homomorphism from CPC into fusion calculus.*

Proof: By Theorem 9.3.3. □

Thus there is no valid encoding or homomorphism from CPC into fusion calculus despite them both supporting a notion of symmetry. Observe that Theorem 9.1.6 could also be adapted to show that fusion calculus is unable to render the intensionality of CPC.

This concludes the expressiveness results for CPC. An alternative path of development for CPC is by implementing the theory in a programming language. Details of this implementation are in Appendix 10. The next chapter presents applications exploiting CPC in a programming language.

Chapter 10

Applications

This chapter develops two major trading examples. The first introduces basic Concurrent **bondi** [Con11] syntax while redeveloping and extending the traders example from Section 6.3. By contrast with the previous development, the focus here is upon Concurrent **bondi** syntax and on extending the example to consider multiple traders. The second exploits the flexibility and expressive power of both sequential and concurrent Concurrent **bondi** programs. This includes discussion of Concurrent **bondi**'s sequential features and how their interplay with concurrency can be exploited for distributed computing on heterogeneous data structures and programming paradigms.

The essence of trade is the discovery of a compatible trading partner and the exchange of information to complete the deal. The symmetric pattern-unification and information exchange of CPC provide a natural language to express such trades. As in Section 6.3 trading can be considered in two phases. The first phase exploits the structure of patterns to express infor-

mation and then pattern-unification to allow traders to discover their shared interest. The second phase exploits the ability to exchange information in a single interaction to complete the deal fairly and without opportunity for incomplete trades.

More practically, trade often involves collaboration between systems that have different data representations and limited information about one aother. Further computation and flexibility is required to easily support trade between systems that have different programming paradigms. The interplay of sequential and concurrent intensionality can be utilised to develop trading applications that support distributed computing across heterogeneous systems, paradigms and data [GWJ11].

Note that Concurrent **bondi** is an augmentation of the **bondi** programming language and interpreter [bon11] to support CPC. The basis of **bondi** is to implement the key concepts of pattern calculus and show how this can express many programming styles with a small pattern-matching core [GWHJ07, JK09, Jay09, bon11, GWJ11]. Several design decisions have been made to integrate sequential and concurrent computation in Concurrent **bondi**, most significantly: the interplay of Concurrent **bondi** programs and CPC patterns, Concurrent **bondi** syntax for CPC, and typing for CPC patterns and processes. As discussion of these design decisions and implementing CPC in a programming language appear in Appendix A, the rest of this chapter will focus on the examples and present Concurrent **bondi** through them. Details of Concurrent **bondi** syntax with respect to CPC

are in Section A.1, however readers familiar with conventions of Objective Caml's functional and imperative style [Cam11], and Java's object-oriented style [GJS05] should be able to follow this chapter without reading the appendix.

10.1 Trade

This section redevelops the example presented in Section 6.3 to introduce CPC syntax and programs in Concurrent **bondi**. (The following presentation highlights the full example code that is available here `http://www-staff.it.uts.edu.au/~tgwilson/concurrent_bondi/section_10.1.bon`.) Recall that the scenario is of two potential traders, a buyer and a seller, who wish to engage in trade. To successfully complete a transaction the traders need to progress through two stages; discovering one other and exchanging information. The discovery phase is resolved when compatible traders find each other by unifying on some common pattern or information. The exchange phase occurs when the buyer and seller have agreed upon a transaction. Now information is exchanged in a single interaction, preventing any incomplete trades from occurring.

There are three stages to the original development in Section 6.3 that increase in sophistication. The first focuses upon discovery. The second introduces a registrar and identity validation. The third uses protected names to ensure privacy. Here, a further stage introduces a market with many

buyers and sellers as well as a broker that acts as both buyer and seller.

Although the redevelopment could simply use (restricted) names for all information, the example is clearer when Concurrent **bondi** algebraic data types (ADTs) are used. These can also be exploited in the success states to display legible results.

The first step is to define some useful ADTs for the stock information.

```
datatype Price = Price of Float
with toString += | Price p -> "$" ^ (toString p);;
datatype Stock = Stock of String and Int and Price
with toString += | Stock s i p -> s ^ " " ^
                        (toString i) ^ " " ^ (toString p);;
```

The first line declares an ADT for prices that are floating point numbers in the style of OCaml. The second line is a novel feature of Concurrent **bondi** that allows dynamic addition of cases to existing functions. The default `toString` function would result in `"Price"` concatenated with the floating point number as a string. Here a special case is added with `+=` that matches the pattern `Price p` of a `Price` data type applied to a floating point number `p` and instead yields `"$"` concatenated with the string representation of `p`. The double semi-colon `;;` terminates a declaration in Concurrent **bondi**. These techniques are repeated for the `Stock` ADT.

Similar declarations define the bank accounts (with number and name) and share certificates (with company and number of shares).

```
datatype BankAccount = BankA of Int and String
with toString += | BankA n s -> (toString n) ^ " " ^ s;;
datatype Certificate = Cert of String and Int
with toString += | Cert s i -> s ^ " " ^ (toString i);;
```

Also declare a data type of identities for later use.

```
datatype Identity = Id of Int
with toString += | Id i -> "ID" ^ (toString i);;
```

These ADTs can be used to declare the information used in redeveloping the solutions from before.

```
let s = Stock "ABC" 100 (Price 0.38);;
let b = BankA 123456 "Buyer";;
let c = Cert "ABC" 100;;
let bid = Id 0;;
let sid = Id 1;;
```

Here `s` is the stock information the traders are interested in; `b` is the buyer's bank account information; `c` is the seller's share certificates; and `bid` and `sid` are the buyer and seller identities, respectively. Observe that `let` declarations are similar to OCaml with Concurrent **bondi** constructors being applied to their arguments (rather than a single tuple of all the arguments). This supports partial application of constructors and shall be exploited later with the stock to separate out price information.

Solution 1

Consider two traders, a buyer and a seller. The buyer is interesting in purchasing stock described by `s` and is willing to pay using bank account information `b`. The buyer wishes to find a compatible trader who will provide a name `m` used to exchange bank account information for share certificates (bound by `x`). The first pattern that supports unification of `s` and binding `m` is represented as `s, \m` where the comma denotes tupling of the components and the backslash denotes a binding name. The second pattern to unify on `m` and exchange `b` for some information bound to `x` is similarly represented by `m, b, \x`. Combining these into a process is through cases of the form

`cpc p -> P` where `p` is a pattern and `P` a Concurrent **bondi** program. Thus, the Concurrent **bondi** program to represent the buyer and print the purchase result can be defined as follows.

```
cpc s, \m ->
    cpc m, b, \x ->
        println ("Bought " ^ (toString x));;
```

The basic structure of the program for the seller is the same as the buyer, the only significant difference is the creation of a restricted name. This is declared with the `rest` keyword, for example `rest n in P` creates a fresh (unique internal value) name `n` for the process `P`. Thus the seller can be defined with the following program.

```
rest n in
    cpc s, n ->
        cpc n, \y, c ->
            println ("Bill " ^ (toString y));;
```

Like the buyer, the seller unifies on `s` and shares `n`, then uses `n` to exchange `c` for information bound by `y` and completes by printing the bank account to charge.

Interaction between the buyer and seller in Concurrent **bondi** becomes possible when both are declared and consequently added to a common data-space. This is illustrated below using the Concurrent **bondi** interpreter.

```
~~ cpc s, \m ->
       cpc m, b, \x ->
           println ("Bought " ^ (toString x));;
it: Unit
~~ rest n in
       cpc s, n ->
           cpc n, \y, c ->
               println ("Bill " ^ (toString y));;
it: Unit
```

```
"Bill 123456 Buyer"
"Bought ABC 100"
~~
```

The double tilde `~~` is the prompt of the Concurrent **bondi** interpreter that accepts declarations and responds with the type and value of the program. Thus the result of each anonymous declaration is `it: Unit` to indicate unit type and value. The printed messages `"Bill 123456 Buyer"` and `"Bought ABC 100"` are the output of the traders successfully interacting.

Solution 2

The first solution allows the traders to discover each other and exchange information atomically to complete the transaction. The second solution adds identities for the traders and a registrar to keep track of registered traders, thus ensuring that traders are trustworthy. The traders now offer their identity to potential partners and then confirm with the registrar that their potential trade partner is a valid trader, i.e. trustworthy.

The discovery phase now includes the buyer exchanging their identity `bid` for the potential partner's identity, bound by `j`. The buyer then checks with the registrar if the potential partner is a valid trader. This is achieved by interacting with the registrar via a a shared name `nb` and checking the registrar knows `j` with unification providing a name, bound to `m`, to then complete the transaction. The exchange phase continues as before using the name provided by the registrar.

```
cpc s, bid, \j ->
    cpc nb, j, \m ->
        cpc m, b, \x ->
            println ("Bought " ^ (toString x))
```

The first pattern now exchanges the buyer's identity for the seller's, bound to `j`. The buyer then consults the registrar using `nb` to validate `j` with the registrar providing the name to complete the exchange.

The seller is defined simmilarly with their identity `sid` and shared name with the registrar `ns`.

```
cpc s, \j, sid ->
    cpc ns, j, \m ->
        cpc m, \y, c ->
            println ("Bill " ^ (toString y))
```

The registrar is defined as a restriction of a name `n` and the parallel composition `|` of two processes to interact with the two traders. The restriction scopes a name `n` to provide to the traders. Then each of the sub-processes unifies with a trader using the shared name, `nb` or `ns`, and confirms the identity of the other trader while providing the name `n`. After interaction with a trader the registrar does nothing, i.e. performs the unit program `()`.

```
rest n in
    cpc nb, sid, n -> ()
  | cpc ns, bid, n -> ()
```

Observe that while rather simple, such a registrar can easily be extended to support a multitude of traders.

Running these processes in parallel with appropriate restricted names to share between the traders and the registrar yields the following results.

```
~~ rest nb ns in
    (cpc s, bid, \j -> cpc nb, j, \m -> cpc m, b, \x ->
        println ("Bought " ^ (toString x)))
  | (cpc s, \j, sid -> cpc ns, j, \m -> cpc m, \y, c ->
        println ("Bill " ^ (toString y)))
  | (rest n in cpc nb, sid, n -> () | cpc ns, bid, n -> ());;
it: Unit
```

```
"Bill 123456 Buyer"
"Bought ABC 100"
~~
```

The share information `s` allows the buyer and seller to discover each other and swap identities `bid` and `sid`. The next two interactions involve the buyer and seller validating each other's identity and inputting the identifier to complete the transaction and print the results as before.

Although this solution satisfies the desire to validate that traders are legitimate, the freedom of matching allows for malicious processes to interfere. Specifically the negotiation and validation are vulnerable to the promiscuous process that offers some dummy information `a` (or `"anything"`) in exchange for capturing two other pieces of information. Adding such a promiscuous process to the scenario can interfere with the traders and registrar, preventing successful trade.

```
let a = "anything";;
rest nb ns in
    (cpc s, bid, \j -> cpc nb, j, \m -> cpc m, b, \x ->
        println ("Bought " ^ (toString x)))
  | (cpc s, \j, sid -> cpc ns, j, \m -> cpc m, \y, c ->
        println ("Bill " ^ (toString y)))
  | (rest n in cpc nb, sid, n -> () | cpc ns, bid, n -> ())
  | (cpc \z1, \z2, a -> println ("Stole: " ^ (toString z1)
        ^ " and " ^ (toString z2 )));;
```

Running this in Concurrent **bondi** *may* yield the following result

```
it: Unit
"Stole: ABC 100 $0.38 and ID0"
```

when the promiscuous process has interacted with the buyer to steal the desired stock information `ABC 100 $0.38` and the buyer's identity `ID0`. As the trading requires an open environment to discover partners, this behaviour is acceptable even if annoying.

More problematic is an alternative result of running the same program that yields

```
it: Unit
"Stole: ... and ID1"
```

where the promiscuous process has interacted with the buyer's attempt to validate the seller, stealing the restricted name `nb` (that displays as `...` as it is an internal value only) and the seller's identity `ID1`. Here the promiscuous process has stolen information that should have been kept private.

Note that as Concurrent **bondi**'s implementation of CPC is not deterministic both results are possible. Details on how this is implemented are in Sections A.3 & A.4.

Solution 3

Recall that the vulnerability exploited by the promiscuous process above can be repaired using protected names. Names are protected in Concurrent **bondi** by adding the tilde `~` in front of the name. One names have been protected, the processes can be run as before with the same results.

```
~~ rest nb ns in
    (cpc s, bid, \j -> cpc ~nb, j, \m -> cpc ~m, b, \x ->
        println ("Bought " ^ (toString x)))
  | (cpc s, \j, sid -> cpc ~ns, j, \m -> cpc ~m, \y, c ->
        println ("Bill " ^ (toString y)))
  | (rest n in cpc ~nb, ~sid, n -> () | cpc ~ns, ~bid, n -> ());;
it: Unit
"Bill 123456 Buyer"
"Bought ABC 100"
~~
```

This ensures that other processes can only interact with the traders during the discovery phase, which will not lead to a successful transaction. The

registrar will only interact with the traders (by Lemma 6.2.6) as all of the registrar's patterns have protected names known only to the registrar and a trader.

