Universiteit Leiden
The Netherlands

Opleiding Informatica

Coalgebras of Session Types:

Defining a syntax independent framework

Alex Keizer

Supervisors:
Henning Bassold & Jorge A. Pérez

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

15/07/2020

Abstract

Session type systems provide a robust way of checking whether a communication channel is used according to some protocol. Most work on session types, however, tends to be heavily based on syntax, and the syntax used differs between authors. In this thesis, we aim to provide a more structured system, based in the theory of labelled transition systems and coalgebras. We also define bisimulation and simulation on these coalgebras such that they correspond to type equivalence and subtyping, respectively. Furthermore, we define a coalgebra on the syntax of two existing type systems, showing how to relate syntax-based systems to our coalgebraic system, and how to instantiate our type rules on syntactical expressions. Finally, we give a type checking algorithm and show that it will always terminate for a certain class of coalgebras, which includes the coalgebra on expressions.

Contents

1 Introduction **1**
- 1.1 The problem . . . 1
- 1.2 Thesis overview . . . 2

2 Session Types **2**
- 2.1 Subtyping . . . 3
- 2.2 Syntax . . . 4
- 2.3 Pi-Calculus . . . 6

3 Category Theory **8**
- 3.1 Set Operations . . . 8
- 3.2 Relations and Orderings . . . 9
- 3.3 Functors . . . 9
- 3.4 Coalgebra . . . 10
- 3.5 Bisimulation . . . 10

4 Session Coalgebra **11**
- 4.1 F-Coalgebras . . . 13
- 4.2 Coalgebra of Types . . . 15

5 Relations on Types **17**
- 5.1 Bisimulation . . . 18
- 5.2 Duality . . . 19
- 5.3 Parallelizability . . . 20
- 5.4 Simulation and Subtyping . . . 22
- 5.5 Composites . . . 24
- 5.6 Decidability of Coalgebraic Type Relations . . . 26

6 Type Rules **28**

7 Algorithmic Type Checking **34**
7.1 Normalization and Unpacking 38
7.2 Algorithmic Soundness 40

8 Conclusions and Further Research **43**

References **44**

1 Introduction

Type systems are commonplace in modern programming languages, coming in many forms and shapes. The type systems used by most popular languages today, however, can only provide compile-time guarantees about properties that hold at any point during the execution of a program.

A static type system can, for example, guarantee that a function is always called with an integer as argument, but not (in an ergonomic, developer-friendly way) that a certain function is only called after another function.

Behavioural typing emerged as a way to describe such dynamic behaviour of a program, which eventually led to, among others, session typing. In the interest of simplicity, session types are a type system for the pi-calculus (see Section 2.3). The pi-calculus is a theoretical language designed to model asynchronous behaviour with minimal syntax, similarly to the lambda calculus for sequential programs. It features processes running in parallel, which communicate over channels. A channel is some object, with two channel ends, such that a process that holds one end can either write a value, or read a value that was written to the other end. Unix pipes and sockets are straightforward examples, but things like shared memory could also be modelled with channels.

To return to our comparison of type systems: a channel that can only hold integers (to which only integers can be written, and thus only ever read) can still be modelled in a language with a static type system (like `C++`). Once we add constraints like an integer written to the channel must, at some later point, be followed by a Boolean value, it becomes very difficult to guarantee that the channels are used correctly. Session types let us accurately describe this dynamically evolving relation between what is sent over the channel. In this work we will be focused on binary session types, describing communications between exactly two participants.

1.1 The problem

The lambda calculus has a canonical type system which provides the basis of many (sequential) functional programming languages, for example, Haskell. We would like to have a generic session type system as well, so implementations can reuse the type rules and soundness proof of the generic system.

Currently, most work on session types defines a syntax for their types, and then proceeds to do their definitions and proofs directly using that syntax. To make matters worse, the syntax tends to differ between works. It is possible, and indeed regularly done, to show how one syntax translates into another, but we would like to work with a more structured representation.

This thesis proposes a coalgebraic structure. We will later see what the definition of a coalgebra is, but one can think of the relation between (syntactical) type expressions and coalgebras as very similar to the relation between a regular expression and the finite automaton for the same language. Expressions tend to be more human read- and writable, but the structured nature of automata allows for easier reasoning and proofs. In the case of session types, there is the added benefit that a coalgebra can be defined on all kinds of syntax, enabling more formal reuse of our type system and proofs.

1.2 Thesis overview

This chapter contains the introduction; Section 2 introduces session types and the pi-calculus; Section 3 gives some definitions and a brief overview of functors and coalgebras; Section 4 defines the functor whose coalgebras represent session types; Section 5 defines bisimulation, simulation, duality and parallelizability of these coalgebras; Section 6 gives the type rules of our system, followed by a type checking algorithm. Section 8 concludes.

This bachelor thesis was written under the supervision of Henning Bassold and Jorge Pérez and is presented at the Leiden Institute of Advanced Computer Science.

2 Session Types

A session type, like ?`int`.!`bool`.*end*, is a sequence of actions allowed on a channel end. Recall that a channel consists of two connected ends; whatever is written to one end can be read from the other. The simplest type is *end*, the completed protocol. It describes a channel on which no actions are allowed, usually indicating the end of a session. We indicate a session type can receive a value with ?`int`; sending a value looks like !`int`. The type of a channel end is not constant, we need to describe what the channel does after sending or receiving that `int` value. This *continuation type* is written after the action. A channel end of type ?`int`.!`bool`.*end* can receive a single integer (after which the type is !`bool`.*end*) and send single a Boolean value, before closing the channel (because the type became *end*).

Protocols tend to not be strictly linear sequences; there could be points where messages of different types are allowed, and the behaviour after each message type may differ. Two processes following such a protocol need to agree on which variant they will follow, this choice is communicated by sending ($\oplus$) or receiving (&) a label. A choice that is made by a process internally, leading to a label sent, is called *internal choice*, while a choice that is determined by receiving such a label is *external*.

Take, for example, a service for doing some mathematical operations "in the cloud":

$$\& \begin{cases} \mathit{mul}: & ?[\texttt{int}, \texttt{int}].!\texttt{int}.\mathit{end} \\ \mathit{neg}: & ?\texttt{bool}.!\texttt{bool}.\mathit{end} \end{cases}$$

An external choice is offered between multiplying two integers or negating a Boolean value. A channel of this type first has to receive the *mul* or *neg* label and then continues as the corresponding continuation type (?[`int`,`int`].!`int`.*end* for *mul* or ?`bool`.!`bool`.*end* for *neg*).

If a client of our service wanted to do multiple operations, they would need to start a new session for each operation. Ideally, we would like to reuse a single channel. The μ operator binds a variable to itself, allowing recursive types: in the type $\mu X.T$, every occurrence of X is bound to $\mu X.T$. Removing a μ operator, by replacing each occurrence of the bound variable with its value, is called unfolding. The type $\mu X.?\texttt{int}.X$ unfolds into $?\texttt{int}.\mu X.?\texttt{int}.X$ and like this, repeated behaviour can be modelled.

Let us augment our example type with recursion:

$$\mathit{server} = \mu X.\, \& \begin{cases} \mathit{mul}: & ?[\texttt{int}, \texttt{int}].!\texttt{int}.X \\ \mathit{neg}: & ?\texttt{bool}.!\texttt{bool}.X \\ \mathit{quit}: & \mathit{end} \end{cases}$$

Instead of closing the channel after an operation is done, we continue as the bound type X. That is, after sending the expected integer or Boolean reply, the type expects to be sent another *mul*, *neg* or *quit* label. The third, *quit*, was added so the server can be informed when the client is done and wants to close the channel.

