Formalizing π-Calculus in Guarded Cubical Agda

Niccolò Veltri
Department of Software Science
Tallinn University of Technology
Estonia
niccolo@cs.ioc.ee

Andrea Vezzosi
Department of Computer Science
IT University of Copenhagen
Denmark
avez@itu.dk

Abstract

Dependent type theories with guarded recursion have shown themselves suitable for the development of denotational semantics of programming languages. In particular Ticked Cubical Type Theory (TCTT) has been used to show that for guarded labelled transition systems (GLTS) interpretation into the denotational semantics maps bisimilar processes to equal values. In fact the two notions are proved equivalent, allowing one to reason about equality in place of bisimilarity.

We extend that result to the π-calculus, picking early congruence as the syntactic notion of equivalence between processes, showing that denotational models based on guarded recursive types can handle the dynamic creation of channels that goes beyond the scope of GLTSs.

Hence we present a fully abstract denotational model for the early π-calculus, formalized as an extended example for Guarded Cubical Agda: a novel implementation of Ticked Cubical Type Theory based on Cubical Agda.

CCS Concepts • **Theory of computation → Process calculi**; **Type theory**; **Denotational semantics**.

Keywords ticked cubical type theory, denotational semantics, π-calculus, guarded recursion

ACM Reference Format:
Niccolò Veltri and Andrea Vezzosi. 2020. Formalizing π-Calculus in Guarded Cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372885.3373814

1 Introduction

Developing the denotational semantics of a programming language with non-trivial effects, such as concurrency and non-determinism, can quickly lead to fairly involved constructions, e.g. domain theory and powerdomains, which then can be cumbersome to manipulate when trying to establish properties of the semantics, and a barrier for their widespread adoption. Nakano [2000] introduced a modality to give an axiomatic view of recursion as the limit of a sequence of approximations. Since then the modality has been redubbed $\triangleright$, pronounced "later", and extended to dependent type theory as a way to reproduce step-indexing and domain theoretic arguments without having to directly deal with the indexing nor the domains, [Birkedal et al. 2019, 2011; Bizjak et al. 2016]. The expressive power of the later modality comes from a guarded fixpoint combinator $\mathsf{fix}_A : (\triangleright A \to A) \to A$ ensuring the existence of a solution for all guarded recursive equations in any type. Logically, this corresponds to a version of Löb's axiom for the $\triangleright$ modality and enables complex patterns of recursion, including (guarded) negative recursive types. Its applicability was demonstrated by Paviotti et al's formalization of PCF [Paviotti et al. 2015] and Møgelberg and Paviotti's formalization of FPC [Møgelberg and Paviotti 2019] in Guarded Dependent Type Theory [Bizjak et al. 2016], while Møgelberg and Veltri [2019] develop the interaction of the later modality with Higher Inductive Types (HITs), and in particular use the finite powerset to handle non-determinism and formalize a denotational semantics for Milner's Calculus of Communicating Systems (CCS). While the above results are formal in the sense of being obtained in a formal language, they have not been machine verified because of the lack of native support for the later modality in proof assistants.

We remedy this situation by introducing Guarded Cubical Agda (GCA): a proof assistant based on Cubical Agda [Vezzosi et al. 2019] with support for guarded recursion. Hence this paper is also meant as an introduction and demonstration of the current features of Guarded Cubical Agda and all the results we present are formalized in GCA. We will include selected parts of the code to illustrate the use of the $\triangleright$ modality. In particular GCA implements the presentation adopted by Ticked Cubical Type Theory (TCTT) [Møgelberg and Veltri 2019], by extending the typechecking algorithm to handle the non-structural scoping of TCTT's tick variables, which are used to introduce and eliminate elements of $\triangleright A$. We hope in the future to extend the implementation to cover more features of Guarded Dependent Type Theory,

such as clock quantification, which would enable to derive coinductive types from guarded ones.

To further demonstrate the use of guarded recursion for the denotational semantics of process calculi, in this paper we tackle Milner's π-calculus [Milner et al. 1993], which is the base for many other calculi. Compared to CCS, the π-calculus gains more expressivity by allowing processes to locally create new channels and then share them dynamically with other processes. This extra flexibility, combined with the ability to compare channel names, adds an extra challenge to developing a model, on top of what is needed for concurrency. Traditionally, the denotational semantics of π-calculus is developed in specific categories of functors from the category of injective renamings Inj, for handling names, to the category of profunctors [Cattani et al. 1997] or the category of domains [Fiore et al. 1996; Stark 1996], in which is possible to handle recursion and non-determinism. We handle names in the form of well-scoped de Bruijn indexes, while we manage recursion and non-determinism via guarded recursion and the countable powerset as a HIT. We obtain a denotational model whose equality corresponds to guarded early congruence of processes, i.e. guarded early bisimilarity for every possible renaming. We chose the early semantics as for a certain class of processes early congruence corresponds to barbed congruence, but the technique could be applied to the late semantics as well. Our bisimilarity is guarded in the sense that the processes onto which the original ones step only have to be later bisimilar. This means that in some cases the bisimilarity of two processes cannot be outright refuted, because it only implies false at the next time step. This shortcoming would be addressed by the introduction of clock quantification into the theory, as it allows to encapsulate time dependencies and safely project out of the later modality.

Supplementary Material. Guarded Cubical Agda and the full Agda formalization are submitted as supplementary material to the paper. In the file README.agda we import all the relevant modules and we describe what they implement.

Structure of the Paper. In Section 2 we give an introduction to Guarded Cubical Agda. First we give an overview of Cubical Agda, then we add the support for guarded recursion via ticks. In the second half of the section we present some programming examples to showcase the new features. We also recall some basic notions from Homotopy Type Theory that are employed in the rest of the paper. In Section 3 we introduce the countable powerset datatype as a HIT. In Section 4 we define the syntax of the early π-calculus, together with a structural congruence and an early operational semantics. In Section 5 we present a sound interpretation of π-calculus in a denotational model constructed using the countable powerset and guarded recursion. This interpretation is also computationally sound, meaning that the dynamic behaviour of syntactic processes, specified by the operational semantics, is reflected in the denotational universe via a semantic transition relation. In Section 6 we introduce syntactic early congruence and prove the denotational semantics fully abstract with respect to it. In Section 7 we discuss related work and in the final Section 8 we summarize our achievement and consider possible future directions of work.

2 Guarded Cubical Agda

2.1 Cubical Agda

Cubical Agda implements the features of Cubical Type Theory (CTT) [Cohen et al. 2018], which make it possible to support principles like univalence and function extensionality [Univalent Foundations Program 2013], without introducing axioms that would block computation.

In CCT, univalence is a theorem that states that equality of types corresponds to equivalence. A function $f : A \to B$ is an equivalence if it has "contractible fibers", i.e., if the preimage of any element in B is a singleton type. Any function with an inverse is an equivalence. We write $A \simeq B$ for the type of equivalences between A and B, so univalence is more formally phrased as $(A \equiv B) \simeq (A \simeq B)$. In particular we have a function $\mathsf{ua} : A \simeq B \to A \equiv B$ which turns equivalences into equalities. Since from any proof of equality built as $\mathsf{ua}\ e$ we need to be able to extract the equivalence e, this means that the representation of equality has to accommodate such information. Cubical Type Theory takes inspiration from the topological interpretation of Homotopy Type Theory and represents equalities as paths, i.e. maps from an interval object.

In Cubical Agda we have a primitive interval type I with two endpoints $\mathsf{i0}$ and $\mathsf{i1}$[1]. On top of that, PathP is the primitive type of paths in a family of types $\mathrm{A} : \mathsf{I} \to \mathsf{Set}$,

$$\mathsf{PathP} : \forall\ \{\ell\}\ (A : \mathsf{I} \to \mathsf{Set}\ \ell) \to A\ \mathsf{i0} \to A\ \mathsf{i1} \to \mathsf{Set}\ \ell$$

An element $p : \mathsf{PathP}\ A\ a_0\ a_1$ is eliminated by application to an element $\mathrm{r} : \mathsf{I}$ of the interval, obtaining $p\ r$ of type $A\ r$. Unlike a function type, such an application can compute even when p is unknown by using the endpoints a_0 and a_1 stored in the type: $p\ \mathsf{i0} = a_0$ and $p\ \mathsf{i1} = a_1$. Analogously, creating a path is done with a lambda abstraction $\lambda\ i \to \mathrm{t} : \mathsf{PathP}\ A\ a_0\ a_1$, but this incurs the extra requirement to match the endpoints: $\mathrm{t}(\mathsf{i0}/\mathrm{i}) = a_0$ and $\mathrm{t}(\mathsf{i1}/\mathrm{i}) = a_1$.