The advantage of protected names is that once discovery is completed then the rest of the interactions are performed in private (by Lemma 6.2.6). Of course this does not prevent the promiscuous process from interfering with the discovery phase, however without open discovery new trade partners cannot be found in the first place. The following output from Concurrent **bondi** shows an execution of the code with protected names.

```
~~ rest nb ns in
    (cpc s, bid, \j -> cpc ~nb, j, \m -> cpc ~m, b, \x ->
        println ("Bought " ^ (toString x)))
  | (cpc s, \j, sid -> cpc ~ns, j, \m -> cpc ~m, \y, c ->
        println ("Bill " ^ (toString y)))
  | (rest n in cpc ~nb, ~sid, n -> () | cpc ~ns, ~bid, n -> ())
  | (cpc \z1, \z2, a -> println ("Stole: " ^ (toString z1)
        ^ " and " ^ (toString z2 )));
it: Unit
"Bill 123456 Buyer"
"Bought ABC 100"
~~ %status;;
Process with ID 9:(Pair Pair (\z1) (\z2) a ->
    println
      ^ ("Stole: ")
        (^ (toString (z1))
          (^ (" and ") (toString (z2)))))
~~
```

Further, the `%status;;` commend displays the processes in the data-space, here showing that the promiscuous process has been left over without any other process to interact with.

The solution could be extended further: although the share information is never examined in detail, it could be partially matched against and

some information filled in during the discovery stage. This would allow discovery based on partial information, for example: specify a company code and price, but not the number of shares `Stock "ABC" \v (Price 0.38)`; or specify only the price and accept any company or number of shares `Stock \u \v (Price 0.38)`. The seller could also offer partial share information, although this may be a very risky business strategy! Observe that either trader can protect any component of the pattern if they wish to ensure that the other party exactly meets that criterion.

Another possibility is to allow for some checking of the integrity of the patterns being communicated. Given some standard language for the representation of data, such as the data types here or XML, this could be checked by the matching. For example, a valid bank account may be required to be constructed by `BankA` with the account number and account name. Thus, a pattern to input only valid bank accounts, binding the account number to `u`, the name to `v` could be `BankA \u \v`. Thus, any pattern that successfully matches must be identically constructed. Indeed, this could be developed further to account for XML and web services such as in `PiDuce` [BLM05].

Some of these techniques for unifying on parts of data structures or binding only the components of interest are used in the next section. Others are exploited in the second example later in the chapter.

Market

A market can be created that allows many buyers and sellers to interact. Here the stock can be created with a partially applied constructor

```
~~ let sc = Stock "ABC" 100;;
```

```
sc: Price -> Stock
~~
```

that has the same company information but no price. Observe that the type of `sc` is a function type `Price -> Stock` that when supplied with an argument of type `Price` reduces to something of type `Stock`.

Now create a market with five buyers (with identities `bid1` to `bid5`) and five sellers (with identities `sida` to `side`). To simplify the registrar definition the buyers shall nominate a channel to be used for the later transaction and inform the registrar. The sellers shall nominate the price (`pa` to `pe`) they are offering to sell shares at. Lastly, to clarify the code, a print function `print_trade` outputs the details of a transaction from the buyer, while the sellers remain silent.

A single buyer can now be defined by the following program.

```
cpc sc \p, bid1, \j ->
    rest m in
     cpc ~nb1, j, m ->
         cpc ~m, b, \x ->
             print_trade bid1 j sc p
```

The first pattern `sc \p, bid1, \j` unifies with stock and binds `p` to the price being asked for the stock while offering the buyer's identity `bid1` for the seller's, bound by `j`. Observe that the application of `sc` to `\p` unifies with a stock data structure, for example `Stock "ABC" 100 (Price 0.40)` and binding `p` to `Price 0.40`.

A single seller can be defined as follows

```
cpc sc pa, \j, sida ->
    cpc ~nsa, j, \m ->
        cpc ~m, \y, c -> ()
```

that offers stock at price `pa` and identity `sida` in exchange for the buyer's identity bound to `j`. The remainder is symmetric to the buyer process except that no output is generated upon completion of the trade.

The market can now be defined with the buyers and sellers all competing to find partners, as shown below.

```
~~ rest nb1 nb2 nb3 nb4 nb5 nsa nsb nsc nsd nse in
  (cpc sc \p, bid1, \j -> rest m in cpc ~nb1, j, m -> cpc ~m, b, \x
      -> (print_trade bid1 j sc p))
| (cpc sc pa, \j, sida -> cpc ~nsa, j, \m -> cpc ~m, \y, c -> ())
| (cpc sc \p, bid2, \j -> rest m in cpc ~nb2, j, m -> cpc ~m, b, \x
      -> (print_trade bid2 j sc p))
| (cpc sc pb, \j, sidb -> cpc ~nsb, j, \m -> cpc ~m, \y, c -> ())
| (cpc sc \p, bid3, \j -> rest m in cpc ~nb3, j, m -> cpc ~m, b, \x
      -> (print_trade bid3 j sc p))
| (cpc sc pc, \j, sidc -> cpc ~nsc, j, \m -> cpc ~m, \y, c -> ())
| (cpc sc \p, bid4, \j -> rest m in cpc ~nb4, j, m -> cpc ~m, b, \x
      -> (print_trade bid4 j sc p))
| (cpc sc pd, \j, sidd -> cpc ~nsd, j, \m -> cpc ~m, \y, c -> ())
| (cpc sc \p, bid5, \j -> rest m in cpc ~nb5, j, m -> cpc ~m, b, \x
      -> (print_trade bid5 j sc p))
| (cpc sc pe, \j, side -> cpc ~nse, j, \m -> cpc ~m, \y, c -> ())
(* Registrar process here, left out for brevity *);;
it: Unit
"Buyer ID14 bought "ABC" for $38. from ID101"
"Buyer ID13 bought "ABC" for $37. from ID104"
"Buyer ID12 bought "ABC" for $39. from ID103"
"Buyer ID15 bought "ABC" for $40. from ID102"
"Buyer ID11 bought "ABC" for $41. from ID105"
~~
```

The printing shows that all the buyers and sellers have found a partner to trade with, although some buyers have paid more than others.

Once a market is established where both buyers and sellers can trade, traders could be added that act as both buyer and seller to generate profit. Consider the scenario above with an additional trader that first buys some

shares and then sells them at an increased price. Here the generic trader has their own identity `tid` and corresponding communication channel to the registrar `nt`. Their succeed state prints out the profit they made on trading.

```
cpc sc \p, tid, \j ->
  rest m in
    cpc ~nt, j, m ->
      cpc ~m, bt, \x ->
        let np = Price (match p with
                        | Price p -> p * 1.1)
        in
        cpc sc np, \j, tid ->
          cpc ~nt, j, \m ->
            cpc ~m, \y, c ->
              (match p with
              | Price p ->
              println ( "Made $"
                       ^ (toString (p * 0.1 * 100.))
                       ^ " profit!")))
```

Observe that the first four lines behave the same as a buyer, purchasing stock and providing the trader's bank account information `bt`. A new price `np` is calculates as the old price plus 10% and then the trader behaves as a seller and sells the stock with the new price. After completing the sale the profit is then calculated and displayed.

Running the market again with the trader may yield the following output.

```
it: Unit
"Buyer ID11 bought "ABC" for $41. from ID105"
"Buyer ID14 bought "ABC" for $37. from ID104"
"Buyer ID12 bought "ABC" for $40. from ID102"
"Buyer ID15 bought "ABC" for $39. from ID103"
"Buyer ID13 bought "ABC" for $41.8 from ID999"
"Made $3.8 profit!"
```

Here the trader has made a small profit and one buyer has paid slightly more for their stock.

This illustrates the implementation of CPC in Concurrent **bondi** is able to represent CPC processes and captures the data space, unification, and non-determinism required. The next section develops a more complex example that also exploits the interplay with sequential Concurrent **bondi**.

10.2 Services

More general trading scenarios involve collaboration between heterogeneous systems in open environments to perform a trade. Consider the problem of shopping for a term deposit from a variety of different banks. These banks not only use different shapes of data, but also different programming paradigms. By combining sophisticated pattern-matching with pattern-unification a simple, distributed solution can be programmed in Concurrent **bondi**. (The following presentation highlights the full example code that is available here `http://www-staff.it.uts.edu.au/~tgwilson/concurrent_bondi/section_10.2.bon`.)

The example is of banks that offer a similar product, namely term deposits, and is developed through four stages. The first presents Concurrent **bondi** data structures using both ADTs and object-orientation. The second exploits dynamic patterns to traverse heterogeneous data structures within a single closed system. The third generalises to many distributed systems in an open environment by exploiting pattern-unification. The last shifts from finding information to completing a transaction in this setting.

Data Structures

Concurrent **bondi**'s support for multiple programming paradigms is illustrated by developing two banks: the NSW banks that represents its data using ADTs; and the Vic banks that uses object-oriented classes.

Bank accounts at NSW bank of type `Account` are of the form `Acct n x` where `n` is the name of the account (a string) and `x` is the balance of the account (a float):

```
datatype Account = Acct of String and Float
```

in the familiar functional programming style. Term deposits are declared similarly, with the bank name, product name, minimum deposit, rate, period and government guarantee.

```
datatype TermDepositADT =
   TDADT of String and String and Int and Float and Int and Bool
```

The NSW bank is represented by `Bank` with lists of accounts and term deposits.

```
datatype BankADT = BankADT of String and List Account
                        and List TermDepositADT
with toString += | BankADT n _ _ -> n
```

The NSW bank containing two customer accounts and two term deposits is given below.

```
let acct1 = Acct "John Citizen" 2222.00;;
let acct2 = Acct "Jane Doe" 2736.30;;
let tdnsw1 = TDADT "NSW" "Standard TD" 1000 4.7 12 True;;
let tdnsw2 = TDADT "NSW" "Short and cheap TD" 500 3.3 1 False;;
let nsw = BankADT "NSW" [acct1,acct2] [tdnsw1,tdnsw2];;
```

The Vic bank is implemented using mostly object-oriented classes, which are modelled upon those of Java [GJS05]. The term deposits are defined below, with a minimum deposit, rate and period.

```
class TermDepositOO {
minDep: Int;                (* Minimum deposit *)
rate: Float;                (* Rate *)
period: Int;                (* Period *)
(* Get and set methods. *)
with toString += | (x:TermDepositOO) -> "" }
```

Observe that in addition to using a different format, the Vic bank does not store the bank name within its term deposits. Also like when declaring ADTs, the generic `toString` function can be augmented to alter the printing of classes; here printing nothing.

The Vic bank is also represented by a class defined below.

```
class BankOO {
name: String;               (* Bank name *)
accts: List Account;        (* List of accounts, same ADT as NSW *)
tds: List TermDepositOO;    (* List of term deposit classes. *)
(* Get and set methods. *)
with toString += | (x:BankOO) -> x.getName() }
```

Here the extension to the `toString` function matches the pattern `(x:BankOO)` to bind `x` to an object of type `BankOO` and then uses the method `getName()` to display the bank name.

The Vic bank is created as follows.

```
let tdvic = new TermDepositOO;;
tdvic.setMinDep(2000);
tdvic.setRate(4.6);
tdvic.setPeriod(3);;
let vic = new BankOO ;;
vic.setName("Vic");
vic.setAccts([acct2]);
vic.setTDs([tdvic]);;
```

Dynamic Patterns

The relevant attributes of term deposits are chosen to be the name of the bank, the minimum amount, rate of return, and period. Routine techniques are used to display this information.

```
let display b m r p = println ("Term deposit from " ^ b ^
    " with minimum $" ^ (toString m) ^ ", rate " ^ (toString r) ^
    "%, and period " ^ (toString p) ^ " months.")
```

Harder is to recognise a term deposit as such within a larger, arbitrary data structure, and to extract the relevant attributes. As the pattern (or shape) for a term deposit varies between banks this needs to be given by a *shape parameter* `shape` to the function.

```
let findTDs f = fun
  (shape: lin (String -> Int -> Float -> Int ->  a)) ->
    iter (| {b,m,r,p} shape b m r p -> f b m r p
          | _ -> ())
```

The first argument `f` of `findTDs` is some program to accept the desired attributes of any term deposits found, for example `display`. The second argument `shape` is used to create the pattern `shape b m r p` which will match against term deposits. Note that the pattern here is defined in pure pattern calculus style and has a list of binding symbols `{b,m,r,p}` that also appear as arguments to the function `shape`. Here the program `shape b m r p` must be evaluated before matching can occur, that is the pattern is *dynamic*. Pattern parameters such as `shape` must be *linear*, to avoid duplication or elimination of pattern binders, as indicated by the keyword `lin` applied to its type. The `iter` function uses path polymorphism (as introduced in Chapter 5 and in the literature [GWHJ07, Jay09]) to traverse any data structure applying the anonymous function that matches term deposits and applies `f`.

Finding and displaying term deposits is now as simple as providing the shape and bank as arguments to `findTDs display`. Create an ADT for banks that contains the shape and data.

```
datatype Bank = Bank of (lin (String -> Int -> Float -> Int -> a))
                        and b
```

Note that the types for the result of the pattern and for the bank data are existential here. That is; they do not appear on the left hand side of the data type declaration and so are not universally quantified. Now

```
let allTDs = iter (| Bank shape bank -> findTDs display shape bank
                   | _ -> ())
```

will display all term deposits found within any bank that is in turn within any data structure containing, say, businesses. Observe that all this has been written before any actual shapes have been defined.

Now define the shape for term deposits in the NSW bank as follows.

```
lin shapeNSW b m r p = TDADT b _ m r p _
```

Here `lin` indicates that a linear term is being declared, and the wildcards `_` are used to ignore the information that is not required. The arguments to the shape are the binding symbols that will be bound in the matching process. That is, `shapeNSW` is a function that accepts binding symbols and returns a pattern containing each of those binding symbols exactly once.

The shape for the Vic bank uses a helper function and fills in the missing bank name with an explicit value.

```
let getTDVicData (x:TermDepositOO[a]) =
    ("Vic",x.getMinDep(),x.getRate(),x.getPeriod());;
lin shapeVic b m r p = view(getTDVicData,(b,m,r,p))
                       as (_:TermDepositOO);;
```

The pattern `view(getTDVicData,(b,m,r,p)) as (_:TermDepositOO)` has a two separate pattern-matching stages when evaluated. The first ensures the argument is any object of the class `TermDepositOO`. If this succeeds, then the second matches `(b,m,r,p)` against the result of applying `getTDVicData` to the argument (the term deposit object).

