# A Formalisation of Biochemical Process Languages in Lean

*Jonathan Coates*

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2020

# Abstract

The continuous π-calculus is a process algebra for modelling the behaviour of biochemical molecular systems. I have developed a mechanisation of the calculus within the Lean theorem prover, and shown the semantics are sound. I extend the semantics to allow an arbitrary equivalence relation.

I defined one such alternative relation, which allowed us to create a computable version of the semantics. This was then used to demonstrate several cπ examples have the behaviour expected from standard systems biology.

# Table of Contents

# Chapter 1

# Introduction

As biological research progresses, we find ourselves increasingly dependent on the ability to model complex biochemical systems[1, 2]. There are two primary ways of generating such models. The mathematical method, which gives a series of equations to solve, and the computational method which provides a program to 'execute'.

One approach to building computational models is to view biomolecular systems as a series of communicating processes, with message passing representing reactions between molecules. Much work has been done on using process algebras, such as Regev et. all in [3]. Of interest to us is the *continuous* $\pi$*-calculus* (or c$\pi$), devised by Kwiatkowski and Stark[4].

Loosely derived from the $\pi$-calculus, $c\pi$'s semantics operate on a real vector space of species. It provides a compositional way to create interacting molecules and generate ordinary differential equations (ODEs) describing how concentrations of involved molecules change over time.

While the continuous $\pi$-calculus provides many interesting features, the definitions and properties of $c\pi$ have only been expressed by humans on pen and paper. The proofs for some of these properties are incredibly nuanced and so, even under the most watchful eye, there is always the risk of flaws. The solution is to describe the calculus in a *theorem prover*, providing a machine-checked proof for every proposition. Doing so gives us a concrete guarantee that our theorems are correct.

Interactive theorem provers, or *proof assistants*, are one class of such systems. These provide an interactive environment where the user can develop proofs with help from the computer. Lean[5] is one such interactive theorem prover. Initially developed in 2013 it has a vibrant community, generating a reasonable amount of interest in the mathematical world due to its expressiveness. For these reasons, I decided to use Lean for this project.

## 1.1 Contributions

- The primary contribution is a largely complete mechanised definition of the continuous-$\pi$ calculus within Lean. It contains all thirteen definitions of [4], and proves four of the five theorems, with the remaining one relying on a single axiom.
- In order to obtain a computable semantics, we define an alternative equivalence relation (n-equivalence), which is strictly weaker than structural congruence. We use this to execute several examples, and demonstrate the produced ODEs are what we would expect.

## 1.2 Overview

As part of this project, I built a model of the c$\pi$-calculus within Lean. This model, and the proofs relating to it, are about 5700 lines of Lean code. The code for this may be found at the associated GitHub repository[1].

The remainder of the report is structured as follows.

- In Chapter 2, I introduce the basics of the continuous $\pi$-calculus and present an example using an enzyme system which will be used throughout this document. I also provide a brief introduction to the Lean theorem prover.
- Chapter 3 describes the syntax of $c\pi$ in detail. As definitions are introduced, I present the Lean code required to describe each construct.
- Chapter 4 presents the semantics of the calculus, including the transition system, execution, and generation of ODEs. We also show the required lemmas, largely focusing on those involving structural congruence.
- Chapter 5 discusses several problems that arise when working with structural congruence within a theorem prover. We explore whether an alternative equivalence relation may be used, defining the necessary properties that such a relation must have, and investigate one possible candidate.
- Finally, Chapter 6 and Chapter 7 describe our progress, evaluate our successes and failures, and explore possible future avenues of work.

[1] `https://github.com/continuouspi/lean-cpi`

# Chapter 2

# Background

## 2.1 Continuous $\pi$-calculus

The continuous $\pi$-calculus is a process calculus designed for modelling biological systems. Molecules, and the reactions that they undergo, are expressed as a *species*. Species may then be mixed together at specific concentrations to form a *process*, from which we can derive the ODEs for this system.

Species are the fundamental building block of $c\pi$, and are described by the following grammar.

$$A, B ::= \mathbf{0} \;\mid\; D(\vec{a}) \;\mid\; \sum_{i=0}^{n} \pi_i.A_i \;\mid\; A \,|\, B \;\mid\; (\nu M)A$$

**The inert species** The null - or inert - species $\mathbf{0}$ represents a molecule which will no longer undergo any reactions.

**Species invocation** Species invocation allows you to build species which refer to themselves, or other named species. Given some definition of $D$ ($D(\vec{y}) \overset{\text{def}}{=} A$), any invocation of this species ($D(\vec{a})$) is replaced with $A$, with all names in $y$ replaced with their corresponding names in $a$.

**Parallel composition** Parallel composition $A \,|\, B$ describes two species $A$ and $B$ which may both participate in reactions. Note that these species may react with other molecules in the system, *or each other*.

**Guarded choice** Guarded choice, written as $\Sigma_{i=0}^{n} \pi_i.A_i$ or $\pi_0.A_0 + \cdots + \pi_n.A_n$, denotes a series of mutually exclusive reactions. The *prefix* $\pi$ describes the interaction which will take place, with species $A$ being the reaction's product.

**Restriction** Name restriction $(\nu M)A$ introduces new names into the current scope, making them accessible only to $A$. This allows for dynamic interaction between molecules.

An associated *affinity network* $\langle M, f\rangle$ defines the set of bound names $M$, along with a symmetric function $f : M \times M \to \mathbb{R}_{\geq 0}$. This function defines the 'affinity' between two names, effectively describing how well they interact with each other.

### 2.1.1 Example of an enzyme system

One useful example, which I refer to throughout this document, is the simple enzyme and substrate system pictured in Figure 2.1.

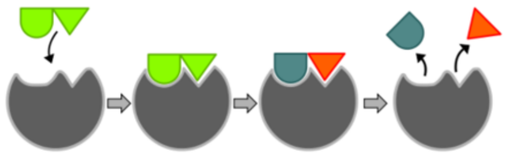
Figure 2.1: The reaction between enzyme and substrate. The enzyme (grey) and substrate (green) bind, producing two products (blue and orange).[6].

The enzyme system starts with two molecules, the substrate $S$ and enzyme $E$. These bind together to form the complex $C$, as seen in Figure 2.1. Once the two molecules have bound together, they react to form the original enzyme (which may partake in further reactions) and two products $P_1$, $P_2$, which degrade at their own rate.

This behaviour of the substrate, enzyme and products can be expressed in the continuous $\pi$-calculus as seen in Figure 2.2. While this system is relatively simple, it makes use of all the features of $c\pi$.

$$\begin{aligned} S &\overset{\text{def}}{=} s(x,y).(x.S + y.(P_1 \mid P_2)) \\ E &\overset{\text{def}}{=} \nu(u,r,t : M)e\langle u,r\rangle.t.E \\ P_1 = P_2 &\overset{\text{def}}{=} \tau@k_{\text{degrade}}.\mathbf{0} \\ \\ \Pi &\overset{\text{def}}{=} c_1 \cdot E \parallel c_2 \cdot S \end{aligned}$$

Figure 2.2: Our enzyme system as a series of $c\pi$ expressions.

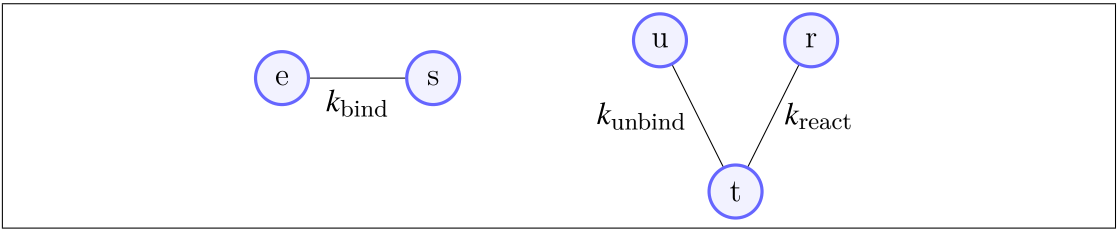

Figure 2.3: The global affinity network *Aff* and local affinity network $M$.

Given the definitions of species and substrate, as well as the affinity networks in Figure 2.3, one can determine what transitions the system undergoes and at what rate those transitions occur.

Figure 2.4 demonstrates how the reactions in Figure 2.1 translate to transitions between various species. One can see the enzyme and substrate reacting at rate $k_{\text{bind}}$ to form some complex, which then goes on to participate in further reactions.

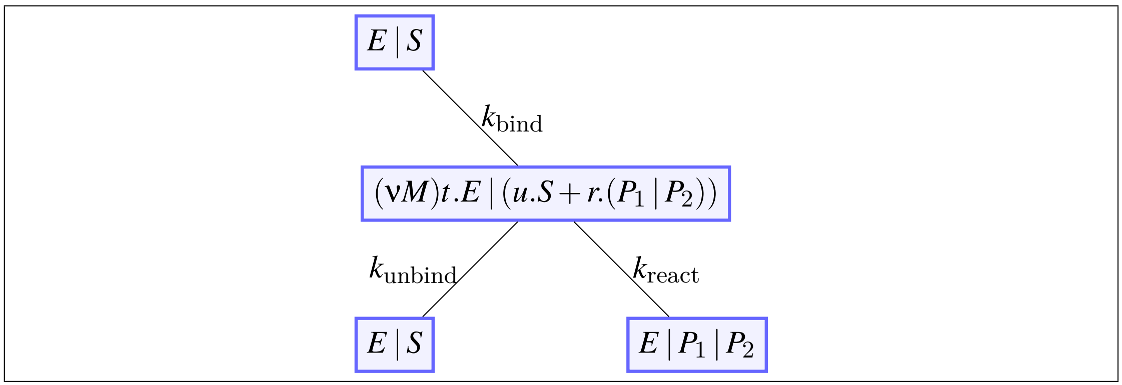

Figure 2.4: Transitions the enzyme system undertakes.

## 2.2 The L∃∀N Theorem Prover

Lean is an open source theorem prover and programming language developed by Microsoft research[5]. Like Coq and Agda, Lean is based on the theory of dependent types, using the Calculus of Constructors with inductive types[7, 8].

Lean's syntax is somewhat reminiscent of ML and Caml, featuring many familiar syntactical constructs like `let` and `match`. Top level definitions are introduced with the `def` keyword.

```
/-- Define a variable "zero" with the type "ℕ" equal to "0". -/
def zero : ℕ := 0

/-- Define a function which takes natural x as a parameter and returns x + 1. -/
def add_one (x : ℕ) := x + 1

example : ℕ := add_one zero
```

Inductive types are introduced by the `inductive` keyword. Here our `tree` type is parameterised by a type variable α.

```
inductive tree (α : Type) : Type
| empty {} : tree
| node : tree → α → tree → tree

-- Constructors are namespaced by default, so must be accessed using `tree.zero'
example : tree ℕ := tree.node tree.empty zero tree.empty
```

Such inductive definitions can then be pattern matched upon, either using the `match` expression or within a function definition.

```
def tree.append {α : Type} : tree α → tree α → tree α
| tree.empty b := b
| (tree.node l x r) b := tree.node l x (tree.append r b)
```

**Type classes** Lean also supports type classes, using them extensively in the standard library. For example, the `++` operator can be overloaded by providing an instance of the `has_append` class.

```
namespace tree
  variable (α : Type)

  instance : has_append (tree α) := ⟨ tree.append ⟩
end tree
```

**Tactics** Lean has support for tactics. These are separate procedures which, instead of working on values, work on the current 'tactic state', building up an appropriate proof or program for the current goal. Lean, and its companion library mathlib[9], come with a wealth of existing tactics. New ones can be added within Lean itself, making it its own metalanguage.

```
-- Using the simp tactic to find a proof that `a + b + c' equals 'b + a + c'.
example (a b c : ℕ) : a + b + c = b + a + c := by simp
```

**Computational irrelevance** One major distinguishing feature from Coq and Agda, is that Lean's theory also introduces the axiom of proof irrelevance[10]. This means that any two propositions (values of sort `Prop`) are indistinguishable in a computational context.

A consequence of this is that we cannot write a non-trivial function which consumes a proposition and yields a non-proposition value. For instance, given the proposition $\exists\, x, x > 0$, we cannot write a function which extracts the $x$.

```
-- induction tactic failed, recursor 'Exists.dcases_on' can only eliminate into Prop
example : (∃ (x : ℕ), x > 0) → ℕ
| ⟨ x, h ⟩ := x
```

This property is incredibly useful, as it allows us to safely use classical logic in proof contexts, while not affecting the computability of the system as a whole.

**Quotients** Quotients allow us to define override equality for a type, defining it using an equivalence relation. If we have some type $\alpha$ and equivalence relation

$R : \alpha \times \alpha$, we can define the quotient as the set of elements of $\alpha$ modulo $R$. Thus, if two values $a$ and $b$ are related by $R$, then their quotients are equal.

Lean provides built-in support for this, where `quotient R` is the quotient $\alpha$ modulo $R$ and `⟦ x ⟧` represents the equivalence class of `x : α`.

```
def α' := quotient R /- A quotient under relation R -/

def x' : α' := ⟦ x ⟧ /- The equivalence class of x -/

/- If x ≈ y, then their quotients are equal -/
example : ⟦ x ⟧ = ⟦ y ⟧ := quotient.sound_
```

# Chapter 3

# The Continuous π-calculus

## 3.1 Names

Kwiatkowski and Stark[4, §2.1] define the set of *names* $\mathcal{N}$ as a countably infinite set of lowercase letters $(a, b, \ldots)$. While named variables are easy to reason about, such a scheme ends up being difficult to formalise.

The well-established alternative is to use de Bruijn indices[11, 12]. Instead of using a set of names, variables are represented as a natural representing the number of binders between a variable's definition and usage. However in the case of the cπ-calculus, all binders declare a vector of names. In order to represent this, we view our variables as a pair of naturals; one determining the binding depth, and another representing the "index" of any one variable within the name vector.

One remaining issue with de Bruijn indices is that it is essential to update the index of any variables when removing or adding a new binder. While forgetting to do so will mean later proofs do not go through, it is possible to enforce this directly with an intrinsically typed version of names. Every term, including names, prefixes and species, are indexed by context, which represents the current binder depth and each binders' arity. When one adds or removes a binder, the context changes, and so you are forced to rename all variables.

```
inductive context : Type
| nil : context
| extend : ℕ → context → context

inductive name : context → Type
| zero   {Γ} {n : ℕ} : fin n → name (context.extend n Γ)
| extend {Γ} {n : ℕ} : name Γ → name (context.extend n Γ)
```

Listing 3.1: The definition of contexts and names.

```
/-- A telescope may introduce 0 or 1 binders. -/
inductive telescope : Type
| extend : ℕ → telescope
| preserve : telescope

/-- Apply a telescope to a context. -/
def telescope.apply : telescope → context → context
| (telescope.extend n) Γ := context.extend n Γ
| telescope.preserve Γ := Γ

/-- A prefix expression. -/
inductive prefix_expr (ℍ : Type) : context → telescope → Type
| communicate {} {Γ} (a :  name Γ) (b : list (name Γ)) (y : ℕ)
  : prefix_expr Γ (telescope.extend y)
| spontaneous {} {Γ} (k : ℍ) : prefix_expr Γ telescope.preserve

-- Define some additional notation, and sugar
notation a `#(` b ` ; ` y `)` := prefix_expr.communicate a b y
notation a `#(` y `)` := prefix_expr.communicate a [] y
notation a `#⟨` b `⟩` := prefix_expr.communicate a b 0
notation a `#` := prefix_expr.communicate a [] 0
```

Listing 3.2: Our Lean definition of prefix expressions.