Using PathP we can define the type of path equalities in a type A,

$$\_\equiv\_ : \forall\ \{\ell\}\ \{A : \mathsf{Set}\ \ell\} \to A \to A \to \mathsf{Set}\ \ell$$
$$\_\equiv\_\ \{\_\}\ \{A\}\ x\ y = \mathsf{PathP}\ (\lambda\ \_ \to A)\ x\ y$$

which we will use throughout the paper. Here $\mathsf{Set}\ \ell$ is the notation used by Agda for the type of types, often called a universe, at level ℓ. Universes are stratified into levels to

[1] Plus min, max and reversal operations that turn it into a De Morgan Algebra

avoid Russell-style paradoxes, but we will omit the levels for most of our presentation since they do not play any important role for us.

With this setup, a proof of function extensionality, i.e. that pointwise equal functions are equal, is a simple matter of swapping the order of two arguments,

```
funExt : {f g : A → B} → ((x : A) → f x ≡ g x) → f ≡ g
funExt p i x = p x i
```

More details on how to program and reason with path types can be found in Vezzosi et al. [2019].

2.2 Adding Guarded Recursion via Ticks

Guarded Cubical Agda introduces the ▷ modality by piggybacking on Agda's support for annotated Π types: the implementation introduces a type Tick and a new annotation @tick allowing one to write (@tick α : Tick) → A to denote the type ▷ α.A of TCTT, (c.f. Figure 1). In the rest of the paper we will use the following abbreviations: ▷ A when A does not use α, and ▶ A when it does.

```
▷_ : ∀ {l} → Set l → Set l
▷_ A = (@tick x : Tick) → A

▶_ : ∀ {l} → ▷ Set l → Set l
▶ A = (@tick x : Tick) → A x
```

The type ▶ A is introduced by lambda abstraction, $\lambda\ \alpha \to t$, and eliminated by application, $t\ \alpha$, like a standard function type. However, applying a tick introduces extra restrictions on the term t, as it can only mention variables bound before α. This prevents the construction of a term like $\lambda\ x\ \alpha \to x\ \alpha$ α : ▷▷ A → ▷ A which would move elements from two time steps in the future to just one. What we can have is a map delaying an element to the next time step

```
next : A → ▷ A
next x α = x
```

Interval variables are exempt from the restriction imposed by tick application, as the interval is unaffected by time steps in the model. This allows us to implement an extensionality principle and its inverse, giving a full characterization of equality in ▶ A.

```
later-ext : ∀ {f g : ▶ A} → (▶ λ α → f α ≡ g α) → f ≡ g
later-ext eq i a = eq a i

later-ext-inv : ∀ {f g : ▶ A} → f ≡ g → ▶ λ α → f α ≡ g α
later-ext-inv eq α i = eq i α
```

The last ingredient we will make use of is a guarded fixpoint combinator fix together with a proof fix-eq of its unfolding.

```
fix    : (f : ▷ A → A) → A

fix-eq : (f : ▷ A → A) → fix f ≡ f (next (fix f))
```

We will use fix for defining programs and proving their properties by guarded recursion. In general we would also use fix to define guarded recursive types themselves, however when possible we will rely on Agda's builtin recursive types, like in the following example.

2.2.1 Programming and Reasoning with Ticks

We will exemplify the use of ticks and guarded recursion by defining a map function for the type of finitely branching streams (S A) and proving a simple lemma about it.

We define S A as a record type with two projections, giving us the first element now and a List of tails at the next time step.

```
record S (A : Set) : Set where
  inductive
  constructor _,_
  field
    head : A
    tails : ▷ List (S A)
```

Assuming we have a corresponding mapL for List, we implement a mapping function mapS using guarded recursion.

```
mapS : (A → B) → S A → S B
mapS = fix λ mapS' f xs →
  f (xs .head) , λ α → mapL (mapS' α f) (xs .tails α)
```

The call to fix provides us with the induction hypothesis *mapS'* : ▷ ((A → B) → S A → S B) which we pass to mapL to process the streams in the list of tails. Productivity is guaranteed solely by the fact that we need a tick α to make use of *mapS'*, without having to consider whether mapL preserves it or not. The syntactic guardedness checker of plain Coq or Agda would instead reject a similar definition, complaining that mapL is not an introduction form and thus might force the recursive call too much.

Unfolding the fixpoint in the definition of mapS, we obtain the following equality, witnessed by fix-eq:

```
mapS f xs ≡ f (xs .head) , λ α → mapL (mapS f) (xs .tails α)
```

The ability to combine guarded recursion with existing combinators extends to proofs of equality. For example we can prove that mapS is the identity if mapping the identity function.

```
mapS-id : ∀ (xs : S A) → mapS (λ x → x) xs ≡ xs
```

Starting with fix we can assume *mapS-id'* : ▷ (∀ *xs* → mapS (λ *x* → *x*) *xs* ≡ *xs*) and proceed by this sequence of equalities

```
  mapS (λ x → x) xs
≡ xs .head , λ α → mapL (mapS (λ x → x)) (xs .tails α)
≡ xs .head , λ α → mapL (λ x → x) (xs .tails α)
≡ xs .head , λ α → xs .tails α
= xs
```

$$\frac{\Gamma, \alpha : \mathbb{T} \vdash A}{\Gamma \vdash \triangleright\, \alpha.A} \qquad \frac{\Gamma, \alpha : \mathbb{T} \vdash t : A}{\Gamma \vdash \lambda\, \alpha.t : \triangleright\, \alpha.A} \qquad \frac{\Gamma \vdash t : \triangleright\, \alpha.A}{\Gamma, \beta : \mathbb{T}, \Gamma' \vdash t\ [\beta] : A(\beta/\alpha)} \qquad \frac{\Gamma, \alpha : \mathbb{T}, i : \mathbb{I} \vdash A}{\Gamma, i : \mathbb{I}, \alpha : \mathbb{T} \vdash A} \qquad \frac{\Gamma, \alpha : \mathbb{T}, i : \mathbb{I} \vdash t : A}{\Gamma, i : \mathbb{I}, \alpha : \mathbb{T} \vdash t : A}$$

Figure 1. Selected typing rules of Ticked Cubical Type Theory. The double-line rules are invertible. In TCTT notation, $\mathbb{I}$ is the interval type and $\mathbb{T}$ is the type of ticks.

The first equality follows from fix-eq, while the the second is established through later-ext and *mapS-id'*, the third assumes a similar lemma holds for mapL ($\lambda\ x \to x$), while the last judgmental equality follows from the η-rule for record types. Once again the fact that the inductive hypothesis *mapS-id'* was used only in the context of the tick α is enough to guarantee the proof is well-defined.

2.3 Preliminaries

Here we recall some notions from Homotopy Type Theory and some basic constructions.

It will be useful to distinguish two classes of types: the *(mere) propositions*, for which any two elements are equal, and the *sets*, whose equality type is a proposition.

```
isProp : Set → Set
isProp A = (x y : A) → x ≡ y

isSet : Set → Set
isSet A = (x y : A) → isProp (x ≡ y)
```

The propositional truncation $\| A \|$ is the least proposition with a map from A. We define it as a HIT

```
data ||_|| (A : Set) : Set where
  |_|     : A → || A ||
  squash : ∀ a0 a1 → a0 ≡ a1
```

and derive a recursion principle into any proposition P

```
rec||-|| : ∀ {P : Set} → isProp P → (A → P) → || A || → P
rec||-|| Pprop f | x | = f x
rec||-|| Pprop f (squash x y i) =
  Pprop (rec||-|| Pprop f x) (rec||-|| Pprop f y) i
```

the right hand side of the clause for squash needs to be equal to rec||-|| *Pprop f x* when i = i0, as that is what the left hand side reduces to, and similarly for i = i1. This is easily achieved by using *Pprop* : isProp P.

We write ∃[x : A] P, or ∃[x] P when the type A is understood from the context, for the mere existence of an element x in A satisfying the predicate P, defined as $\| \Sigma\, A\, P \|$. We write Ω for the type of propositions: Σ Set isProp. Given a proposition $P : \Omega$, we write $[P]$: Set for the type underlying P, i.e. the first projection of P. We also have isSet Ω. We define $\bot_\mathsf{p} : \Omega$ as the proposition associated to the empty type, $\lfloor \_ \rfloor_\mathsf{p} : (\mathbb{N} \to \Omega) \to \Omega$ as the countable union of propositions and $x \equiv_\mathsf{p} y : \Omega$ as the proposition associated to the propositional truncation of the path equality $x \equiv y$.