Now combine the NSW and Vic banks into a single data structure

```
let banks = [Bank shapeNSW nsw, Bank shapeVic vic]
```

taking advantage of the existential types to remain type safe. Then evaluating `allTDs banks` yields

```
~~ allTDs banks;;
it: Unit
"Term deposit from NSW with minimum $1000,
      rate 4.7%, and period 12 months."
"Term deposit from NSW with minimum $500,
      rate 3.3%, and period 1 months."
"Term deposit from Vic with minimum $2000,
      rate 4.6%, and period 3 months."
~~
```

showing that the function finds and displays all the term deposits as desired.

Thus a simple solution is able to find information based upon local shapes within arbitrary data structures. However, this approach is limited to a single system and relies upon the function having access to all the bank's data without limitations.

Pattern unification

In the concurrent setting the main challenge is for the bank and customer to discover each other and communicate. Discovering each other requires

that they have some small amount of shared data, namely a parameter that describes term deposits. Here `tdDesc` is a constant pattern to describe shared term deposit data that is understood by both parties. For the rest, each party is free to create its own processes.

For example, the NSW bank process for term deposits will involve a pattern `descNSW` that provides a bank description, and a function `security` that sandboxes the query, as well as the parameters `shapeNSW` and `nsw` employed before. These are combined to form a process.

```
cpc descNSW, ~tdDesc, \q -> security q shapeNSW nsw
```

That is: a process that shares the bank's description `descNSW`; requires the other process be interested in term deposits `~tdDesc`; and binds `q` to a query for the bank's data. Observe that the query `q` is passed through the bank's security and then allowed access to the bank's data within the bank, thus keeping the bank's internals hidden from other processes.

Define a similar process for the Vic bank term deposit service.

```
cpc descVic, ~tdDesc, \q -> security q shapeVic vic
```

To interact with these services the customer must create a process that will unify with a bank's term deposit pattern. This requires accepting the bank's description, being interested in term deposits, and providing a query. Such a query consists of a function similar to `allTDs` that adapts to the shape provided by the bank and then acts on an arbitrary data structure. Of course the modified query communicates its results back to the customer rather than printing the information out on the bank's system. Such a query could be

```
let query session =
    findTDs (fun b m r p -> cpc ~session, b, m, r, p, -> ())
```

that executes the familiar `findTDs` query on the banks data, communicates this back to the customer along `session` and terminates. Note how the higher-order function `findTDs` has been applied to a function that returns a process. The results are displayed using the process returned by the function

```
let displayFromSession session =
    !(cpc ~session, \b, \m, \r, \p -> display b m r p)
```

by replicating `!` a process to accept any number of results along a private channel `session` from the bank and then print them. Then

```
! rest session in
  cpc \bank, tdDesc, (query session) -> (
    println ("Querying " ^ bank);
    displayFromSession session)
```

uses `rest session` to restrict access to the channel `session`, and then replicates the whole, so as to be able to interact with many banks. Summarising, when a bank is discovered by unification the service reports the bank that accepted the query and creates another process that collects all responses from that bank to display them.

Now running this process in parallel with the services from the NSW and Vic banks yields the following.

```
"Querying Vic Bank"
"Querying NSW Bank"
"Term deposit from NSW with minimum $500,
      rate 3.3%, and period 1 months."
"Term deposit from Vic with minimum $2000,
      rate 4.6%, and period 3 months."
"Term deposit from NSW with minimum $1000,
      rate 4.7%, and period 12 months."
```

Observe that the queries were computed in parallel and the results displayed as they arrived with no relation to the structure of the banks' data.

Purchasing

While these steps demonstrate the components, the ultimate goal is to write a program that can find a suitable term deposit from a potentially unknown supplier and make a purchase. With all the ingredients already available, a little modification is all that is required. The problem of discovering services for term deposits is handled with pattern unification. The query sent to the bank can be modified to only find term deposits that meet specific criteria, for example a rate greater than 4.5%. The last ingredient is a way for the purchase to be made, taking a name for the account and the amount to be deposited. This can all be combined into a query.

```
let query session purchase =
findTDs (fun b m r p ->
  if r > 4.5
  then cpc ~session b (\n, \a) ->
    purchase n a
  else ())
```

Here `purchase` is a program to purchase a term deposit provided by the bank the query is run on. Observe that when an appropriate term deposit is found the `session` is used to obtain the name `n` and amount `a` from the customer to complete the purchase.

The purchase code can be defined for the banks by the following functions.

```
let purchaseNSW n a =
  nsw = (| BankADT b acs tds ->
    BankADT b (Cons (Acct n a) acs) tds) !nsw;;
let purchaseVic n a =
  vic.setAccts (Cons (Acct n a) (vic.getAccts ()));;
```

The NSW bank is now stored in a reference to allow for stateful updating. The purchase programs for both banks add a new account with the name and amount provided.

The services for the banks can now be defined with

```
let descNSW = "NSW Bank" in
cpc descNSW, ~tdDesc, \q -> security q purchaseNSW shapeNSW nsw;;
let descVic = "Vic Bank" in
cpc descVic, ~tdDesc, \q -> security q purchaseVic shapeVic vic;;
```

that behaves as before with the addition of providing their own purchasing code to the query.

All that remains is to define the program for the client. The `name` and `amount` for the prospective term deposit are declared, with the body of the code as before. The main difference is that instead of collecting results, the first successful result is used to purchase a term deposit.

```
let name = "Carl Smith" in
let amount = 10000.0 in
begin
rest session in
  (!cpc \bank, tdDesc, (query session) ->
        println ("Querying " ^ bank))
|  (cpc ~session \bank (name, amount) ->
        println ("Deposited with " ^ bank))
end
```

The result of running the programs is as follows.

```
~~ let name = "Carl Smith" in
let amount = 10000.0 in
begin
rest session in
  (!cpc \bank, tdDesc, (query session) ->
        println ("Querying " ^ bank))
|  (cpc ~session \bank (name amount) ->
        println ("Deposited with " ^ bank))
end;sleep 1.0;;
it: Unit
"Querying Vic Bank"
"Querying NSW Bank"
"Deposited with NSW"
~~
```

Here the purchase has been made with the NSW bank, this can be confirmed by showing all the accounts now at the NSW bank.

```
~~ (* Show NSW accounts *)
(| BankADT _ acs _ -> acs)!nsw;;
it: List Account
it = [Acct "Carl Smith" 10000.,
      Acct "John Citizen" 2222.,
      Acct "Jane Doe" 2736.3]
~~
```

Another run may result in the purchase being made with the Vic bank as shown below.

```
~~ let name = "Carl Smith" in
let amount = 10000.0 in
begin
rest session in
  (!cpc \bank, tdDesc, (query session) ->
        println ("Querying " ^ bank))
|  (cpc ~session \bank (name amount) ->
        println ("Deposited with " ^ bank))
end;sleep 1.0;;
it: Unit
"Querying NSW Bank"
"Querying Vic Bank"
"Deposited with Vic"
~~
```

The accounts for the Vic bank can be viewed to confirm that the purchase has been successful by.

```
~~ (* Show Vic accounts *)
vic.getAccts();;
it: List Account
it = [Acct "Carl Smith" 10000.,
      Acct "Jane Doe" 2736.3]
~~
```

This solution to the problem of finding and purchasing term deposits in a collaborative environment shows how patterns can be exploited in both sequential and concurrent computation. The sequential dynamic patterns support queries that adapt to the shape of heterogeneous data and programming styles. Further, pattern-unification supports decentralised discovery of compatible processes and exchange of information, all within an atomic interaction.

Chapter 11

Conclusions

This chapter draws conclusions and discusses future work by first reconsidering the computation square and then the results of this dissertation in finer detail.

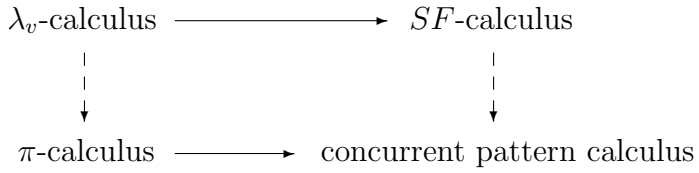
This work has shown that extensional, sequential computation can be generalised by adding intensionality, that respects the usual reduction properties, to be able to represent symbolic computation not representable in merely extensional models. The expressive power of extensionality and intensionality can be characterised by structure completeness, i.e. the ability to support arbirary pattern-matching functions whose patterns are normal forms. This is demonstrated by SF-calculus that is structure complete, and thus able to represent all Turing computable functions on SF-combinators in normal form, for example equality of normal forms.

Shifting to the concurrent setting, process calculi are expected to subsume sequential computation. A popular example is the π-calculus that has encodings of λ-calculus with a specific reduction strategy. The combination of

encoding λ-calculus and the independent parallel reduction of process calculi, yields an extensional, concurrent model of computation.

As process calculi are expected to subsume sequential computation, a natural development is to consider an intensional concurrent computation model. By basing interaction on symmetric pattern-unification, CPC is able to generalise both intensional sequential computation and extensional concurrent computation. Further, as all interactions are inherently between (at least) two parties, CPC increases the symmetry of interaction by shifting away from asymmetric input/output or pattern-matching, to symmetric pattern-unification.

The original *concrete* computation square relates λ_v-calculus, π-calculus, SF-calculus and CPC as follows



where the left side is extensional, the right side intensional, the top side sequential, and the bottom side concurrent. The horizontal arrows are homomorphisms that map application/parallel composition to itself and preserve reduction. The vertical arrows are parallel encodings (Definition 3.1.1) that map application to parallel composition (with some extra machinery) and preserve reduction. Significantly all the arrows are translations $[\![\cdot]\!]$ such that a source term S translates to a target term $T = [\![S]\!]$ and when $S \longrightarrow S'$ then $T \Longrightarrow T'$ for some T' equivalent to $[\![S']\!]$ in the target language. Further, there are no reverse arrows as each arrow signifies an increase in expressive

power. Such a square raises many questions, not all of which are easy to answer.

The square does not commute due to both the choice of translations and the choice of calculi. This stems from the π-calculus' inability to examine the structure of a name and so the parallel encoding from λ_v-calculus translates terms into processes that capture behaviour. Further, these limitations also prevent easily capturing the reductions of λ-calculus and so requiring the choice of a reduction strategy. By contrast the parallel encoding of SF-calculus into CPC is able to exploit the structure of translated terms to directly perform reductions. This does not require fixing a reduction strategy as the reducing process can operate directly upon the translated syntax and so support any reduction rules. Thus, travelling from λ-calculus to CPC via π-calculus yields a behavioural representation of the terms, while travelling via SF-calculus yields separate syntax and reduction rules. Also the homomorphism from π-calculus into CPC is unlike the other arrows in that it is *faithful*; i.e. every reduction of π-calculus yields a single reduction in CPC. While every other arrow preserves reduction to some notion of behavioural equality, the homomorphism from π-calculus into CPC is a direct equivalence of the translations.

The diagonals of the square have not been detailed in this dissertation. From λ_v-calculus to CPC there are already two diagonals via either π-calculus or SF-calculus. A third diagonal via SK-calculus and then directly to CPC has been shown to be a straightforward adaption of the parallel encoding of SF-calculus into CPC, i.e. a parallel encoding of SK-calculus into CPC. The existence of either diagonals between SF-calculus and π-calculus have not

been explored. The lack of a reduction preserving encoding from π-calculus into SF-calculus should be routine. However, the arrow from SF-calculus to π-calculus has yet to be formalised. On the surface the inability to represent structure within a name in π-calculus suggests that such an arrow does not exist, but to prove this is another matter.

Conjecture 1. *There exists no parallel encoding from SF-calculus into π-calculus.*

There are many ways to produce a concrete computation square of calculi that have corners according to extensional, intensional, sequential and concurrent computation. The top left corner could be populated by λ_v-calculus or λ_l-calculus with minimal changes to the proofs. Alternatively, choosing λ-calculus or SK-calculus may also hold, although a parallel encoding into π-calculus may require some work. The bottom left corner is also open to some other calculi mentioned in this dissertation: monadic/polyadic synchronous/asynchronous π-calculus or Linda could replace π-calculus with no significant changes to the results. The top right corner could be populated by any of the structure complete combinatory logics, SF-calculus, SKF-calculus, $SF\mathcal{C}$-calculus, SFC-calculus or $\{S, F, \Omega\}$-calculus, without much effort. It may also be possible to take pure pattern calculus, or other pattern calculi [Jay04, JK06, GW07, JK09, Jay09], at the top right. Similarly, there will be other process calculi that can take the place of CPC at bottom right. (Some variations on CPC are discussed in Section 11.2.)

Many choices of calculi, and thus many concrete computation squares, can be routinely shown to have the same irreversible arrows. However, it is not clear if there are general results which cover the many possibilities

and formalise an *abstract* computation square of families of: extensional $\mathcal{E}$ or intensional $\mathcal{I}$; and sequential $\mathcal{S}$ or concurrent $\mathcal{C}$ calculi.

Another consideration is to generalise the relations between calculi in the manner that Gorla developed valid encodings for process calculi. Thus leading to the following question: is there a single notion of encoding that can be used for every arrow?

11.1 Intensional Sequential Computation

The ability to examine the internal structure of combinators within a combinatory logic is simple and powerful. Usually such examination arises in a more operational setting, such as when a Turing machine acts upon combinator syntax. In general, such encodings are so powerful that ad-hoc techniques are required to ensure that such operations respect combinator semantics.