## 3.2 Prefix Expressions

As mentioned in Section 2.1, prefix expressions are used to represent the reaction a species may participate in. Prefixes come in two forms, *communication* and *spontaneous*.

$$\pi ::= a(\vec{b};\vec{y}) \mid \tau@k$$

**Communication prefix** A communication prefix $a(\vec{b};\vec{y})$ models the interaction between two molecules. This is a combination of sending and receiving in the traditional π-calculus; we send the values in vector $\vec{b}$ across channel $a$, and receive another series of values, binding them to the names in $\vec{y}$.

**Spontaneous prefix** A spontaneous or silent prefix $\tau@k$ models a reaction which occurs without any external interaction, at rate $k \in \mathbb{R}_{\geq 0}$.

Note that the communication prefix binds additional variables, while the spontaneous prefix does not. Thus we need some way of extending a context depending on the kind of prefix. This is done by an additional "telescope" type, which describes how a context should be extended. Prefixes are then indexed by their appropriate telescope.

An alternative formalisation would be to view spontaneous prefix as binding 0 variables. This would allow prefixes to be indexed by the arity of their binder instead, removing the need for telescopes and the various definitions and lemmas related to them. However, this the introduces complexities to the representation of transitions later on[1].

Prefixes, and by extension species and processes, are parameterised by an arbitrary type $\mathbb{H}$. This represents the rate of a reaction (either within an affinity matrix or a spontaneous prefix). The continuous $\pi$ paper uses the set of real numbers $\mathbb{R}$ for this, but we can generalise for any commutative ring, which proves useful when dealing with differential equations.

## 3.3 Species

As we described in Section 2.1, species are defined by the following grammar.

$$A, B ::= \mathbf{0} \mid D(\vec{a}) \mid \Sigma_{i=0}^{n} \pi_i.A_i \mid A \,|\, B \mid (\nu M)A$$

However, it is not immediately clear how such a definition should be translated into Lean.

Here, guarded choice is defined as a list of prefix-species pairs, with species $A$ occurring within the binder potentially introduced by prefix $\pi$. Representing such a type on its own in Lean is straightforward. However, we need to do so within the definition of species themselves. Lean does not support this nesting[2], and so we must find an alternative formulation. I choose to formalise species by separating out the elements of guarded choice into a separate, mutually recursive inductive type, giving the following definition.

$$\begin{aligned} A, B &::= \mathbf{0} \mid D(\vec{a}) \mid \Sigma\, As \mid A \,|\, B \mid (\nu M)A \\ \text{As} &::= [] \mid \pi.A :: As \end{aligned}$$

While this can be represented in Lean, mutually-recursive inductive types do not have first-class support. Instead, they are translated to a single inductive type indexed by a 'tag type' which determines to which type name each constructor belongs. The original definitions then wrap this internal combined type. As a result, tactics such as `induction` or `cases` do not work on mutually recursive types, which makes writing proofs harder than necessary.

In order to avoid this complexity, I apply this translation manually. I introduce a `kind` type, which determines whether each constructor is part of our species (A, B) or an element in our list of choices (As). Our flattened type `whole` is then indexed by this `kind`.

[1]While the proofs are largely the same, Lean finds them much harder to reason about and frequently times out.

[2]This is possible within systems such as Agda, but it does not result in simpler proofs. One cannot map or fold over such a list, as it prevents showing well-foundedness of a proof, and so one must write the mutually-recursive proofs, much like in Lean.

```
inductive kind
| species
| choices

/-- The set of species and choices. -/
inductive whole (ℍ : Type) (ω : context) : kind → context → Type
  /- Species -/
| nil      {} {Γ} : whole kind.species Γ
| apply    {} {Γ} {n} : reference n ω → vector (name Γ) n → whole kind.species Γ
| choice      {Γ} : whole kind.choices Γ → whole kind.species Γ
| parallel    {Γ} : whole kind.species Γ → whole kind.species Γ → whole kind.species
↪  Γ
| restriction {Γ} (M : affinity ℍ) :
    whole kind.species (context.extend M.arity Γ) → whole kind.species Γ

  /- Elements in our list of choices -/
| empty {} {Γ} : whole kind.choices Γ
| cons     {Γ} {f} (π : prefix_expr ℍ Γ f) :
    whole kind.species (f.apply Γ) → whole kind.choices Γ → whole kind.choices Γ

/-- An alias for species within the `whole' datatype. -/
def species (ℍ : Type) (ω : context) := @whole ℍ ω kind.species

/-- An alias for choices within the `whole' datatype. -/
def choices (ℍ : Type) (ω : context) := @whole ℍ ω kind.choices
```

Listing 3.3: Species and guarded choice within Lean.

Our Lean definition of species is given within Listing 3.3. We also define notation for parallel composition (`|ₛ`) and guarded choice (`Σ##`).

## 3.3.1 Expressing our example in Lean

Recalling our example enzyme system, we can now translate the equations in Figure 2.2 into equivalent expressions within Lean.

For instance, the substrate $S \stackrel{\text{def}}{=} s(x,y).(x.S + y.(P_1 \mid P_2))$ can be written as.

```
def S_ : species ℝ ω Γ :=
  s #( 2 ) • Σ# ( whole.cons (x #) (apply S ∅)
              $ whole.cons (y #) (apply P₁ ∅ |ₛ apply P₂ ∅)
              $ whole.empty ) )
```

Likewise our enzyme $E \stackrel{\text{def}}{=} \nu(u,r,t : M)e\langle u,r\rangle.t.E$ becomes

```
def E_ : species ℝ ω Γ :=
  ν(M) (name.extend e #( [u, r] )) • (name.extend t # • apply E ∅)
```

## 3.3.2 Free variables and renaming

Now we have definitions for the syntactic elements of the calculus, we are able to begin work on the main elements of [4]. One definition which, while not explicitly stated, is assumed, is the notion of renaming.

Given some function $\rho$ : `name` $\Gamma \to$ `name` $\Delta$, one should be able to lift this function to map over prefixes and species in contexts $\Gamma$, $\Delta$.

```
/-- Rename prefix expressions -/
def prefix_expr.rename {Γ Δ} : Π {f}, (name Γ → name Δ)
  → prefix_expr ℍ Γ f → prefix_expr ℍ Δ f
| f (a#(b; y)) ρ := (ρ a)#(list.map ρ b; y)
| f τ@k ρ := τ@k

/-- Rename species and choices -/
def species.rename : Π {Γ Δ k}, (name Γ → name Δ)
  → whole ℍ ω k Γ → whole ℍ ω k Δ
/- ... -/
```

While these renaming functions technically allow us to map between arbitrary contexts, in practice we find the only useful operation is to rename into an extended context by incrementing every de Bruijn index ($\rho : \Gamma \to$ `context.extend n` $\Gamma$). However, sometimes we would like to be able to perform the inverse. For instance, consider the species $\nu(M)D(a)$, where $a$ is not bound by $M$. As our binder is never used, ideally we might be able to drop it, rewriting the expression to $D(a)$. However, it is not clear how this could be done using our current renaming functions. We can extend a context, but we cannot shrink it.

My approach is to modify the renaming functions to take an additional proof that the given name is used within the current term. In the aforementioned case, we know that $M$ does not occur within $D(a)$, and so we can use that information when renaming.

In practice, our implementation is a little less refined. Rather than determining whether a specific name is used within a term, we simply check whether *any* variable within a binder is used. This is sufficient for our needs, and makes some definitions a little simpler.

```
/-- Levels represent all names bound by a specific binder. -/
inductive level : context → Type
| zero   {Γ} {n} : level (context.extend n Γ)
| extend {Γ} {n} : level Γ → level (context.extend n Γ)
```

Given the notion of levels, I can then define the notion of a level being "free" in a name, prefix and species. The definitions of these are fairly obvious.

```
/-- Determine if a level is free within a prefix -/
def prefix_expr.free_in : ∀ {Γ} {f}, level Γ → prefix_expr ℍ Γ f → Prop
| ._ ._ l (a#(b; y)) := l ∈ a ∨ ∃ x ∈ b, l ∈ x
| ._ ._ l τ@_ := false
```

Now we have some notion of free variables, we are able to define a refined version of renaming. Our renaming function $\rho$ now takes a name $a$, and a witness that $a$ is used within the prefix to rename.

```
/-- Rename all names within a prefix expression, providing some witness that
    this variable is free within it. -/
def prefix_expr.rename_with {Γ Δ} :
  ∀ {f} (π : prefix_expr ℍ Γ f)
  , (∀ (a : name Γ), name.to_level a ∈ π → name Δ)
```

```
/-- Decrement the level of every variable. -/
def drop_var {Γ} {n}
  (P : level (context.extend n Γ) → Prop) (p : (¬ P level.zero))
  : Π a, P (name.to_level a) → name Γ
| (name.zero idx) q := by contradiction
| (name.extend a) _ := a

/- Optionally construct a restriction, only adding a binder if it is
   needed. -/
example {Γ : context} (M : affinity ℍ)
    (A : species ℍ ω (context.extend M.arity Γ))
  : species ℍ ω Γ
  := if h : level.zero ∈ A then ν(M) A
     else rename_with A (drop_var (λ l, l ∈ A) h)
```

Listing 3.4: Using our renaming functions to drop a binder.

$$
\begin{aligned}
A \mid \mathbf{0} &\equiv A && \\
A \mid B &\equiv B \mid A && \\
(A \mid B) \mid C &\equiv A \mid (B \mid C) && \\
\Sigma_{i=0}^{n} \pi_i . A_i &\equiv \Sigma_{i=0}^{n} \pi_{\sigma_i} . A_{\sigma_i} && \text{where } \sigma \text{ forms a permutation} \\
(\nu M)(A \mid B) &\equiv A \mid (\nu M) B && \text{where } M \notin A \\
(\nu M) A &\equiv A && \text{where } M \notin A \\
(\nu M)(\nu N) A &\equiv (\nu N)(\nu M) A &&
\end{aligned}
$$

Figure 3.1: The rules of structural congruence for species.

```
  → prefix_expr ℍ Δ f
/- ... -/
```

Given this, we can finally construct a function to remove binding levels (Listing 3.4). When given a proof that no variables in the closest binder are used, we can decrease the level of all other variables. We make use of this function in Section 5.2 when normalising terms.

### 3.3.3 Structural congruence

Continuous π defines a *structural congruence* relation over species. This forms an equivalence class of species which are syntactically distinct, but have identical semantics. While this relation is easy to express in writing, it requires a little more care to formalise.

Structural congruence is defined in [4, §2.1] as the transitive closure of the rules in Figure 3.1.

This can be expressed as an inductive type indexed by the species which are structurally congruent. Each congruence rule then becomes a constructor within this type.

```
inductive equivalent : ∀ {Γ} (A B : species ℍ ω Γ), Type
```

The first three rules, which declare that parallel composition forms a commutative monoid, map relatively directly to Lean.

```
| parallel_nil₁   {Γ} {A : species ℍ ω Γ}
  : equivalent (A |ₛ nil) A
| parallel_symm   {Γ} {A B : species ℍ ω Γ}
  : equivalent (A |ₛ B) (B |ₛ A)
| parallel_assoc₁ {Γ} {A B C : species ℍ ω Γ}
  : equivalent ((A |ₛ B) |ₛ C) (A |ₛ (B |ₛ C))
```

It is worth noting that structural congruence, as it is an equivalence relation, is symmetric. Thus, $A \equiv A \,|\, \mathbf{0}$ is also true. While it would be possible to state this using a `symm` constructor, this introduces difficulties later on. For instance, we often need to prove $A \equiv B \to P(A) \to P(B)$ for some property $P$. However, symmetry requires that we must also show $P(B) \to P(A)$, which complicates the proof significantly. In order to avoid this, we additionally define a reversed version of all rules.

```
| parallel_nil₂   {Γ} {A : species ℍ ω Γ}
  : equivalent A (A |ₛ nil)
| parallel_assoc₂ {Γ} {A B C : species ℍ ω Γ}
  : equivalent (A |ₛ (B |ₛ C)) ((A |ₛ B) |ₛ C)
```

Several of the congruence rules for restriction require that the affinity matrix is not used within some species. While we could represent this with our previous notion of free variables (or rather, free binders), we instead use `rename` to increase the binding level of a term.

```
| ν_parallel₁ {Γ} (M : affinity ℍ)
    {A : species ℍ ω Γ} {B : species ℍ ω (context.extend M.arity Γ)}
  : equivalent (ν(M) (rename name.extend A |ₛ B)) (A |ₛ ν(M)B)
| ν_drop₁ {Γ} (M : affinity ℍ) {A : species ℍ ω Γ}
  : equivalent (ν(M) (rename name.extend A)) A
| ν_swap₁ {Γ} (M N : affinity ℍ)
    {A  : species ℍ ω (context.extend N.arity (context.extend M.arity Γ))}
  : equivalent (ν(M)ν(N) A) (ν(N)ν(M) rename name.swap A)
```

As before, we also have symmetric versions of these rules.

While our set of rules so far give us some form of relation, we must add a series of additional constructors to have a complete definition of structural congruence. We define a series of compatibility rules, which make our definition a congruence relation. For instance, if $A \equiv A'$, then $A \,|\, B \equiv A' \,|\, B$.

```
| ξ_parallel₁ {Γ} {A A' B : species ℍ ω Γ}
  : equivalent A A' → equivalent (A |ₛ B) (A' |ₛ B)
| ξ_parallel₂ {Γ} {A B B' : species ℍ ω Γ}
  : equivalent B B' → equivalent (A |ₛ B) (A |ₛ B')
| ξ_restriction {Γ} (M : affinity ℍ)
    {A A' : species ℍ ω (context.extend (M.arity) Γ)}
  : equivalent A A' → equivalent (ν(M) A) (ν(M) A')
| ξ_choice_here {Γ} {f} (π : prefix_expr ℍ Γ f)
    {A A' : species ℍ ω (f.apply Γ)} {As : choices ℍ ω Γ}
  : equivalent A A'
  → equivalent (Σ# (whole.cons π A As)) (Σ# (whole.cons π A' As))
| ξ_choice_there {Γ} {f} (π : prefix_expr ℍ Γ f)
    {A : species ℍ ω (f.apply Γ)} {As As' : choices ℍ ω Γ}
  : equivalent (Σ# As) (Σ# As')
  → equivalent (Σ# (whole.cons π A As)) (Σ# (whole.cons π A As'))
```

Finally, we provide reflectivity and transitivity explicitly.

```
| refl  {Γ} {A : species ℍ ω Γ} : equivalent A A
| trans {Γ} {A B C : species ℍ ω Γ}
  : equivalent A B → equivalent B C → equivalent A C
```

Our `equivalent` type now defines the smallest relation that contains structural congruence. We then show that this relation is symmetric, transitive and reflexive, which allows us to define a quotient type of species.

```
instance {Γ} : setoid (species ℍ ω Γ) := -- ...

/- A set of structurally congruent species. -/
def species' (ℍ : Type) (ω Γ : context) := quotient (@species.setoid ℍ ω Γ)
```

### 3.3.4 Prime species

When introducing the syntax of species in Section 2.1, we describe parallel composition as representing two species in a mixture. When reasoning about species and their semantics, it is useful to be able to distinguish between those which are mixtures of independent species, and those which are not.