3 The Countable Powerset Functor

The *countable powerset* of a type A, denoted P∞ A, is the type whose elements are the subsets of A with countable cardinality. It is a well-known fact that P∞ A is constructible as the free countably-complete join semilattice on A. A *countably-complete join semilattice* is a partially ordered set $(X, \leq)$ with a bottom element $\bot : X$ and a countable join operation $\bigvee : (\mathbb{N} \to X) \to X$. Countably-complete join semilattices admit an equational presentation as an infinitary algebraic theory. In HoTT and CTT, and therefore in TCTT, it is possible to introduce the free object of an algebraic theory as a HIT. Let A be a type, in Ticked Cubical Agda the free countably-complete join semilattice on A is defined as the following HIT:

```
data P∞ (A : Set) : Set where
  ø       : P∞ A
  η       : (a : A) → P∞ A
  sup     : (s : ℕ → P∞ A) → P∞ A
  comm : ∀ x y   → x ∪ y ≡ y ∪ x
  assoc : ∀ x y z → x ∪ (y ∪ z) ≡ (x ∪ y) ∪ z
  idem : ∀ x     → x ∪ x ≡ x
  unit : ∀ x     → x ∪ ø ≡ x
  bound : ∀ s n  → s n ∪ sup s ≡ sup s
  dist : ∀ s x   → sup s ∪ x ≡ sup (λ n → s n ∪ x)
  trunc : isSet (P∞ A)
```

The binary join operator ∪ is defined mutually with P∞ A and it is given as:

```
_∪_ : {A : Set} → P∞ A → P∞ A → P∞ A
x ∪ y = sup (caseNat x y)
```

where caseNat x y returns x if the input number is 0, otherwise it returns y. The partial order on P∞ A is defined as $x \leq y = (x \cup y) \equiv y$.

This presentation of the countable powerset as a HIT has been introduced by Chapman et al. [2019], inspired by the specification of the finite powerset as a HIT by Frumin et al. [2018]. The type P∞ A is the free countably-complete join semilattice on A by construction. In the types of its constructors, it is possible to identify the algebraic theory of countably-complete join semilattices. The constructor ∅ represents the empty set, η builds singleton subsets and sup is the countable union operator. The 0-truncation constructor forces P∞ A to be a set, i.e., to satisfy the principle of uniqueness of identity proofs.

The dependent eliminator of $\mathrm{P}\infty\, A$ is an induction principle from which freeness (the unique mapping property) can be derived. We spell it out for the case in which the type family we are eliminating into is a family of propositions $P : \mathrm{P}\infty\, A \to \Omega$:

```
IndP∞ : {A : Set} (P : P∞ A → Ω)
  → (pø : [ P ø ])
  → (pη : ∀ a → [ P (η a) ])
  → (psup : ∀ s → (∀ n → [ P (s n) ]) → [ P (sup s) ])
  → ∀ x → [ P x ]
```

The countable powerset datatype is a functor, its action on morphisms is recursively defined as:

```
mapP∞ : {A B : Set} → (A → B) → P∞ A → P∞ B
mapP∞ f ø       = ø
mapP∞ f (η a)   = η (f a)
mapP∞ f (sup x) = sup (mapP∞ f ∘ x)
```

The cases for the higher constructors are omitted. $\mathrm{P}\infty$ is also a monad. Its unit is the singleton constructor η, while its Kleisli extension bindP∞ is defined by recursion similarly to the action on morphisms.

The membership operation $\in$ relates an element $a : A$ with a countable subset $x : \mathrm{P}\infty\, A$. Being a member of a subset is a proposition, hence $a \in x : \Omega$. The membership operation is defined by induction on the subset x.

```
_∈_ : ∀ {A} → A → P∞ A → Ω
a ∈ ø       = ⊥ₚ
a ∈ η b     = a ≡ₚ b
a ∈ sup s   = ⨆ₚ (λ n → a ∈ s n)
```

The omitted cases for the higher constructors are dealt with using the univalence axiom.

We end this section with a characterization of membership in mapP∞ $f\, x$. Stating that b is a member of mapP∞ $f\, x$ corresponds to the mere existence of an element a in the subset x such that $f\, a$ is equal to b.

```
∈mapP∞-eq : {A B : Set} (f : A → B) (b : B) (x : P∞ A)
  → [ b ∈ mapP∞ f x ] ≡ ∃[ a ∶ A ] ([ a ∈ x ] × (b ≡ f a))
```

This is proved by invoking the univalence principle for propositions, also called propositional extensionality, so it is enough to show that the two propositions in the conclusion are logically equivalent. Both implications are proved by induction on the subset x.

4 The Early π-calculus

4.1 Names

For the specification of the π-calculus syntax, we assume the existence of a countable set of *names*. For every natural number n, we assume to be given a type Name n, the set containing the first n names. A function of type Name $n \to$ Name m, for some natural numbers n and m, is called a *renaming*. We write InjRen $n\, m$ for the type of *injective* renamings, that are pairs of a renaming f : Name $n \to$ Name m together with a proof of injectivity. We denote the new (largest) name in Name (suc n) as fresh. The set of names comes with a function ι : Name $n \to$ Name (suc n), embedding a set of names into one containing an additional name. The function ι has a partial inverse down, which is only defined on names different from fresh. For every natural number $n : \mathbb{N}$ we assume to be given a subset enum : $\mathrm{P}\infty$ (Name n) containing all the names in Name n. Equality of names is decidable, and we write $x \stackrel{?}{=} y$ for the type of proofs of x and y being equal or unequal.

Given these assumptions on names, we derive some functions that will be employed throughout the whole development. First we define a function lift, lifting a renaming f acting on n names to a renaming acting on suc n names. The lifting of f behaves like f on the first n names and maps the fresh name in the input set to the fresh name in the output set. Clearly the lifting operation is injective.

```
lift : ∀ {n m} → (Name n → Name m)
  → (Name (suc n) → Name (suc m))
```

Then we consider a function snoc, which extends a renaming f : Name $n \to$ Name m to Name (suc n) by sending the fresh name to a given name v : Name m.

```
snoc : ∀ {n m} (f : Name n → Name m) (v : Name m)
  → Name (suc n) → Name m
```

The last operation is called swap, swapping the two largest names in Name (suc (suc n)). This is also an injective renaming, which we call swapI : InjRen (suc (suc n)) (suc (suc n)).

```
swap : ∀ {n} → Name (suc (suc n)) → Name (suc (suc n))
```

4.2 Syntax

Each process can perform an output, an input or a silent action. The type of actions is indexed by two natural numbers, representing the number of free names and the sum of free and bound names, respectively. In the output constructor, the name ch is the channel onto which the name v is transmitted. In the input constructor, the fresh name in Name (suc n) is transmitted on channel ch. In particular, an input action binds the input name.

```
data Act (n : ℕ) : (m : ℕ) → Set where
  out : (ch v : Name n) → Act n n
  inp : (ch : Name n)   → Act n (suc n)
  τ   : Act n n
```

The π-calculus syntax is standard [Milner et al. 1992]. It includes the nil process end and constructors for prefixing $\cdot$, binary choice $\oplus$, parallel composition ||, restriction ν, replication ! and matching guard. The process guard $x\, y\, P$ is usually denoted $[x = y]P$.

```
data Pi (n : ℕ) : Set where
  end   : Pi n
  _·_   : ∀ {m} (a : Act n m) (P : Pi m) → Pi n
  _⊕_   : (P Q : Pi n)                   → Pi n
  _||_  : (P Q : Pi n)                   → Pi n
  ν     : (P : Pi (suc n))               → Pi n
  !     : (P : Pi n)                     → Pi n
  guard : (v w : Name n) (P : Pi n)      → Pi n
```

Free names in a process can be renamed. The action of Pi on renamings is defined by recursion on the input process.

```
mapPi : ∀ {n m} → (Name n → Name m) → Pi n → Pi m
```

The processes in Pi n are to be considered up to a structural congruence relation $\approx$. In the Agda definition of this relation given below, the equivalence and congruence rules are omitted. This selection of structural rules is the one used by Sangiorgi and Walker [2001]. The relation $\approx$ forces the binary sum and parallel composition operations to be commutative monoids, both with unit end. It characterizes the replication operator in terms of parallel composition: given a process P : Pi n, we have $!P \approx P \,||\, !P$. It also characterizes the matching operator: when guard is given the same name w twice, it is structurally congruent to the identity on processes. The rule $\nu||$ states that restricting a parallel composition of processes, when the right process does not depend on the local variable, is the same as the parallel composition of the restricted first process and the unrestricted second process. The rule swapν states that when restricting a process twice it does not matter in which order we restrict the two local variables. The final rule νend equates the nil process with its restriction.

```
data _≈_ {n : ℕ} : (P Q : Pi n) → Set where
  unit⊕     : ∀ {P}     → P ⊕ end ≈ P
  comm⊕     : ∀ {P Q}   → P ⊕ Q ≈ Q ⊕ P
  assoc⊕    : ∀ {P Q R} → (P ⊕ Q) ⊕ R ≈ P ⊕ (Q ⊕ R)
  unit||    : ∀ {P}     → P || end ≈ P
  comm||    : ∀ {P Q}   → P || Q ≈ Q || P
  assoc||   : ∀ {P Q R} → (P || Q) || R ≈ P || (Q || R)
  repl      : ∀ {P}     → ! P ≈ P || ! P
  guardrefl : ∀ {w P}   → guard w w P ≈ P
  ν||       : ∀ {P Q}   → ν (P || mapPi ι Q) ≈ ν P || Q
  swapν     : ∀ {P}     → ν (ν P) ≈ ν (ν (mapPi swap P))
  νend      : ν end ≈ end
```