By beginning with a syntactic characterisation of the factorable forms, factorisation can be guaranteed to respect the semantics, and so yield a symbolic computation. When factorisation and operator equality is combined with extensionality then the resulting calculus is structure complete, in that one is able to represent pattern-matching functions in which arbitrary normal forms are allowed as patterns. Examples include a generic test for structural equality of normal forms, and general forms of the queries popular in database

programming. In turn, structure complete calculi are characterised by their support for four combinators, namely S (for duplication), K (for elimination), F (for factorisation) and `eqatom` for separating irreducible operators.

11.2 Concurrent Pattern Calculus

By basing interaction on symmetric pattern-unification, combined with the usual process calculus primitives, CPC is able to support intensionality in a concurrent setting. Indeed, CPC's support for input, output and equality through pattern-unification allows discovery of compatible partners and exchange of information. This models the essence of trading in the information age, as illustrated by the example of traders who discover each other in the open and then close a deal in private.

Although CPC sits in the bottom right corner of the computation square, there are many variations on CPC that could fulfil the same rôle. The support for intensionality does not require the symmetry of pattern-unification, indeed a simpler calculus could limit interaction to the traditional input and output primitives while retaining intensionality. The development of a pattern-matching calculus can exploit the existing definitions for CPC. For example, reduction can be defined by

$$p \to P \mid q \to Q \quad \longmapsto \quad \sigma P \mid Q \qquad \text{where } \{p \| q\} = (\sigma, \{\}).$$

Here $p \to P$ is an "input" p and $q \to Q$ is an "output" q. Although "outputs" would still interact under this definition, modifying the calculus to prevent this is routine. Observe that such a modified calculus would still support all

the development on the boolean reduction system $\mathcal{B}$ and thus the encoding of SF-calculus. Further, it is straightforward to show that there is a homomorphism from such a calculus into CPC. The lack of a converse would follow from the separation results by symmetry for π-calculus, Linda, Spi calculus and fusion calculus. While pattern-matching alone captures intensionality in the style of structure completeness, the symmetry of interaction has been artificially limited.

Another variation on CPC would be to prevent variable and protected names unifying with each other. That is, variable names only unify with variable names, and protected names only unify with protected names. Although this would simplify some results, it would not yield greater expressive power and would limit the flexibility of discovery in the trading examples.

In addition to variations on the concepts motivating CPC, there are also further paths of development for CPC itself. Implementation in a programming language has been indirectly demonstrated through the applications in Chapter 10. Of course there are many design decisions in converting from calculus to programming language, some of which have been discussed in the notes on the development of Concurrent **bondi** in Appendix A. One of particular interest, even without implementing a programming language, being a type system for CPC. Some preliminary work on this for CPC appears in Section A.2, however a robust, rather than ad-hoc, approach that builds upon the lessons from sequential typing of pattern calculus [Jay09, Part II] could lead to many interesting results.

Lastly, the relation of CPC to pure pattern calculus remains. Although pure pattern calculus and pattern-matching are the motivation for CPC,

the relations are not yet clear. In particular, that pure pattern calculus is more expressive than λ-calculus is ongoing work. Further, there are some preliminary results which show a homomorphism from pure pattern calculus into SF-calculus, thus ensuring that there is (at least an indirect) relation between pure pattern calculus and CPC. A more direct relation is not simply a routine translation of the terms due to two complexities. First, the patterns of pure pattern calculus support reduction while the patterns of CPC do not. Although binding names to functions has been shown to be sufficient for π-calculus, it is not clear whether this still holds when intensionality is in play. Second, pure pattern calculus has a notion of match failure to detect when a pattern cannot be matched against an argument. However, in CPC pattern-unification is used to determine interaction, not structural application, thus detecting match failure is non-trivial. Following from these observations and ongoing work there is likely an encoding from pure pattern calculus into CPC, however the manner and directness of such an encoding are not yet clear.

11.3 Completing the Square

The parallel encoding from SF-calculus into CPC exploits intensionality to match both the operators and structure in translated combinators. By separating the behaviour from the translated combinators CPC is able to encode SF-calculus without having to fix a reduction strategy, as is required by the parallel encodings of Milner. Thus, a straightforward adaptation of these techniques can be used to provide a parallel encoding of SK-calculus (and

thus λ-calculus) into CPC.

The combination of matching arbitrary numbers of symbols and arbitrary structure through CPC's pattern-unification suggests support for general reduction systems. This is clearest in the encodings that translate the source terms into a pattern and exploit a reducing process that operates upon the encoded pattern to perform reductions. The development of encodings of system $\mathcal{B}$ and SF-calculus illustrate how this can be done by examining the syntax and the structure of the reduction rules. This style of translation is similar to encoding the terms onto the tape of a Turing machine, as one process, and the reduction rules into the state, another process.

Conjecture 2. *There is a parallel encoding of any reduction system into CPC.*

Indeed, an elegant and faithful encoding of a Turing machine into CPC is routine by exploiting these techniques and taking a little care in representing the boundless nature of the tape. Combined with the above conjecture this yields two ways to encode a reduction system into CPC, either directly or via a Turing machine. However, a parallel encoding supports modularity of the translation and respects the semantics in a manner that ad-hoc Turing machine encodings do not.

Shifting to the bottom of the computation square, there is a homomorphism from π-calculus into CPC that does not rely upon any behavioural theory. That is, CPC is able to validly encode π-calculus relying only upon structural congruence. The lack of a valid encoding of CPC into π-calculus follows from both CPC's ability to match any number of names in an interaction and CPC's symmetry. That π-calculus cannot preserve the reductional

behaviour of CPC ensures there can be no homomorphism from CPC into π-calculus.

11.4 Behavioural Theory

The behavioural theory of CPC is unusual due to the peculiarities of pattern-unification. In particular, the manner in which variable names may be tested for equality, and the contractive nature of binding names. The first appears in the definition of barbs that are defined with a set of names. Both appear in the comparability relation that leads to an ordering on patterns. This ordering is asymmetric with respect to patterns, yet used to define the (inherently symmetric) bisimulation relation. Although the development of the behavioural theory for CPC follows the standard approach, these peculiarities lead to some unusual constructions.

The definition of barbs for CPC includes parametrisation by a set of names that include both those that *may* be tested for equality and those that *must*. Interestingly, a set is sufficient to represent all the names; that is, the order of appearance in the pattern, number of occurrences, and form (variable or protected) is immaterial to the barb definition. This is in contrast with the barbs of, say Linda, where both order and number of occurrences are part of the definition (along with the size of the tuple).

The flexibility of names that may be tested for equality and the ability to bind arbitrarily complex patterns to a single name leads to some patterns being more general than others. This is captured by the comparability relation that also orders related patterns. The ability to order patterns provides po-

tential for optimisations, in particular for implementing more efficient unification algorithms. Another direction in which such orderings appears is in the development of object-oriented systems. The ordering on patterns provides the basis for developing inheritance hierarchies and thus could elegantly capture object-oriented communication in CPC, indeed structure bases object-orientation has been developed before in pattern calculi [Jay09, Chapters. 6, 11 & 18].

The mixture of the asymmetric comparability of patterns with the symmetric bisimilarity of process seems counter-intuitive. While there are other process calculi whose bisimulation does not require an exact response, ordering of responses based upon structure appears to be new.

11.5 Relations to Other Process Calculi

Although relating CPC to π-calculus completes the computation square, there are other process calculi whose relation to CPC is of particular interest. By exploiting valid encodings suitable translations and separation results are easy to formalise.

The encoding of Linda into CPC is straightforward once a suitable translation of Linda templates and data has been found. That this encoding is valid, and indeed a homomorphism, is simple to prove and, like for π-calculus, does not rely upon behavioural equivalence for CPC. Following this result, all of the process calculi captured by Gorla's work on valid encodings should have homomorphisms into CPC. The lack of a valid encoding into CPC by exploiting symmetry is a trivial reuse of the separation result for π-calculus.

The lack of a valid encoding by intensionality indicates that even a CPC style pattern-matching (not unification) process calculus would be more expressive than Linda.

Although developed to consider security, the Spi calculus terms support intensionality much in the way that SF-calculus does. That is, Spi calculus supports arbitrary structures within terms and intensional reductions upon them. Indeed, the combination of arbitrary structure, (Spi) match, and pair splitting appears to be sufficient to encode intensionality. However, to support a parallel encoding either additional tagging or a reduction strategy (similar to the parallel encodings from λ-calculus to π-calculus) would be required.

Conjecture 3. *There is a parallel encoding of SF-calculus into Spi calculus.*

This support for intensionality, and likely support for SF-calculus, make the relation to CPC of particular interest. The valid encoding of Spi calculus into CPC is routine, if long, due to the rich syntax of Spi calculus. Interestingly, the behavioural theory for CPC is required to dispose of processes that cannot interact or reduce that may be a side effect of reducing a translated Spi calculus process. The lack of a valid encoding of CPC into Spi is ensured by CPC's symmetry. Interestingly, there is likely an adaptation of the proof by intensionality (for separating CPC from Linda) that would work for Spi calculus as well. The crucial point is that any (attempt to produce a valid) encoding into Spi calculus would yield reductions when there is none in CPC, and thus lead to violating either divergence reflection or success sensitiveness. Thus, even without symmetry, CPC's linking of intensionality with communication has expressive power than cannot be captured by Spi

calculus.

Since the separation results for CPC and the other process calculi so far can all be proved by exploiting CPC's symmetry, the relation to fusion calculus is of particular interest. This interest is piqued further by observation that fusion calculus was partially motivated by a desire to model sequential computation and in particular *strong lazy* λ-calculus. The lack of a valid encoding of fusion calculus into CPC is due to the peculiarities of name fusion, as there are fusion calculus processes that would require a ternary interaction in CPC. However, that there is also no valid encoding of CPC into fusion calculus is ensured by both the ability of CPC to test any number of names for equality in a single interaction, and by CPC's symmetry. Thus, fusion calculus and CPC are unrelated and appear to be focused in different directions.

11.6 Applications

Programming with Concurrent **bondi** is able to exploit the expressiveness of CPC to generalise computation into a concurrent setting. In particular, pattern-unification supports discovery of compatible processes and information exchange while also allowing private channels of communication. This combination of behaviours provides the essence of trading in the information age.

Exploiting CPC alone, traders can create patterns for desired deals and advertise this to discover partners. Credentials can be exchanged and then used to verify each other before closing the deal, via another exchange of

information, in private. Since interactions in CPC are atomic, discovery and trade are immediate without any possibility of partial transactions failing mid-way through, and with each trader on equal standing.

Combining CPC's expressive power with Concurrent **bondi**'s dynamic patterns and support for heterogeneous programming styles allows for solutions to distributed programming in open collaborative environments. Here agents can discover collaborators and communicate programs that work across agents and structures to find information. These can be exploited further to then use that information to trade using Concurrent **bondi** as a common environment.

To conclude, intensionality yields greater expressive power in both sequential and concurrent computation. Even greater expressive power is achieved by generalising from pattern-matching to pattern-unification and exploiting the inherent symmetry of process calculi. In addition to supporting these concepts, CPC provides a natural language to express trade where pattern-unification can be exploited to discover trade partners and exchange information. Capturing all of these in a programming language, Concurrent **bondi** is able to support open, collaborative, distributed and heterogeneous programs.

Appendix A

Implementation

This appendix discusses an alternative development for a process calculus; implementation in a programming language [PT97, NFP98, PMR99, cpp10, JoC11]. The focus here is upon design decisions, algorithms and machinery required to augment an existing programming language to support CPC.

The augmentations to support CPC have been made to the **bondi** programming language and interpreter [bon11]. This decision is due to **bondi** being built upon a foundation of pattern-matching as an implementation of the concepts captured by pure pattern calculus [JK09, Jay09, bon11]. This allows **bondi** to support many programming styles, including: functional, imperative, relation, and object-orient, with a small pattern-matching core [GWHJ07, Jay09, bon11, GWJ11]. As **bondi** is already built upon pattern-matching and pure pattern calculus, it is natural to augment **bondi** to support the pattern-unification of CPC to yield Concurrent **bondi** [Con11].

Although **bondi** is a good starting point, there are several design decisions to consider when supporting both existing **bondi** and CPC. Pure pat-

tern calculus style data structures are built with application and constructors, ideally CPC compound patterns should unify with applicative data structures and support constructors. As free names (variables) in **bondi** could appear in CPC patterns there needs to be some reasonable way to support reduction and imperative features in CPC patterns. The **bondi** language is strongly typed and so CPC in Concurrent **bondi** will require types.

Once the design decisions have been made, the augmented Concurrent **bondi** requires algorithms to support CPC rules that account for both the CPC specification and the design decisions. Of particular interest are: the two algorithms to do pattern-unification and process interaction; and the implementation of the process space to support both concurrency and non-determinism.

The rest of the appendix is structured as follows. Section A.1 discusses language design decisions to support CPC in Concurrent **bondi**. Section A.2 develops types for CPC that are compatible with the **bondi** type system and type inference. Section A.3 presents the algorithms for pattern-unification and process interaction. Section A.4 considers implementing the process space to support concurrency.

A.1 Syntax

The **bondi** programming language syntax is styled after Objective Caml (OCaml) [Cam11] for functional and imperative programming, with Java [GJS05] style syntax for object-orientated programming. As the majority of **bondi** and Concurrent **bondi** syntax is very similar to OCaml detail shall

be elided here except to highlight differences and design decisions. Examples of Concurrent **bondi** programs exploiting functional, imperative and object-oriented programming are presented in Chapter 10 as well as in the literature [GWHJ07, GW07, Jay09, GW10, GWJ11].