Kwiatkowski et. all. [4, §2.2] defines a species $A$ to be *prime* if it cannot be decomposed into a parallel composition of non-trivial species. More precisely, $A$ is prime if $A \not\equiv \mathbf{0}$, and, if $A \equiv B \mid C$, then $B$ or $C$ is congruent to $\mathbf{0}$.

```
def prime (A : species ℍ ω Γ) : Prop
  := ¬ A ≈ nil ∧ ∀ B C, A ≈ (B |ₛ C) → B ≈ nil ∨ C ≈ nil

def prime_species (ℍ : Type) (ω Γ : context) : Type
  := { A : species ℍ ω Γ // prime A }
```

Referring back to our example in Section 2.1.1, we can that both $S$ and $E$ are prime, but our intermediate product $(\nu M)t.E \mid (u.S + r.(P_1 \mid P_2))$ is not.

Intuitively, we can see that it should be possible to 'decompose' any species into a list of prime species. However, defining such a function proves problematic. For now, we will assume it exists, and revisit this within Chapter 5.

```
constant prime_decompose {Γ} : species ℍ ω Γ → list (prime_species ℍ ω Γ)
```

## 3.4 Processes

While species can be viewed in isolation, ultimately we wish to model a solution of multiple molecules, each with an initial concentration. For that purpose, we define *processes*, a collection of species with some associated concentrations.

```
inductive process (ℂ ℍ : Type) (ω Γ: context) : Type
| one      : ℂ → species ℍ ω Γ → process
| parallel : process → process → process

infix ` ◯ `:60 := process.one
infixr ` |ₚ `:50 := process.parallel
```

Listing 3.5: Our definition of processes in Lean.

$$
\begin{aligned}
P \parallel (c \cdot \mathbf{0}) &\equiv P \\
P \parallel Q &\equiv Q \parallel P \\
P \parallel (Q \parallel R) &\equiv (P \parallel Q) \parallel R \\
(c \cdot A) \parallel (d \cdot A) &\equiv (c+d) \cdot A \\
c \cdot A &\equiv c \cdot B \qquad \text{where } A \equiv B \\
c \cdot (A \,|\, B) &\equiv c \cdot A \parallel c \cdot B
\end{aligned}
$$

Figure 3.2: The rules of structural congruence for processes.

Processes are defined by the following grammar, which translates to Lean incredibly intuitively (Listing 3.5).

$$P ::= c \cdot A \;\mid\; P \parallel Q$$

Following from our intuition of processes, as a mixture of multiple species, it makes sense to define a congruence relation over processes. Much like species, this relation defines mixture composition ( $\parallel$ ) as a commutative monoid, with several additional properties relating parallel composition and concentrations (Figure 3.2). This can be expressed in Lean in much the same way that structural congruence over species was (Listing 3.6).

With that, we have completed our mechanisation of cπ's syntax.

```
inductive equiv : process ℂ ℍ ω Γ → process ℂ ℍ ω Γ → Prop
| refl  {A}     : equiv A A
| trans {A B C} : equiv A B → equiv B C → equiv A C
| symm  {A B}   : equiv A B → equiv B A

-- Compatibility and parallel rules as with species.

| join  {A} {c d} : equiv (c ◯ A |ₚ d ◯ A) ((c + d) ◯ A)
| split {A B} {c : ℂ} : equiv2 (c ◯ (A |ₛ B)) (c ◯ A |ₚ c ◯ B)
```

Listing 3.6: Structural congruence of processes in Lean.

# Chapter 4

# Semantics of cπ

cπ's semantics are defined in two states, discrete and continuous. The *discrete semantics* are defined using a *labelled transition system*, which describes the behaviour of each process, and what it may evolve in to. The resulting transitions are then used within the *continuous semantics* to compute a vector space which describes how concentrations change over time, and from there compute the differential equations for each process.

## 4.1 Concretions

When modelling the semantics, some transitions produce a *concretion*[13, §5.5][14, §3.3.1]. A concretion can be viewed as a species which has the potential to take part in a reaction, but what it will react with has not yet been determined.

At their core, concretions can be thought of as corresponding with the 'communication' prefix expression (Section 3.2). They are formed from a base 'abstraction' term $(\vec{b};\vec{y})A$. The abstraction communicates on some channel, sending the values $\vec{b}$, and receiving another series of values, which are bound to $\vec{y}$. This base term is then contained within a context of other species.

$$F,G ::= (\vec{b};\vec{y})A \;\mid\; F\,|\,A \;\mid\; A\,|\,F \;\mid\; (\nu M)F$$

When we come to work with concretions, it is useful to know the arity of $\vec{b}$ and $\vec{y}$, to ensure two concretions are 'compatible'. To facilitate that, we encode the two arities into the type signature of concretions (Listing 4.1).

As with species and processes, concretions also have a structural congruence (Figure 4.1). As one might expect, this relation has very similar congruence rules to that of species. This can be defined in Lean in much the same way as our equivalence on species or processes was.

```
inductive concretion (ℍ : Type) (ω : context) : context → ℕ → ℕ → Type
| apply {Γ} {b} (bs : vector (name Γ) b) (y : ℕ)
  : species ℍ ω (context.extend y Γ)
  → concretion Γ b y
| parallel₁ {Γ} {b y} : concretion Γ b y → species ℍ ω Γ → concretion Γ b y
| parallel₂ {Γ} {b y} : species ℍ ω Γ → concretion Γ b y → concretion Γ b y
| restriction {Γ} {b y} (M : affinity ℍ)
  : concretion (context.extend M.arity Γ) b y
  → concretion Γ b y

notation `#(` b ` ; ` y `)` A := concretion.apply b y A

infixr ` |₁ ` := concretion.parallel₁
infixr ` |₂ ` := concretion.parallel₂

notation `ν'(` M `) ` A := concretion.restriction M A

inductive equiv : ∀ {Γ} {b y}, concretion ℍ ω Γ b y → concretion ℍ ω Γ b y → Prop
```

Listing 4.1: The definition of concretions in Lean.

$$
\begin{aligned}
F \mid \mathbf{0} &\equiv F \\
F \mid A &\equiv A \mid F \\
(F \mid A) \mid B &\equiv F \mid (A \mid B) \\
(A \mid F) \mid F &\equiv A \mid (F \mid B) \\
F \mid A &\equiv F \mid B && \text{where } A \equiv B \\[1em]
(\nu M)(A \mid F) &\equiv A \mid (\nu M) F && \text{where } M \notin A \\
(\nu M)(F \mid A) &\equiv F \mid (\nu M) A && \text{where } M \notin F \\
(\nu M) F &\equiv F && \text{where } M \notin F \\
(\nu M)(\nu N) F &\equiv (\nu N)(\nu M) F \\[1em]
(\vec{b};\vec{y}) A &\equiv (\vec{b};\vec{y}) B && \text{where } A \equiv B \\
(\vec{b};\vec{y})(A \mid B) &\equiv A \mid (\vec{b};\vec{y}) B && \text{where } \vec{y} \notin A
\end{aligned}
$$

Figure 4.1: The rules of structural congruence for concretions.

```
private def pseudo_apply_app {a b} :
  ∀ {Γ}, vector (name Γ) a → species ℍ ω (context.extend b Γ)
  → concretion ℍ ω Γ b a → species ℍ ω Γ
| Γ as A (#(bs; y) B) :=
  species.rename (name.mk_apply bs) A |ₛ species.rename (name.mk_apply as) B
| Γ as A (F |₁ B) := pseudo_apply_app as A F |ₛ B
| Γ as A (B |₂ F) := B |ₛ pseudo_apply_app as A F
| Γ as A (ν'(M) F) := ν(M) (pseudo_apply_app _ (species.rename _ A) F)

def pseudo_apply {a b} :
  ∀ {Γ}, concretion ℍ ω Γ a b → concretion ℍ ω Γ b a → species ℍ ω Γ
| Γ (#(bs; y) A) F' := pseudo_apply_app bs A F'
| Γ (F |₁ A) F' := pseudo_apply F F' |ₛ A
| Γ (A |₂ F) F' := A |ₛ pseudo_apply F F'
| Γ (ν'(M) F) F' := ν(M) (pseudo_apply F (rename name.extend F'))
```

Listing 4.2: Pseudo application in Lean, defined as two separate functions.

### 4.1.1 Pseudo-application

Recall that concretions represent a species which has the potential to react. When two concretions react with each other, the resulting product is determined via *pseudo-application*, denoted as $F \circ G$.

The base case of pseudo-application operates on two abstraction terms, $(\vec{a};\vec{x})A \circ (\vec{b};\vec{y})B$:

$$(\vec{a};\vec{x})A \circ (\vec{b};\vec{y})B \stackrel{\text{def}}{=} A\{\vec{b}/\vec{x}\} \,|\, B\{\vec{a}/\vec{y}\}$$

Such an application is only defined when $|\vec{a}| = |\vec{y}|$ and $|\vec{b}| = |\vec{x}|$. However, as we track the arity of concretions within the type signature, it is easy to enforce this requirement.

All other cases are defined by induction over the structure of concretions.

$$\begin{aligned}
(\vec{a};\vec{x})A \circ (F \,|\, B) &\stackrel{\text{def}}{=} (((\vec{a};\vec{x})A \circ F)) \,|\, B & \qquad (A \,|\, F) \circ G &\stackrel{\text{def}}{=} A(F \circ G) \\
(\vec{a};\vec{x})A \circ (B \,|\, F) &\stackrel{\text{def}}{=} B \,|\, ((\vec{a};\vec{x})A \circ F) & (F \,|\, A) \circ G &\stackrel{\text{def}}{=} (F \circ G) \,|\, A \\
(\vec{a};\vec{x})A \circ (\nu M)F &\stackrel{\text{def}}{=} (\nu M)((\vec{a};\vec{x})A \circ F) & (\nu M)(F) \circ G &\stackrel{\text{def}}{=} (\nu M)(F \circ G)
\end{aligned}$$

You can view this definition as first recurring over the left-hand concretion, until the base case is reached, then recurring on the right-hand concretion. We choose to make this explicit in our definition (Listing 4.2), splitting the work into two functions. Doing so means the any recursion only operates on one concretion, simplifying the work needed to show the recursion terminates.

```
| Γ (F |₁ A) (ν'(M) G) := begin
  -- ...
  ... ≈ ((ν(M) pseudo_apply (rename name.extend F) G) |ₛ A)
  ... ≈ ((ν(M) pseudo_apply G (rename name.extend F)) |ₛ A)
      : ξ_parallel₁ (ξ_restriction M (pseudo_apply.symm (rename name.extend F) G))
  -- ...
end
| Γ (ν'(M) F) (G |₁ B) := begin
  -- ...
  ... ≈ ((ν(M) pseudo_apply F (rename name.extend G)) |ₛ B)
  ... ≈ ((ν(M) pseudo_apply (rename name.extend G) F) |ₛ B)
      : ξ_parallel₁ (ξ_restriction M (pseudo_apply.symm F (rename name.extend G)))
  -- ...
end
```

Listing 4.3: Two cases in our proof that $F \circ G \equiv G \circ F$.

I also show that pseudo-application commutes with renaming. In turn, this can be used to show that pseudo application is commutative modulo congruence, namely $F \circ G \equiv G \circ F$.

As [4, §2.2] states, this can be shown by induction over the definition of $F \circ G$, and thus by induction over $F$ and $G$ themselves. However, attempting to perform induction on two concretions at once can prove troublesome.

As Lean is intended for theorem proving, it requires that every recursive function has a proof that it is *well-founded*, or rather that it terminates. If such a requirement was not included, it would be possible to write a proof for $\perp$ simply by recurring forever.

Most of the time, Lean is able to show that a function is well-founded, as some term will shrink. One peels away a single constructor at a time, and recurs on the (smaller) child terms. However, this is not always the case. In our definitions `pseudo_apply` and `pseudo_apply_app` from Listing 4.2, one concretion shrinks but, in the presence of binders, the other species or concretion grows larger, as we must rename it.

Lean allows you to define custom *measures* for your recursive function, as well as tactics for showing such measures decrease on every recursive call. For instance, in our definition of `pseudo_apply_app`, we can define our measure as the size of the concretion.

```
using_well_founded {
  rel_tac := λ _ _,
  -- x is a tuple of our arguments (Γ, as, A, F). x.snd.snd.snd corresponds to F.
  `[exact ⟨_, measure_wf (λ x, concretion.sizeof ℍ ω x.fst b a x.snd.snd.snd ) ⟩ ],
  dec_tac := tactic.fst_dec_tac,
}
```

Such a measure is not suitable for our proof of commutativity. For instance, consider the case where $F$ is a parallel composition and $G$ a $\mu$ binder. The proofs for the two cases look relatively similar (Listing 4.3).

```
private def depth : ∀ {Γ} {b y}, concretion ℍ ω Γ b y → ℕ
| _ _ _ (#(_; _) _) := 1
| _ _ _ (F |₁ _) := depth F + 1
| _ _ _ (_ |₂ F) := depth F + 1
| _ _ _ (ν'(M) F) := depth F + 1

private lemma depth.over_rename :
  ∀ {Γ Δ} {b y} (ρ : name Γ → name Δ) (F : concretion ℍ ω Γ b y)
  , depth F = depth (rename ρ F) := /- ... -/
```

Listing 4.4: A function to measure the depth of a concretion, along with a proof that depth is preserved across renames,

However, in one case we recur using `symm (rename name.extend F) G)` and the other using `symm F (rename name.extend G)`. There is no built-in measure which allows us to do this. While $F$ or $G$ may shrink, the other concretion grows.

The solution is to define a `depth` function (Listing 4.4), which describes how many layers our concretion has until it reaches a species. We can then show that depth is preserved over renaming. This function is then used to define a custom measure for `pseudo_app.symm`, as the depth of $G$ will always decrease.

Finally, we wish to show that pseudo application commutes with equivalence. Or rather, if $F \equiv F'$ and $G \equiv G'$ then $F \circ G \equiv F' \circ G'$.

It is sufficient to show this is true for only one side of the application (namely $F \equiv F' \Rightarrow F \circ G \equiv F' \circ G$), and use commutativity of pseudo-application to show the full lemma.

Finally, we define a type `concretion'`, which is the quotient of concretions modulo structural congruence. Given our previous lemma, we can define a version of pseudo application which operates on equivalence classes of concretions, producing an equivalence classes of species.

```
def pseudo_apply.quotient {Γ a b}
  : concretion' ℍ ω Γ a b → concretion' ℍ ω Γ b a
  → species' ℍ ω Γ
| F G := quotient.lift_on₂ F G
  (λ F G, ⟦ pseudo_apply F G ⟧)
  (λ F G F' G' eqF eqG, quot.sound (pseudo_apply.equiv eqF eqG))
```

## 4.2 Labels and Productions

Transitions describe how a species will evolve or react in a system. Each transition is composed of three terms; a species, which describes the molecule which partakes in this reaction, a *label* which describes the behaviour or rate of this transition, and a *production*, which is the result of this transition.

Labels appear in three forms, *communicate*, *spontaneous* and *affinity*.

```
inductive kind
| species
| concretion

inductive label (ℍ : Type) : context → kind → Type
| apply {} {Γ} (a : name Γ) : label Γ kind.concretion
| spontaneous {Γ} (rate : ℍ) : label Γ kind.species
| of_affinity {} {Γ} (k : upair (name Γ)) : label Γ kind.species

inductive production (ℍ : Type) (ω : context) (Γ : context) : kind → Type
| species (A : species ℍ ω Γ) : production kind.species
| concretion {b y} (F : concretion ℍ ω Γ b y) : production kind.concretion
```

Listing 4.5: Our definition of labels and productions.

**Communicate** The communicate transition $A \xrightarrow{a} F$ converts a species $A$ to a concretion $F$ which will interact using channel $a$. For a specific concretion $(\vec{b};\vec{y})B$, species $A$ communicates on channel $a$, sending $b$ and receiving $y$ before evolving into $B$.