Session types describe only one end of a channel, but two ends forming a channel together must have related types. Otherwise, a process expecting to read an `int` could have been sent an entirely different kind of value. This relation is called duality, and it intuitively describes opposite types.

- If a value of a certain type is written, the dual type needs to read a value of the same type
- If an external choice is offered between a set of labels, the dual type needs to make an internal choice between the same labels
- The corresponding continuation types must always be dual

A formal definition of duality is further explored in Section 5.

The dual of a type can often[1] be made by exchanging input with output, reads become writes and external choices become internal. The dual of the mathematical service example is

$$client = \mu X.\ \oplus \begin{cases} mul: & ![\texttt{int}, \texttt{int}].?\texttt{int}.X \\ neg: & !\texttt{bool}.?\texttt{bool}.X \\ quit: & end \end{cases}$$

To reiterate: a channel is some object we can write to and read from. A channel has two channel ends, each of which has its own session type. These two types need to be dual to each other for communication over the channel to occur properly.

This introduction should be enough to follow the rest of this work, but the interested reader is invited to look at [H+16] for a more in-depth overview off the literature on session types.

2.1 Subtyping

Subtyping is a relation between types that describes when one type (say, `int`) is can be used wherever another (such as `real`) was expected. If `int` is a subtype of `real`, then all `int` values can be seen as valid `real`s, so any function expecting a `real` argument can safely be called with an `int` value. Subclassing, as featured in object oriented programming languages, is a well-known form of subtyping. Subtyping for session types was introduced by Gay and Hole in [GH05].

Intuitively, if a process P is well-formed with a channel of type $T = !\texttt{int}.end$, it must only ever send an `int` value over the channel. Every such value is also valid for a channel of type $U = !\texttt{real}.end$. After substituting U for T as type of the channel in the process P, it remains well-formed. Therefore, type U is a subtype of T. Notice that if output is changed to input, the order of the two types in the relation is switched: although U is a subtype of T, the dual $T' = ?\texttt{int}.end$ is a subtype of $U' = ?\texttt{real}.end$.

Imagine a client of our mathematical service that is only interested in multiplying natural numbers (yet is still capable of handling integer responses), its type would be:

$$client2 = \mu X.\ \oplus \begin{cases} mul: & ![\texttt{nat}, \texttt{nat}].?\texttt{int}.X \\ quit: & end \end{cases}$$

[1] This only works for tail-recursive types, types that only have recursion variable as continuations, not for types like $\mu X.?X.T$

$$
\begin{array}{rcl}
S & ::= & ?[\widetilde{T}].S \\
& | & ![\widetilde{T}].S \\
& | & \&\{\widetilde{l} : \widetilde{S}\} \\
& | & \oplus\{\widetilde{l} : \widetilde{S}\} \\
& | & end \\
& | & X \\
& | & \mu X.S \\
\\
T_{GH} & ::= & d \\
& | & S \\
& | & \frown[\widetilde{T}]
\end{array}
$$

Figure 1: Gay and Hole's grammar

$$
\begin{array}{rcl}
q & ::= & lin|un \\
\\
p & ::= & ?T.T \\
& | & !T.T \\
& | & \&\{\widetilde{l} : \widetilde{T}\} \\
& | & \oplus\{\widetilde{l} : \widetilde{T}\} \\
\\
T_V & ::= & q\,p.T \\
& | & end \\
& | & X \\
& | & \mu X.T \\
& | & d
\end{array}
$$

Figure 2: Vasconcelos' grammar

Under the assumption that `nat` is a subtype of `int`, the *mul* branch in *client*2 is a subtype of the *mul* branch in *client*. Additionally, the *neg* option is left out of the choice, but both *mul* and *quit* were part of the original *client* type. A well-formed process will never send *neg* over a channel of type *client*2, but that same process would still be well-formed if the extra option was allowed on its channel. Here too, the direction of input or output matters. The subtype of an *internal* choice may limit its options and the subtype of an *external* choice can offer extra options.

This new *client*2 type is a subtype of the original *client* type, which was dual to the *server* type. This means that a process using *client*2 and a process using the *server* type can communicate with each other. We did not need to change the server to accommodate a specialized client.

Two types T and U are called *type-equivalent* if they are both subtypes of each other (i.e., T is a subtype of U and U is a subtype of T). Because either type can be used where the other was expected, they are freely interchangeable.

Subtyping allows us to type check a process with a single session type (like !`int`.*end*), and know that it is guaranteed to be well-formed for every subtype (such as !`real`.*end*). Because of this guarantee, we can be more flexible in the type rules, reducing the amount of rejected processes that can be shown to execute without error.

2.2 Syntax

Different authors use slightly different syntax for session types, and almost all proofs and definitions are done directly on syntax. As a result, it is hard to reuse work from other authors. This work aims to introduce a system that is not dependent on any single syntax.

We do still need a syntax for examples, and to compare the syntax-less system with previous work on session types. We will use the syntax introduced in [GH05] and [Vas12], whose grammars can be found in Fig. 1 and Fig. 2, respectively. The definitions of this section, the type rules of Section 6 and the type checking algorithm of Section 7 are also based on these two works.

Throughout this work we will use $\widetilde{T}$ to refer to a list $T_1, T_2, ..., T_n$ of types, (distinct) labels $\widetilde{l}$, or other items, and $\{\widetilde{l} : \widetilde{T}\}$ for a list $\{(l_1 : T_1), ..., (l_n : T_n)\}$ of label/type pairs. We assume there are infinitely many labels $l_1, l_2, ...$ and recursion variables $X_1, X_2, ...$

There is a distinction between *linear* and *unrestricted* types. A channel end of a linear type must only ever be held by a single *thread* (non-composite process) at a time, while unrestricted channel ends can be shared by any amount of threads. This is why unrestricted types are also called shared types. Gay and Hole's grammar strictly separates linear and unrestricted types; a linear type is either *end*, recursive $\mu X.T$ or an action (marked by ?, !, & or $\oplus$) combined with one or more continuation types. The only kind of unrestricted type is $\widetilde{\frown}[\widetilde{T}]$, which indicates a bidirectional channel over which tuples of the appropriate types can be sent or received infinitely.

Vasconcelos moves the linear/unrestricted distinction to a qualifier. The basic actions form pre-types, p, to which then a *lin* or *un* qualifier gets prepended. Note that $lin?\texttt{int}.T$ is the same as $?\texttt{int}.T$. To make examples more readable, we will omit the *lin* qualifier and trailing *end* in types (e.g., ?`int` is short for $lin?\texttt{int}.end$).

Much of the grammars is the same, but Vasconcelos' seems to allow for a lot more complexity in unrestricted types. For example, $un?\texttt{int}.un!\texttt{bool}$. While this is a valid string of the grammar, the type rules forbid us from typing any process with such a type. It is, however, considered valid to have a linear type continue as an unrestricted type (but not the other way around).

Strings of these grammars are further required to be contractive, containing no substrings of the form $\mu X_1.\mu X_2...\mu X_n.X_1$, and closed, containing no free recursion variables. Variables are bound by the μ operator, giving rise to the standard definition of free variables. The sets of strings generated by T_{GH} and T_V satisfying these constraints will be called $Type_{GH}$ and $Type_V$ respectively. We will mostly make statements about both grammars simultaneously, using $Type$ to refer to the union of these two sets.

In these grammars, d refers to basic types: `int`, `bool`, `Object`, types that aren't session types. We deliberately leave these unspecified, so it is easy to implement session types on top of an existing (static) type system. The set of all basic types, i.e., all values d can take in the grammars, will be represented by D. We do assume there is some decidable subtyping preorder[2] relation, written as $\leq_D$, on them. A trivial subtyping relation can be constructed through identity, assuming identity is decidable.