4.3 Operational Semantics

The early operational semantics presented here has first been introduced by Milner et al. [1993]. The type of transition labels include a silent action τ, free and bound outputs out and bout, and free and bound inputs inp and binp. The original presentation has only one notion of input labels, but the operational semantics has special cases for when the transmitted name is fresh, for which we use binp. Similarly to type actions, the type of labels is also indexed by the number of free names and the sum of free and bound names.

```
data Label (n : ℕ) : ℕ → Set where
  τ     : Label n n
  out   : (ch v : Name n) → Label n n
  bout  : (ch : Name n)   → Label n (suc n)
  inp   : (ch z : Name n) → Label n n
  binp  : (ch : Name n)   → Label n (suc n)
```

The operational semantics is displayed in Figure 2. It is given as a ternary transition relation _[_]↦_ relating a process P : Pi n, a label a : Label $n\,m$ and a second process Q : Pi m. We are defining an *early* operational semantics, since renaming of input names happens during the execution of an input action, not in the communication phase. Having labels for both free and bound input implies the presence of two possible transitions from a process trying to input on a channel ch. In the rule INP, the fresh name in the process P is substituted with the passed value v. In contrast, in the rule BINP, a transition happens without receiving any value, implying that the bound parameter will be instantiated at a later stage. The CONV rule allows us to factor out some repetition, e.g. we do not need rules for $!P$, and the structural congruence relation will be easy enough to handle in our denotational model. All the other transitions in Figure 2 are standard.

We conclude this section with some definitions that will be employed in Section 5.3. The function labelBinds takes a label a : Label $n\,m$ and returns a function on natural numbers which is the successor function suc if a is a binding action, and is the identity otherwise.

```
labelBinds : ∀{n m} → Label n m → ℕ → ℕ
```

Using labelBinds, we define a function mapLabel, which renames all the free names in a given label.

```
mapLabel : ∀{n m k} (f : Name n → Name m)
  → (a : Label n k) → Label m (labelBinds a m)
```

We also define a function labelLift, which takes a label a : Label $n\,m$ and a renaming f, and returns lift f if a is a binding action, and it returns f otherwise.

```
labelLift : ∀{n m k} (a : Label n m) (f : Name n → Name k)
  → Name m → Name (labelBinds a k)
```

5 Denotational Semantics

As the denotational semantic domain, we modify and adapt to TCTT the categorical model of Cattani et al. [1997]. The starting point for understanding this construction is viewing the syntax of a process calculus, together with its operational semantics, as a labelled transition system (LTS). An LTS can be modelled categorically as a coalgebra f :

$$\frac{}{\mathsf{out}\ ch\ v \cdot P\ [\ \mathsf{out}\ ch\ v\ ]\!\mapsto P}\ \text{OUT} \qquad \frac{}{\mathsf{inp}\ ch \cdot P\ [\ \mathsf{inp}\ ch\ v\ ]\!\mapsto \mathsf{mapPi}\ (v/\mathsf{fresh})\ P}\ \text{INP} \qquad \frac{}{\mathsf{inp}\ ch \cdot P\ [\ \mathsf{binp}\ ch\ ]\!\mapsto P}\ \text{BINP}$$

$$\frac{}{\tau \cdot P\ [\ \tau\ ]\!\mapsto P}\ \text{TAU} \qquad \frac{P\ [\ a\ ]\!\mapsto P'}{P \parallel Q\ [\ a\ ]\!\mapsto P' \parallel Q}\ \text{PAR} \qquad \frac{P\ [\ \mathsf{out}\ ch\ v\ ]\!\mapsto P' \quad Q\ [\ \mathsf{inp}\ ch\ v\ ]\!\mapsto Q'}{P \parallel Q\ [\ \tau\ ]\!\mapsto P' \parallel Q'}\ \text{COM}$$

$$\frac{P\ [\ \mathsf{bout}\ ch\ w\ ]\!\mapsto P' \quad Q\ [\ \mathsf{binp}\ ch\ w\ ]\!\mapsto Q'}{P \parallel Q\ [\ \tau\ ]\!\mapsto \nu\ (P' \parallel Q')}\ \text{CLOSE} \qquad \frac{P\ [\ a\ ]\!\mapsto P'}{P \oplus Q\ [\ a\ ]\!\mapsto P'}\ \text{SUM} \qquad \frac{P\ [\ a\ ]\!\mapsto P'}{\nu\ P\ [\ a\ ]\!\mapsto \nu\ P'}\ \text{RES}$$

$$\frac{P\ [\ \mathsf{out}\ ch\ \mathsf{fresh}\ ]\!\mapsto P'}{\nu\ P\ [\ \mathsf{bout}\ ch\ ]\!\mapsto P'}\ \text{OPEN} \qquad \frac{P\ [\ a\ ]\!\mapsto Q \quad P \approx P' \quad Q \approx Q'}{P'\ [\ a\ ]\!\mapsto Q'}\ \text{CONV}$$

Figure 2. The early operational semantics, in the style of Milner et al. [1993].

$X \to F\,X$, for a certain endofunctor F on a certain category $\mathbb{C}$ [Jacobs 2016]. Intuitively, objects in the category $\mathbb{C}$ are sets of states and the functor F characterizes the branching of transition systems, as in describing the set of possible transitions and successive states. A coalgebra f associates to each state x the collection of transitions initiating from x. The final coalgebra of F then provides a universe for the denotational semantics of an LTS, as each coalgebra f gives rise to a unique map of coalgebras from states X to the final coalgebra of F.

In our setting, we choose $\mathbb{C}$ to be the category $\mathsf{Set}^{\mathsf{Inj}}$ of covariant presheaves over the category Inj of injective renamings. In Inj, objects are natural numbers and morphisms between n and m are elements of InjRen $n\,m$. A presheaf over Inj is therefore a type family $X : \mathbb{N} \to \mathsf{Set}$ acting on injective renamings. We consider the endofunctor F on $\mathsf{Set}^{\mathsf{Inj}}$ to be $\mathsf{F}\,X\,n := \mathsf{P}\infty\,(\mathsf{Step}\,X\,n)$, where $\mathsf{Step}\,Y\,n := \Sigma\{m : \mathbb{N}\}.\ \mathsf{Label}\,n\,m \times Y\,m$. Given a presheaf X, we call actF X the action of $\mathsf{F}\,X$ on injective renamings. We call mapF the *internal* functorial action of F:

```
mapF : ∀ {X Y n} → (∀ {m} → InjRen n m → X m → Y m)
  → F X n → F Y n
```

This action is internal since the type $\forall\{m\} \to \mathsf{InjRen}\,n\,m \to X\,m \to Y\,m$ is the exponential Y^X (modulo naturality condition) of presheaves X and Y in $\mathsf{Set}^{\mathsf{Inj}}$.

The π-calculus syntax Pi, which in Section 4.2 we have shown acting on (non-necessarily injective) renamings via mapPi, is an object of $\mathsf{Set}^{\mathsf{Inj}}$ and it carries an F-coalgebra structure $\mathsf{step} : \forall\{n\} \to \mathsf{Pi}\,n \to \mathsf{F}\,\mathsf{Pi}\,n$, recursively defined on the input process. The coalgebra step completely characterizes the dynamic behaviour of the early π-calculus, that is step P is the set of transitions initiating from P. More precisely, each transition $P\,[a]\!\mapsto Q$ in the inductively defined operational semantics of Figure 2 corresponds to a proof of (a, Q) being a member of step P, modulo structural congruence $\approx$:

```
opsem-eq : ∀ {n m} (P : Pi n) a (Q : Pi m)
  → ∥ P [ a ]↦ Q ∥ ≡ (a , Q) ≈∈≈step P
```

where $(a, Q) \approx\!\in\!\approx \mathsf{step}\,P$ states that there merely exists processes P' and Q' such that $P \approx P'$, $Q \approx Q'$ and $(a\,,\,Q')$ is a member of step P':

```
_≈∈≈step_ : ∀ {n} → Step Pi n → Pi n → Set
(a , Q) ≈∈≈step P =
  ∃[ P' ] ∃[ Q' ] (P ≈ P' × Q ≈ Q' × [ (a , Q') ∈ step P' ])
```

Let us analyze the behavior of step on the matching operator guard $x\,y$, since this motivates the choice to consider only *injective* renamings in the indexing category Inj. Given $P : \mathsf{Pi}\,n$, the set of transitions starting from the process guard $x\,y\,P$ is the set of transitions starting from P if $x = y$, while it is the empty set for $x \neq y$. If the category Inj contains also non-injective renamings, then the coalgebra step would not be a morphism in $\mathsf{Set}^{\mathsf{Inj}}$, i.e. a natural transformation, between the presheaves Pi and F Pi. In fact, given a process P and a renaming function f sending two distinct names x and y to a common name z, we would have

$$\mathsf{actF}\ \mathsf{Pi}\ f\ (\mathsf{step}\ (\mathsf{guard}\ x\ y\ P)) = \mathsf{actF}\ \mathsf{Pi}\ f\ \emptyset = \emptyset$$

while

$$\begin{aligned}&\mathsf{step}\ (\mathsf{mapPi}\ f\ (\mathsf{guard}\ x\ y\ P))\\ &= \mathsf{step}\ (\mathsf{guard}\ z\ z\ (\mathsf{mapPi}\ f\ P))\\ &= \mathsf{step}\ (\mathsf{mapPi}\ f\ P)\end{aligned}$$

which is non-empty whenever mapPi $f\,P$ can perform transitions. So the naturality condition of step would fail if it involved arbitrary renamings like f, while injective renamings do not cause this problem as they cannot conflate names.