**Spontaneous** A spontaneous transition $A \xrightarrow{\tau@k} B$ represents a species which evolves from $A$ to $B$ at rate $k$, akin to the spontaneous prefix expression.

**Of affinity** An affinity transition $A \xrightarrow{\tau\langle a,b\rangle} B$ represents a species which evolves from $A$ to $B$, but where the rate of this reaction is determined by the affinity between $a$ and $b$. When this reaction occurs within $(\nu M)$, this is replaced by an equivalent transition with a $\tau@M(a,b)$ label.

As transitions may produce a species or concretion, we introduce a `kind` type, indexing transitions using it. This is then used to determine a transition's labels and production, thus ensuring they are well formed.

As affinity networks are symmetric, we have that $\tau\langle a,b\rangle$ is equivalent to $\tau\langle b,a\rangle$. We represent this by defining an unordered pair, and using it within the `of_affinity` constructor (Listing 4.5). We discuss the construction of out `upair` type in more detail within Appendix A.

## 4.3 Transitions

At this point, we have everything needed to describe the transitions a species may undergo. Transitions are defined as a type indexed by their source species, a *lookup function*, the transition's label and the result of this transition.

The lookup function maps references to their corresponding species. We require that any species invocation results in a guarded choice. This does not limit the power of the calculus, but ensures it is possible to enumerate all transitions.

```
def lookup (ℍ : Type) (ω Γ : context) := ∀ n, reference n ω → species.choices ℍ ω
↪  (context.extend n Γ)
```

```
inductive transition :
  Π {Γ} {k}
  , species ℍ ω Γ → lookup ℍ ω Γ → label ℍ Γ k → production ℍ ω Γ k
  → Type
```

**Guarded choice** All transitions originate from guarded choice. Every element of the choice produces a spontaneous or communication label, corresponding to the prefix expression of that element.

$$\frac{0 \leq j \leq n \; \pi_j = a_j(\vec{b}_j;\vec{y}_j)}{\sum_{i=0}^{n} \pi_j.A_i \xrightarrow{a_j} (\vec{b}_j;\vec{y}_j)A_j} \quad \textsc{Choice-1}$$

$$\frac{0 \leq j \leq n \; \pi_j = \tau@k}{\sum_{i=0}^{n} \pi_j.A_i \xrightarrow{\tau@k} A_j} \quad \textsc{Choice-2}$$

These two rules translate to Lean fairly directly. We mirror the structure of choices, defining constructors `choice_1` and `choice_2` which transition from a `whole.cons` term to a production.

In order to enumerate all arms of a choice, the `ξ_choice` constructor extends the list of choices that this transition operates from.

```
| choice₁ {Γ ℓ} (a : name Γ) {n} (b : list (name Γ)) (b_len : list.length b = n)
    (y : ℕ) (A : species ℍ ω (context.extend y Γ)) (As : species.choices ℍ ω Γ)
  : transition (Σ# species.whole.cons (a#(b; y)) A As)
               ℓ (#a)
               (production.concretion (#⟨ b, b_len ⟩; y) A))

| choice₂ {Γ ℓ} (k : ℍ) (A : species ℍ ω Γ) (As : species.choices ℍ ω Γ)
  : transition (Σ# species.whole.cons (τ@k) A As) ℓ τ@'k (production.species A)

| ξ_choice {Γ ℓ f} {π : prefix_expr ℍ Γ f}
    {A : species ℍ ω (f.apply Γ)} {As : species.choices ℍ ω Γ}
    {k} {l : label ℍ Γ k} {E : production ℍ ω Γ k}
  : transition (Σ# As) ℓ l E
  → transition (Σ# species.whole.cons π A As) ℓ l E
```

For instance, the species $a.A + \tau@k.B$ would produce transitions `choice₁ a _ _ _ _ _` and `ξ_choice (choice₂ k _ _)`.

**'Compatibility' transitions** Any transition from species $A$ may be converted into an equivalent transition from a larger species, such as $A \mid B$ or $(\nu M)A$. When doing so, both the input species and resulting production are wrapped in the same constructor, while the label is preserved.

$$\frac{A \xrightarrow{\alpha} E}{A\,|\,B \xrightarrow{\alpha} E\,|\,B} \qquad \textsc{Par-Left}$$

$$\frac{B \xrightarrow{\alpha} E}{A\,|\,B \xrightarrow{\alpha} A\,|\,E} \qquad \textsc{Par-Right}$$

$$\frac{A \xrightarrow{\alpha} E \quad \alpha \notin M}{(\nu M)A \xrightarrow{\alpha} (\nu M)E} \qquad \textsc{Res-1}$$

All three rules span any label $\alpha$ and production $E$, meaning the constructs on the right hand side (such as $E\,|\,B$) apply to both species and concretions. The easiest way to implement this in Lean would be to have a function which either uses species or concretion parallel composition (`_ |ₛ _` and `_ |₁ _`) respectively, and pass the production through that function. However, using a function application as part of a type's index causes problems (which we discuss in more detail in Section 6.2).

The alternative solution is less elegant, but sufficient for our needs. We duplicate all three rules, having species and concretion variants. While this does lead to some code duplication, as we must now handle two near-identical constructors, fortunately this is less than one might expect. As the kind of production is often known, we can generally avoid having to match against both variants of the constructor.

```
| parL_species {Γ ℓ A} B {l : label ℍ Γ kind.species} {E}
  : transition A ℓ l (production.species E)
  → transition (A |ₛ B) ℓ l (production.species (E |ₛ B))
| parL_concretion
    {Γ ℓ A} B {l : label ℍ Γ kind.concretion} {b y} {E : concretion ℍ ω Γ b y}
  : transition A ℓ l (production.concretion E)
  → transition (A |ₛ B) ℓ l (production.concretion (E |₁ B))

| parR_species {Γ ℓ} A {B} {l : label ℍ Γ kind.species} {E}
  : transition B ℓ l (production.species E)
  → transition (A |ₛ B) ℓ l (production.species (A |ₛ E))
| parR_concretion /- … -/ → transition (A |ₛ B) ℓ l (production.concretion (A |₂ E))

| ν₁_species
    {Γ ℓ} (M : affinity ℍ) {A} {l : label ℍ Γ kind.species} {l' : label ℍ
    ↪  (context.extend M.arity Γ) kind.species} {E}
  : l' = label.rename name.extend l
  → transition A (lookup.rename name.extend ℓ) l' (production.species E)
  → transition (ν(M) A) ℓ l (production.species (ν(M) E))
| ν₁_concretion /- … -/ → transition (ν(M) A) ℓ l (production.concretion (ν'(M) E))
```

The `ν₁` rule defines two labels `l` and `l'` and an equality between them. It would be possible to remove the second label, and use the right hand side of the equality directly. However, doing so would result in a function application within the type index which, as mentioned above, is something we want to avoid. We do not have the same issue with the lookup function, as that is invariant between all constructor cases, and so does not cause the same problems.

**Species invocation** We can view transitions from a species invocation term in a similar manner. Instead of wrapping a transition from a sub-term, they look up the species in the current environment.

$$\frac{B \xrightarrow{\alpha} E \quad D(\vec{y}) \overset{\text{def}}{=} B}{D(\vec{b}) \xrightarrow{\alpha} E\{\vec{b}/\vec{y}\}} \qquad \textsc{Defn}$$

The translation into Lean is surprisingly direct, though with several minor differences. As our environment maps species names to a list of choices, rather than a full species, the input transition's type is adjusted accordingly.

We also perform substitution before constructing the transition, rather than afterwards. This should[1]not change the set of allowed transitions, but does simplify later proofs.

```
| defn
    {Γ k n} {α : label ℍ Γ k} (ℓ : lookup ℍ ω Γ)
    (D : reference n ω) (as : vector (name Γ) n)
    (B : species.choices ℍ ω Γ) {E}
  : B = species.rename (name.mk_apply as) (ℓ n D)
  → transition (Σ# B) ℓ α E
  → transition (species.apply D as) ℓ α E
```

**Parallel species** When two species are in a mixture together, they have the potential to interact with each other. This is modelled by the Com-1 transition.

$$\frac{A \xrightarrow{a} F \quad B \xrightarrow{b} G \quad F \circ G \downarrow}{A \mid B \xrightarrow{\tau\langle a,b\rangle} F \circ G} \qquad \textsc{Com-1}$$

Due to our encoding of a concretion's arity within its type, we do not need the additional constraint that $F \circ G$ is defined. However, we do need a level of indirection on both $F \circ G$ and $\tau\langle a,b\rangle$ in order to avoid function applications on an index.

```
| com₁
    {Γ ℓ x y} {A B : species ℍ ω Γ} {a b : name Γ}
    {F : concretion ℍ ω Γ x y} {G : concretion ℍ ω Γ y x}
    {FG : species ℍ ω Γ} {α : label ℍ Γ kind.species}

  : FG = concretion.pseudo_apply F G
```

[1]cπ requires that every recursive cycle of species involves a prefix guard. However, expressing this requirement in Lean, and showing it is equivalent is a difficult task and was not attempted.

```
structure finset (α : Type*) :=
  (val : multiset α)
  (nodup : nodup val)

class fintype (α : Type*) :=
  (elems : finset α)
  (complete : ∀ x : α, x ∈ elems)
```

Listing 4.6: The definition if finite sets and enumerable types.

```
  → α = τ⟨ a, b ⟩
  → transition A ℓ (#a) (production.concretion F)
  → transition B ℓ (#b) (production.concretion G)
  → transition (A |ₛ B) ℓ α (production.species FG)
```

**Binders and local behaviour** Finally, as mentioned in Labels and Productions, a $\tau\langle a,b\rangle$ transition may be replaced by a $\tau@M(a,b)$ transition when within a $(\nu M)$ binder. This is handled by the Com-2 rule.

$$\frac{A \xrightarrow{\tau\langle a,b\rangle} B \quad a,b \in M \quad M(a,b)\downarrow}{(\nu M)A \xrightarrow{\tau@M(a,b)} (\nu M)B} \qquad \textsc{Com-2}$$

```
| com₂
    {Γ ℓ} (M : affinity ℍ) {A B : species ℍ ω (context.extend M.arity Γ)}
    {p : upair (fin M.arity)} {p' : upair (name (context.extend M.arity Γ))}
    (k : ℍ)
  : M.get p = some k
  → p' = p.map name.zero
  → transition A (lookup.rename name.extend ℓ) τ⟨ p' ⟩ (production.species B)
  → transition (ν(M) A) ℓ τ@'k (production.species (ν(M) B))
```

While the type of transitions explicitly state their output and label, we also find it useful to describe any transition from a specific species.

```
def transition_from {Γ} (ℓ : lookup ℍ ω Γ) (A : species ℍ ω Γ) : Type
  := Σ k (α : label ℍ Γ k) E, A [ℓ, α]⟶ E
```

### 4.3.1 Enumerating transitions

In order to evaluate the behaviour of a species, we must have a way to enumerate all transitions from it. Conceptually, this is quite simple, and is paid little heed by [4]. However, building such a procedure in Lean is a non-trivial task.

In order to enumerate the transition set, we use mathlib's `fintype α` class. We construct such a type by providing a multiset of values of type `α`, along with a proof that the multiset has no duplicates and all values appear within it.

In order to insert an element into a `finset`, one must provide a proof that the element does not already occur within the set. Similarly, one may only map over

```
private def enumerate_parallel_ts {Γ} {ℓ : lookup ℍ ω Γ} (A B : species ℍ ω Γ)
: fintype (transition.transition_from ℓ A)
→ fintype (transition.transition_from ℓ B)
→ finset (transition.transition_from ℓ (A |ₛ B))
| As Bs :=
  finset.union_disjoint
    (finset.map
      (com₁.embed ℓ A B)
      ((finset.product As.elems Bs.elems).subtype (com₁.is_compatible ℓ A B)))
    (finset.union_disjoint
      (As.elems.map (parL.embed A B))
      (Bs.elems.map (parR.embed A B))
      (λ x memL memR, begin show false, /- ... -/ end))
     (λ x memL memR, begin show false, /- ... -/ end))

private lemma enumerate_parallel_complete {Γ} {ℓ : lookup ℍ ω Γ}
    (A B : species ℍ ω Γ)
    (As : fintype (transition.transition_from ℓ A))
    (Bs : fintype (transition.transition_from ℓ B))
  : ∀ x, x ∈ enumerate_parallel_ts A B As Bs
| ⟨ k, α, E, com₁ eqFG eqα tf tg ⟩ := /- ... -/
| ⟨ k, α, E, parL_species _ t ⟩ := /- ... -/
-- ...
```

Listing 4.7: Enumerating all transitions from species $A \mid B$.

`finset`s using injective functions, and perform unions on disjoint sets[2].

How an instance of such a structure can be computed should be fairly clear. We simply recur over the structure of a species, enumerating transitions for sub-terms and then transforming those transitions using the appropriate constructors for this species.

For instance, in order to enumerate the transition from the parallel composition $A \mid B$, we take the union of:

- All transitions from $A$, wrapped using the Par-Left constructor.
- All transitions from $B$, wrapped using the Par-Right constructor.
- The cross product of transitions from $A$ and $B$, restricted to those which can be converted into a Com-1 transition.

The Lean translation of this union is incredibly verbose, due to the additional proofs of non-intersection and non-membership required (Listing 4.7). We continue this process for all species constructors. Listing 4.8 gives an example of enumerating the transitions of the parallel composition of two basic species.

[2]It is possible to avoid this restriction by defining a decision procedure for equality of elements. However, given the complex nature of transitions, writing such a function is frustrating.

```
def A := a# • nil |ₛ b# • nil
#eval (fintype.elems (transition_from ℓ A))
/-
{(0.0#.0 | 0.1#.0) [#0.0]⟶ (([];0)0 | 0.1#.0),
 (0.0#.0 | 0.1#.0) [#0.1]⟶ (0.0#.0 | ([];0)0),
 (0.0#.0 | 0.1#.0) [τ( 0.0 , 0.1 )]⟶ (0 | 0)}
-/
```

Listing 4.8: Enumerating all transitions of the species $a.0 \,|\, b.0$. The name `0.0` corresponds to channel $a$, and `0.1` to $b$.

```
def equivalent_of :
  ∀ {Γ ℓ k} {A : species ℍ ω Γ} {B : species ℍ ω Γ} {α : label ℍ Γ k}
     {E : production ℍ ω Γ k}
  , species.equivalent A B → A [ℓ, α]⟶ E
  → Σ' (E' : production ℍ ω Γ k) (eq : E ≈ E'), B [ℓ, α]⟶ E'
  := /- ... -/

noncomputable def equivalent_of.map {Γ ℓ} {A B : species ℍ ω Γ} (h :
↪  species.equivalent A B)
    {k} {α : label ℍ Γ k}
  : (Σ (E : production ℍ ω Γ k), A [ℓ, α]⟶ E)
  → (Σ (E : production ℍ ω Γ k), B [ℓ, α]⟶ E)
| ⟨ E, t ⟩ :=
  let ⟨ E', _, t' ⟩ := equivalent_of h t in
  ⟨ E', t' ⟩
```

Listing 4.9: Constructing an equivalent transition for structurally congruent species.

## 4.3.2 Transitions of structurally congruent species

In order to show that structural congruence is a behavioural equivalence, we must show that the transition sets of two structurally congruent species are equivalent.

Theorem 9 [4, §2.2] states that if $A \equiv B$, then there is some bijection $\phi : Trans(A) \to TransB$ such that if $\phi(A \xrightarrow{\alpha} E) = B \xrightarrow{\alpha'} E'$ then $\alpha = \alpha'$ and $E \equiv E'$.

In order to show this we first provide a function which maps from one transition to another with the same label and a structurally congruent production. This can be used to show that such a function $\phi$ exists.