For the examples in this work, we will be assuming our basic types are: `nat`, `int`, `real` and `bool`. We will also assume `nat` is a subtype of `int`, which is, in turn, a subtype of `real`.

Two expressions (type, process or otherwise) are α-equivalent when one can be transformed into another by renaming bound variables (along with all references). Clearly, there is no need to distinguish between a type $\mu X.?\texttt{int}.X$ and $\mu Y.?\texttt{int}.Y$. As is usual, we will be working up to α-equivalence, meaning we will not address all possible α-equivalent cases.

We will also follow Barendregt's Variable Convention ([Bar84]) and assume all bound variables are distinct from all other variables (free or bound) in scope when binding—recall that we already assumed there are no free variables in our types, but this is not always true for processes. This assumption is allowed because any bound variable can be renamed to a fresh name while remaining α-equivalent.

Due to Barendregt's convention capture-avoiding substitution of a session type S for variable X, written as $T[S/X]$, is straightforward. If X is a free name of an expression, every occurrence of X must be a reference to the free variable. If X is not free, substitution does nothing. The most common use-case of substitution in types is to unfold a recursive type. The variable X in $T = \mu X.S$ refers to T, so substitution does not change the behaviour of the type.

[2] A preorder is a relation that is reflexive and transitive

$$
\begin{array}{rcll}
P, Q & ::= & x(\widetilde{y}).P & \text{bind input from channel } x \text{ to variable } y \\
& | & \overline{x}\langle\widetilde{y}\rangle.P & \text{output } y \text{ on channel } x \\
& | & x \rhd \{\widetilde{l} : \widetilde{P}\} & \text{offer choices } l_1, l_2, ... \\
& | & x \lhd l.P & \text{make choice } l \\
& | & P \mid P & \text{composition} \\
& | & !P & \text{replication} \\
& | & \mathbf{0} & \text{finished process} \\
& | & (\nu xy) & \text{channel creation}
\end{array}
$$

Figure 3: Process Syntax

Definition 2.1. *The unfolding of a recursive type $\mu X.T$ is defined recursively*

$$unfold(\mu X.T) = unfold(T[\mu X.T/X])$$

And for all other T in Type $unfold$ is the identity: $unfold(T) = T$.

The assumption that all types are contractive ensures $unfold(T)$ terminates for all types T. As all types are required to be closed, we know that $unfold(T)$ can never be a variable X. Any such variable would have to be bound somewhere before use, meaning it would have been substituted. Furthermore, unfolding a closed type always yields another closed type, as each removed binder always causes a substitution of the bound variable.

2.3 Pi-Calculus

The pi-calculus is a language designed to model concurrent programs. As a theoretical language, it is not meant for writing actual programs, but the simple nature makes it ideal for reasoning about concurrency.

As briefly introduced earlier, the pi-calculus consists of—possibly parallel—processes which can communicate over channels. The syntax for processes is given in Fig. 3:

This version of the pi-calculus, based upon the language of [Vas12], defines special language constructs for communicating choices through synchronization signals.

Choices can either be made based on the internal state of the process, which it then communicates over the channel with $\lhd$, or a process can wait to receive a synchronization signal from the channel with $\rhd$, and proceed accordingly. Regular values, which can either be a value of some basic data type or a delegated channel, are sent like $\overline{x}\langle 2\rangle$ or received, and bound to a name, with $x(y)$.

Composition ($P \mid Q$) defines two processes that can proceed in parallel. Sometimes we want to run multiple copies of a process in parallel, this is expressed by replication ($!P$). A channel is created by scope restriction with the ν operator, binding each end of the channel to a variable—(νxy) should explicitly not be read as short for $(\nu x)(\nu y)$. These variables x and y are also referred to as *covariables*, to indicate that they form a channel together. Finally, $\mathbf{0}$ is simply a process that does nothing.

Take, for example, the process

$$(\nu xy)(\overline{x}\langle 2\rangle.P \mid y(z).Q)$$

Reduction

$$(\nu xy)(\overline{x}\langle v\rangle.P \mid y(z).Q \mid R) \longrightarrow (\nu xy)(P \mid Q[v/z] \mid R) \qquad \text{[R-COM]}$$

$$(\nu xy)(x \lhd l_i.P \mid y \rhd \{\widetilde{l} : \widetilde{Q}\} \mid R) \longrightarrow (\nu xy)(P \mid Q_i \mid R) \qquad \text{[R-SYNC]}$$

$$\frac{P \longrightarrow Q}{(\nu xy)P \longrightarrow (\nu xy)Q} \text{[R-CREAT]} \qquad \frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R} \text{[R-PAR]}$$

$$\frac{P \equiv P' \quad P \longrightarrow Q \quad Q \equiv Q'}{P' \longrightarrow Q'} \qquad \text{[R-CONG]}$$

Structural congruence

Parallel composition:

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

Scope restriction:

$(\nu xy)(\nu vw)P \equiv (\nu vw)(\nu xy)P \qquad (\nu xy)\mathbf{0} \equiv \mathbf{0}$

Replication:

$!P \equiv P \mid !P$

Parallel scope restriction:

$(\nu xy)(P \mid Q) \equiv (\nu xy)P \mid Q$ if x any y are not free names of Q

Figure 4: Process semantics

It creates a channel with covariables x and y. The left process sends the value 2 and continues as P, the right process receives a value, binds it to z and then continues as Q. After a single execution step, the process will become $(\nu xy)(P \mid Q[2/z])$, where $Q[2/z]$ is the substitution of value 2 for the free variable z in process Q.

It is important to note that substitution in processes is not always defined, e.g., substituting a value like 1 for a variable that is used as channel is invalid. When writing $P[v/z]$, we will assume the appropriate operation is defined.

The semantics of execution are defined in Fig. 4 as the reduction relation. We say a process P reduces to Q, written $P \longrightarrow Q$, when there is a single execution step yielding Q from P. This can be when two processes communicate, in R-COM, or synchronize, in R-SYNC, over a channel formed by two covariables, as indicated by the scope restriction.

R-CREAT shows that scope restriction allows reductions in the underlying process and R-PAR allows composed processes to proceed in parallel. Finally, R-CONG says that structurally congruent processes have the same reductions.

Two processes are structurally congruent if they are identical in behaviour, but not necessarily in structure. It is the smallest congruence relation satisfying the (standard) axioms in Fig. 4.

3 Category Theory

Before introducing the structure we will use to represent types, we are going to give a brief introduction to category theory, explaining what coalgebras are. Category theory is a very general field: a category is some collection of "objects" and "morphisms" or arrows between these objects. We will be limiting ourselves to the category *Set*, whose objects are sets and whose morphisms are functions mapping from one set, the domain, to another, the codomain.

3.1 Set Operations

We begin by introducing a few relevant operations and their notations.

The *product* $X \times Y$ of two sets is the set $\{(x, y) \mid x \in X, y \in Y\}$ of all pairs with the first value taken from X and the second taken from Y. A function $f : X \to Y$ assigns an element of Y to each element of X. The set Y^X collects all possible functions mapping from X to Y.

The *coproduct* $X + Y$, or $X \coprod Y$, of sets is the same as their disjoint union. The disjoint union of a list of sets is constructed by prepending some index of the originating set to each element.

$$\coprod_{a \in A} X_a = \{(a, x) \mid a \in A,\ x \in X_a\}$$

If X and Y are disjoint (i.e., they share no elements), the disjoint union $X + Y$ is isomorphic to the regular union $X \cup Y = \{x \mid x \in X \text{ or } x \in Y\}$. The coproduct of two functions $f_a : X_a \to Y_a$ and $f_b : X_b \to Y_b$ has the signature

$$f_a + f_b : X_a + X_b \to Y_a + Y_b$$

The domain and codomain are coproducts, so their first element is the index a or b, marking the originating set. This index determines which function needs to be applied.

$$(f_a + f_b)(a, x) = (a, f_a(x)) \qquad (f_a + f_b)(b, x) = (b, f_b(x))$$

The *pairing* of two functions $f : X \to Y$ and $g : V \to W$ is the function

$$\langle f, g \rangle : X \cap V \to Y \times W$$

defined by

$$\langle f, g \rangle(x) = (f(x), g(x))$$

Both paired functions are applied to the argument. Hence, the domain is the intersection and codomain is the product of the original (co)domains.