For the denotational semantic domain, we consider the guarded recursive type

```
record Proc (n : ℕ) : Set where
  inductive
  constructor Fold
  field
    Unfold : P∞ (Step (λ m → ▹ Proc m) n)
```

Proc is the final coalgebra of the functor $\mathsf{F'}\,X := \mathsf{F}\,(\lambda\,m \to \triangleright\,X\,m)$ and satisfies the type equivalence:

$$\mathsf{Proc}\ n \simeq \mathsf{P}\infty\,(\Sigma(m : \mathbb{N}).\ \mathsf{Label}\ \ n\,m \times \triangleright\,\mathsf{Proc}\ m)$$

This implies that, given a coalgebra $f : \forall\{n\} \to X\,n \to$ F' $X\,n$, there exists a unique coalgebra morphism $\mathsf{eval}_f : \forall\{n\} \to X\,n \to \mathsf{Proc}\,n$. The final coalgebra of F can be constructed in an extension of TCTT with clocks using universal clock quantification, similarly to how coinductive types are derived from guarded recursive types in Guarded Dependent Type Theory [Bizjak et al. 2016]. The study of this extended type system and the lifting of the development presented in this paper to the extended setting are left for future work.

Proc acts on injective renamings via a function mapProc.

$$\mathsf{mapProc} : \forall\,\{n\ m\} \to \mathsf{InjRen}\ n\ m \to \mathsf{Proc}\ n \to \mathsf{Proc}\ m$$

We call elements of Proc n *semantic processes*, or simply processes when it is clear from context that we are working in the denotational universe. A semantic process is a countably-branching, possibly non-wellfounded tree. Given a process $P : \mathsf{Proc}\,n$, members of Unfold P are pairs in Step $(\lambda\,m \to \triangleright\,\mathsf{Proc}\,m)\,n$ that we call *semantic transitions*, or simply transitions when there is no ambiguity with their syntactic counterparts.

In the rest of the section we give a detailed description of the denotational semantic domain. First we define a general notion of a model of π-calculus, that we call π-model. We will find out that Proc is not a suitable π-model (a shortcoming also recognized by Cattani et al. [1997] for their categorical universe) and we will move to a new domain PiMod which soundly models all the features of π-calculus.

5.1 Algebras and Models

The syntax of the π-calculus presented in Section 4 can be characterized as the term algebra for a particular second-order algebraic theory in the sense of Fiore and Mahmoud [2010]. Second-order algebraic theories come with their notion of algebras and models, which we now introduce for the specific case of the π-calculus.

A *π-algebra* is a type family $X : \mathbb{N} \to \mathsf{Set}$ equipped with operations corresponding to the generators of the π-calculus syntax. That is, the following record type is inhabited for all $n : \mathbb{N}$:

```
record isPi-alg (X : ℕ → Set) (n : ℕ) : Set where
  field
    endX  : X n
    actX  : ∀ {m} → Act n m → X m → X n
    sumX  : X n → X n → X n
    parX  : X n → X n → X n
    νX    : X (suc n) → X n
    guardX : Name n → Name n → X n → X n
    !X    : X n → X n
```

The notions of structural congruence and operational semantics lift to π-algebras X acting on renamings, that is fitted with a function $\mathit{map}_X : \forall\{m\,n\} \to (\mathsf{Name}\,m \to \mathsf{Name}\,n) \to X\,m \to X\,n$. Given such a π-algebra X, we say that a type family $\mathsf{StructCong}_X : \forall\{n\} \to X\,n \to X\,n \to \mathsf{Set}$ is a *structural congruence on X* if it is closed under the rules of the syntactic structural congruence $\approx$. Similarly, we say that a type family $\mathsf{OpSem}_X : \forall\{n\,m\} \to X\,n \to \mathsf{Label}\,n\,m \to X\,m \to \mathsf{Set}$ is an *operational semantics on X* if it is closed under the rules of the syntactic operational semantics in Figure 2.

The type family Pi is a π-algebra, called the term (or initial) π-algebra. This means that, given another π-algebra X, we can define a function $[\![\ -\ ]\!]_X : \forall\{n\} \to \mathsf{Pi}\,n \to X\,n$ by recursion on the input process. Notice though that a π-algebra X that has a structural congruence $\mathsf{StructCong}_X$ and an operational semantics OpSem_X does not generally model all the rules of Figure 2, in particular the rule INP for free input transition. This is because the semantic substitution operator map_X of a π-algebra X is in no way connected to the syntactic substitution operator mapPi. This shortcoming can be rectified by refining the notion of π-algebra to the one of π-model.

A *π-model* is a π-algebra X acting on renaming via map_X, that has a structural congruence $\mathsf{StructCong}_X$ and an operational semantics OpSem_X on it, such that map_X is functorial and moreover $[\![\,\mathsf{mapPi}\,f\,P\,]\!]_X \equiv \mathit{map}_X\,f\,[\![\,P\,]\!]_X$. Following Fiore and Mahmoud [2010], we can prove that each π-model X respects structural congruence and operational semantics, i.e. $P \approx Q$ implies $\mathsf{StructCong}_X\,[\![\,P\,]\!]_X\ [\![\,Q\,]\!]_X$ and $P\,[\,a\,]\!\mapsto Q$ implies $\mathsf{OpSem}_X\,[\![\,P\,]\!]_X\,a\,[\![\,Q\,]\!]_X$. The proofs proceed by induction on the derivations of $P \approx Q$ and $P\,[\,a\,]\!\mapsto Q$, respectively.

Notice that instances of π-models are objects in the category $\mathsf{Set}^{\mathsf{Fin}}$ of covariant presheaves over the category Fin of (non-necessarily injective) renamings. This is because in the rule INP of Figure 2 the fresh name in P needs to be substituted for the value v, and this is a non-injective renaming. Therefore we cannot hope to obtain a sound model of π-calculus in $\mathsf{Set}^{\mathsf{Inj}}$.

Nevertheless, the category $\mathsf{Set}^{\mathsf{Inj}}$ still plays a role in our development. In fact, the construction of the π-algebra structure on the π-model PiMod of Section 5.3 takes place in two stages. First we show how to interpret the syntax of π-calculus (except prefixing) in Proc, which, as discussed at the beginning of Section 5, is an element of $\mathsf{Set}^{\mathsf{Inj}}$. Then we show how to lift these construction to PiMod, which is an object of $\mathsf{Set}^{\mathsf{Fin}}$, and how to soundly interpret the prefixing operation, the structural congruence and the operational semantics.

5.2 Operations on Semantic Processes

We now define some operations on the processes in Proc by guarded recursion. More precisely, given a proof $ih : \triangleright\,(\forall n \to \mathsf{isPi\text{-}alg\text{-}no\text{-}act}\ \mathsf{Proc}\,n)$, where isPi-alg-no-act is the record type family isPi-alg without projection actX, we construct an element of $\forall n \to \mathsf{isPi\text{-}alg\text{-}no\text{-}act}\ \mathsf{Proc}\,n$. At the end, we take the fixpoint of the obtained function. In other

words, we are defining several functions by mutual guarded recursion. Let $n : \mathbb{N}$. Given a tick α and a natural number m, we write parX α and νX α (omitting m) for the parallel composition and restriction projections of $ih\ \alpha\ m$.