Patterns

One particular feature of **bondi** is the implementation of data structures in the style of pure pattern calculus. That is; compound data structures are terms headed by a constructor and built by application. Thus, the list containing the numbers 1 to 3 is represented by

```
[1,2,3] = Cons 1 (Cons 2 (Cons 3 Nil))
```

using the typical list constructors `Cons` and `Nil`. As pure pattern calculus, and **bondi**, allow patterns that match any compound, the same is preferred in implementing compound patterns for CPC in Concurrent **bondi**.

Thus the first step in developing the Concurrent **bondi** language is to define the syntax for CPC patterns. The translations from CPC to Concurrent **bondi** are given by

Variable name	$[\![x]\!]$	=	`x`
Protected name	$[\![\ulcorner x \urcorner]\!]$	=	`~x`
Binding name	$[\![\lambda x]\!]$	=	`\x`
Compound	$[\![p \bullet q]\!]$	=	$[\![p]\!]\ [\![q]\!]$.

The variable names are translated directly to variables in Concurrent **bondi**. The protected names are prefixed with a tilde `~` to denote their protected status. The binding names are prefixed by a backslash, chosen for similarity

to the λ.Compounds are translated component wise with the bullet replaced by application to align with other Concurrent **bondi** (and pure pattern calculus) compounds.

The choice to indicate binding names with backslashes is to keep the syntax as close to CPC as possible. This is distinct from the dynamic patterns of pure pattern calculus that have a sequence of binding names at the beginning of the pattern to denote the status of symbols (names). As CPC patterns cannot lose their binding symbols through reductions there is no need to track this information separately and so a separate sequence for binding symbols is not required.

Although this translation of patterns is adequate to support CPC theory, the interplay with the other aspects of Concurrent **bondi** is much more elegant if constructors and primitive datum (integers, booleans, strings, etc.) are supported in CPC patterns. So in addition to the translation above, the Concurrent **bondi** class of patterns also supports data structures and datum, with either variable or protected status. Thus, the pattern to bind the tail of a list headed by `1` to `t` can be written

```
~Cons ~1 \t
```

using a protected constructor `~Cons` and a protected datum `~1`.

Combining all these, with brackets as required, allows CPC patterns to support arbitrarily complex structures while seamlessly supporting Concurrent **bondi** datum in patterns.

Processes

The Concurrent **bondi** language extensions for processes are given by

Parallel composition	$[\![P \mid Q]\!]$	=	$[\![P]\!]$ `|` $[\![Q]\!]$
Replication	$[\![!P]\!]$	=	`!`$[\![P]\!]$
Restriction	$[\![(\nu x)P]\!]$	=	`rest x in` $[\![P]\!]$
Case	$[\![p \rightarrow P]\!]$	=	`cpc` $[\![p]\!]$ `->` $[\![P]\!]$.

The parallel composition is translated directly to the translation of processes separated by the bar `|`. The replication is also straightforward with the bang symbol `!` in front of the translated process. Restrictions use the keyword `rest` and are similar to `let` declarations on the understanding that values for the names are not required. Cases are declared similar to anonymous functions with the keyword `cpc` beginning the declaration, followed by the translation of the pattern, the arrow symbol `->`, and then the (translation of the) body.

Like let declarations, the restriction can be used to declare any number of names with a single use of the `rest` keyword. For example the process

$$(\nu x)(\nu y)(\nu z)P$$

can be represented by

`rest x y z in` $[\![P]\!]$

rather than `rest x in rest y in rest z in` $[\![P]\!]$.

As the null process can be represented by any Concurrent **bondi** program

```
type p_term =
  | Papply of p_term * p_term
...
(*> CPC *)
  | Pname of name_form * identifier
  | Pcname of name_form * identifier
  | Pdname of name_form * p_term
  | Pparr of p_term * p_term
  | Prest of identifier * p_term
  | Prepl of p_term
  | Ppcase of p_term * p_term
and name_form = Variable | Protected | Binding
(*< CPC *)
```

Figure A.1: CPC parsed term constructors

that contains no CPC cases. For simplicity the unit program `()` shall be used to denote the null process when required.

Parsing

This section discusses the parsing of Concurrent **bondi** syntax, using Lex and Yacc [LMB92], into algebraic data types (ADTs). Some detail is presented where useful to understanding the Concurrent **bondi** interpreter and as these ADTs appear later in the code for typing in Section A.2.

To support CPC several new constructors were added for the parsed data types `p_term` shown in Figure A.1. Note that `(*> CPC *)` denotes the beginning of code blocks modified for Concurrent **bondi** and `(*< CPC *)` denotes the end. The `Pname` directly corresponds to the names of CPC with a name form (`Variable`, `Protected` or `Binding`) and an identifier (a string). The `Pcname` supports constructors being used in patterns, e.g. `Cons` or `Nil` for

"cpc"	CPC
"in"	IN
"rest"	REST
"status"	DIRECTIVE_status
"Un"	UN
"\\"	CPCBIND
"~"	CPCPRO
"("	LPAREN
")"	RPAREN
"|"	BAR
"!"	BANG
"->"	RARROW
","	COMMA

Figure A.2: Glossary of lexer tokens

lists, and can be denoted as variable or protected. Datum are supported by `Pdname` which can be denoted as variable or protected. Compounds are represented using the application `Papply` of the components of the pattern. Note that this is the same application as use for pure pattern calculus terms to apply functions and build data structures. Parallel composition is represented by `Pparr` of the two parsed terms (processes). Restriction `Prest` has the name (identifier) being restricted and the process as a parsed term. The replication `Prepl` is simply the process being replicated. The CPC case `Ppcase` consists of the pattern and the body, both as parsed terms.

The code to parse CPC syntax is divided into five declarations; datum patterns, name patterns, compound patterns, name lists, and processes. The declarations make use of a number of tokens, rather than explain each in detail a glossary is provided in Figure A.2.

The support for datum in patterns is managed by a separate collection of parsing rules. These construct appropriate parsed term forms and are

defined as follows.

```
/* Datum as they can appear in CPC patterns, a subset of all
 * datum forms and not treated as general terms. */
datumPattern:
  | LPAREN RPAREN       { Pconstructor "Un" }
  | INTEGER             { p_datum (Int $1) }
  | FLOAT               { p_datum (Float $1) }
  | CHARACTER           { p_datum (Char $1) }
  | STRING              { p_datum (String $1) }
  | TRUE                { p_datum (Bool true) }
  | FALSE               { p_datum (Bool false)}
  | UN                  { Pconstructor "Un" }
```

Here `p_datum` is a helper function to convert datum into parsed terms.

Patterns that are atoms are parsed by the `namePattern` code that also identifies the form of the atom; variable, protected or binding. The direct translation of names (and variables from **bondi**) are lower case identifiers `L_IDENT` and can be of any form. Constructors (`U_IDENT`) and datum (`datumPattern`) can be either variable or protected.

```
/*  Name patterns are CPC patterns that are atomic, i.e. not
 * build from compounds. This includes names (variable,
 * protected and binding), and also constructors (variable
 * or protected) and datum (variable or protected). */
namePattern:
  | LPAREN namePattern RPAREN { $2 }
  | L_IDENT         { Pname (Variable, $1) }
  | CPCPRO  L_IDENT { Pname (Protected,$2) }
  | CPCBIND L_IDENT { Pname (Binding,  $2) }
  | U_IDENT         { Pcname (Variable,$1)}
  | CPCPRO U_IDENT  { Pcname (Protected,$2) }
  | datumPattern    { Pdname (Variable,$1) }
  | CPCPRO datumPattern { Pdname (Protected,$2) }
```

Compound patterns are parsed by the code declared by `compoundPattern`. For elegance, tuples can be declared directly in patterns and are denoted with

a comma. This is syntactic sugar for declaring nested instances of the `Pair` constructor as used throughout Concurrent **bondi** programs.

```
/* Compound CPC patterns are built with applications, this
 * approach ensures they are left associative for consistent
 * interplay with pattern calculus. */
compoundPattern:
  | namePattern
        { $1 }
  | LPAREN compoundPattern RPAREN
        { $2 }
  | compoundPattern LPAREN compoundPattern RPAREN
        { Papply ($1,$3) }
  | compoundPattern namePattern
        { Papply ($1,$2) }
  | compoundPattern COMMA LPAREN compoundPattern RPAREN
        { Papply (Papply (Pcname(Variable,"Pair"),$1),$4) }
  | compoundPattern COMMA namePattern
        { Papply (Papply (Pcname(Variable,"Pair"),$1),$3) }
```

Processes are parsed directly, with the only non-trivial example being the restriction where `nameList` parses a list of names and `multirest` folds over this list to produce the parsed term.

```
/* Process forms: bracketed processes, parallel composition,
 * replication, restriction and CPC case. */
process:
  | LPAREN process RPAREN               { $2 }
  | process BAR process                 { Pparr($1,$3) }
  | BANG process                        { Prepl $2 }
  | REST nameList IN process            { multirest $2 $4 }
  | CPC compoundPattern RARROW pTerm    { Ppcase ($2,$4) }
```

A.2 Typing

Once syntax has been parsed in the Concurrent **bondi** interpreter, the parsed data syntax tree is transformed to inferred data. This transformation per-

forms type inference and type checking for Concurrent **bondi** programs. As **bondi** is strongly typed, it follows that Concurrent **bondi** must also be strongly typed. The rest of this section discusses types and type rules for CPC as implemented in Concurrent **bondi**, including some detail of the type inference implementation. As the type inference requires some delicacy and the **bondi** type system is not formalised anywhere, the approach here is to provide type derivation rules for CPC syntax and then highlight the type inference code in the Concurrent **bondi** interpreter. The implementation is consistent with the existing **bondi** type system and exploits existing type inference algorithms. Thus the typing for Concurrent **bondi** is consistent with **bondi**, however as there is no formalised type system no claims can be proved about the implementation.

Type Derivation

The type derivation rules utilise a type context Γ that is a mapping from variables (names) to types (meta-variables U, V, W, X, Y, Z). The domain and range of type contexts are as expected, with the free type variables, denoted $\mathsf{FTV}(\Gamma)$, being the union of the free type variables of all the types in the range of Γ.

The type derivation rules for patterns are given in Figure A.3. Variable names and protected names have their types given by the type context. Binding names must be a fresh type variable that does not appear in the context otherwise. The two rules for typing patterns relate to the interplay with sequential Concurrent **bondi** programs. The first, ensures that any communicable pattern must have a type consistent with application; that is

the left hand side must be a function type, and the right hand side must be an appropriate argument type. The second, is for when a pattern is not communicable, in which case the types of the components can be unrelated. (Here a fresh type is used for the pattern as a whole, although it is of no relevance to the rest of the type derivation.)

The rule that requires communicable compound patterns to have types that support application is due to the following Concurrent **bondi** program.

```
cpc x y -> () | cpc \z -> z
```

Observe that as the pattern `x y` could be bound to a single name `z` and then evaluated, the application of `x` to `y` must be type safe (and in this case yield `Unit`). This is only required for communicable patterns as compounds of protected or binding names cannot be bound to a single name. (Of course if they are applied elsewhere in the program then the typing infers appropriate types.)

Observe that due to Concurrent **bondi**'s support for tuples in patterns, an alternative translation of compound patterns that keeps the typing simple is

$$[\![p]\!] \bullet q \quad = \quad [\![p]\!] \, , [\![q]\!]$$

where the bullet is replaced by a comma. Although this prevents some interplay with other Concurrent **bondi** programs, this approach simplifies pure CPC programs.

The type derivation rules for processes are straightforward on the understanding that all processes have unit type, as detailed in Figure A.4. The parallel composition of processes has the unit type if both processes are of

$$\frac{}{\Gamma; x : X \vdash x : X} \qquad \frac{}{\Gamma; x : X \vdash \ulcorner x \urcorner : X}$$

$$\frac{}{\Gamma; x : X \vdash \lambda x : X} \quad X \notin \mathsf{FTV}(\Gamma)$$

$$\frac{\Gamma \vdash p : U \to V \quad \Gamma \vdash q : U}{\Gamma \vdash p \bullet q : V} \quad (p \bullet q) \text{ is communicable}$$

$$\frac{\Gamma \vdash p : U \quad \Gamma \vdash q : V}{\Gamma \vdash p \bullet q : X} \quad \begin{array}{l}(p \bullet q) \text{ is not communicable} \\ X \notin \mathsf{FTV}(\Gamma)\end{array}$$

Figure A.3: Type derivation rules for patterns

unit type. Replication of a process is the unit type when the process is the unit type. Restriction of a name in a process is of unit type if the process has unit type after the type context is extended to map the restricted name to a fresh type variable. The only complex rule is for cases. Here the type context Γ is extended by Δ that maps all the binding names of the pattern to fresh type variables. If the extended context $\Gamma; \Delta$ shows that the pattern p has a type, and the body t (a possibly any Concurrent **bondi** program) has type unit, then the case $p \to t$ has unit type.