Both the functions from Listing 4.9 operate on `species.equivalent A B` rather than the standard `A ≈ B`. As the relation `_ ≈ _` is a proposition, it is erased at runtime, and so we cannot use it to construct a value, such as a transition. However, the main definition of congruence (Section 3.3.3) is a **`Type`**, and so may be used in a computational context.

In order to show that `equivalent_of.map` is a bijection, it should be sufficient to show that it is its own inverse. However, the definition of `equivalent_of` is a little over 400 lines of code, and so such a proof will be similarly long. For now, we axiomise this property (Listing 4.10).

One other complication arises when dealing with various congruence rules dealing

```
axiom equivalent_of.map_map {Γ ℓ} {A B : species ℍ ω Γ} (h : species.equivalent A B)
  {k} {α : label ℍ Γ k} (t : Σ (E : production ℍ ω Γ k), A [ℓ, α]⟶ E)
: equivalent_of.map h.symm (equivalent_of.map h t) = t

/-- Show that two equivalent species's transition sets are isomorphic. -/
noncomputable def equivalent_of.is_equiv {Γ ℓ} {A B : species ℍ ω Γ}
    (h : species.equivalent A B) {k} {α : label ℍ Γ k}
  : (Σ (E : production ℍ ω Γ k), A [ℓ, α]⟶ E) ≃ (Σ (E : production ℍ ω Γ k), B [ℓ,
  ↪  α]⟶ E)
  := { to_fun := equivalent_of.map h,
       inv_fun := equivalent_of.map h.symm,
       left_inv := equivalent_of.map_map h,
       right_inv := λ x, /- ... -/ }
```

Listing 4.10: An isomorphism between transition sets of congruent species. We currently axiomise the fact that this function is a bijection.

with $(\nu M)$ binders. For instance, consider the congruence $(\nu M)A \equiv A$. Given some transition $(\nu M)A \xrightarrow{\alpha} E$, we must find a suitable transition $A \xrightarrow{\alpha} E'$. This is easy to do, however the resulting transition is in the wrong scope; it originates from `species.rename name.extend A`, rather than `A` directly.

Thus we find we need a function to "undo" the renaming of a species, and yield an equivalent transition. Such a function most likely exists, but for deriving it is tricky. For now, we axiomatise it.

```
protected constant rename_from :
  ∀ {Γ Δ ℓ k}
    {A : species ℍ ω Γ} {l : label ℍ Δ k} {E : production ℍ ω Δ k}
    (ρ : name Γ → name Δ)
  , species.rename ρ A [lookup.rename ρ ℓ, l]⟶ E
  → Σ'(l' : label ℍ Γ k) (E' : production ℍ ω Γ k)
    , pprod (A [ℓ , l']⟶ E') (label.rename ρ l' = l ∧ production.rename ρ E' = E)
```

Regrettably, usage of this function makes `equivalent_of` non-computable. However, this is not too serious, as this function is only used within proofs, which do not require computability.

## 4.4 Continuous Semantics

### 4.4.1 Vector spaces

[14, §3.3.3] defines the semantics of species and processes as operating in two vector spaces, the *process space* and *interaction space*.

**The process space** $\mathbb{P}$ is an infinite dimensional vector space $\mathbb{R}^{\mathcal{S}\#}$, where $\mathcal{S}\#$ is the set of prime species.

```
def process_space (ℂ ℍ : Type) (ω Γ : context) [add_monoid ℂ]
  := fin_fn (prime_species' ℍ ω Γ) ℂ
```

Like processes, this is parameterised over two types. ℂ, the codomain of the vector space, describes the concentration (or concentration gradient). $\mathbb{H}$ describes the rate of reaction, such as that defined in affinity networks. Typically these may both be instantiated to $\mathbb{R}$ (or some similar, computable type), though we leave them generic.

**Interaction space** $\mathbb{D}$ is an infinite dimensional vector space $\mathbb{R}^{\mathcal{S}\# \times \mathcal{C} \times \mathcal{N}}$, where $\mathcal{C}$ is the set of concretions. This is described in much the same way.

```
def interaction_space (ℂ ℍ : Type) (ω Γ : context) [add_monoid ℂ]
  := fin_fn
    (prime_species' ℍ ω Γ × (Σ (b y), concretion' ℍ ω Γ b y) × name Γ)
    ℂ
```

Both these definitions make use of a `fin_fn α β` type. This models vector spaces in much the same way as [14, §3.3.3]. We define a function $f : \alpha \to \beta$ and the function's *support*, such that $x \in \text{support}(f) \iff f(x) \neq 0$.

```
structure fin_fn (α : Type*) (β : Type*) [has_zero β] :=
  (space : α → β)
  (support : finset α)
  (support_iff : ∀ x, space x ≠ 0 ↔ x ∈ support)
```

Given such a definition, it is relatively easy to provide definitions for addition, subtraction and scalar multiplication for our `fin_fn`, and so show it forms a semi-module. From our definition of vector spaces, and some suitable function to compute the prime decomposition of species, we can define the *species embedding* $\langle - \rangle : \mathcal{S} \to \mathbb{P}$.

$$\langle A \rangle \overset{\text{def}}{=} \sum_{B \in \text{primes}(A)} \mathbf{1}_B$$

We use the notation $\mathbf{1}_B$ to denote a basis vector. This may be defined as a function $\lambda\, A.\ \texttt{if}\ A \equiv B\ \texttt{then}\ 1\ \texttt{else}\ 0$, or `fin_fn.single B 1` within Lean.

```
def to_process_space {Γ} (A : species' ℍ ω Γ) : process_space ℂ ℍ ω Γ
  := (prime_decompose' A).sum' (λ A, fin_fn.single A 1)
```

Here `prime_decompose'` operates on quotients or equivalence classes of species instead, mapping them to a multiset of quotients of prime species. Using the properties of `sum'` and prime decomposition, it is simple to show the required properties, such as $\langle A \mid B \rangle = \langle A \rangle + \langle B \rangle$.

```
lemma to_process_space.parallel {Γ} (A B : species ℍ ω Γ)
  : (to_process_space ⟦ A |ₛ B ⟧ : process_space ℂ ℍ ω Γ)
  = to_process_space ⟦ A ⟧ + to_process_space ⟦ B ⟧
  := by simp only [to_process_space, prime_decompose_parallel', multiset.sum'_add]
```

It's also useful to provide a similar embedding from processes to process spaces. This does not have any meaning within the semantics, but proves useful when operating with ODEs.

```
private def interaction_tensor_worker (conc : ℍ ↪ ℂ)  : /- ... -/
| ⟨ A, ⟨ bF, yF, F ⟩, x ⟩ ⟨ B, ⟨ bG, yG, G ⟩, y ⟩ :=
  option.cases_on (M.f x.to_idx y.to_idx) 0 (λ aff,
    if h : bF = yG ∧ yF = bG then begin
      rcases h with ⟨ ⟨ _ ⟩, ⟨ _ ⟩ ⟩,
      from conc aff • ( to_process_space (pseudo_apply' F G)
                      - fin_fn.single A 1 - fin_fn.single B (1 : ℂ)),
    end else 0)

def interaction_tensor (conc: ℍ ↪ ℂ)
  : interaction_space ℂ ℍ ω (context.extend M.arity context.nil)
  → interaction_space ℂ ℍ ω (context.extend M.arity context.nil)
  → process_space ℂ ℍ ω (context.extend M.arity context.nil)
| x y := fin_fn.bind₂ x y (interaction_tensor_worker conc)
```

Listing 4.11: The interaction tensor defined in Lean. This is defined as a function which operates on a single basis, which is then applied to the whole vector space.

```
def process.to_space {Γ} : process ℂ ℍ ω Γ → process_space ℂ ℍ ω Γ
| (c ◯ A) := c • to_process_space ⟦ A ⟧
| (P |ₚ Q) := process.to_space P + process.to_space Q
```

### 4.4.2 Interaction tensor

The interaction tensor $- \oslash - : \mathbb{D} \to \mathbb{D} \to \mathbb{P}$ combines the potentials of two interaction spaces. This behaves somewhat similarly to the two Com transitions, taking some potential transitions $A \xrightarrow{x} F$ and $B \xrightarrow{y} G$ and evolving into a new species $F \circ G$.

The interaction tensor is defined in terms of a function operating on the basis values.

$$\mathbf{1}_{A,F,x} \oslash \mathbf{1}_{B,G,y} \overset{\text{def}}{=} \begin{cases} \text{Aff}(x,y) \times (\langle F \circ G \rangle - \mathbf{1}_A - \mathbf{1}_B) & x,y \in \text{Aff and } F \circ G \downarrow \\ 0 & \text{otherwise} \end{cases}$$

This is relatively easy to define in Lean. We first define a worker, which operates on a single basis. This checks the two preconditions, and then evaluates the body appropriately. We then apply this function to the cross product of the two vector spaces to obtain the resulting process spaceListing 4.11. Both these functions take an embedding `conc : ℍ ↪ ℂ`, which converts rates of reactions to concentration gradients within the process space.

Given pseudo application and the affinity network are commutative, it's easy to show that the interaction tensor is commutative. We can also show that it forms a monoid homomorphism, distributing over addition.

$$\partial(c \cdot A) \overset{\text{def}}{=} c \cdot \sum_{B \in \text{primes}(A)} \sum_{B \xrightarrow{x} F} \mathbf{1}_{B,F,x}$$

$$\partial(P \parallel Q) \overset{\text{def}}{=} \partial Q + \partial P$$

$$\frac{d(c \cdot A)}{dt} \overset{\text{def}}{=} \sum_{B \in \text{primes}(A)} \sum_{B \xrightarrow{\tau@k} F} (k \times c \times (\langle C \rangle - \mathbf{1}_B)) + \frac{1}{2}(\partial(c \cdot A) \oslash \partial(c \cdot A))$$

$$\frac{d(P \parallel Q)}{dt} \overset{\text{def}}{=} \frac{P}{dt} + \frac{P}{dt} + \partial P \oslash \partial Q$$

Figure 4.2: The continuous semantics of cπ, defined by induction on processes.

```
def potential_interaction_space {Γ} {ℓ : lookup ℍ ω Γ} {A : prime_species ℍ ω Γ}
  : transition.transition_from ℓ A.val → interaction_space ℂ ℍ ω Γ
| ⟨ _, # a , @production.concretion _ _ _ b y G, tr ⟩
  := fin_fn.single ⟨ ⟦ A ⟧, ⟨ b, y, ⟦ G ⟧ ⟩, a ⟩ 1
| ⟨ _, τ@'_, E, tr ⟩ := 0
| ⟨ _, τ⟨_⟩, E, tr ⟩ := 0

def immediate_process_space {Γ} {ℓ : lookup ℍ ω Γ} (conc : ℍ ↪ ℂ)
    {A : prime_species ℍ ω Γ}
  : transition.transition_from ℓ A.val → process_space ℂ ℍ ω Γ
| ⟨ _, # a , _, tr ⟩ := 0
| ⟨ _, τ@'k, production.species B, tr ⟩
  := conc k • (to_process_space ⟦ B ⟧ - fin_fn.single ⟦ A ⟧ 1)
| ⟨ _, τ⟨ n ⟩, _, tr ⟩ := 0
```

Listing 4.12: Computing the resulting vector and interaction space of single transition.

### 4.4.3 Process behaviour

Finally, we are ready to compute the behaviour of a process $P$. This is defined in two steps, the *interaction potential* $\partial P \in \mathbb{D}$, derived from the potential interactions of a process, and the *immediate behaviour* $\frac{dP}{dt} \in \mathbb{P}$, derived from the immediate actions a process may undergo.

In order to convert the behaviour described in Figure 4.2 into Lean, we must find a way to express the inner sum. Rather than summing a specific kind of transition (such as $B \xrightarrow{x} F$ or $A \xrightarrow{\tau@k} F$), we operate on all transitions, returning 0 if it is not of the correct form (Listing 4.12.

We then map over all prime species and transitions, in order to obtain the semantics for a single-species process. We also define a version of these semantics which operates on quotients of species instead. This can be done fairly directly given that the two transitions sets are isomorphic (as-per Section 4.3.2). From there, it's simple to define the complete potential and immediate behaviour of a process.

Given our previous lemmas about how `potential_interaction_space` and `immediate_process_space`

```
/-- The vector space of potential interactions of a process (∂P). -/
def process_potential {Γ} (ℓ : lookup ℍ ω Γ)
: process ℂ ℍ ω Γ → interaction_space ℂ ℍ ω Γ
| (c ◯ A) := c • potential_interaction_space.from_species ℓ A
| (P |ₚ Q) := process_potential P + process_potential Q

/-- The vector space of immediate actions of a process (dP/dt)-/
def process_immediate (M : affinity ℍ) (conc : ℍ ↪ ℂ)
    (ℓ : lookup ℍ ω (context.extend M.arity context.nil))
  : process ℂ ℍ ω (context.extend M.arity context.nil)
  → process_space ℂ ℍ ω (context.extend M.arity context.nil)
| (c ◯ A)
  := c • immediate_process_space.from_species conc ℓ A
  + (½ : ℂ) • (process_potential ℓ (c ◯ A) ⊘[conc] process_potential ℓ (c ◯ A))
| (P |ₚ Q)
  := process_immediate P + process_immediate Q
  + (process_potential ℓ P ⊘[conc] process_potential ℓ Q)
```

Listing 4.13: Our continuous semantics from Figure 4.2, defined in Lean.

hold over structural congruence, it's relatively trivial to show that the same is true for the potential and immediate behaviour.

Some attention should be shown to the structural rules describing how mixtures of processes are structurally congruent.

$$(c \cdot A) \parallel (d \cdot A) \equiv (c+d) \cdot A$$
$$c \cdot (A \,|\, B) \equiv c \cdot A \parallel c \cdot B$$

While both of these are 'obvious' by inspecting the definitions of $\frac{dP}{dt}$ and $\delta P$, showing these in a theorem prover can be tricky due to the amount of arithmetic manipulation required.

Thankfully, Lean includes an `abel` tactic, which is able to reason about abelian groups and commutative monoids. This allows us to automate much of the tedious legwork which would otherwise be required.

```
calc  c • dA + c • dB + (iA + (½ : ℂ) • iAB + ((½ : ℂ) • iAB + iB))
    = c • dA + c • dB + (iA + iB + ((½ : ℂ) • iAB + (½ : ℂ) • iAB)) : by abel
... = c • dA + c • dB + (iA + iB + iAB)
      : by rw [← add_smul, ← half_ring.one_is_two_halves, one_smul]
... = c • dA + iA + (c • dB + iB) + iAB : by abel
```

## 4.4.4 Extraction of ODEs

Finally, we must be able to translate our cπ models into ODEs. This can be done in much the same way as [14, §3.4]. As with the algorithm presented by Kwiatkowski, first we provide a way to enumerate all possible species within the system, and then execute it using a symbolic representation.

```
def all_species.finset (M : affinity ℍ)
    (ℓ : lookup ℍ ω (context.extend M.arity context.nil))
  : ℕ → finset (prime_species' ℍ ω (context.extend M.arity context.nil))
  → all_species M ℓ
| 0 As := all_species.incomplete As
| (nat.succ n) As :=
  let As' := (process_immediate.quot M ℓ fin_poly.C.embed
                (process.from_prime_multiset fin_poly.X As.val)).support in
  if eq : As' ⊆ As then all_species.complete As eq
  else all_species.finset n (As ∪ As')

/-- Get all species in the transition graph for a process. -/
def all_species.process (M : affinity ℍ)
    (ℓ : lookup ℍ ω (context.extend M.arity context.nil))
  : ℕ → process ℍ ℍ ω (context.extend M.arity context.nil) → all_species M ℓ
| fuel P := all_species.finset M ℓ fuel (process.to_space P).support
```

Listing 4.14: Enumerating the complete transition space of a process.