Empty Process. Modelled using the empty set of the countable powerset:

```
endProc : Proc n
endProc = Fold ø
```

Binary Sums. Modelled using the binary union of the countable powerset:

```
sumProc : Proc n → Proc n → Proc n
sumProc P Q = Fold (Unfold P ∪ Unfold Q)
```

Matching. We check if the two given names are equal. If they are, we return the input process, otherwise we return Fold applied to the empty set.

```
guardProc : (x y : Name n) → Proc n → Proc n
guardProc x y P with x ≟ y
... | yes p = P
... | no p = Fold ø
```

Parallel Composition. The parallel composition of two processes can step in three different ways: either the left process steps, the right process steps, or they synchronize:

```
parProc : Proc n → Proc n → Proc n
parProc P Q = Fold (stepL P Q ∪ stepR P Q ∪ synch P Q)
```

The function stepL builds the semantic transitions in which the left process P steps. For any future process $P' : \triangleright$ Proc m resulting from a transition initiating from P, we take the parallel composition of P' with Q. The resulting process exists in the next time step, as evidenced by the presence in scope of the tick α. The parallel composition operation parX α is obtained from the guarded recursive hypothesis. The names in the process Q need to be renamed via fst f, which is the function underlying f : InjRen $n\ m$.

```
stepL : Proc n → Proc n → F' Proc n
stepL P Q =
  mapF (λ f P' α → parX α (P' α) (mapProc f Q))
       (Unfold P)
```

The function stepR is defined analogously, with the roles of P and Q being swapped. Synchronization is specified in two steps. First we define an auxiliary function synch'. Its arguments $aP' = (a\ ,\ P')$ and $bQ' = (b\ ,\ Q')$ are semantic transitions initiating from P and Q respectively. If a and b can communicate, that is if $a =$ out $ch\ v$ and $b =$ inp $ch\ v$, then the parallel composition of P and Q make a silent transition to the parallel composition of P' and Q'. Communication can also happen if $a =$ bout ch and $b =$ binp ch, in which case P and Q make a silent transition to the restriction of the parallel composition of P' and Q'. No synchronization happens in all other cases for a and b.

```
synch' : (aP' bQ' : Step (λ m → ▹ Proc m) n) → F' Proc n
synch' (out ch v , P') (inp ch' z , Q') with ch ≟ ch' | v ≟ z
... | yes p | yes q = η (τ , λ α → parX α (P' α) (Q' α))
... | _ | _ = ø
synch' (bout ch , P') (binp ch' , Q') with ch ≟ ch'
... | yes p = η (τ , λ α → νX α (parX α (P' α) (Q' α)))
... | _ = ø
synch' _ _ = ø
```

The set synch $P\ Q$ is obtained by taking the union of synch' applied to all possible semantic transitions aP' and bQ' initiating from P and Q.

```
synch : Proc n → Proc n → F' Proc n
synch P Q =
  bindP∞ (Unfold P) (λ aP' → bindP∞ (Unfold Q)
    (λ bQ' → synch' aP' bQ' ∪ synch' bQ' aP'))
```

Restriction. Restriction is also defined in two steps. First we define an auxiliary function stepν. Its argument $aP' = (a\ ,\ P')$ is a semantic transition that we think initiating from a certain semantic process P. The set stepν aP' is a subsingleton set which contains an element precisely when aP' generates a transition initiating from the restriction of P. This can happen in two cases. If a is a label not depending on the fresh name of Name n, then the transition aP' lifts to a transition between P' and P, as in the rule res of Figure 2. In case a is a free output label, if the output channel is not fresh and the outputted name is fresh, then the transition aP' implies a transition from the restriction of P' to P, as in the rule opn of Figure 2. The cases where the action is of the form binp and inp are omitted, since they are defined analogously to the cases of bout and the first case of out, respectively.

```
stepν : (aP' : Step (λ m → ▹ Proc m) (suc n)) → F' Proc n
stepν (τ , P') = η (τ , (λ α → νX α (P' α)))
stepν (out ch v , P') with ch ≟ fresh | v ≟ fresh
... | no p | no q =
  η (out (down ch p) (down v q) , λ α → νX α (P' α))
... | no p | yes q = η (bout (down ch p) , P')
... | yes p | _ = ø
stepν (bout ch , P') with ch ≟ fresh
... | no p =
  η (bout (down ch p) , λ α → νX α (mapProc swapI (P' α)))
... | yes p = ø
```

The semantic restriction νProc P is obtained by taking the union of stepν applied to all possible transitions aP' initiating from P.

```
νProc : Proc (suc n) → Proc n
νProc P = Fold (bindP∞ (Unfold P) stepν)
```

Replication. The replication of a process P can step in the following ways: either P steps or P synchronizes with itself, and in both cases the resulting process continues in parallel with the replication of P. This suggests the specification of the semantic replication operator via a nested guarded fixpoint. In the definition below, the function stepL' is a variant of stepL with the second argument taken from $\triangleright$ Proc n instead of Proc n, such that stepL $P\,Q$ = stepL' P (next Q).

```
!Proc : Proc n → Proc n
!Proc P =
  Fold (fix (λ !P → stepL' (Fold (Unfold P ∪ synch P P))
                           (λ α → Fold (!P α))))
```

Notice that this is equivalent to the definition of unguarded replication in the domain theoretic model of Stark [1996], which, when translated to our setting, can be paraphrased as follows: the replication of a process P steps if parProc P P steps and the resulting process continues in parallel with the replication of P.

Another possibility would have been to use the semantic definition of replication of Cattani et al. [1997], more explicitly described by Cattani in his PhD thesis [Cattani 1999, Section 7.4.3]. In our setting, this amounts to introduce a domain structure on Proc n and define replication as the least upper bound of an ω-chain built from iterating parallel composition. Naively, one would take the order on Proc n to be subset inclusion and the least upper bound operation to be countable union. Then, given a semantic process P, one can define an ω-chain whose n-th position consists of n parallel copies of the process P and take !Proc P as the least upper bound of this ω-chain. This turns out to be incorrent. In fact, the structural congruence $!\,P \approx P \,||\, !\,P$ could alternatively be introduced in the operational semantics as the rule:

$$\frac{P \,||\, !\,P\,[\,a\,]\!\mapsto Q}{!\,P\,[\,a\,]\!\mapsto Q}$$

But our naive definition of semantic replication is only able to model the following rule:

$$\frac{n : \mathbb{N} \qquad \overbrace{P \,||\, \ldots \,||\, P}^{n}\,[\,a\,]\!\mapsto Q}{!\,P\,[\,a\,]\!\mapsto Q}$$

This rule does not properly capture the behaviour of replication: the process $P \,||\, \ldots \,||\, P$, consisting of n parallel copies of P, is unable to create an $(n+1)$-th copy of P. This shows that the naive domain structure we put on Proc n is wrong, and the correct domain structure is more involved. Nevertheless, defining replication following Cattani's idea would force us to explicitly work with the domain structure of Proc n, which is unnatural in a type theory with guarded recursion where solutions to recursive equations are generally constructed using the fixpoint combinator fix.

Interpreting the Structural Congruence. The rules generating the syntactic congruence $\approx$ do not involve the prefixing operation. These rules also hold in Proc n up to propositional equality. That is Proc has a structural congruence $\mathsf{StructCong}_{\mathsf{Proc}}\,P\,Q := P \equiv Q$, i.e. it satisfies the laws up to path equality, even without Proc being a π-algebra. Notice that mapPi appears in the rule $\nu||$, but this can be modelled using mapProc because the renaming is injective.

The proof that $\mathsf{StructCong}_{\mathsf{Proc}}$ is a structural congruence on Proc proceeds by guarded recursion, which is to say that the rules are proved by mutual guarded recursion. This mutual definition is necessary because the proof of some laws require the validity of other laws under a tick. For example, in order to show that the semantic parallel composition is associative, we require both the associativity *and* the commutativity of the parallel composition to hold at the next time step. Both proofs are given to us by the guarded recursive hypothesis.

Interpreting the Operational Semantics. It is possible to interpret in Proc the rules of the operational semantics that do not involve prefixing, by defining $\mathsf{OpSem}_{\mathsf{Proc}}\,P\,a\,Q := [\,(a\,,\,\mathsf{next}\,Q) \in \mathsf{Unfold}\,P\,]$. The semantic version of these rules is proved by mutual guarded recursion. These proofs heavily rely on the equality ∈mapP∞-eq introduced in the end of Section 3, which characterizes the membership into subsets of the form mapP∞ f x.

5.3 A π-model

As discussed in the beginning of Section 5.2, Proc is not a π-algebra, and in particular not a π-model. To obtain a sound interpretation of π-calculus we need to move to a different denotational domain:

```
record PiMod (n : ℕ) : Set where
  field
    elem : ∀ {m} → (Name n → Name m) → Proc m
    elem-nat : ∀ {m l}
      → (f : InjRen m l) (ρ : Name n → Name m)
      → mapProc f (elem ρ) ≡ elem (fst f ∘ ρ)
```

Elements of PiMod n are families of semantic processes indexed by renamings from Name n, satisfying a naturality condition. The condition makes sure that for renamings that differ only by permutations or introduction of new names, the family elem will produce processes that only differ in the same way. The consequence is that the family only really gives us something new when the given renaming conflates existing names, which is what Proc n alone failed to represent. Formally, PiMod is obtained as the right Kan extension of Proc along the inclusion i : Inj → Fin.