The *co-pairing* of two functions $f_a : X_a \to Y_a$ and $f_b : X_b \to Y_b$ is the function

$$[f_a, f_b] : X_a + X_b \to Y_a \cup Y_b$$

defined by

$$[f_a, f_b](a, x) = f_a(x) \qquad [f_a, f_b](b, y) = f_b(x)$$

The co-pairing is essentially the coproduct without the index a or b in the result.

Functions $f : X \to Y$ and $g : Y' \to Z$ where the codomain Y is a subset of (or equal to) the domain Y', can be *composed* into a function

$$g \circ f : X \to Z$$

The result of the composite is the same as applying g to the result of f.

$$(g \circ f)(x) = g(f(x))$$

Do note that the functions are applied right-to-left, $g \circ f$ is commonly read as "g after f"

The *powerset* of a set is $\mathcal{P}(\mathcal{X})$ and contains all subsets of X. Recall that a set is finite if it has less element than $\mathbb{N}$. The set of all finite, non-empty subsets is

$$\mathcal{P}^{+}_{<\aleph_0}(X) = \{Y \mid Y \subseteq X \quad Y \neq \emptyset \quad |Y| < |\mathbb{N}|\}$$

3.2 Relations and Orderings

A (binary) relation on X is a subset of $X \times X$. Relations have traditionally been written infix, so we continue to write $x \ R \ y$ to mean $(x, y) \in R$. Some relations have certain properties which make them of special interest: a *preorder* is a relation which is transitive ($x \ R \ y$ and $y \ R \ z$ implies $x \ R \ z$) and reflexive (every element is related to itself), a *partial order* is a preorder that is also anti-symmetric ($x \ R \ y$ and $y \ R \ x$ implies $a = b$) and an *equivalence* is a preorder that is also symmetric ($x \ R \ y$ implies $y \ R \ x$). For example, $\leq$ is a partial order on numbers, and equality is an equivalence.

Relations can be composed as well. The composite $R_1; R_2$ contains all pairs (x, z) for which there is some y with $x \ R_1 \ y$ and $y \ R_2 \ z$. Note that the left relation is applied first, unlike function composition.

A set L equipped with a partial order R is a *complete lattice* if each subset of L has both a supremum (minimal item that is bigger than or equal to each element) and an infimum (maximal item less than or equal to each element) in L. Notice that if you take the order to mean less than or equals, bigger than or equals is given by the transpose ($x \ R^T \ y$ if and only if $y \ R \ x$).

The set of all relations on X, the powerset $\mathcal{P}(X \times X)$, ordered by the subset relation $\subseteq$ forms such a complete lattice. The supremum of a subset is simply the union of all elements (recall that the elements of a powerset are themselves sets) and the infimum is the intersection. Such a union or intersection of subsets always give another subset of $X \times X$, and is thus an element of the powerset.

3.3 Functors

A functor maps between categories, it assigns an object $F(X)$ to each object X and a morphism $F(f) : F(X) \to F(Y)$, to each morphism $f : X \to Y$ of a category. Additionally, a functor F must preserve:

- **Identity:** For any object X, the functor must map the identity morphism of X to the identity morphism of $F(X)$. The identity function of a set simply maps each element to itself, $id(x) = x$.

- **Composition:** For any two morphisms $f : Y \to Z$ and $g : X \to Y$, applying F to the composite, $h_1 = F(f \circ g)$, and composing $h_2 = F(f) \circ F(g)$ must yield the same morphism $h_1 = h_2$.

A broader overview of functors, and category theory in general, can be found in [Awo10].

3.4 Coalgebra

A coalgebra of a specific functor F, often called F-coalgebra, is a pair of an object X and a morphism $c : X \to F(X)$. The interpretation of a coalgebra is entirely dependent on the functor. A common application is in the study of automata or more general labelled transition systems. Rutten provides great examples in [Rut19].

Deterministic finite automata, for example, are usually defined as a tuple $(Q, \Sigma, \delta, q_0, A)$ of a set of states Q, an input alphabet Σ, a transition function $\delta : Q \to Q^\Sigma$, an initial state $q_0 \in Q$ and a set of accepting states $A \subseteq Q$. When purely reasoning about the structure, the input alphabet can be assumed constant between automata and the initial state may be factored out. The information in A and δ can be combined into a function $c : Q \to \{0, 1\} \times X^\Sigma$:

$$c(q) = \begin{cases} (1, \delta(q)) & \text{if } q \in A \\ (0, \delta(q)) & \text{if } q \notin A \end{cases}$$

The pair (Q, c) is a coalgebra of the functor $F(X) = \{0, 1\} \times X^\Sigma$. By interpreting the structure as a coalgebra, we can analyse it with concepts like bisimulation.

3.5 Bisimulation

We are often not interested in the exact structure of a coalgebra, but in the observable behaviour. What is observable, and what is not, depends on the kind coalgebra in question. For states of an automaton, we observe the accepted language. A session type is observed through the processes that can be typed with it. Two coalgebras can have distinct internal structures, but if they behave the same (e.g., they are finite automata accepting the same languages), then they are equivalent.

Let (X, c) be some coalgebra and R be a relation on X. A relation is just a set of pairs, so it has the projections $\pi_1 : R \to X$ and $\pi_2 : R \to Y$. R is a bisimulation if there is a function $c_R : R \to F(R)$ such that the following diagram commutes:

$$\begin{array}{ccccc} X & \xleftarrow{\pi_1} & R & \xrightarrow{\pi_2} & Y \\ \downarrow{\scriptstyle c} & & \downarrow{\scriptstyle c_R} & & \downarrow{\scriptstyle c} \\ F(X) & \xleftarrow[F(\pi_1)]{} & F(R) & \xrightarrow[F(\pi_2)]{} & F(X) \end{array}$$

The diagram commutes if all paths with the same start and endpoint are equivalent. That is, if $c_i \circ \pi_i \equiv F(\pi_i) \circ c_R$ for $i = 1$ and $i = 2$. This equivalence distinguishes between observable properties (which must match) and non-observable properties (which may differ). Bisimilar session types, and labelled transition systems in general, are allowed to transition to different states, as long as those states are also pair-wise bisimilar.

The bisimilarity relation—which is the union of all bisimulations—is an equivalence relation, it is symmetric. Sometimes we need a coalgebra that can match the behaviour of another, but is not necessarily equivalent. In case of finite automata, one might accept a subset of another automaton's language. Simulations are constructed very similar to bisimulations, but the equivalency of paths is weakened to some preorder. Section 5 will precisely define bisimulation and simulation on the coalgebras representing session types.

4 Session Coalgebra

In this section, we will discuss the main topic—and new contribution—of this thesis: the functor F such that coalgebras of this functor describe a session type. Such F-coalgebras can be defined on any kind of set. Most notably, we would like to define a F-coalgebra on the set $Type$ of expressions introduced in section 2.

Unless otherwise specified, a type T is understood to be some element of a F-coalgebra. This could be the coalgebra on $Type$, or any arbitrary F-coalgebra.