Type Inference

The Concurrent **bondi** interpreter has a type inference algorithm that attempts to find a type while transforming a parsed term to an inferred term. This is initiated by the `infer` function that accepts the parsed term and an expected type. When successful the result is a tuple of: the inferred term,

$$\frac{\Gamma \vdash P : \texttt{Unit} \quad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash (P \mid Q) : \texttt{Unit}} \qquad \frac{\Gamma \vdash P : \texttt{Unit}}{\Gamma \vdash (!P) : \texttt{Unit}}$$

$$\frac{\Gamma; x : X \vdash P : \texttt{Unit}}{\Gamma \vdash ((\nu x)P) : \texttt{Unit}} \quad X \cap \mathsf{FTV}(\Gamma) = \{\}$$

$$\frac{\Gamma; \Delta \vdash p : P \qquad \Gamma; \Delta \vdash t : \texttt{Unit}}{\Gamma \vdash (p \to t) : \texttt{Unit}} \quad \begin{array}{l} \Delta = \{x : X\} \text{ where } x \in \mathsf{bn}(p) \\ \mathsf{range}(\Delta) \cap \mathsf{FTV}(\Gamma) = \{\} \end{array}$$

Figure A.4: Type derivation rules for processes

the final type, and the type substitution required to map the expected type to the final type. Failure to infer a type is handled by exceptions. The `infer` function in turn calls the `inf` function that takes the parsed term, a type scheme environment (a mapping from variables to types), a type substitution (initially the identity substitution), and the expected type. When successful the result is a pair of: the type substitution for unification, and the inferred term. Additional inferred term constructors to support CPC are shown in Figure A.5 and closely align with their parsed term counterparts with the addition of type information (`i_type`) for patterns. The main body of the `inf` function switches on the parsed term's constructor as show below.

```
and inf = function
(* inf term (sEnv,fixed,constraints,sub,expectedTy) = (sub',term')
 * term
 * sEnv = type scheme environment for free variables in the term
 * fixed = type variables to be fixed in substitution
 * sub = the initial type substitution
 * expectedTy = the expected type
 * The arguments need not be normal w.r.t. the substitution *)
...
```

```
type i_term =
...
(*> CPC *)
  | Tdname of P_data.name_form * i_term * i_type
  | Tcname of P_data.name_form * term_variable * int * i_type
  | Tname of P_data.name_form * term_variable * int * i_type
  | Tpapp of i_term * i_term * i_type
  | Prll of i_term * i_term
  | Rest of term_variable * i_term
  | Repl of i_term
  | Pcase of i_term * i_term
(*< CPC *)
```

Figure A.5: CPC inferred term constructors

```
(*> CPC *)
  | Pparr (p1,p2) -> infer_parallel_composition p1 p2
  | Prepl (p) -> infer_replication p
  | Prest (n,p) -> infer_restriction n p
  | Ppcase (p,s) -> infer_cpc_case p s
  | Pname _
  | Pdname _
  | Pcname _ -> (fun _ _ _ _ -> typeError []
                  "Unexpected CPC name/constructor, aborting.")
(*< CPC *)
```

The inferences for processes are all handled by appropriate helper functions As the helper function for cases manages the patterns separately, instances of atoms (`Pname`, `Pdname` and `Pcname`) in `inf` are errors and raise an appropriate exception.

The functions for parallel composition and replication are straight forward; infer a type for the process and unify with the unit type before returning the appropriate inferred term and type substitution. The more interesting functions are for the restriction and cases.

The restriction creates an instance of the unit type `uty` and a fresh type variable `newty`. The type scheme is augmented to `sEnv1` with the fresh variable for the restricted name. This is used to infer the type for the process with the expected type being `Unit`. If this succeeds then continue building the type substitution by unifying the expected type for the process with the unit type. Return the resulting type substitution and the process converted to a inferred term.

```
(* Bind the restricted name to a new variable and infer the
 * process under the restriction. *)
and infer_restriction name proc sEnv fixed sub0 expectedTy =
  let uty = cvar "Unit" in
  let newty = (TyV(nextTypeVar(),0),Simple) in
  let sEnv1 = TMap.add (Var name) newty sEnv in
  let (sub2,proc2) = inf proc sEnv1 fixed sub0 uty in
  let sub3 = unify fixed sub2 uty expectedTy in
  (sub3,Rest(Var name,proc2))
```

The type inference for cases is more complex as patterns require significant type inference and checking, as well as requiring unification with type information from the body. Further, the scope of variables and constructors must be managed for both free and binding names. Lastly, the type information for patterns needs to be preserved in the inferred term for use in pattern unification.

To assist in the inference for cases there are three straightforward helper functions. The `communicable` function determines if a pattern is communicable or not, returning a boolean. The `getVarInfo` functions returns a pair of the index and type of a variable, and raises an exception if the variable is not in the local or global environment, i.e. is not declared anywhere. The `pushSub` function that takes a type substitution and applies it to the type

information carried in inferred patterns.

Most of the complexity in the inference for cases is in inferring the pattern. The `infer_cpc_pattern` function takes a type environment (mapping from variables to types), a type substitution, an expected type, and a pattern as input. The result is a tuple of; a type environment (a mapping from binding names to fresh type variables), a type substitution (unifying the types of the pattern where required), and the pattern as an inferred term.

The transformation for binding names, variables, constructors and datum is straightforward with detail given below. Observe that the inferred patterns contain type information to ensure type safety in evaluation (discussed in Section A.3). The binding name is mapped to the expected type (a fresh type variable) in the environment, and the expected type is added to the inferred binding name. Note that duplicate binding names are detected from the environment and raise exceptions during type inference. A variable or protected name has its index and type determined by the environment. The type is unified with the expected type, and the index and expected type used to create the inferred term for the name. Constructors in patterns have the substitution and inferred constructor generated by the inference for constructors. This information is used to create the inferred term of the constructor in the pattern. Datum are similarly handled by obtaining the substitution and inferred term by using the existing inference for datum. The results are used to create the inferred pattern for a datum.

```
(* Helper function to infer a CPC pattern, returns:
 * - environment of binding names and their types
 * - updated type substitution
 * - inferred pattern *)
let rec infer_cpc_pattern env sub pTy = function
| Pname(Binding,x) ->
```

```
    if TMap.mem (Var x) env
    then termError [Tvar(Var x,0)] "is a duplicate binding symbol"
    else (TMap.add (Var x) (pTy,Simple) env,
            sub,Tname(Binding,Var x,0,pTy))
| Pname(form,x) ->
    let (n,ty) = getVarInfo (Var x) in
    let sub1 = unify fixed sub pTy ty in
    (env,sub1,Tname(form,Var x,n,pTy))
| Pcname(form,c) ->
    let (sub1,c1) = infer_constructor c env fixed sub pTy in
    begin
      match c1 with
      | Tconstructor(Var c2,n) ->
            (env,sub1,Tcname(form,Var c2,n,pTy))
      | _ -> termError [c1] "when expecting constructor"
    end
| Pdname(form,Pdatum(d)) ->
    let (sub1,d1) = infer_datum d env fixed sub pTy in
    (env,sub1,Tdname(form,d1,pTy))
```

The inference for compound patterns depends on whether the pattern (as a whole) is communicable or not. When the pattern is communicable then the left hand component should be a function type $U \to T$ and the right hand component should be of type U. The type inference for this infers types for the components separately, augmenting the environment and substitution, and transforming the pattern. The substitution is then applied to the inferred function type by `applySub sub2 funTy`, and quantifications removed with `inst_tyscheme`, and the resulting type examined. If the result is a function type from `aTy` to `rTy`, then unify the argument type `argTy` with `aTy`, and the expected result type `pTy` with `rTy`. Return the appropriate environment, substitution and pattern. If the inferred type for the left hand component is not a function type, then unify it with a function type from the right hand component to the expected type `funty argTy pTy`. Again return the appropriate environment, substitution and pattern.

```
(* Communicable compound patterns must have types that support
 * evaluation. *)
| Papply(p,q) as patt when communicable patt ->
      let funTy = TyV(nextTypeVar(),0) in
      let argTy = TyV(nextTypeVar(),0) in
      let (env1,sub1,p1) = infer_cpc_pattern env  sub  funTy p in
      let (env2,sub2,q2) = infer_cpc_pattern env1 sub1 argTy q in
      begin
        match inst_tyscheme (applySub sub2 funTy) with
        | Funty(aTy,rTy) ->
            let sub3 = subunify fixed sub2 aTy argTy in
            let sub4 = subunify fixed sub3 rTy pTy in
            (env2,sub4,Tpapp(p1,q2,pTy))
        | fTy ->
            let sub3 = subunify fixed sub2 fTy (funty argTy pTy) in
            (env2,sub3,Tpapp(p1,q2,pTy))
      end
```

Otherwise when the pattern is not communicable then it does not need to support evaluation of the left hand component applied to the right. Here is it sufficient to generate the environment, substitution and pattern components without any relation between the types. Indeed, fresh types are used for the components and the expected type used for the compound pattern as a whole. (Note that the expected type is not checked for any type unification, its value does not matter as will be discussed in Section A.3.)

```
(* Pattern is not communicable, the type of the "function" and
 * "argument" components do not relate to each other. *)
| Papply(p,q) ->
      let pTy0 = TyV(nextTypeVar(),0) in
      let (env1,sub1,p1) = infer_cpc_pattern env sub pTy0 p in
      let qTy1 = TyV(nextTypeVar(),0) in
      let (env2,sub2,q2) = infer_cpc_pattern env1 sub1 qTy1 q in
      (env2,sub2,Tpapp(p1,q2,pTy))
```

Once the type inference for patterns is defined, the type inference for cases is straightforward. Generate a type environment, type substitution and

inferred pattern by starting with an empty type environment, the existing type substitution, and a fresh type variable for the pattern. The resulting type environment `sEnv1` is then folded into the original type environment `sEnv0` to allow the binding names to overshadow earlier declarations. The body is then inferred with the expected type of `Unit`. The resulting type substitution is used to (sub)unify the expected type for the case with the unit type. The resulting type substitution is then applied to the types in the inferred pattern. Lastly the type substitution and transformed case are returned.

```
(* The inference goes as follows
 * - create a fresh type for the pattern
 * - infer the pattern with the fresh type and empty binding
 *   environment
 * - use binding environment to update substitution environment
 * - infer the body with updated environment and sub
 * - subunify to final sub
 * - use sub to update type annotations in the pattern
 *   (removing dependence on sub)
 * - return sub and inferred CPC case *)
let pTy = TyV(nextTypeVar(),0) in
let (sEnv1,sub1,patt1) =
            infer_cpc_pattern TMap.empty sub0 pTy patt in
let sEnv2 = TMap.fold TMap.add sEnv1 sEnv0 in
let (sub2,body2) = inf body sEnv2 fixed sub1 (cvar "Unit") in
let sub3 = subunify fixed sub2 (cvar "Unit") expectedTy in
let patt2 = pushSub sub3 patt1 in
(sub3,Pcase(patt2,body2))
```

Although the detail of the **bondi** type system has not been formalised anywhere, by exploiting the existing functions and algorithms the type inference for CPC constructs is consistent with Concurrent **bondi** type. This includes exploiting sub-unification algorithms to handle object-orientation, and instantiating type schemes to support quantified types.

A.3 Interacting

Once Concurrent **bondi** programs have been transformed by parsing and type inference, the inferred terms are passed to an evaluator that performs reductions to yield values. This section highlights the additions to the evaluator to support evaluation of processes, in particular pattern unification and process interaction.

Evaluation

Evaluation in the Concurrent **bondi** interpreter is handled by an `eval` function that accepts a value environment and an inferred term and reduces this to a value. The value environment is a mapping from variables to values and accumulates all bindings that occur within an evaluation of a term. Similar to the inference function `inf`, `eval` switches on the inferred term and calls helper functions to handle different term forms. The additions to the `eval` function for CPC are shown below.

```
let rec (eval : value_env * i_term -> value) = fun (vEnv,term) ->
  match term with
...
(*> CPC *)
  | Prll(p1,p2) -> eval_parallel_composition vEnv p1 p2
  | Rest(x,p) -> eval_restriction vEnv x p
  | Repl(p) -> eval_replication vEnv p
  | Pcase(_,_) -> eval_process_case vEnv term
(*< CPC *)
```

As interactions for processes are handled separately to support concurrency, the helper functions for processes perform optimisations and then queue the resulting processes for adding to the data-space (details on the data-space

are in Section A.4). The optimisations always yield either a replication or a case for adding to the data-space.

For parallel composition the optimisation is straightforward: each processes is evaluated separately ultimately ending up in the queue. The parallel processes are evaluated in random order to prevent potential determinism.

A restriction that appears in the `eval` function is top level, that is not under any replication or case. Thus it is is safe to instantiate the restricted name to a unique (internal only) value and evaluate the process under restriction.

A replication can be handled in three separate ways depending on what process form is immediately under the replication. If the replication is a parallel composition then each of the processes is treated as a separate replication

$$!(P \mid Q) \sim !P \mid !Q$$

as by Theorem 8.7.3. If the replication is of another replication then one is dropped

$$!!P \simeq !P$$

as by Theorem 8.7.4. Otherwise the process under the replication is a restriction or a case. In both of these scenarios the process is queued for adding to the data-space. The queue accepts a tuple consisting of: a locking function for the process; an unlocking function for the process; the local value environment; and the process as an inferred term. The locking and unlocking is required for safe concurrency, details in Section A.4, while the value environment and process are required for interaction and reduction.

```
(* Remove excess replications, continue until the process
```

```
 * under the replication is either a restriction or a case. *)
and eval_replication pEnv = function
  (* !(P|Q) is close enough to !P|!Q that we shall use the
   * second form to simplify the matching algorithm. *)
  | Prll(p1,p2) -> let _ = (eval_replication pEnv p1,
                            eval_replication pEnv p2) in
                   Vdatum Datum.Un
  (* !(!P) is close enough to !P that we use the latter. *)
  | Repl(p) -> eval_replication pEnv p
  (* Replications can always be locked and require no effect
   * to unlock. *)
  | p ->  let l,ul = (fun () -> true),(fun () -> ()) in
          add_to_proc_queue (l,ul,pEnv,Repl(p))
```

Note that as processes have the unit value when evaluated, the result of `eval_replication` (and `add_to_proc_queue`) is the unit value (given by `Vdatum Datum.Un`).

Cases cannot be optimised and are simply queued for adding to the data-space with appropriate locks.

Pattern unification

The first major algorithm to be implemented from CPC is the unification of patterns. This can be implemented as an algorithm that accepts two patterns and determines if they can be unified. When successful, the result should include the substitutions generated by unification. If they cannot be unified then failure is reported.