One thing to note is that the transition space of a system may not be finite. It is possible to derive a term which may 'diverge', producing ever larger species. In order to avoid this, we provide our function to enumerate the species with an amount of fuel, which reduces after every iteration. The function may then return a complete set or, if it runs out of fuel, a partial one.

Then given a complete set of prime species, we can convert them into a process and execute them, producing a polynomial.

```
def as_ode (M : affinity ℍ) (ℓ : lookup ℍ ω (context.extend M.arity context.nil))
  : finset (prime_species' ℍ ω (context.extend M.arity context.nil))
  → process_space
      (fin_poly (prime_species' ℍ ω (context.extend M.arity context.nil)) ℍ)
      ℍ ω (context.extend M.arity context.nil)
| As := process_immediate.quot M ℓ fin_poly.C.embed
    (process.from_prime_multiset (λ x, fin_poly.X x) As.val)
```

For reasons we will explore in Chapter 5, we are currently unable to execute such a function. However, if we provide the complete species space of system manually, we can achieve the same result. We can do this for our example enzyme system from Section 2.1.1[3].

```
def system : process ℂ ℍ ω Γ :=
  (fin_poly.X (apply S ∅)) ◯ (apply S ∅) |ₚ
  (fin_poly.X E'_) ◯ E'_ |ₚ
  (fin_poly.X C'_) ◯ C'_ |ₚ
  (fin_poly.X (apply P₁ ∅)) ◯ (apply P₁ ∅) |ₚ
  (fin_poly.X (apply P₂ ∅)) ◯ (apply P₂ ∅)

#eval process_immediate aff ℓ conc system
```

With a little bit of post-processing in order to replace species with their names, we receive the following ODEs.

[3]The complete Lean code for the enzyme system may be found in Appendix B.

$$
\begin{aligned}
\frac{dE}{dt} &= -k_{\text{bind}} \cdot E \cdot S + k_{\text{react}} \cdot C + k_{\text{unbind}} \cdot C \\
\frac{dS}{dt} &= -k_{\text{bind}} \cdot E \cdot S + k_{\text{unbind}} \cdot C \\
\frac{dP_1}{dt} &= -k_{\text{decay}} \cdot P_1 + k_{\text{react}} \cdot C \\
\frac{dP_2}{dt} &= -k_{\text{decay}} \cdot P_2 + k_{\text{react}} \cdot C \\
\frac{dC}{dt} &= -k_{\text{react}} \cdot C - k_{\text{unbind}} \cdot C + k_{\text{bind}} \cdot E \cdot C
\end{aligned}
$$

Figure 4.3: The expected ODEs for our enzyme system.

```
/-
  ((-1•(k_bind))•(E•S) + (1•(k_react) + 1•(k_unbind))•(S)) • E
  ((-1•(k_bind))•(E•S) + (1•(k_unbind))•(S)) • S
  ((-1•(k_degrade))•(P₁) + (1•(k_react))•(S)) • P₁
  ((-1•(k_degrade))•(P₂) + (1•(k_react))•(S)) • P₂
  ((-1•(k_react) + -1•(k_unbind))•(S) + (1•(k_bind))•(E•S)) • C
-/
```

While this result is somewhat confusing to read, due to the nature of our polynomial representation, it is possible to see these are equal to the expected ODEs from Figure 4.3.

At this point, we have completed our definition of cπs semantics, and shown most of its main lemmas. As such, our main mechanisation of [4, 14] is complete.

# Chapter 5

# Alternative Equivalences

We now have a relatively complete formalisation of the continuous $\pi$-calculus, and are theoretically able to complete an end-to-end translation from species and processes to the resulting ODEs. However, there are several problems which make actually computing the ODEs much harder than expected.

Recall our definition of process spaces and interaction spaces from Section 4.4.1. Both of these are effectively defined as functions mapping from sets of structurally congruent species to $\mathbb{R}$ (or some other concentration scalar). Thus, in order to evaluate a function at a specific point, we need a way to determine if two species are structurally congruent.

While it is possible to define a *decision procedure* for a single congruence rule, doing so for the transitive closure proves much harder. We must be able to show, or refute, the infinite number of proofs that two species are congruent.

Similar problems arise when working with prime species (Section 3.3.4). Kwiatkowski [14, Appendix A] presents a 'simple procedure' to compute the prime decomposition of any species. While we are able to translate it into Lean (see Appendix C), as it relies on classical logic, it cannot be executed. This makes it useless if we wish to generate ODEs within Lean.

In this chapter, we explore alternatives to structural congruence. We determine what properties an equivalence relation over species must have, and work with one possible candidate.

## 5.1 Semantics using Alternative Equivalences

When we originally explored the continuous semantics of c$\pi$ in Section 4.4, all definitions operated using equivalence classes of species and concretions under structural congruence. We generalise our previous definitions, replacing structural congruence with an abstract equivalence relation.

Let $\approx$ be a decidable equivalence relation over species and concretions. We use $[A]$ to represent the $\approx$-equivalence class of $A$ (written as `⟦ A ⟧` in Lean).

```
class cpi_equiv (ℍ : Type) (ω : context) :=
  [species_equiv {} : ∀ Γ, setoid (species ℍ ω Γ)]
  [concretion_equiv {} : ∀ Γ b y, setoid (concretion ℍ ω Γ b y)]
  [decide_species {} : ∀ Γ, decidable_rel (species_equiv Γ).r]
  [decide_concretion {} : ∀ Γ b y, decidable_rel (concretion_equiv Γ b y).r]

  (prime_decompose {Γ} : species ℍ ω Γ → multiset (prime_species ℍ ω Γ))
  ( pseudo_apply {Γ} {a b : ℕ}
  : concretion' ℍ ω Γ a b → concretion' ℍ ω Γ b a
  → species' ℍ ω Γ )
  /- ... -/
```

Figure 5.1: An arbitrary equivalence relation for species and concretions, and the properties it must have.

Given such an equivalence relation, we can redefine our previous definitions in such a way that they work with our alternative equivalence.

**Prime species** $\approx$-prime species are defined in much the same way as $\equiv$-prime species. Namely, a species is prime if $A \not\approx \mathbf{0} \land \forall BC.A \approx (B\,|\,C) \rightarrow B \approx \mathbf{0} \lor C \approx \mathbf{0}$.

**Prime decomposition** We require some procedure $\mathcal{P}$ which maps a species into a multiset of prime-species, such that:

- $\mathcal{P}(\mathbf{0}) = \emptyset$
- $\mathcal{P}(A\,|\,B) = \mathcal{P}(A) + \mathcal{P}(B)$
- For all prime species $A$, $\mathcal{P}(A) = \{A\}$
- If $A \approx B$, then $\mathcal{P}(A) = \mathcal{P}(B)$ up to equivalence of species.

**Pseudo-application** While the definition of pseudo-application remains as before, we require that $F \circ G \approx F' \circ G'$ when $F \approx F'$ and $G \approx G'$, in much the same way that we do for structural congruence.

These requirements may easily be encoded into Lean as a typeclass (Figure 5.1). While they are sufficient in order to define the semantics using arbitrary relations, in order to show that the semantics are *sound* (namely, equivalent species yield identical semantics), we require several additional properties. Pseudo-application must be commutative up to equivalence, and equivalent species must have isomorphic transition sets. We define these requirements in a separate type class, as this allows us to execute the semantics for potentially unsound systems, while still allowing proofs about sound relations.

## 5.2 n-Equivalence

Now we have defined the necessary properties an equivalence relation must have, we explore one possible candidate. Taking inspiration from [14, Appendix A],

$$
\begin{array}{rlll}
\text{Parallel composition} & & & \\
\text{within guarded choice} & \sigma & ::= A^* & \\
& & \mid\ A^* \mid \sigma & \\
& & & \\
\text{Species within guarded choice} & \tau & ::= \sigma \ \mid\ \mathbf{0} & \\
& & & \\
\text{Species within } (\nu M) \text{ binders} & \gamma & ::= A^* & \text{when } M \in A^* \\
& & \mid\ A^* \mid \gamma & \text{when } M \in A^* \\
& & & \\
\text{Atomic species} & A^* & ::= D(\vec{a}) & \\
& & \mid\ \Sigma_{i=0}^{n} \pi_i . \tau_i & \\
& & \mid\ (\nu M)\gamma &
\end{array}
$$

Figure 5.2: The definition of atomic species.

we define a procedure to normalise terms, and say two species are n-equivalent if they normalise to a syntactically *identical* term.

Our normalisation procedure maps a single species, to a list of *atomic* species which, when in parallel composition, are congruent to the original one. An atomic species normalises to itself, and is defined using the grammar in Figure 5.2.

The language of atomic species enforces several invariants, which eliminates many of the original structural congruence rules, such as $A \mid \mathbf{0} \equiv A$. Any species which are congruent only using these eliminated rules, will be n-equivalent.

However, we do not impose any ordering on $\nu$ binders, guarded choice or parallel composition. As a result, some species which are structurally congruent (such as $A \mid B \equiv B \mid A$) are not n-equivalent. While it would be possible to enforce an ordering on choice and parallel composition, dealing with $\nu$ exchange is much harder.

Normalisation yields a list of species, which are congruent to the original species and are all atomic. This may be used to define a simpler normalisation function, which maps species to species. Finally, we may define n-equivalence (Listing 5.1).

The most useful property of this relation is that it is decidable, as normalisation is a computable function. Given that normalised terms are congruent to their original species, n-equivalence implies structural congruence, making it a strictly weaker relation. Furthermore, all atomic species are also n-prime, meaning that normalisation computes the prime decomposition of species.

Now we have an equivalence relation for species, we must also decide on an relation for concretions. While it would be ideal to define a similar normalisation function for concretions, due to time constraints, we settled on using syntactic

```
inductive atom :
  ∀ {sk : kind} (k : kind' ℍ sk) {Γ : context}
  , whole ℍ ω sk Γ → Prop

def equivalence_of : ∀ {k} {Γ}, whole ℍ ω k Γ → Type
| kind.species Γ A :=
  Σ' (Bs : list (species ℍ ω Γ))
  , A ≈ parallel.from_list Bs
  ∧ ∀ B ∈ Bs, normalise.atom normalise.kind'.atom B
| kind.choices Γ A := /- ... -/

def normalise_to : ∀ {k} {Γ} (A : whole ℍ ω k Γ), equivalence_of A
/- ... -/

def normalise : ∀ {k} {Γ}, whole ℍ ω k Γ → whole ℍ ω k Γ
| kind.species Γ A := parallel.from_list (normalise_to A).fst
| kind.choices Γ A := (normalise_to A).fst

def normalise.equiv {Γ : context} (A B : species ℍ ω Γ) : Prop := normalise A =
↪  normalise B
```

Listing 5.1: Atomic species and n-equivalence, as defined in Lean.

equality. Again, this is trivially computable and is sufficient for our requirements.

These relations and definitions are sufficient in order to *execute* c$\pi$ terms. However, in order to show that n-equivalence produces a sound semantics, we must also show that pseudo-application is commutative up to n-equivalence, and n-equivalent species have equivalent transition sets.

Unfortunately, this is not the case. Consider the pseudo-application of two concretions $F \overset{\text{def}}{=} (\nu M)(;)A$ and $G \overset{\text{def}}{=} (\nu N)(;)B$. Pseudo application of $F$ and $G$ yields different terms, depending on the order of the operands. $F \circ G = (\nu M)(\nu N)(A|B)$, but $G \circ F = (\nu N)(\nu M)(B|A)$. As previously discussed, normalisation does not handle $\nu$ exchange, and so these species are not n-equivalent.

This, in turn, means that parallel composition (and thus process mixtures) are not symmetric, which is problematic. It would be possible to define an alternative version of pseduo-application which sorts its arguments using some lexicographical ordering. This would be effective, though rather inelegant.

More troubling is that n-equivalence of species does not imply equivalence of their transition sets. As we do not have a specialised equivalence relation on concretions, transitions from $A$ and normalise($A$) will not have equivalent productions. Defining a similar n-equivalence relation on concretions should be sufficient to show this property, but due to time constraints no attempt has been made at this.

As a result, extraction of ODEs is not possible directly. We do not have an isomorphism between processes and process spaces, as mixtures are not commutitive, and so it is not possible to enumerate the entire transition space. Instead, one can enumerate the transition space by hand, and then compute the ODEs from a ‘complete’ process. This is the technique we used to compute the examples in Section 4.4.4.

Sadly, in its current state, n-equivalence is a workable, but unsound equivalence relation. While this is still a useful counterpart to the unworkable but sound structural congruence, it is not ideal.

# Chapter 6

# Evaluation

We have managed to derive an almost complete mechanisation of the continuous $\pi$-calculus, and explored alternative equivalences which make it more practical in a computational setting. This section discusses successes and failures in the project, and provides some observations about where theorem proving can be a help and hindrance.

## 6.1 Our Mechanisation of Continuous-$\pi$

Our mechanisation of the two c$\pi$ papers, [14] and [4] are largely complete. However, as discussed in Chapter 5, our implementation is imperfect, as it is not computable. In this section, we review what could be done to improve the mechanisation, any remaining axioms which must be proven, and why one may encounter further obstacles when attempting to do so.

**Axioms** There are seven axioms remaining in our code base, though only a few are directly relevant to the main work of the project. We provide a full list in Appendix D, though a summary of the key axioms are as follows.

- Given a renaming function $\rho$, for any transition $\rho(A) \xrightarrow{\alpha} E$, there exists another transition $A \xrightarrow{\alpha'} E'$, such that $\rho(\alpha') = \alpha$ and $\rho(E') = E$. More informally, if we have a transition where the species, label and production have been renamed, it should be possible to recover the original terms.
- The function $\phi$ which maps between transitions of congruent species (Section 4.3.2) is a bijection.
- The classical version of prime decomposition (Chapter 5 and Appendix C) obeys the properties we would expect of prime decomposition. Namely, $\mathcal{P}(A \,|\, B) = \mathcal{P}(A) + \mathcal{P}(B)$ and $\mathcal{P}(A) = \mathcal{P}(B)$ when $A \equiv B$.
- Similarly, if a species $A$ is n-prime, its prime decomposition is a singleton multiset $\{A\}$.

The first of these axioms is especially concerning, as it is a much bigger statement than the other axioms. However, work on a proof was considered impractical, as discussed further in Section 6.2.1.

I believe the remaining axioms are true. However, little attempt has been made to show them due to time constraints and their relative unimportance for the main body of this work.

**Progress** The first few stages of the project were very productive, and I was able to mechanise the syntax of species and concretions with few problems. However, as work continued throughout the first and second semester, there were significant hurdles which slowed progress down dramatically.

Regrettably the second half of the project was very much dominated by working with equivalence relations and prime species, and so I was unable to complete all the work I had initially intended. I had planned to do some work with the Bond Calculus [15], but as time was spent on other aspects of the project, this was not attempted.

One interesting thing to note is that the difficulties I encountered were not where my supervisor or I expected them to be. We thought that working with real numbers and c$\pi$'s vector semantics would be the most challenging. However, they proved surprisingly straightforward, in part due to Lean's extensive support for reasoning about monoids and modules (discussed in Section 6.2). While working with real numbers in a theorem prover is often difficult, as they are non-computable, we were able to avoid this problem by working with an abstract commutative monoid instead of a concrete type.