Session types are similar to labelled transition systems. Indeed, the functor F resembles the functor of these systems. A regular labelled transition system is a collection of states and a function mapping each state to a set of label-destination pairs. Session types are deterministic; there can only be one destination for each combination of source and label. We also associate each state with extra information, independent of any specific transition.

Part of this static information is an operation, represented by an element of the set O.

$$O = \{com, branch, end, type, par\}$$

Definition 4.1. *The* operation *of a state describes what kind of action it represents.* com *marks the sending or receiving of a value,* branch *an internal or external choice,* end *the completed protocol,* type *a basic data type, and* par *an unrestricted (parallel) type.*

Some operations can be either input or output. The set P contains these two polarities.

$$P = \{in, out\}$$

Definition 4.2. *The* polarity *of a com or branch state can either be* in *or* out*. The polarity distinguishes a receiving action or external choice (input) from a sending action or internal choice (output).*

Note that pairs in $\{com, branch\} \times P$ directly correspond to the actions of a session type:

$? = (com, in)$	receive value
$! = (com, out)$	send value
$\& = (branch, in)$	external choice
$\oplus = (branch, out)$	internal choice

We will be using these markers to abbreviate the pairs.

Recall that D is the set of all basic data types, let $\mathbb{L}$ be the set of all choice labels, let $\mathbb{1} = \{*\}$ be a unit set and let $\mathbb{2} = \{*_1, *_2\}$ be a set with exactly two elements.

We define $\mathbb{N}_+$ as the set of all natural numbers, excluding 0. Variables n, i will denote elements of $\mathbb{N}_+$. The variable l will be used to refer to a label, i.e., an element of $\mathbb{L}$. Variables $L, L_1, L_2, ...$ refer to finite, non-empty subsets of $\mathbb{L}$. The variable p can be a pretype (strings of the p-production in the grammar of Fig. 2), but will mostly denote a polarity of P. These uses should be sufficiently different that the context will disambiguate.

The functor $F : Set \to Set$ is defined on sets as

$$F(X) = \coprod_{a \in A} X^{B_a + C_a}$$

where

$$\begin{array}{rll} A = & \{com\} \times P \times \mathbb{N}_+ & B_{com,p,n} = \{1, ..., n\} \\ & & C_{com,p,n} = \mathbb{1} \\ & \cup \{branch\} \times P \times \mathcal{P}^+_{<\aleph_0}(\mathbb{L}) & C_{branch,p,L} = L \\ & \cup \{end\} & C_{end} = \emptyset \\ & \cup \{type\} \times D & C_{type,d} = \emptyset \\ & \cup \{par\} & C_{par} = \mathbb{2} \end{array}$$

And $B_a = \emptyset$ for any not yet defined.

Elements of $F(X)$ are a pair of an element a of A and a transition function $f : B_a + C_a \to X$. The tuple a gives information directly associated with a specific state: the operation, polarity, number of values communicated, labels of a *branch* or the basic type of a *type* state. B_a and C_a give the domain of the transition function. Transitions from B_a lead to the types to be communicated in a *com* state, and C_a always gives the continuation state(s).

The functor must also map any function

$$f : X \to Y$$

to a function

$$F(f) : F(X) \to F(Y)$$

Elements of $F(X)$ are pairs (a, g), with $a \in A$ and $g : B_a + C_a \to X$. We map to $F(Y)$ by composing the supplied function f with the transition function g.

$$F(f)(a, g) = (a, f \circ g)$$

Theorem 4.3. *F is a functor*

Proof. Functors need to preserve identity and composition. First, consider the identity function id_X, recall that composing identity with a function does not change anything. When applying F to id_X we get:

$$\begin{array}{rl} F(id_X)(a, g) & = (a, id_X \circ g) \\ & = (a, g) \\ & = id_{FX}(a, g) \\ F(id_X) & = id_{FX} \end{array}$$

In the case of composition, we see

$$\begin{aligned}
F(f \circ g)(a,h) &= (a,\ (f \circ g) \circ h) \\
&= (a,\ f \circ (g \circ h)) && \text{associativity} \\
&= F(f)(a,\ g \circ h) && \text{definition of } F(f) \\
&= F(f)(F(g)(a,h)) && \text{definition of } F(g) \\
&= (F(f) \circ F(g))(a,h) && \text{composition} \\
F(f \circ g) &= F(f) \circ F(g)
\end{aligned}$$

Showing F satisfies both conditions. □

Specifically, F is a polynomial functor in a single variable, as defined in 1.5 of [GK09].

4.1 F-Coalgebras

To define a F-coalgebra (X, c), we need to construct a function c that maps from X to $F(X)$. Tuples in $F(X)$ are quite complex, so we will build c up as a combination of simpler functions.

Each element of $F(X)$ is a pair (a, f) of an element of A and a transition function $f : B_a + C_a \to X$. A tuple in A always has an operation, an element of O, which determines the rest of the shape.

We will be assuming a function $\text{op} : X \to O$ and constructing a function $s : X \to A$. There is an induced partition of X for both:

$$\begin{aligned}
X_o &= \{x \in X \mid \text{op}(x) = o\} && \text{for all } o \in O \\
X_a &= \{x \in X \mid s(x) = a\} && \text{for all } a \in A
\end{aligned}$$

Recall that the coproduct of a family of sets is a set of pairs with some index and an element. For the induced partition on operations, these are pairs $(\text{op}(x), x)$. We can map from X to this coproduct by pairing the function op, or s, with the identity on X.

$$\begin{aligned}
\langle \text{op}, id_x \rangle :&\ \ X \to \coprod_{o \in O} X_o \\
\langle s, id_x \rangle :&\ \ X \to \coprod_{a \in A} X_a
\end{aligned}$$

The functions which make up the coalgebra c are:

op	$: X \to O$	maps each state to an operation
pol	$: X_{com} + X_{branch} \to P$	maps *com* and *branch* states to a polarity
ar	$: X_{com} \to \mathbb{N}_+$	maps *com* states to an arity
la	$: X_{branch} \to \mathcal{P}^+_{<\aleph_0}(\mathbb{L})$	maps *branch* states to a (finite, non-empty) set of labels
δ_a	$: X_a \to X^{B_a + C_a}$	maps states with $s(x) = a$ to their transition function

The function $s : X \to A$, which gives the static information of each state, is defined as $s = [s_{com}, s_{branch}, ..., s_{par}] \circ \langle \text{op}, id_x \rangle$, using the functions:

$$\begin{aligned}
s_{com} &= \langle \text{op}, \text{pol}, \text{ar} \rangle \\
s_{branch} &= \langle \text{op}, \text{pol}, \text{la} \rangle \\
s_{end} &= \text{op}_{end} && (\text{op restricted to } X_{end} \to \{end\}) \\
s_{type} &= \langle \text{op}, \text{da} \rangle \\
s_{par} &= \text{op}_{par} && (\text{op restricted to } X_{par} \to \{par\})
\end{aligned}$$

Each part of the co-pairing has the signature $s_o : X_o \to A$. We use the pairing of op and the identity function to map from X to the coproduct of domains X_o, resulting in the signature $s : X \to A$.

Recall that each assumed global transition function δ_a has the signature

$$\delta_a : X_a \to X^{B_a + C_a}$$

The coproduct of these functions is

$$\coprod_{a \in A} \delta_a : \coprod_{a \in A} X_a \to \coprod_{a \in A} X^{B_a + C_a}$$

The codomain is exactly $F(X)$, but the domain is the coproduct $\coprod_{a \in A} X_a$, rather than X itself. We can adjust this with a pairing of s and the identity. The function c is defined as

$$c = (\coprod_{a \in A} \delta_a) \circ \langle s, id_x \rangle$$

Whose signature is the required $c : X \to F(X)$.