Implementing such an algorithm is more complex than simply implementing the pattern-unification rules, the algorithm must also account for: constructors, datum, reducible variables, imperative constructs, and type unification.

Consider a few illustrative examples of pattern-unification to be supported by the algorithm. Consider the unification of the two names (or variables) `x` and `x`. Although they may have the same identifier, i.e. `"x"`, they may have different scope. For example:

```
let x = 3 in (cpc x -> () | rest x in cpc x -> ())
```

where one `x` is bound to `3` and the other is a restricted name. As these two patterns should not unify, the algorithm must evaluate names in patterns and compare their values. Note that this is not necessarily trivial as each pattern has its own value environment and so both will appear to be `x` until the appropriate environment is checked.

Another example is when one pattern has structure and the other does not. For example, consider the following program:

```
let ls = [1] in cpc ls -> () | cpc Cons 1 Nil -> println "[1]" .
```

A naive unification algorithm would note that the first pattern `ls` is a name and the second a compound and fail to unify. Of course the patterns can be unified if `ls` is instantiated to `Cons 1 Nil`. However, immediately instantiating `ls` to its binding (in this case `Cons 1 Nil`) in the pattern is not suitable either, consider the sequence of declarations below (each declaration is terminated by `;;`).

```
let x = Ref 3;;
let y = Ref 5;;
cpc x -> () | cpc y -> println "Found 5";;
x = 5;;
```

If `x` was immediately instantiated to a copy of the reference to `3` then the patterns would never unify, however when `x` is assigned the value `5` then unification should succeed. Therefore, to account for these examples the

unification algorithm evaluates free names in patterns only when required for unification.

Yet another complexity for the unification algorithm is to ensure type safety. The process

```
cpc \f -> f 1 2
```

should only unify when the binding name `f` is bound to a function of type `Int -> Int -> Unit`. The pattern-unification must also support appropriate type unification, including handling sub-typing, instantiation of type variables, and type quantification.

All of these complexities are handled in the pattern-unification algorithm, as well as implementing the basic specification of CPC's pattern-unification. The rest of this section discusses how these are handled by the pattern-unification algorithm.

The main function to perform pattern-unification is `unify1` that accepts: a pair of value environments; a pair of substitutions; and a pair of patterns. The value environments are the respective local mappings (from variables to values) for each of the patterns. The substitutions are those generated by unification so far.

When both patterns are variable or protected names then their values are computed (with `eval`) and compared. If the values are equal then the result is `Some` of the substitutions so far (no additional bindings). If the values are not equal then they cannot be unified as indicated by `None`.

```
(* When simple names/variables are involved, compare their
 * evaluated values. *)
| (Tname(P_data.Variable,pv,pn,_),Tname(P_data.Variable,qv,qn,_))
| (Tname(P_data.Variable,pv,pn,_),Tname(P_data.Protected,qv,qn,_))
| (Tname(P_data.Protected,pv,pn,_),Tname(P_data.Variable,qv,qn,_))
```

```
| (Tname(P_data.Protected,pv,pn,_),Tname(P_data.Protected,qv,qn,_))
        -> if evalVar pEnv pv pn = evalVar qEnv qv qn
           then Some (subp,subq)
           else None
```

If one pattern `p` is a binding name then several checks need to be made to see if the patterns can be unified. The other pattern `q` is passed to the `communicable` function that here returns a pair of a boolean, if the pattern is communicable or not, and the type of the pattern `qTy`. When `q` is communicable then attempt to unify the types of `p` and `q` (`pTy` and `qTy` respectively). Successful type unification finds a type substitution that makes `pTy` and `qTy` equal, which is then added to the environment along with the binding of `p` to `q`. This substitution is then passed back to indicate success. Failure to unify types results in failure to unify the patterns. Here and for the rest of the section when there are two symmetric pieces of code only one shall be shown. In this scenario the following code is for when `p` is a binding name.

```
(* If one pattern is a binding name then ensure the other is
 * communicable and determine it's type. Check that
 * the binding is type safe, if so bind, otherwise fail. *)
| (Tname(P_data.Binding,bp,_,pTy),qv) ->
  begin
    match communicable qv with
    | (false,_) -> None
    | (true,qTy) ->
      try
        begin
          let sub = unify_for_cpc (getTySub pEnv)
                        (getTySub qEnv) pTy qTy in
          Some(TMap.add bp (eval (qEnv,(unCPCPattern qv)))
                (TMap.add (Mvar(-1)) (VtySub(sub)) subp),
               subq)
        end
      with _ -> None
  end
```

When both patterns are compounds, represented by applications, then attempt to unify their respective left hand components. If this succeeds then pass the augmented substitutions into recursively unifying the right hand components, otherwise return failure to unify.

```
(* Do application component wise. *)
| (Tpapp(p1,p2,_),Tpapp(q1,q2,_)) ->
    begin
      match unify1 (pEnv,qEnv) (subp,subq) (p1,q1) with
      | Some (subp1,subq1) ->
                unify1 (pEnv,qEnv) (subp1,subq1) (p2,q2)
      | None -> None
    end
```

It is possible that a variable or protected name is bound to an application and the other pattern is a compound. To catch this possibility, find the value bound to the name and determine if it is an application. When the value is a compound, create a new pattern, maintaining the variable or protected status, that is a compound with fresh names and mappings in the value environment. The unification can then proceed as before. Note that the components of values found during evaluation cannot be accurately typed, so do not attempt to unify with binding names as this would break type safety. Here the `freePattern` function ensures a pattern has no binding names[1].

```
(* One is an application and the other is a non-binding name,
 * should evaluate the name to see if it is also an application
 * (bondi data structure).
 * ... *)
| (Tpapp(_,_,_) as p,Tname(qf,qv,qn,qTy))
    when qf != P_data.Binding && freePattern p ->
    begin
```

[1]When the type system is disabled `freePattern` always returns `true` as type rules are not enforced.

```
      match evalVar qEnv qv qn with
      | Vapply(qVal1,qVal2) ->
          let qv1 = nextvar () in
          let qv2 = nextvar () in
          let qEnv1 = TMap.add qv1 qVal1
                               (TMap.add qv2 qVal2 qEnv) in
          let q1 = Tname(qf,qv1,0,qTy) in
          let q2 = Tname(qf,qv2,0,qTy) in
          unify1 (pEnv,qEnv1) (subp,subq) (p,Tpapp(q1,q2,qTy))
      | _ -> None
    end
```

Constructors and datum in patterns are handled in a somewhat similar manner; create a fresh term variable and add a mapping from that variable to the value in the local environment. Unification then continues with the variable in the pattern and the modified environment.

```
(* Constructors in patterns can add complexities, easiest to
 * augment local environment. *)
| (Tcname(pt,pv,pn,pTy),q) ->
        let pv1 = nextvar () in
        let pEnv1 = TMap.add pv1 (Vconstructor (pv,pn)) pEnv in
        unify1 (pEnv1,qEnv) (subp,subq) (Tname(pt,pv1,0,pTy),q)
...
(* Datum in patterns, simply treat like a constructor. *)
| (Tdname(pt,Datum(pd),pTy),q) ->
        let pv1 = nextvar () in
        let pEnv1 = TMap.add pv1 (Vdatum pd) pEnv in
        unify1 (pEnv1,qEnv) (subp,subq) (Tname(pt,pv1,0,pTy),q)
```

If all these fail then there is no unification of the patterns so return failure.

```
  (* Unable to unify, fail. *)
  | _ -> None
```

Unification of patterns is started by the `unify` function that accepts the value environments and associated patterns, this in turn calls `unify1` beginning with empty substitutions.

Interaction

Now that unification can be computed, the next step is to determine if two arbitrary processes can interact. This is a little more complex than simply checking if processes are cases and then trying to unify their patterns, as cases may be underneath some combination of parallel composition, replication and restriction.

The algorithm to compute whether two processes can interact is somewhat similar to the pattern-unification algorithm. The algorithm takes two value environments and two processes, and returns either `Some` (of an OCaml function) when interaction occurs or `None` if the processes cannot interact. The main difference is that successful interaction returns a function within the interpreter to evaluate the processes yielded by interaction.

The rest of this section highlights the algorithm for testing interactions between processes. Observe that some delicacy is required to ensure no processes are lost during interaction. For example, if the processes being considered are P and Q, and $P = P_1 \mid P_2$ such that $P_1 \mid Q \longmapsto R$ then the result must be $R \mid P_2$.

The inner function that handles the majority of testing for interactions between processes is `interactable1` that switches on the processes to try and find cases. If either of the processes is a restriction then instantiate the restricted name to a fresh (internal only) value in the appropriate value environment and then continue under the restriction. As before, when the code handles two symmetric scenarios, only one shall be shown here.

```
let rec interactable1 pE qE p q =
  match(p,q) with
  (* Restrictions that must be underneath a replication, can
```

```
   * instantiate the name to try and match. *)
  | (Rest(n,p1),_) ->   let n1 = nextvar() in
                        let pE1 = TMap.add n (Vvar(n1)) pE in
                        interactable1 pE1 qE p1 q
```

When parallel composition is encountered then both processes in parallel must be tested for interaction. To avoid deterministic behaviour this is done in random order. If an interaction occurs with one of the processes from a parallel composition then the other process is added to the returned function so that it is not lost. This is handled by the following code.

```
(* Parallel composition underneath a replication/restriction.
 * Note that this only tests against p and q matching each
 * other, internal reductions are done elsewhere. *)
| (Prll(p1,p2),_) ->
      begin
        let pA,pB = if Random.int 2 = 1 then p1,p2 else p2,p1 in
        match interactable1 pE qE pA q with
        | Some cont -> Some (fun () -> cont ();
                             ignore (eval(pE,pB)))
        | None ->
            begin
              match interactable1 pE qE pB q with
              | Some cont -> Some (fun () -> cont ();
                                   ignore (eval(pE,pA)))
              | None -> None
            end
      end
```

Replications require that if an interaction occurs the replication is maintained, similar to the parallel process that did not interact. This is straightforward and handled as below.

```
(* Replication, just try to interact with a copy. If successful we
 * are under a restriction already so we must have a copy of
 * the replicating process with the right restriction/scoping. *)
| (Repl(p1),_) ->
      begin
```

```
        match interactable1 pE qE p1 q with
        | Some cont -> Some (fun () -> cont ();
                             ignore (eval(pE,Repl(p1))))
        | None -> None
      end
```

Otherwise when both processes are cases, if their patterns can be unified then create a function to evaluate the resulting bodies with value environments augmented by the substitutions.

```
(* Two cases, try to match their patterns. *)
| (Pcase(pp,pb),Pcase(qp,qb)) ->
      begin
        match unify pE qE pp qp with
        | Some (ps,qs) -> Some (fun () ->
              begin
              (* Generate the processes to continue with. *)
                let pE1 = TMap.fold TMap.add ps pE in
                let qE1 = TMap.fold TMap.add qs qE in
                let _,_ = (eval (pE1,pb)),(eval (qE1,qb)) in ()
              end)
        | None -> None
      end
```

This finds any interactions between two processes but has two limitations; top level replicating processes will create copies of themselves, and there is no detection of reductions within a process.

The copying of replications is required to ensure those within a process are not lost. However, top level replications do not need to be copied. To manage this the `interactable` function is called on top level processes that strips off the outer replications and then calls `interactable1`.

Interactions within a process are handled by a similar function `self_interaction` that finds interactions within a process. Due to the optimisations any interactions within a process must be under a top level replication. This makes

managing self interactions easier and so the result of `self_interact` is always the OCaml unit value.

A.4 Concurrency

Now that the machinery for evaluation and interaction is defined it remains to manage the processes in a concurrent setting. Management of the processes has two main aspects; the management of the global process environment, and testing for interactions of processes.

The process environment is represented as a hash table keyed on process identifiers (integers) that stores a tuple consisting of; a locking function, an unlocking function, the value environment for the process, and the process as an inferred term. The process identifiers are generated by a concurrency safe function that increments a counter. The locking and unlocking functions are to prevent a process from interacting with more than one other process at a time. For processes that are cases at the top level this is based on a mutual exclusion lock For replicating processes, dummy functions that always allow locking and unlocking are sufficient.

The management of the environment is done by a constant loop that goes through three stages; adding new processes to the environment, testing for interactions, and removing processes from the environment. Each loop also incorporates a short (100ms) pause to improve load balancing and responsiveness of the interpreter. The code for this is in the `procManager` function shown in Figure A.6.

Adding processes is done by locking the process queue and then adding

```
(* An evaluation engine for CPC, goes through a constant
 * cycle to add processes to the environment, try to match
 * them all and then clean up. *)
let procManager () =
  let addprocs () =
    pqlock();
    randomiter (fun x -> Hashtbl.add procEnv (procGen ()) x)
               !proc_queue;
    proc_queue := [];
    pqunlock()
  in
  while true do
    Thread.delay 0.1;
    addprocs ();
    Hashtbl.iter self_interaction procEnv;
    let (r,ks) = Hashtbl.fold topfoldinteract procEnv (false,[]) in
    if r
    then List.iter (Hashtbl.remove procEnv) ks
  done
```

Figure A.6: CPC process manager

the waiting processes in random order. Each process is given a freshly generated process identifier and added to the environment. Once all the queued processes have been added, the queue is reset to empty and unlocked.

Testing for interactions is handled by two functions `self_interaction` and `topfoldinteract`. The first finds interactions within a replication and always returns unit. The second finds interactions between processes and returns a (possibly empty) list of identifiers for processes that should be removed from the process environment. If interactions have occurred that may result in processes being listed for removal, then the list is used to remove the identified processes.