PiMod is an object of $\mathsf{Set}^{\mathsf{Fin}}$. It has a functorial action on renamings called mapPiMod, defined by copatterns [2] as follows:

```
mapPiMod : {n m : ℕ} → (Name m → Name n)
  → PiMod m → PiMod n
elem (mapPiMod ρ P) ρ'       = elem P (ρ' ∘ ρ)
elem-nat (mapPiMod ρ P) f ρ' = elem-nat P f (ρ' ∘ ρ)
```

Moving from Proc to PiMod gives us extra flexibility that we use to implement the prefixing operation and build a π-algebra structure.

Lifting the Operations on Proc. Using the semantic operations on Proc introduced in Section 5.2, it is possible to define a π-algebra structure on PiMod. All the operations can be lifted pointwise. E.g., the parallel composition on PiMod n is defined by copatterns as:

```
parPiMod : PiMod n → PiMod n → PiMod n
elem (parPiMod P Q) ρ = parProc (elem P ρ) (elem Q ρ)
elem-nat (parPiMod P Q) f ρ =
  parRen f
    • cong₂ parProc (elem-nat P f ρ) (elem-nat Q f ρ)
```

where • constructs the sequential composition of two paths and parRen proves that parProc commutes with injective renamings.

```
parRen : ∀ {n m} {P Q : Proc n} (f : InjRen n m)
  → mapProc f (parProc P Q)
      ≡ parProc (mapProc f P) (mapProc f Q)
```

The other operations of Section 5.2 also commute with injective renamings and therefore lift in a similar way to PiMod.

Prefixing. Prefixing is defined by pattern-matching on the action. The silent action and output cases are straightforward. The interpretation of prefixing with an input action is the union of two sets, the first corresponding to the rule BINP and the second to the rule INP of the operational semantics. The instantiation of the bound input name in the action with the passed value v is performed using the function snoc introduced in Section 4.1. The proof of the naturality conditions is omitted.

```
actPiMod : ∀ {m} → Act n m → PiMod m → PiMod n
elem (actPiMod τ P) ρ =
  Fold (η (τ , next (elem P ρ)))
elem (actPiMod (out ch v) P) ρ =
  Fold (η (out (ρ ch) (ρ v) , next (elem P ρ)))
elem (actPiMod (inp ch) P) ρ =
  Fold (η (binp (ρ ch) , next (elem P (lift ρ)))
    ∪ mapP∞ (λ v → inp (ρ ch) v , next (elem P (snoc ρ v)))
            enum)
```

[2] Here in particular we have record projection copatterns, as we are defining an element of a record type by specifying how it should behave when any of the two projections are applied to it.

Interpreting Congruence and Operational Semantics. The π-algebra PiMod implements a structural congruence defined as $\mathsf{StructCong}_{\mathsf{PiMod}}\ P\ Q := P \equiv Q$. The proof that this is indeed a structural congruence on PiMod follows straightforwardly from the structural congruence on Proc discussed in the end of Section 5.2.

PiMod also implements an operational semantics, defined as the following transition relation:

```
_[_]↦PiMod_ : ∀ {n m}
  → PiMod n → Label n m → PiMod m → Set
P [ a ]↦PiMod Q = ∀ {l} (ρ : Name _ → Name l)
  → [ (mapLabel ρ a , next (elem Q (labelLift a ρ)))
      ∈ Unfold (elem P ρ) ]
```

This is an extension of the transition relation $\mathsf{OpSem}_{\mathsf{Proc}}$ specified in the end of Section 5.2, in which the names in the label a are renamed according to the environment ρ. This renaming is performed via the functions mapLabel and labelLift introduced in the end of Section 4.3.

This transition relation is indeed an operational semantics on PiMod. The rules not involving prefixing follow straightforwardly from their counterpart in $\mathsf{OpSem}_{\mathsf{Proc}}$. We refer the interested reader to our Agda formalization for the interpretation of the remaining rules involving prefixing.

6 Syntactic Early Congruence and Full Abstraction

The difficulty of handling input actions is a well-known issue in the meta-theory of the π-calculus, which leads various notions of bisimilarity to not be congruence relations [Sangiorgi and Walker 2001]. In this section we introduce the notion of *guarded* early bisimilarity on syntactic processes and show how to refine it to obtain an early congruence relation. The attribute "guarded" indicates that this notion of bisimilarity is defined by guarded recursion. Afterwards we go on to prove that the denotational model PiMod of Section 5.3 is fully abstract wrt. the early congruence relation. Full abstraction says that two syntactic processes are related by early congruence if and only if their semantic interpretations are related by a semantic version of bisimilarity, usually expressed via open maps [Cattani et al. 1997]. In TCTT, a notion of bisimilarity can be defined for all guarded recursive types and proved equivalent to path equality [Møgelberg and Veltri 2019]. This implies that, in our setting, full abstraction corresponds to the equivalence between syntactic congruence of processes and equality of their interpretations.

We parameterize the definition of early bisimilarity by an arbitrary transition relation T on syntactic processes

$$T : \forall\{n\ m\} \to \mathsf{Pi}\ n \to \mathsf{Label}\ n\ m \to \mathsf{Pi}\ m \to \mathsf{Set}$$

not necessarily the operational semantics of Figure 2. We define the relation BisimT T to hold on processes P and Q

whenever the following statement is satisfied: if P makes an a-labelled transition to P' in the transition system T, then there merely exists Q' such that Q makes an a-labelled transition to Q' in the system T and later P' and Q' are related by BisimT T, and dually with P and Q swapping roles.

```
record BisimT
    (T : ∀ {n m} → Pi n → Label n m → Pi m → Set)
    (n : ℕ) (P Q : Pi n) : Set where
  inductive
  constructor FoldB
  field
    UnfoldB :
      (∀ {m} (a : Label n m) (P' : Pi m) → T P a P'
        → ∃[ Q' ∶ Pi m ] (T Q a Q' × ▷ BisimT T m P' Q'))
      ×
      (∀ {m} (a : Label n m) (Q' : Pi m) → T Q a Q'
        → ∃[ P' ∶ Pi m ] (T P a P' × ▷ BisimT T m Q' P'))
```

We write Bisim for the bisimilarity relation when T is the transition system of Figure 2. This notion of bisimilarity is not a congruence since it does not preserve input actions. In fact, following the example in [Sangiorgi and Walker 2001, page 96], it is possible to describe two early bisimilar processes which are not early congruent. Early bisimilarity can be refined to early congruence by closing it wrt. name substitution:

```
EarlyCong : (n : ℕ) → Pi n → Pi n → Set
EarlyCong n P Q = ∀ {m} (ρ : Name n → Name m)
  → Bisim m (mapPi ρ P) (mapPi ρ Q)
```

We say that the π-model PiMod is *fully abstract* if, for all syntactic processes P and Q, the type EarlyCong $n\ P\ Q$ is logically equivalent to the type $[\![P]\!]_{\text{PiMod}} \equiv [\![Q]\!]_{\text{PiMod}}$. The proof of full abstraction is achieved through several auxiliary results.

- In the beginning of Section 5 we introduced a transition relation ≈∈≈step. We also presented an equality opsem-eq, fully characterizing the operational semantics of Figure 2 in terms of ≈∈≈step. Let BisimS be early bisimilarity on the transition relation ≈∈≈step. The equality opsem-eq implies that Bisim $n\ P\ Q$ and BisimS $n\ P\ Q$ are logically equivalent for all processes P, Q : Pi n.
- The relation BisimS is analogous to the semantic notion of bisimilarity considered by Møgelberg and Veltri [2019] instantiated to the case of the guarded recursive type Proc n. The extensionality principle associated to Proc n then states that BisimS $n\ P\ Q$ is logically equivalent to $\text{eval}_{\text{step}}\ P \equiv \text{eval}_{\text{step}}\ Q$, for all processes P, Q : Pi n, where $\text{eval}_{\text{step}}$ is the unique coalgebra morphism into Proc n associated to the coalgebra step.
- Finally, one can prove that, given a syntactic process P : Pi n and a renaming ρ : Name n → Name m, the semantic process $\text{eval}_{\text{step}}$ (mapPi $\rho\ P$) is path equal to elem $[\![P]\!]_{\text{PiMod}}\ \rho$. In other words, first applying the syntactic substitution ρ to P and then interpreting the resulting process in Proc m (using the evaluation map given by the finality of Proc as a coalgebra for F') is the same as interpreting P into the π-model PiMod (using the evaluation map given by the initiality of Pi as a π-algebra), then taking the semantic process at environment ρ.