Notice there is only one function that fits the signature $f_\emptyset : \emptyset \to X$, which is an "empty" function mapping nothing. Hence, δ_{end} and δ_{type} are uniquely defined by X.

We will generally use a single transition function $\delta : X \to \bigcup_{a \in A} X^{B_a + C_a}$, defined as

$$\delta = [\delta_a, \delta_b, ...] \circ \langle s, id_x \rangle$$

Where the co-pairing ranges over all elements of A. As long as the mappings of δ have the appropriate types, a definition of δ can be transformed into definitions for these functions δ_a, δ_b, etc.

It might seem odd to encode each basic type as a state of the coalgebra, rather than including a list of basic types in a *com* tuple. This allows uniform handling of regular and higher-order communications. A higher-order type uses a session type as the type to be sent or received, such as in $?(!\texttt{int}).T$. A channel end of this type expects to receive another channel, of type $!\texttt{int}$. Session delegation is often used to establish a private (linear) session over an unrestricted channel. If we had defined A as something like

$$... \cup (\{com\} \times P \times D^+) \cup ...$$

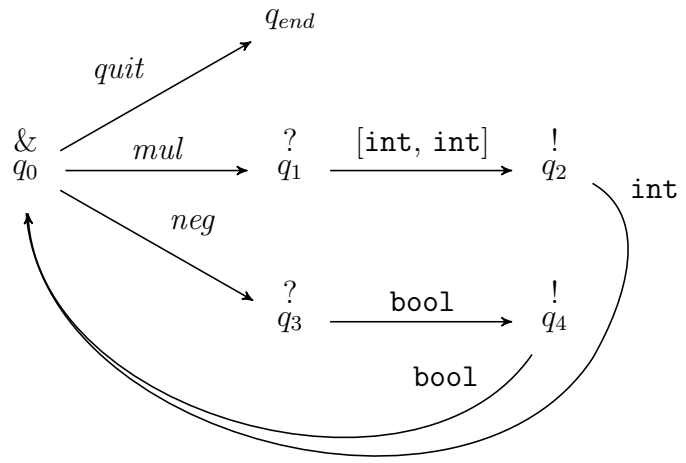
Where D^+ is a non-empty list of basic data types, then session delegation would not have been possible (or would need a special case).

Example

Let us revisit the *server* type from Section 2.

$$\mu X. \, \& \begin{cases} quit : & end \\ mul : & ?[\texttt{int}, \texttt{int}].!\texttt{int}.X \\ neg : & ?\texttt{bool}.!\texttt{bool}.X \end{cases}$$

The used labels are $\mathbb{L} = \{mul, neg, quit\}$. A coalgebra for the same behaviour could be



The graph is quite similar to the expression. Formally, the coalgebra is the pair (X, c), with:

$$X = \{q_0, q_1, q_2, q_3, q_4, q_{end}, q_{int}, q_{bool}\}$$

	q_0	q_1	q_2	q_3	q_4	q_{end}
op	*branch*	*com*	*com*	*com*	*com*	*end*
pol	*in*	*in*	*out*	*in*	*out*	
$\delta(q_i)(*)$		q_2	q_0	q_4	q_0	
$\delta(q_i)(1)$		q_{int}	q_{int}	q_{bool}	q_{bool}	
$\delta(q_i)(2)$		q_{int}				
$\delta(q_i)(quit)$	q_{end}					
$\delta(q_i)(mul)$	q_1					
$\delta(q_i)(neg)$	q_3					

	q_{int}	q_{bool}
op	*type*	*type*
da	`int`	`bool`

Although the basic types are shown as labels in the diagram, they are, in fact, transitions. There are two transitions from q_1 to q_{int}, one from q_2 to q_{int}, etc. This visualization makes the graph more compact and, in our opinion, clearer.

4.2 Coalgebra of Types

The following table defines a function $c_{Type} : Type \to F(Type)$ in terms of the individual parts $\mathrm{op}, \mathrm{pol}, \delta, \mathrm{ar}$ and la. Together with the set $Type$ of expressions, this defines a coalgebra directly on the syntax of session types.

T	c_{Type}			
	$\text{op}(T)$	$\text{pol}(T)$	$\delta(T)$	$\text{ar}(T)$
$?[T_1, ..., T_n].S$	com	in	$\delta(T)(*) = S$	n
$![T_1, ..., T_n].S$		out	$\delta(T)(i) = T_i$	
$lin?T'.S$	com	in	$\delta(T)(*) = S$	1
$lin!T'.S$		out	$\delta(T)(1) = T'$	

There are two separate cases, one for *com* types from Gay and Hole and one for those from Vasconcelos.

T	$\text{op}(T)$	$\text{pol}(T)$	$\delta(T)$	$\text{la}(T)$
$(lin)\&\{\widetilde{l} : \widetilde{T}\}$	branch	in	$\delta(T)(l_i) = T_i$	$\{l_1, ..., l_n\}$
$(lin) \oplus \{\widetilde{l} : \widetilde{T}\}$		out		

Because *branch* types are the same between the two systems, besides the *lin* qualifier in Vasconcelos, they are handled together. The set of labels $\text{la}(T)$ is guaranteed to be finite, by virtue of an expression being a finite string.

T	$c_{Type}(T)$
end	(end)
d	$(type, d)$
$\mu X.T$	$c_{Type}(unfold(\mu X.T))$

The completed protocol *end* and basic types d are straightforward. Recursive types are handled according to their unfolding. Recall that contractivity ensures that $unfold$ always terminates. As our types are closed, all recursion variables are substituted during the unfolding of their binder. Consequently, we do not need to define c on these variables.

T	$\text{op}(T)$	$\delta(T)$
$\widehat{}[T_1, ..., T_n]$	par	$f(*_1) = ?[T_1, ..., T_n].\widehat{}[\widetilde{T}]$ $f(*_2) = ![T_1, ..., T_n].\widehat{}[\widetilde{T}]$
$un\ p$	par	$f(*_1) = f(*_2) = lin\ p$

Finally, unrestricted types need to be handled separately again. A *par* state wraps two underlying types, additionally marking them as unrestricted. Here, p refers to the pretypes defined in the grammar of Fig. 2. In bidirectional types $\widehat{}[\widetilde{T}]$, we need to allow bothe reading and writing. Looping back to the original type and repeating infinitely, in either case. Vasconcelos' *un* qualified types have the same behaviour as the *lin* variants, just unrestricted. They also don't have multiple behaviours, so the *par* state has only one distinct underlying type.

5 Relations on Types

With our functor defined, we can formally and precisely define bisimulation, simulation and duality of types. There are many ways to define these relations. We will be defining them as the greatest postfixpoint of some monotonic function. We will also be defining which unrestricted types are valid and which are not, by defining their parallelizability.

Let (X, c) be some coalgebra. For each of bisimulation, simulation and duality, we are going define a function from relations on X, represented by the preordered set $Rel_X = (\mathcal{P}(X \times X), \subseteq)$, to relations on $F(X)$ (i.e., Rel_{FX}). This function is then composed with the inverse c^* of c on relations, to get a function from Rel_X to Rel_X.

Definition 5.1. *The* inverse $c^* : Rel_{FX} \to Rel_X$ *yields all pairs* (x, y) *for which* $(c(x), c(y))$ *is in the original relation;* $c^*(R) = \{(x, y) \mid (c(x), c(y))$ *in* $R\}$

Rel_X, as a powerset ordered on $\subseteq$, is a complete lattice. A fixed point of a function f is any x for which $x = f(x)$, a postfixpoint is any x for which $x \subseteq f(x)$. The Knaster-Tarski Theorem (see [Tar55]) states that if a function $f : Rel_X \to Rel_X$ is monotonic (i.e., $x \subseteq y$ implies $f(x) \subseteq f(y)$), then there exists a (unique) greatest postfixpoint, which is also the greatest fixed point. By definition of greatest (on Rel_X), such a postfixpoint is a superset of all postfixpoints. Consequently, to prove a pair (x, y) is in the greatest postfixpoint, we need only prove there exists some postfixpoint containing (x, y).