The last aspect of process management is to introduce concurrency. As all

the relevant functions have been made concurrency safe, it is possible to run any numbers of `procManager` functions in parallel to support concurrency. The Concurrent **bondi** interpreter creates two instances of `procManager` in separate threads, as well as running the original evaluator code including the pre-processing of CPC process in a third thread. This allows multiple interactions to occur at once, as well as computations for other Concurrent **bondi** programs.

The rest of this section overviews how processes in the process environment are tested for interaction.

Choosing processes

All of the processes in the process environment need to be tested for interactions. This involves both testing for interactions within replications, and testing if any two processes in the environment can interact with each other.

The interaction of processes with themselves is handled by the function `self_interaction` that performs any interactions and returns unit. Such interactions are trivial to handle by the process manager. As interactions within a process will evaluate the resulting programs themselves there is no need to manage the evaluation. Further, as these always occur within replications there is no need to consider locking or remove the replicated process from the process environment.

The interaction between processes is more complex and is implemented with two functions that fold over the process environment. The general approach is to attempt to lock each process in the process environment by an outer `topfoldinteract` function. If a process can be locked, then attempt

to interact with other processes in the environment via `foldinteract`. When interactions occur then; evaluate the bodies, record that an interaction has occurred, and pass back any process identifiers of processes that should be removed from the environment. The remainder of this section details how this is implemented.

The `topfoldinteract` function begins with a pair of; a boolean to indicate if interactions have occurred, and an empty list of processes to be removed from the environment. This is then folded over the processes environment and attempts to lock each process in turn using the process' locking function `pl`. If locking is successful, then the inner function `foldinteract` is called with the outer process' information. If the inner call returns that interaction occurred then this is passed back out along with an updated list of processes to remove from the environment (the `to_clean` function identifies the relevant process identifiers). Note that when interactions occur the process is not unlocked. Otherwise if no interaction occurred then the (outer) process is unlocked with `pul` and the ongoing record of interaction success and processes to remove is passed on.

```
(* A top level fold that tries to interact every process in
 * the process environment with every other process. Uses
 * foldinteract (above) with each process to try every
 * possible pair of processes. Accumulates whether any
 * interactions have occurred (succ) and also a list of
 * processes to be removed from the process environment (ks).
 * Note: Now updated to do multiple interactions per pass. *)
let topfoldinteract ppid (pl,pul,pE,p) (succ,ks) =
  if pl()
  then
    begin
      let (r,_,_,_,ks1) = Hashtbl.fold foldinteract procEnv
                                       (false,ppid,pE,p,ks) in
      if r
```

```
      then (true,to_clean ppid ks1 p)
      else (pul();(succ,ks))
    end
  else (succ,ks)
```

Observe that `topfoldinteract` may lock many processes in a single fold over the process environment and thus allows any number of interactions to occur per pass.

The inner function `foldinteract` is called when one process is already locked to test if that process can interact with any other. If no interaction with the outer process has already occurred `succ` and the inner process can be locked `ql`, then both processes are tested for interaction with `interactable`. If they do interact then the bodies are evaluated immediately and the interaction is reported along with an updated list of processes to remove from the environment. As before when interaction occurs the process that interacted is not unlocked. If no interaction occurs then the inner process is unlocked and the original information is passed along.

```
(* A foldable function that attempts to interact a CPC process
 * with it's ID (ppid), environment pqE), processes (p) against
 * every other process (qE,q) in the environment. Result is
 * either false if no interaction and the input, or true if
 * there is a interaction with the list of processes IDs to
 * remove from the environment. *)
let foldinteract qqid (ql,qul,qE,q) (succ,ppid,pE,p,ks) =
  if succ || not (ql ())
  then (succ,ppid,pE,p,ks)
  else match interactible pE qE p q with
       | Some cont -> (cont (); (true,ppid,pE,p,to_clean qpid ks q))
       | None -> (qul();(succ,ppid,pE,p,ks))
;;
```

This approach ensures that every process is tested for interaction with every other process in the process environment. Although this can be inefficient

with the constant testing of processes that may never interact, it supports the interplay between imperative and object-oriented features within Concurrent **bondi**. Specifically, references, arrays or objects that appear in CPC patterns may change their value between attempts to find interactions, and so must be tested regularly. Note that in a language without these features it may be possible to test for interaction only when adding processes to the environment, and when replications create infinite interactions.

Bibliography

[Abr90] Samson Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lvy. Explicit substitutions. *Journal of Functional Programming*, 1(04):375–416, 1991.

[ACS98] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.

[AG97] Martn Abadi and Andrew Gordon. Reasoning about cryptographic protocols in the spi calculus. In Antoni Mazurkiewicz and Jzef Winkowski, editors, *CONCUR '97: Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 59–73. Springer Berlin / Heidelberg, 1997.

[Bar85] Henk P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics.

Elsevier Science Publishers B.V., 1985. BAR h 85:1 1.Ex.

[BB90] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94, New York, NY, USA, 1990. ACM.

[BBM04] Michele Boreale, Maria Grazia Buscemi, and Ugo Montanari. D-fusion: a distinctive fusion calculus. In *In Proc. APLAS04, LNCS 3302*, pages 296–310. Springer, 2004.

[BG07] Michele Bugliesi and Marco Giunti. Secure implementations of typed channel abstractions. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 251–262, New York, NY, USA, 2007. ACM.

[BGZZ98] Nadia Busi, Roberto Gorrieri, Gianluigi Zavattaro, and Mura Anteo Zamboni. Comparing three semantics for linda-like languages. *Theoretical Computer Science*, 240, 1998.

[BLM05] Allen L. Brown, Cosimo Laneve, and L. Gregory Meredith. Piduce: A process calculus with native XML datatypes. In *In Proc. of EPEW05/WS-FM05, volume 3670 of Lect*, pages 18–34. Springer, 2005.

[bon11] bondi. bondi, 2011. `http://bondi.it.uts.edu.au`.

[BPV03] Michael Baldamus, Joachim Parrow, and Björn Victor. Spi calculus translated to π-calculus preserving may testing. Technical report, 2003.

[BPV05] Michael Baldamus, Joachim Parrow, and Björn Victor. A fully abstract encoding of the *pi*-calculus with data terms. In *ICALP*, pages 1202–1213, 2005.

[Cam11] The Caml language. The Caml language: Home, 2011. `http://caml.inria.fr/`.

[CCAV08] Diletta Cacciagrano, Flavio Corradini, Jesús Aranda, and Frank D. Valencia. Linearity, persistence and testing semantics in the asynchronous pi-calculus. *Electron. Notes Theor. Comput. Sci.*, 194:59–84, January 2008.

[CCP07] D. Cacciagrano, F. Corradini, and C. Palamidessi. Separation of synchronous and asynchronous communication via testing. *Theor. Comput. Sci.*, 386:218–235, October 2007.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[CHS72] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic*, volume II. North-Holland, Amsterdam, 1972.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[CM03] Marco Carbone and Sergio Maffeis. On the expressive power of polyadic synchronisation in π-calculus. *Nordic J. of Computing*, 10:70–98, May 2003.

[Con11] Concurrent bondi. Concurrent bondi, 2011. `http://www-staff.it.uts.edu.au/~tgwilson/concurrent_bondi/`.

[cpp10] CppLINDA: C++ LINDA implementation, 2010. Retrieved 18 November 2010, from http://sourceforge.net/projects/cpplinda/.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[dBP94] Frank S. de Boer and Catuscia Palamidessi. Embedding as a tool for language comparison. *Inf. Comput.*, 108:128–157, January 1994.

[DGP07] R. De Nicola, D. Gorla, and R. Pugliese. Basic observables for a calculus for global computing. *Information and Computation*, 205(10):1491–1525, 2007.

[Fis93] Michael J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6:259–287, 1993.

[GA97] A.D. Gordon and M. Abadi. A calculus for cryptographic protocols: The spi calculus. *4th ACM Conference on Computer and Communications Security*, pages 36 – 47, 1997.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GGJ10] Thomas Given-Wilson, Daniele Gorla, and Barry Jay. Concurrent pattern calculus. In C. Calude and V. Sassone, editors, *Proc. of 6th International IFIP Conference on Theoretical Computer Science (IFIP-TCS 2010)*, IFIP. Springer, 2010.

[GJS05] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley Professional, third edition, 2005.

[GLT89] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[Gor08a] D. Gorla. Comparing communication primitives via their relative expressive power. *Information and Computation*, 206(8):931–952, 2008.

[Gor08b] D. Gorla. Towards a unified approach to encodability and separation results for process calculi. In F. van Breugel and M. Chechik, editors, *Proc. of 19th International Conference on Concurrency Theory (CONCUR'08)*, number 5201 in LNCS, pages 492–507. Springer, 2008.

[GP02] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

[GW07] Thomas Given-Wilson. *Interpreting the Untyped Pattern Calculus in **bondi***. Honours Thesis, University of Technology, Sydney, Sydney, Australia, August 2007.

[GW10] Thomas Given-Wilson. Concurrent pattern calculus in **bondi**. *Young Researchers Workshop on Concurrency Theory (YR-CONCUR)*, 2010.

[GWHJ07] Thomas Given-Wilson, Freeman Huang, and Barry Jay. Multi-polymorphic programming in bondi, 2007. `http://www.progsoc.org/~sanguinev/files/multi-polymorphism.pdf`.

[GWJ11] Thomas Given-Wilson and Barry Jay. Getting the goods with concurrent **bondi**. *Proceedings of the Fourth Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES)*, 2011.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[HMP08] Bjørn Haagensen, Sergio Maffeis, and Iain Phillips. Matching systems for concurrent calculi. *Electron. Notes Theor. Comput. Sci.*, 194:85–99, January 2008.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, New York, NY, USA, 1986.

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.

[HY95] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437–486, 1995.

[Jay04] C.B. Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):911–937, November 2004.

[Jay09] Barry Jay. *Pattern Calculus: Computing with Functions and Data Structures*. Springer, 2009.

[JGW11] Barry Jay and Thomas Given-Wilson. A combinatory account of internal structure. *Journal of Symbolic Logic*, 76(3):807–826, 2011.

[JK06] Barry Jay and Delia Kesner. Pure pattern calculus. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27–28, 2006, Proceedings (ed: P. Sestoft)*, pages 100–114, 2006. Revised version at `www-staff.it.uts.edu.au/~cbj/Publications/purepatterns.pdf`.

[JK09] Barry Jay and Delia Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):34 pages, 2009.

[JoC11] JoCaml. The JoCaml system, 2011. `http://jocaml.inria.fr/`.

[Kea69] John T. Kearns. Combinatory logic with discriminators. *Journal of Symbolic Logic*, 34(4):561–575, 1969.

[Kea73] John T. Kearns. The completeness of combinatory logic with discriminators. *Notre Dame Journal of Formal Logic*, 14(3):323–330, 1973.

[Kle35] Stephen Kleene. A theory of positive integers in formal logic. *American Journal of Mathematics*, 57:153–173 and 219–244, 1935.

[Kle52] S.C. Kleene. *Introduction to Methamathematics*. van Nostrand, 1952.

[LMB92] John Levine, Tony Mason, and Doug Brown. *lex & yacc, 2nd Edition (A Nutshell Handbook)*. O'Reilly, October 1992.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil90] Robin Milner. Functions as processes. In *Proceedings of the seventeenth international colloquium on Automata, languages*

and programming, pages 167–180, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[Mil93] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, 1992.

[MS92] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proceedings of ICALP '92*, volume 623 of *LNCS*, pages 685–695. Springer, 1992.

[Nes00] Uwe Nestmann. What is a "good" encoding of guarded choice? *Inf. Comput.*, 156:287–319, January 2000.

[Nes06] Uwe Nestmann. Welcome to the jungle: A subjective guide to mobile process calculi. In *CONCUR*, pages 52–63, 2006.

[New87] Isaac Newton. *Philosophiæ Naturalis Principia Mathematica*. 1687.

[NFP98] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[Pal03] Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous π-calculi. *Mathematical. Structures in Comp. Sci.*, 13:685–719, October 2003.

[Par08] Joachim Parrow. Expressiveness of process algebras. *Electron. Notes Theor. Comput. Sci.*, 209:173–186, April 2008.

[PMR99] G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conference on Software Engineering (ICSE'99)*, pages 368–377. ACM Press, 1999.

[PSVV06] Catuscia Palamidessi, Vijay Saraswat, Frank D. Valencia, and Bjorn Victor. On the expressiveness of linearity vs persistence in the asychronous pi-calculus. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, pages 59–68, Washington, DC, USA, 2006. IEEE Computer Society.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In *PROOF, LANGUAGE AND INTERACTION: ESSAYS IN HONOUR OF ROBIN MILNER*, pages 455–494. MIT Press, 1997.

[PV98] Joachim Parrow and Björn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proc. of LICS*, pages 176–185. IEEE Computer Society, 1998.

[PV04] Iain Phillips and Maria Grazia Vigliotti. Electoral systems in ambient calculi. In *In Proceedings of 7th International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 2004*, pages 408–422. Springer-Verlag, 2004.

[PV06] Iain Phillips and Maria Grazia Vigliotti. Leader election in rings of ambient processes. *Theor. Comput. Sci.*, 356:468–494, May 2006.

[Ros73] Barry K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *J. ACM*, 20(1):160–187, 1973.

[Sch24] M. Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92(3 - 4):305–316, 1924.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[Tur36] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[WG04] Lucian Wischik and Philippa Gardner. Strong bisimulation for the explicit fusion calculus, 2004.

[WG05] Lucian Wischik and Philippa Gardner. Explicit fusions. *Theor. Comput. Sci.*, 340(3):606–630, 2005.

[YFY96] C. Yuen, M. Feng, and J. Yee. BaLinda suite of languages and implementations. *Journal of Software Systems*, 32:251–267, 1996.