Furthermore, the definition of c$\pi$'s continuous semantics, as defined by [4, §2.2], is flawed. This problem is identified and rectified within [14, §3.3.3]. However when initially defining the semantics I had not realised this, and so implemented the original (and unsound) version. As a result, I was unable to prove that equivalent processes have identical semantics. This led to me identifying the same flaws that Kwiatkowski and Stark had in 2010.

While this problem was found by hand by Kwiatkowski, I think it is still somewhat compelling that my machine checked proof was able to identify the problem much sooner.

Despite these problems, we were able to successfully build several c$\pi$ models within Lean, and verify that it produced the expected ODEs. Aside from our enzyme example (Section 2.1.1, model at Appendix B), we also translated several simple signalling pathways from Stanely Wang's work[16] (Appendix E, Appendix F, Appendix G)

## 6.2 Working with Lean

Our choice to use Lean for this project definitely influenced the progress made. Its support for type classes and tactics, as well as the extensive standard library, significantly reduced the amount of work required to show various lemmas. However, it is not entirely clear whether Lean, or theorem provers in general, are currently suited to tackle every task that this paper required.

### 6.2.1 Dependent types

Lean is a dependently typed language, and we made extensive use of this within our mechanisation. This allows us to encode many invariants within the type system, such as the arity of binding groups (Section 3.1) and concretions (Section 4.1). In one sense, this was a success. However, there are times when this can become a burden.

For instance, recall our definitions of prefix expressions (Section 3.2). Our original implementation omitted the 'telescope' index, instead deriving it from the species itself.

```
inductive prefix_expr (ℍ : Type) : context → Type
| communicate {} {Γ} (a :  name Γ) (b : list (name Γ)) (y : ℕ)
  : prefix_expr Γ
| spontaneous {} {Γ} (k : ℍ) : prefix_expr Γ

def prefix_expr.extend {Γ} : prefix_expr ℍ Γ → context → context
| (_(_ ; y)) Δ := context.extend y Δ
| (τ@_) Δ := Δ
```

However, one then requires a proof that a context within a prefix is the same as that within a renamed prefix, namely `π.extend Δ = (π.rename ρ).extend Δ`. While such a proof is simple, applying it while maintaining type safety becomes difficult. Our use of telescopes sidesteps this issue, as they do not change across renames.

Secondly consider our formalisation of transitions, which is a relatively direct translation of the original transition rules. This is a contrast to systems using HOL, such as [17], which do not have such an intuitive inductive definition. However, the complex definition of transitions is not without its problems.

For instance, in section Section 4.3 we try to avoid using the result of function application within a type index, and instead introduce an additional variable and equality. Our definition of COM-1 produces `production.concretion FG`, with the additional equality `FG = concretion.pseudo_apply F G`, rather than the more intuitive `production.concretion (concretion.pseudo_apply F G)`.

This indirection is required in order to allow us to case-split on transitions. If we have a transition $A \xrightarrow{\alpha} (\nu M)B$, then in order to case split, we must introduce a unification constraint `concretion.pseudo_apply F G` $\sim (\nu M)A$, which cannot be solved by Lean.

One major problem with more complex types, such as transitions, is that Lean takes an unreasonable amount of proof to type check some cases. For instance, our proof that congruent species have equivalent transition setsSection 4.3.2 takes over a minute to check, with several definitions taking 20 to 30 seconds. This poor performance is one of the reasons I did not attempt a proof to show the function over transitions was a bijection.

### 6.2.2 Tactics

As mentioned in the background (Section 2.2), Lean supports tactics, in the style of Coq or Isabelle. While we do not define our own tactics, we make heavy use of three builtin ones; `rw`, `simp` and `abel`.

The first two of these tactics take proofs of equality, and apply them to the goal. `rw h` takes a proposition `h : a = b`, and will rewrite the goal, replacing all occurrences `a` with `b`. The tactic also accepts a list of propositions, applying each of them in turn.

`simp`, is a natural extension of `rw`. Instead of accepting a single lemma, it searches for any lemmas in scope with the `@[simp]` annotation. It then applies them repeatedly, until the term cannot be simplified any further. Many of our lemmas can be dispatched with the `simp` tactic, or at least reduced to a goal which is easier to manage.

One problem with the simplifier, is that it does not support commutativity lemmas, such as $a+b=b+a$, as this would result in a non-terminating rewrite system. Sadly this means that the simplifier is not much use when working with more complex arithmetic expressions, as it is unable to reorder terms. The `abel` tactic is designed for this case, applying the rules of abelian groups in order to prove an equality.

Lean's tactic mode also supports case splitting, via the `cases` tactic. While this is equivalent to pattern matching within a function definition, it allows us to apply a tactic to every case. This can dramatically reduce the amount of work required to show a proof. For instance, when showing that renaming of species is injective, we case split on both input species, and then simplify and check for contradictions *on every case*. This often means we only need to show one or two goals of the remaining, rather than one for every case.

```
lemma rename.inj :
  ∀ {Γ Δ k} {ρ : name Γ → name Δ}
  , function.injective ρ → function.injective (@rename ℍ ω Γ Δ k ρ)
| Γ Δ _ ρ inj nil B eq := begin
  cases B;
  simp only [rename.nil, rename.invoke, rename.parallel, rename.choice,
             rename.restriction] at eq;
  contradiction,
end
```

# Chapter 7

# Conclusion

Over the course of this project, I have built a largely complete mechanisation of the continuous π-calculus, its semantics and generation of a process's ODEs. I have shown that the semantics are sound under structural congruence, and started work on finding alternative equivalences which are easy to reason about in a computational setting.

Several concepts from cπ, most specifically structural congruence, do not translate well to a theorem prover. Many lemmas, which are easy to state, and are fairly obvious are hard or infeasible to show under the rigorous requirements of Lean.

None the less, we were able to use our mechanisation to produce ODEs of several examples, and verify that their behaviour was correct.

## 7.1 Future Work

While our mechanisation of cπ is mostly complete, there are still several area where further work could be done.

The most obvious candidate is showing our remaining axioms, especially proving that the function to map between transitions of congruent species truly is a bijection. I do not believe the proofs for the remaining axioms would be especially involved. However, they are most likely time consuming, due to complex nature of structural congruence.

There is still much work which could be done with alternative equivalence relations. Firstly, it would be good to define a normalisation procedure for concretions, and thus potentially show isomorphism of transition sets under n-equivalence. Combining this with a variant of pseudo-application which sorts its arguments, as discussed in Section 5.2, would allow us to show the semantics are sound under n-equivalence.

An alternative avenue to explore would be to combine our normalisation procedure with the *auxiliary congruence*, as defined in [14, Appendix A]. This relationship is simpler to reason about, meaning it may be feasible to write a decision procedure for. However, it has the desirable property of being equivalent to structural congruence.

While less related, it would be interesting to mechanize the closely related Bond calculus[15]. It is not clear what techniques from my work would be applicable to the Bond calculus, but I suspect there would be a reasonable amount of overlap.

## 7.2 Related Work

Much work has been done by on mechanising the π-calculus, and other derived process calculi. However, most existing work has had very different goals to our own.

To my knowledge, this work is the first to focus on producing an executable semantics, rather than proving properties about the calculi.

"Proof-relevant π-calculus"[11] uses a very similar formalisation of its calculus to this work. Using Agda, it formalises the π-calculus using a labelled transition system. Names are defined using de Bruijn indices indexed on their context, and as such we share several of their definitions and proofs relating to names and renaming. They also encode transitions as an inductive data type, though their definition is significantly simpler due to the lack of affinity networks and concretions. However, they do not consider structural congruence, leaving that for future work.

"A Full Formalisation of π-Calculus Theory in the Calculus of Constructions"[12] also works with labelled transitions system. It shows that structural congruence is a bisimulation, as well as many other classical properties of the π-calculus.

While most papers use de Bruijn indices, several [18, 19] use Isabelle's 'nominal' package instead. This library provides utilities for dealing with syntax trees involving binders, eliminating the need to write proofs relating to substitution and renaming. It would be interesting to see if a similar package would be possible to implement using Lean's meta-programming capabilities.

"Multisets and Structural Congruence of the π-calculus with Replication"[20] discusses several of the same problems with structural congruence that we did in Chapter 5. It shows that structural congruence is equivalent to *multiset congruence*. Engelfriet and Gelsema show multiset congruence is decidable, meaning that structural congruence also is.

# Bibliography

[1] Jasmin Fisher and Thomas A. Henzinger. “Executable cell biology”. In: *Nature Biotechnology* 25.11 (2007), pp. 1239–1249. ISSN: 1546-1696. DOI: 10.1038/nbt1356. URL: https://doi.org/10.1038/nbt1356.

[2] Vladimir Likić et al. “Systems Biology: The Next Frontier for Bioinformatics”. In: *Advances in bioinformatics* 2010 (Nov. 2010), p. 268925. DOI: 10.1155/2010/268925.

[3] Aviv Regev, William Silverman, and Ehud Shapiro. “Representation and Simulation of Biochemical Processes using the $\pi$-calculus Process Algebra”. In: *Pacific Symposium on Biocomputing*. Vol. 6. 2001, pp. 459–470.

[4] Marek Kwiatkowski and Ian Stark. “The Continuous -Calculus: A Process Algebra for Biochemical Modelling”. In: Oct. 2008, pp. 103–122. DOI: 10.1007/978-3-540-88562-7_11.

[5] Leonardo de Moura et al. “The Lean Theorem Prover (System Description)”. In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21401-6.

[6] Ian Stark, Marek Kwiatkowski, and Chris Banks. *Exploring Variation in Biochemical Pathways with Continuous pi*. SynthSys: Synthetic and Systems Biology. June 14, 2012. URL: http://homepages.inf.ed.ac.uk/stark/evocpi-slides-sbm.pdf.

[7] Thierry Coquand and Gérard P. Huet. “The Calculus of Constructions”. In: *Inf. Comput.* 76.2/3 (1988), pp. 95–120. DOI: 10.1016/0890-5401(88)90005-3. URL: https://doi.org/10.1016/0890-5401(88)90005-3.

[8] Peter Dybjer. “Inductive families”. In: *Formal Aspects of Computing* 6 (Jan. 1994), pp. 440–465. DOI: 10.1007/BF01211308.

[9] The Lean community. *mathlib*. URL: https://github.com/leanprover-community/mathlib.

[10] Leonardo de Moura aJeremy Avigad and Soonho Kong. *Theorem Proving in Lean*. Oct. 13, 2019. URL: https://leanprover.github.io/theorem_proving_in_lean/index.html.

[11] Roly Perera and James Cheney. “Proof-relevant $\pi$-calculus”. In: *CoRR* abs/1604.04575 (2016). arXiv: 1604.04575. URL: http://arxiv.org/abs/1604.04575.

[12] Daniel Hirschkoff. “A Full Formalisation of $\pi$-Calculus Theory in the Calculus of Constructions”. In: *Theorem Proving in Higher Order Logics*. Ed. by Elsa L. Gunter and Amy Felty. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 153–169. ISBN: 978-3-540-69526-4.

[13] Joachim Parrow. "CHAPTER 8 - An Introduction to the π-Calculus". In: *Handbook of Process Algebra*. Ed. by J.A. Bergstra, A. Ponse, and S.A. Smolka. Amsterdam: Elsevier Science, 2001, pp. 479–543. ISBN: 978-0-444-82830-9. DOI: `https://doi.org/10.1016/B978-044482830-9/50026-6`. URL: `http://www.sciencedirect.com/science/article/pii/B9780444828309500266`.

[14] Marek Kwiatkowski. "Formal computational framework for the study of molecular evolution". In: (2010).

[15] Thomas Wright and Ian Stark. "The Bond-Calculus: A Process Algebra for Complex Biological Interaction Dynamics". In: *arXiv preprint arXiv:1804.07603* (2018).

[16] Stanley Wang. "Modelling Biological Systems as Communicating Processes". In: (2016).

[17] Thomas F. Melham. "A Mechanized Theory of the Pi-Calculus in HOL." In: *Nord. J. Comput.* 1.1 (1994), pp. 50–76.

[18] Jesper Bengtson and Joachim Parrow. "A Completeness Proof for Bisimulation in the π-calculus Using Isabelle". In: *Electronic Notes in Theoretical Computer Science* 192.1 (2007), pp. 61–75.

[19] Jesper Bengtson and Joachim Parrow. "Formalising the Pi-calculus Using Nominal Logic". In: *arXiv preprint arXiv:0809.3960* (2008).

[20] Joost Engelfriet and Tjalling Gelsema. "Multisets and Structural Congruence of the π-calculus with Replication". In: *Theoretical Computer Science* 211.1-2 (1999), pp. 311–337.

# Chapter 8

# Appendices

## A Unordered Pairs in Lean

Unordered pairs are surprisingly easy to represent in Lean, thanks to its implementation of quotients (see Section 2.2).

I mirror [10, §11] by defining a pair which stores two elements of the same type. I then define a relation which states two pairs are equivalent if they have equal elements in either order, namely $(a,b) \approx (a,b)$ and $(a,b) \approx (b,a)$. Showing that this relation is an equivalence relation, allows us to define a setoid for pairs.

```
protected structure pair (α : Type*) := (fst snd : α)

protected def equiv : pair α → pair α → Prop
| ⟨ a₁, b₁ ⟩ ⟨ a₂, b₂ ⟩ := (a₁ = a₂ ∧ b₁ = b₂) ∨ (a₁ = b₂ ∧ a₂ = b₁)

instance : setoid (pair α) := setoid.mk upair.equiv /- ... -/
```

Using these definitions, one may then define the type of unordered pairs as the set of pairs modulo our equivalence relation. This can be done using Lean's `quotient` type.

```
def upair (α : Type*) : Type* := quotient (@upair.setoid α)

protected def upair.mk (a b : α) : upair α := ⟦ ⟨ a, b ⟩ ⟧
```

Two unordered pairs are equal iff their underlying pairs are equivalent. This means that `upair.mk a b = upair.mk b a`, as we might expect.

From the definition of unordered pairs, we can then write functions which operate on them. For instance, given a symmetric function, one can extract a value out of our pair.

```
protected def lift (f : α → α → β)
  : (∀ a b, f a b = f b a) → upair α → β
| comm p := quot.lift_on p (λ p, f p.fst p.snd) (λ ⟨ a₁, b₁ ⟩ ⟨ a₂, b₂ ⟩ r, begin
  rcases r with ⟨ ⟨ _ ⟩, ⟨ _ ⟩ ⟩ | ⟨ ⟨ _ ⟩, ⟨ _ ⟩ ⟩,
  from rfl, from comm _ _,
end)
```

`quot.lift_on` takes the quotient to operate on, a function to apply, and a proof that the function returns the same value for all equivalent pairs. As our function is symmetric, this is easy to show.

## B Example Enzyme System in Lean

This is the complete description of the example enzyme system (Section 2.1.1) within Lean.