We can show the inverse c^* is monotonic.

Lemma 5.2. *The function* c^* *distributes over the union of sets*

Proof. Let R and S be any relations of Rel_{FX}. By definition, $c^*(R) = \{(x, y) \mid (c(x), c(y)) \in R_{FX}\}$ and $c^*(S) = \{(x, y) \mid (c(x), c(y)) \in S_{FX}\}$, their union becomes:

$$\begin{aligned} c^*(R) \cup c^*(S) &= \{(x, y) \mid (c(x), c(y)) \in R \text{ or } (c(x), c(y)) \in S\} \\ &= \{(x, y) \mid (c(x), c(y)) \text{ in } R \cup S\} \\ &= c^*(R_{FX} \cup S_{FX}) \end{aligned}$$

□

Lemma 5.3. *The function* c^* *is monotonic*

Proof. Let R and S be any elements of Rel_{FX} with $S \subseteq R$. As a superset, $R = S \cup R - S$. By distribution, $c^*(R) = c^*(S \cup (R - S)) = c^*(S) \cup c^*(R - S)$, which implies $c^*(S) \subseteq c^*(R)$. □

If c^* is composed with another monotonic function, the composite must be monotonic as well

Lemma 5.4. *If a function* $f : Rel_X \to Rel_{FX}$ *is monotonic, the composite* $(c^* \circ f)$ *is monotonic*

Proof. By assumption, $R_x \subseteq R_y$ implies $f(R_x) \subseteq f(R_y)$. By Lemma 5.3, this implies $c^*(f(R_x)) \subseteq c^*(f(R_y))$, hence $R_x \subseteq R_y$ implies $(c^* \circ f)(R_x) \subseteq (c^* \circ f)(R_y)$, which is the condition for monotonicity. □

5.1 Bisimulation

Two states of a coalgebra are said to be bisimilar if they describe the same behaviour. Two states can only ever be bisimilar if their operation, polarity and labels are equal (or both undefined). Furthermore, the states that are transitioned to, for the same label, need to be bisimilar as well. We define the function $f_\sim : Rel_X \to Rel_{FX}$ as

$$\begin{aligned}
f_\sim(R) = \; & \{\, ((com, p, n, f), (com, p, n, f')) && \mid \begin{array}{l} f(*)\ R\ f'(*), \text{ and} \\ f(i)\ R\ f'(i) \quad \text{for all } i \le n \end{array} \,\} \\
\cup \; & \{\, ((branch, p, L, f), (branch, p, L, f')) && \mid f(l)\ R\ f'(l) \quad \text{for all } l \text{ in } L \,\} \\
\cup \; & \{\, ((end, f_\emptyset), (end, f_\emptyset)) \,\} && \\
\cup \; & \{\, ((type, d, f_\emptyset), (type, d', f_\emptyset)) && \mid d \le_D d' \wedge d' \le_D d \,\} \\
\cup \; & \{\, ((par, f), (par, f')) && \mid f(*_1)\ R\ f'(*_1) \text{ and } f(*_2)\ R\ f'(*_2) \,\} \\
\cup \; & \{\, ((par, f), (par, f')) && \mid f(*_1)\ R\ f'(*_2) \text{ and } f(*_2)\ R\ f'(*_1) \,\}
\end{aligned}$$

Recall that we only assumed a preorder on basic types. Preorders aren't necessarily anti-symmetric; thus, it is possible for two distinct basic types to both be a subtype of each other. Such types are equivalent, so we consider them bisimilar. As any type is a subtype of itself, two states with the same basic type are always bisimilar.

We consider parallel types as an unordered pair; necessitating a case for both possible pairings of the transitions.

To prove $cf_\sim$, defined as $cf_\sim = (c^* \circ f_\sim)$, has a greatest postfixpoint, we need to prove it is monotonic

Theorem 5.5. *The function $cf_\sim$ has a greatest postfixpoint*

Proof. The result of $f_\sim(R)$ is only ever positively dependent on R, adding an element to R never removes an element from $f_\sim(R)$. As such, it must be monotonic. The result of composing two monotonic functions (using the same ordering) must be monotonic by transitivity. By the Knaster-Tarski Theorem, $cf_\sim$ must have a greatest postfixpoint. □

Definition 5.6. *A relation R is called a bisimulation if it is a postfixpoint of $cf_\sim$. The greatest postfixpoint is the bisimilarity relation $\sim$.*

Notice that bisimilarity is symmetric, transitive and reflexive, forming an equivalence relation.

Lemma 5.7. *The bisimilarity relation $\sim$ is symmetric*

Proof. All properties of $f_\sim(R)$ that don't depend on R are symmetric. For any bisimulation R, the transpose $R^T = \{(x, y) \mid (y, x) \in R\}$ must also be a bisimulation. The transpose $\sim^T$ being a bisimulation implies $y \sim x$ for all $x \sim y$. □

Lemma 5.8. *The bisimilarity relation $\sim$ is transitive*

Proof. Recall that $\le_D$ is assumed to be transitive, so all properties not related to R are transitive. The composition of $\sim$ with itself must be a bisimulation. □

Lemma 5.9. *The bisimilarity relation $\sim$ is reflexive*

Proof. Let $R = \{(x, x) \mid x \in X\}$ be the diagonal of X. Both elements of any pair (x, x) have the same transition function. Applying an argument to the same function twice must yield a pair (y, y), which is part of the diagonal R. All other required properties hold for an equal pair, so the diagonal is a bisimulation. □

Corollary 5.10. *The bisimilarity relation $\sim$ is an equivalence relation*

5.2 Duality

The basis of duality is the exchange of polarity: the dual of input is output and the dual of output is input.

$$\overline{in} = out \qquad \overline{out} = in$$

The function $f_\perp : Rel_X \to Rel_{FX}$ is defined as

$$\begin{aligned}
f_\perp(R) = \; & \{\, ((com, p, n, f), (com, \overline{p}, n, f')) && \mid \begin{array}{l} f(*)\ R\ f'(*), \text{ and} \\ f(i) \sim f'(i) \quad \text{for all } i \leq n \end{array} \,\} \\
\cup \; & \{\, ((branch, p, L, f), (branch, \overline{p}, L, f')) && \mid f(l)\ R\ f'(l) \quad \text{for all } l \text{ in } L \,\} \\
\cup \; & \{\, ((end, f_\emptyset), (end, f_\emptyset)) \,\} \\
\cup \; & \{\, ((par, f), (par, f')) && \mid f(*_1)\ R\ f'(*_1) \text{ and } f(*_2)\ R\ f'(*_2) \,\} \\
\cup \; & \{\, ((par, f), (par, f')) && \mid f(*_1)\ R\ f'(*_2) \text{ and } f(*_2)\ R\ f'(*_1) \,\}
\end{aligned}$$

To communicate properly, *com* types need to agree on the types to be sent. The continuation types must be dual to ensure the entirety of the behaviours are dual. *branch* types need to have the same labels, and again, dual continuations. The completed protocol *end* is only dual to itself. Duality is strictly defined on session types, a basic type can never be dual to anything. A *par* type is dual if the underlying types are dual, taking into account either ordering of underlying types.

This function is monotonic by the same argument as for bisimilarity.