Putting everything together, we obtain the following sequence of equivalences, proving the denotational semantics fully abstract wrt. early congruence:

$$
\begin{aligned}
&\text{EarlyCong}\ n\ P\ Q \\
&= \forall\ \rho \to \text{Bisim}\ m\ (\text{mapPi}\ \rho\ P)\ (\text{mapPi}\ \rho\ Q) \\
&\simeq \forall\ \rho \to \text{BisimS}\ m\ (\text{mapPi}\ \rho\ P)\ (\text{mapPi}\ \rho\ Q) \\
&\simeq \forall\ \rho \to \text{eval}_{\text{step}}\ (\text{mapPi}\ \rho\ P) \equiv \text{eval}_{\text{step}}\ (\text{mapPi}\ \rho\ Q) \\
&\simeq \forall\ \rho \to \text{elem}\ [\![P]\!]_{\text{PiMod}}\ \rho \equiv \text{elem}\ [\![Q]\!]_{\text{PiMod}}\ \rho \\
&\simeq [\![P]\!]_{\text{PiMod}} \equiv [\![Q]\!]_{\text{PiMod}}
\end{aligned}
$$

7 Related Works

Over the years there have been many formalizations of the π-calculus in proof assistants such as Agda [Perera and Cheney 2018], Coq [Hirschkoff 1997; Honsell et al. 2001], and Nominal Isabelle [Bengtson and Parrow 2009]. These works, while employing diverse techniques for the handling of names, do not construct a denotational semantics of the π-calculus, which is our focus.

As mentioned in the introduction, the denotational semantics of the π-calculus have been formulated as constructions internal to categories of presheaves over domains [Fiore et al. 1996; Stark 1996] or profunctors [Cattani et al. 1997]. Compared to those, other than relying on guarded recursion, our presentation explores more explicitly the relationship of the model with the coalgebraic view of transitions systems, which we use in our proof of full abstraction. Moreover, it is interesting to note how the particular split between free inp and bound binp input labels of Milner et al. [1993] is mirrored by the specific representation of maps from names used in the cited works to represent input transitions.

8 Conclusions and Future Work

In this paper we presented a fully abstract denotational semantics of the early π-calculus, mechanized in Guarded Cubical Agda. This provides a further example of the usefulness of guarded recursion for the development of semantics of programming languages with hard to tackle features such as concurrency. Verifying our development in a proof assistant helped us keep track of how the various parts of

the semantics fit together, and gave us confidence that we were not abusing the induction hypothesis provided by the guarded fixpoint combinator. We are hopeful that the possibility to mechanize such proofs will make not only Ticked Cubical Type Theory but also guarded recursion in general more accessible.

The interplay between guarded recursion and Higher Inductive Types is nicely exhibited by the type of semantic processes: Proc is a guarded recursive type family defined using the countable powerset datatype. Notice that in our development we never used the countable union operation of P∞, and a sound denotational model of early π-calculus could indeed be given using the finite powerset of Frumin et al. [2018] instead. One may naïvely think that countable union is essential for modelling the replication operation !, but a combination of binary union and guarded recursion is indeed sufficient, as we showed in Section 5.2. Our denotational domain could be used for modelling a variant of early π-calculus with countably-infinite sums, for which the notion of early congruence coincide with barbed congruence [Sangiorgi and Walker 2001, Theorem 2.4.36].

Recently, Danielsson [2018] investigated bisimilarity up-to techniques in Agda using sized types and discovered a correlation between these techniques and size-preserving functions. We are interested in studying up-to techniques in our setting and understanding if a similar characterization is possible using the later modality.

In the future we plan to integrate Guarded Cubical Agda into the next major release of the Agda proof assistant, and extend it to support clock quantification. Formally verifying the correctness and decidability of the typechecking algorithm are also left for future work.

Acknowledgments

Niccolò Veltri was supported by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001). Andrea Vezzosi was supported by a research grant (13156) from VILLUM FONDEN.

References

Jesper Bengtson and Joachim Parrow. 2009. Formalising the pi-calculus using nominal logic. *Logical Methods in Computer Science* Volume 5, Issue 2 (June 2009). https://doi.org/10.2168/LMCS-5(2:16)2009

Lars Birkedal, Ales Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2019. Guarded Cubical Type Theory. *J. Autom. Reasoning* 63, 2 (2019), 211–253. https://doi.org/10.1007/s10817-018-9471-7

Lars Birkedal, Rasmus Ejlers Mogelberg, Jan Schwinghammer, and Kristian Stovring. 2011. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science (LICS '11)*. IEEE Computer Society, Washington, DC, USA, 55–64. https://doi.org/10.1109/LICS.2011.16

Ales Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. In *Proc. of the 19th Int. Conf. on Foundations of Software Science and Computation Structures, FOSSACS 2016*. 20–35. https://doi.org/10.1007/978-3-662-49630-5_2

Gian Luca Cattani. 1999. *Presheaf Models for Concurrency*. Ph.D. Dissertation. University of Aarhus.

Gian Luca Cattani, Ian Stark, and Glynn Winskel. 1997. Presheaf Models for the pi-Calculus. In *Proc. of the 7th Int. Conf. on Category Theory and Computer Science, CTCS 1997*. 106–126. https://doi.org/10.1007/BFb0026984

James Chapman, Tarmo Uustalu, and Niccolò Veltri. 2019. Quotienting the delay monad by weak bisimilarity. *Math. Struct. in Comp. Sci.* 29, 1 (2019), 67–92. https://doi.org/10.1017/S0960129517000184

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In *Types for Proofs and Programs (TYPES 2015) (LIPIcs)*, Vol. 69. 5:1–5:34.

Nils Anders Danielsson. 2018. Up-to techniques using sized types. *PACMPL* 2, POPL (2018), 43:1–43:28. https://doi.org/10.1145/3158131

Marcelo P. Fiore and Ola Mahmoud. 2010. Second-Order Algebraic Theories - (Extended Abstract). In *Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings*. 368–380. https://doi.org/10.1007/978-3-642-15155-2_33

Marcelo P. Fiore, Eugenio Moggi, and Davide Sangiorgi. 1996. A Fully-Abstract Model for the pi-Calculus (Extended Abstract). In *Proc. of the 11th Ann. IEEE Symp. on Logic in Computer Science, LICS 1996*. 43–54. https://doi.org/10.1109/LICS.1996.561302

Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. 2018. Finite sets in homotopy type theory. In *Proc. of the 7th ACM SIGPLAN Int. Conf. on Certified Programs and Proofs, CPP 2018*. ACM, 201–214. https://doi.org/10.1145/3167085

Daniel Hirschkoff. 1997. A full formalisation of π-calculus theory in the calculus of constructions. In *Theorem Proving in Higher Order Logics*, Elsa L. Gunter and Amy Felty (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 153–169.

Furio Honsell, Marino Miculan, and Ivan Scagnetto. 2001. π-calculus in (Co)inductive-type theory. *Theoretical Computer Science* 253, 2 (2001), 239 – 285. https://doi.org/10.1016/S0304-3975(00)00095-5 ICEATCS'97.

Bart Jacobs. 2016. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge Tracts in Theoretical Computer Science, Vol. 59. Cambridge University Press. https://doi.org/10.1017/CBO9781316823187

Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Inf. Comput.* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

Robin Milner, Joachim Parrow, and David Walker. 1993. Modal Logics for Mobile Processes. *Theor. Comput. Sci.* 114, 1 (1993), 149–171. https://doi.org/10.1016/0304-3975(93)90156-N

Rasmus Ejlers Møgelberg and Marco Paviotti. 2019. Denotational semantics of recursive types in synthetic guarded domain theory. *Math. Struct. in Comp. Sci.* 29, 3 (2019), 465–510. https://doi.org/10.1017/S0960129518000087

Rasmus Ejlers Møgelberg and Niccolò Veltri. 2019. Bisimulation as path type for guarded recursive types. *PACMPL* 3, POPL (2019), 4:1–4:29. https://doi.org/10.1145/3290317

Hiroshi Nakano. 2000. A Modality for Recursion. In *Proc. of the 15th Ann. IEEE Symp. on Logic in Computer Science, LICS 2000*. 255–266. https://doi.org/10.1109/LICS.2000.855774

Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. 2015. A Model of PCF in Guarded Type Theory. *Electr. Notes Theor. Comput. Sci.* 319 (2015), 333–349. https://doi.org/10.1016/j.entcs.2015.12.020

Roly Perera and James Cheney. 2018. Proof-relevant π-calculus: a constructive account of concurrency and causality. *Mathematical Structures in Computer Science* 28, 9 (2018), 1541–1577. https://doi.org/10.1017/S096012951700010X

Davide Sangiorgi and David Walker. 2001. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press.

Ian Stark. 1996. A Fully Abstract Domain Model for the pi-Calculus. In *Proc of the 11th Ann. IEEE Symp. on Logic in Computer Science, LICS 1996*. 36–42. https://doi.org/10.1109/LICS.1996.561301

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study.

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *PACMPL* 3, ICFP, Article 87 (2019), 29 pages. https://doi.org/10.1145/3341691