Rates of reaction ($\mathbb{H}$) are defined as a polynomial, with string 'variables'. This allows us to define the various rates (such as $k_{\text{bind}}$) as variables, rather than using concrete values.

```
open_locale normalise

def k_bind : ℍ := fin_poly.X "k_bind"
def k_degrade : ℍ := fin_poly.X "k_degrade"
def k_unbind : ℍ := fin_poly.X "k_unbind"
def k_react : ℍ := fin_poly.X "k_react"

def aff : affinity ℍ := affinity.mk_pair k_bind -- x, y

def M : affinity ℍ -- u, r, t
  :=  affinity.mk 3 0 2 k_unbind -- u - t
  ∘[] affinity.mk 3 1 2 k_react -- r - t

def ω : context := context.extend 0 (context.extend M.arity (context.extend 0
↪   (context.extend 0 context.nil)))
def Γ : context := context.extend aff.arity context.nil

def s : name Γ := name.zero ⟨ 0, nat.succ_pos 1 ⟩
def e : name Γ := name.zero ⟨ 1, lt_add_one 1 ⟩

@[pattern] def S : reference 0 ω := reference.zero 0
@[pattern] def E : reference M.arity ω := reference.extend $ reference.zero M.arity
@[pattern] def P₁ : reference 0 ω := reference.extend ∘ reference.extend $
↪   reference.zero 0
@[pattern] def P₂ : reference 0 ω := reference.extend ∘ reference.extend ∘
↪   reference.extend $ reference.zero 0

def x {Γ} : name (context.extend 2 Γ) := name.zero ⟨ 0, nat.succ_pos 1 ⟩
def y {Γ} : name (context.extend 2 Γ) := name.zero ⟨ 1, lt_add_one 1 ⟩

def u {Γ} : name (context.extend M.arity Γ) := name.zero ⟨ 0, nat.succ_pos 2 ⟩
def r {Γ} : name (context.extend M.arity Γ) := name.zero ⟨ 1, int.coe_nat_lt.mp
↪   trivial ⟩
def t {Γ} : name (context.extend M.arity Γ) := name.zero ⟨ 2, lt_add_one 2 ⟩

-- S = s(x, y). (x. S + y. (P|P'))
def S_ : choices ℍ ω Γ :=
  s #( 2 ) •' Σ# ( whole.cons (x #) (apply S ∅)
                 ∘ whole.cons (y #) (apply P₁ ∅ |ₛ apply P₂ ∅)
                 $ whole.empty )
```

```
-- E = ν(u, r, t : M) . e(u, r). t. E)
def E_ : choices ℍ ω (context.extend M.arity Γ) :=
  (name.extend e #⟨ [u, r] ⟩) •' (name.extend t # • ν(M) apply E (u :: r :: t :: ∅))

def P_ : choices ℍ ω Γ := τ@k_degrade •' nil

def ℓ : lookup ℍ ω Γ
| _ S := species.rename name.extend S_
| _ E := E_
| _ P₁ := species.rename name.extend P_
| _ P₂ := species.rename name.extend P_
| (nat.succ n) (reference.extend (reference.extend a))
  := by { cases a, cases a_a, cases a_a_a }

/- Various intermediates -/
def E'_ {Γ} : species ℍ ω Γ := ν(M) apply E (u :: r :: t :: ∅)
def C'_ : species ℍ ω Γ :=
  ν(M) ( ( Σ# ( whole.cons (u#) (apply S ∅)
              $ whole.cons (r#) (apply P₁ ∅ |ₛ apply P₂ ∅)
              $ whole.empty ) )
       |ₛ t# • E'_)

def ℂ : Type := fin_poly (species ℍ ω Γ) ℍ
instance : half_ring ℂ := fin_poly.half_ring _ _
instance : has_repr ℂ := fin_poly.has_repr _ _
def conc : ℍ ↪ ℂ := fin_poly.C.embed

def system : process ℂ ℍ ω Γ :=
  fin_poly.X "S" ○ (apply S ∅) |ₚ
  fin_poly.X "E" ○ E'_ |ₚ
  fin_poly.X "S" ○ C'_ |ₚ
  fin_poly.X "P₁" ○ (apply P₁ ∅) |ₚ
  fin_poly.X "P₂" ○ (apply P₂ ∅)

#eval process_immediate aff ℓ conc system
```

## C Computing Prime Species

We define a Lean version for a classical procedure to compute primes, as given in [14, Appendix A].

This uses a custom well-founded measure to prove termination, much like the original proof. The proofs of `have _ : depth _ < depth A` scattered within the definition aid Lean in showing the function terminates.

```
noncomputable def do_prime_decompose {Γ} :
  ∀ (A : species ℍ ω Γ)
  , Σ' (As : list (prime_species ℍ ω Γ))
    , A ≈ parallel.from_list (list.map subtype.val As)
| A :=
  if is_nil : A ≈ nil then
    ⟨ [], is_nil ⟩
  else if has_decomp : ∃ B C, ¬ B ≈ nil ∧ ¬ C ≈ nil ∧ A ≈ (B |ₛ C) then
    let B := classical.some has_decomp in
    let C := classical.some (classical.some_spec has_decomp) in
    have h : ¬B ≈ nil ∧ ¬C ≈ nil ∧ A ≈ (B |ₛ C) := classical.some_spec
     ↪  (classical.some_spec has_decomp),
    have lB : depth B < depth A := begin
        have : depth A = depth (B |ₛ C) := depth_eq h.2.2, rw this, unfold depth,
        from lt_add_of_pos_right _ (nat.pos_of_ne_zero (λ x, h.2.1 (depth_nil_rev
         ↪  x)))
      end,
    have lC : depth C < depth A := begin
        have : depth A = depth (B |ₛ C) := depth_eq h.2.2, rw this, unfold depth,
        from lt_add_of_pos_left _ (nat.pos_of_ne_zero (λ x, h.1 (depth_nil_rev x)))
      end,
    let Bs := do_prime_decompose B in
    let Cs := do_prime_decompose C in
    suffices this : A ≈ parallel.from_list (list.map subtype.val (Bs.1 ++ Cs.1)),
      from ⟨ Bs.1 ++ Cs.1, this ⟩,
    calc  A
        ≈ (B |ₛ C) : h.2.2
    ... ≈ (parallel.from_list (list.map subtype.val Bs.1) |ₛ parallel.from_list
     ↪  (list.map subtype.val Cs.1))
        : trans (equiv.ξ_parallel₁ Bs.2) (equiv.ξ_parallel₂ Cs.2)
    ... ≈ parallel.from_list (list.map subtype.val Bs.1 ++ list.map subtype.val Cs.1)
          : (parallel.from_append _ _).symm
    ... ≈ parallel.from_list (list.map subtype.val (Bs.1 ++ Cs.1)) : by rw
     ↪  list.map_append
  else
    suffices this : prime A, from ⟨ [ ⟨ A, this ⟩ ], refl _ ⟩,
    ⟨ is_nil, λ B C eq,
      if nilB : B ≈ nil then or.inl nilB else
      if nilC : C ≈ nil then or.inr nilC else
      false.elim (has_decomp ⟨ B, C, nilB, nilC, eq ⟩) ⟩
using_well_founded {
  rel_tac := λ _ _, `[exact ⟨_, measure_wf depth ⟩ ],
  dec_tac := tactic.fst_dec_tac',
}
```

## D Remaining axioms

These are the remaining axioms within our Lean code. We discuss these in more detail in Section 6.1

### Transition systems

```
axiom equivalent_of.map_map {Γ ℓ} {A B : species ℍ ω Γ} (h : species.equivalent A B)
    {k} {α : label ℍ Γ k} (t : Σ (E : production ℍ ω Γ k), A [ℓ, α]⟶ E)
  : equivalent_of.map h.symm (equivalent_of.map h t) = t
```

### Prime decomposition for structural congruence

```
axiom prime_decompose_parallel {Γ} (A B : species ℍ ω Γ)
  : prime_decompose (A |ₛ B)
  = prime_decompose A + prime_decompose B
axiom prime_decompose_equiv {Γ} {A B : species ℍ ω Γ}
  : A ≈ B
  → multiset.map quotient.mk (prime_decompose A)
  = multiset.map quotient.mk (prime_decompose B)
```

### Normalisation and n-equivalence

```
axiom drop_atom :
    ∀ {Γ} {sk} {k : kind' ℍ sk} {n} {A : whole ℍ ω sk (context.extend n Γ)}
      (h : level.zero ∉ A)
    , atom k A → atom k (drop h)

axiom normalise_to.prime {Γ} (A : species ℍ ω Γ)
  : prime A → (normalise_to A).fst = [A]
```

**Semantics** This axiom shows that there is an embedding from process spaces to processes, rather than a simple function. While we do not rely on it within our work, I thought it would be an interesting property to show.

```
axiom process.from_inverse {Γ} :
  function.left_inverse process.to_space' (@process.from_space ℍ ω _ ℂ _ Γ)
```

## E Synthesis and Degradation in Lean

*Synthesis and Degradation*[16, §3.2.1], is a simple example of a common behaviour within biochemical systems. A signal $S$ promotes the generation of product $R$, which happens at rate $k_1$. $R$ itself is produced at an ambient rate $k_2$, and decays at rate $k_2$. This can translated to the following c$\pi$ system[16, §4.1], and the Lean code given in Listing E.1.

$$\begin{aligned} A &\overset{\text{def}}{=} \tau@k_0.(A \mid R) \\ S &\overset{\text{def}}{=} \tau@k_1.(S \mid R) \\ R &\overset{\text{def}}{=} \tau@k_2.\mathbf{0} \end{aligned}$$

This evaluates to `(-1•(R•k₂) + 1•(S•k₁) + 1•(k₀)) • 2([])`, where `2([])` refers to a species invocation of $R$. This is equivalent to the differential equation $\frac{dR}{dt} = k_0 + k_1 S - k_2 R$, which is correct.

## F Phosphorylation and Dephosphorylation in Lean

*Phosphorylation* is the process of adding a phosphate group, via the signal $S$, to a molecule $R$ at rate $k_1$, producing another molecule $RP$. This the decomposes back to the original $R$ molecule at rate $k_2$[16, §3.2.2]. This can be translated into c$\pi$[16, §4.2] and then Lean, as seen in Listing F.1.

```
def k_ambient : ℍ := fin_poly.X "k₀"
def k_react : ℍ := fin_poly.X "k₁"
def k_degrade : ℍ := fin_poly.X "k₂"

def aff : affinity ℍ := ∅

@[pattern] def A : reference 0 ω := reference.zero 0
@[pattern] def S : reference 0 ω := reference.extend $ reference.zero 0
@[pattern] def R : reference 0 ω := reference.extend ∘ reference.extend $
 ↪  reference.zero 0

def A_ : choices ℍ ω Γ := τ@k_ambient •' (apply A ∅ |ₛ apply R ∅)
def S_ : choices ℍ ω Γ := τ@k_react •' (apply S ∅ |ₛ apply R ∅)
def R_ : choices ℍ ω Γ := τ@k_degrade •' nil

def ℓ : lookup ℍ ω Γ
| _ A := species.rename name.extend A_
| _ S := species.rename name.extend S_
| _ R := species.rename name.extend R_
| (nat.succ n) (reference.extend (reference.extend a)) := by { cases a, cases a_a }

def system : process ℂ ℍ ω Γ :=
  1 ◯ (apply A ∅) |ₚ
  fin_poly.X "S" ◯ (apply S ∅) |ₚ
  fin_poly.X "R" ◯ (apply R ∅)

#eval process_immediate aff ℓ conc system
```

Listing E.1: Synthesis and Degradation in Lean.

```
def k_react : ℍ := fin_poly.X "k₁"
def k_degrade : ℍ := fin_poly.X "k₂"

def aff : affinity ℍ := affinity.mk_pair k_react

@[pattern] def R : reference 0 ω := reference.zero 0
@[pattern] def S : reference 0 ω := reference.extend $ reference.zero 0
@[pattern] def RP : reference 0 ω := reference.extend ∘ reference.extend $
↪  reference.zero 0

def a {Γ} : name (context.extend 2 Γ) := name.zero 0
def b {Γ} : name (context.extend 2 Γ) := name.zero 1

def R_ : choices ℍ ω Γ := a# •' apply RP ∅
def S_ : choices ℍ ω Γ := b# •' apply S ∅
def RP_ : choices ℍ ω Γ := τ@k_degrade •' apply R ∅

def ℓ : lookup ℍ ω Γ := λ n a, begin
  cases a with _ _ _ _ _ a, from species.rename name.extend R_,
  cases a with _ _ _ _ _ a, from species.rename name.extend S_,
  cases a with _ _ _ _ _ a, from species.rename name.extend RP_,
  cases a with _ _ _ _ _ a,
end

def system : process ℂ ℍ ω Γ :=
  fin_poly.X "S" ◯ (apply S ∅) |ₚ
  fin_poly.X "R" ◯ (apply R ∅) |ₚ
  fin_poly.X "RP" ◯ (apply RP ∅)

#eval process_immediate aff ℓ conc system
```

Listing F.1: Phosphorylation in Lean.

$$R \overset{\text{def}}{=} a.RP$$
$$S \overset{\text{def}}{=} b.S$$
$$RP \overset{\text{def}}{=} \tau@k_2.R$$

This produces ODEs $\frac{dRP}{dt} = k_1 S \cdot R - k_2 RP$ and $\frac{dR}{dt} = k_2 RP - k_1 S \cdot R$, as we might expect. Wang also presents an alternative translation into c$\pi$, as the above did not work with original C$\pi$-IDE. Thankfully, our implementation behaves correctly, so I did not require this alternative.

## G Perfectly Adapted Response in Lean

This system extends the simple example of Appendix E, adding a second signalling pathway using an additional species $X$. A *perfect adapation* system has a response element, which is independent from the initial signal S. Sudden changes to the signal will produce a response, but after time this returns back to the steady state.

The translation to c$\pi$ is more complex than the synthesis and degradation example[16, p. 4.4]. Species R and X communicate on sites a and b, reacting at rate $k_2$. The signal $S$ produces either $R$ or $X$, at rates $k_1$ and $k_3$ respectively.

$$\begin{aligned}
R &\overset{\text{def}}{=} a.\mathbf{0} \\
S &\overset{\text{def}}{=} \tau@k_1(S \mid R) + \tau@k_3(S \mid X) \\
X &\overset{\text{def}}{=} \tau@k_4\mathbf{0} + b.X
\end{aligned}$$

The Lean translation (Listing G.1) produces two ODEs, $\frac{dR}{dt} = k_1S - k_2X \cdot R$ and $\frac{dX}{dt} = k_3S - k_4X$, which are identical to the expected behaviour.

```
def k1 : ℍ := fin_poly.X "k₁"
def k2 : ℍ := fin_poly.X "k₂"
def k3 : ℍ := fin_poly.X "k₃"
def k4 : ℍ := fin_poly.X "k₄"

def aff : affinity ℍ := affinity.mk_pair k2

@[pattern] def R : reference 0 ω := reference.zero 0
@[pattern] def S : reference 0 ω := reference.extend $ reference.zero 0
@[pattern] def X : reference 0 ω := reference.extend ∘ reference.extend $
↪  reference.zero 0

def a {Γ} : name (context.extend 2 Γ) := name.zero 0
def b {Γ} : name (context.extend 2 Γ) := name.zero 1

def R_ : choices ℍ ω Γ := a# •' nil
def S_ : choices ℍ ω Γ
  := whole.cons τ@k1 (apply S ∅ |ₛ apply R ∅)
   ∘ whole.cons τ@k3 (apply S ∅ |ₛ apply X ∅)
   $ whole.empty
def X_ : choices ℍ ω Γ
  := whole.cons τ@k4 nil
   ∘ whole.cons (b#) (apply X ∅)
   $ whole.empty

def ℓ : lookup ℍ ω Γ := λ n a, begin
  cases a with _ _ _ _ _ a, from species.rename name.extend R_,
  cases a with _ _ _ _ _ a, from species.rename name.extend S_,
  cases a with _ _ _ _ _ a, from species.rename name.extend X_,
  cases a with _ _ _ _ _ a,
end

def system : process ℍ ℍ ω Γ :=
  fin_poly.X "S" ◯ (apply S ∅) |ₚ
  fin_poly.X "R" ◯ (apply R ∅) |ₚ
  fin_poly.X "X" ◯ (apply X ∅)

#eval process_immediate aff ℓ (function.embedding.refl _) system
```

Listing G.1: Perfectly Adapted Response in Lean.