Definition 5.11. *A relation R is called a duality if it is a postfixpoint of $cf_\perp = (c^* \circ f_\perp)$. The greatest postfixpoint is the duality relation $\perp$.*

Lemma 5.12. *The duality relation $\perp$ is symmetric*

Proof. Just like bisimulation, all properties not related to R are symmetric, so the transpose of any duality is itself a duality. □

It is useful to have a duality function mapping any x to their dual $\overline{x}$, if they have one. Define the partial function $\overline{\cdot}$ on tuples of $F(X)$ to exchange polarity recursively for all continuations.

$$\begin{aligned}
c(x) = (com, i, n, f) \quad & \to \overline{c(x)} = (com, \overline{i}, n, f') && \begin{array}{l} f'(*) = \overline{f(*)} \\ f'(i) = f(i) \quad \text{for all } i \leq n \end{array} \\
c(x) = (branch, i, L, f) \quad & \to \overline{c(x)} = (branch, \overline{i}, L, f') && f'(l) = \overline{f(l)} \quad \text{for all } l \in L \\
c(x) = (end, f_\emptyset) \quad & \to \overline{c(x)} = (end, f_\emptyset) \\
c(x) = (type, d, f_\emptyset) \quad & \to \overline{c(x)} \text{ undefined}
\end{aligned}$$

Not every state of a coalgebra (X, c), for which duality is defined, is guaranteed to have a dual state in X. Because we would still like to use the duality function on such state, $\overline{\cdot}$ does not map

to X, but to the set of fresh variables $X' = \{x' \mid \overline{c(x)} \text{ is defined}\}$. We define $c'(x') = \overline{c(x)}$, which forms a coalgebra together with the original

$$(\, X + X',\ [c, c']\,)$$

By construction, $x \perp x'$, so we define $\overline{x} = x'$. This extension $X + X'$ is defined purely by the original coalgebra. When assuming an arbitrary X, we can implicitly assume it is actually an extended $X = Y + Y'$. We do want to avoid having to create another extension when using the duality function on one of these fresh variables. We see, using induction, that for any copy x' the duality function yields $\overline{c'(x')} = c(x)$. We can simply define $\overline{x'} = x$. Finally, basic types don't have duals; when writing $\overline{x}$, we will assume x is not a basic type.

On syntax, a duality function was originally defined to map $\overline{\mu X.?X.end}$ to $\mu X.!X.end$. As pointed out in [GTV20], these types are not, in fact, dual. The meaning of X was altered in construction, so the two types do not agree on the type to be sent. Our definition avoids this problem by abstracting away from binders and variables into the underlying structure.

Let $T = \mu X.?X.end$ be the type in question; the variable X is bound to T. Using the coalgebra of types:

$$c(T) = (com, in, 1, f) \qquad f(*) = end \qquad f(1) = T$$

The completed protocol isn't changed by duality; $c(end) = \overline{c(end)}$, so $end \sim \overline{end}$. Expanding $\overline{c(T)}$ yields

$$\overline{c(T)} = (com, out, 1, f') \qquad f'(*) = \overline{f(*)} \sim end \qquad f'(1) = T$$

A bisimilar expression would be $T' = \mu X.!T.end$, which is indeed dual to $\mu X.?X.end$.

5.3 Parallelizability

The major difference between a linear type and an unrestricted type is that a channel end of unrestricted type may be copied and shared between parallel processes. Each process uses the channel independently, without informing the others. Furthermore, there is no mechanism to coordinate which process receives which message. If a something is sent on the channel, it could be read by a process that just started using the channel, a process that is almost done using the channel, or a process that is anywhere in between.

In practice, this means an unrestricted channel can only carry data of one type or labels of one specific set (exclusively, if the channel carries data, it cannot carry any kind of labels, and vice versa). However, the structure of the functor F allows us to define arbitrarily complex unrestricted types. For example:

$$\mu X.un?\texttt{int}.un?\texttt{bool}.X$$

This expression is in $Type$, so it is a proper session type. We define the parallelizability predicate $\text{par}(x)$ to mark unrestricted types that can be used without errors. The previous example is not parallelizable.

We distinguish data transitions, which give the data to be communicated, from continuation transitions, which describe how a type will continue to behave after an action. In terms of a F-coalgebra, data transitions have elements of $\mathbb{N}_+$ as argument to the transition function (i.e., they are described by $\delta(T)(i)$ for some state T and natural number i), whereas continuations have elements of $\mathbb{1}$, $\mathbb{2}$ or $\mathbb{L}$ as argument.

So, an unrestricted channel x of type T can be copied between an arbitrary amount of processes. Each of these uses the channel without synchronizing with the other processes. At an arbitrary point during execution, the type of the channel in one of these processes can only ever be a type that is reachable through continuations from the original type T.

Definition 5.13. *The continuation relation $\longrightarrow_C$ of a coalgebra (X, c) is the smallest relation where*

$$\begin{array}{lll} c(x) = (com, p, n, f) & \textit{implies} & x \longrightarrow_C f(*) \\ c(x) = (branch, p, L, f) & \textit{implies} & x \longrightarrow_C f(l) \quad \textit{for all } l \in L \\ c(x) = (par, f) & \textit{implies} & x \longrightarrow_C f(*_1) \ \textit{ and } \ x \longrightarrow_C f(*_2) \end{array}$$

Definition 5.14. *The set of continuations $\langle x \rangle_C$ of any state x of a coalgebra (X, c) is the smallest subset of X, containing x, that is closed under the continuation relation*

If we can guarantee that all these reachable types are communications of the same data types or branches of the same labels, we can guarantee that a single channel of that type can be used by multiple processes in parallel. That is, all pairs of two elements of $\langle T \rangle_C$ must have equivalent data types or labels. There is one exception; a *par* state does not directly describe any interaction with a channel, just the structure of a type, so pairs with at least one *par* state can safely be deemed compatible.

The function $f_{\parallel} : X \to Rel_{FX}$ is defined as

$$\begin{array}{rll} f_{\parallel}(X) = & \{((com, p, n, f), (com, p', n, f')) & \mid f(i) \sim f'(i) \quad \text{for all } i \leq n\} \\ \cup & \{((branch, p, L, f), (branch, p', L, f'))\} & \\ \cup & \{((end, f_\emptyset), (end, f_\emptyset))\} & \\ \cup & F(X) \times \{(par, f)\} & \\ \cup & \{(par, f)\} \times F(X) & \end{array}$$

The polarity of states is allowed to be different and, because all continuations are in $\langle x \rangle_C$ anyway, we do not need to check continuations of any individual pair.

Definition 5.15. *The state compatibility relation $\parallel$ is defined as $\parallel = (c^* \circ f_{\parallel})(X)$*

The function $f_{\parallel}$ maps directly from a set X, rather than a relation, so postfixpoints are not needed for this definition.

Lemma 5.16. *The relation $\parallel$ is symmetric.*

Proof. By definition, $x \parallel y$ if and only if one of x or y in X_{par} or they have the same operation, arity, labels and bisimilar data transitions. By symmetry, one of y or x in X_{par} or they still have the same operation, arity, etc. It follows that $y \parallel x$ □

If all reachable states pairwise have the same data types or labels, then the entire type only allows those data types or labels.

Definition 5.17. *A state x is parallelizable—written $par(x)$—if $v \parallel w$ for all states v and w in $\langle x \rangle_C$.*

Additionally, we may want to check if two types could be combined into a parallelizable type.

Definition 5.18. *Two states x and y are pairwise parallelizable—written $par(x, y)$—if $v \parallel w$ for all states v and w in $\langle x \rangle_C \cup \langle y \rangle_C$.*

Lemma 5.19. *A state x with $c(x) = (par, f)$ is parallelizable if and only $f(*_1)$ and $f(*_2)$ are pairwise parallelizable*

Proof. Let $x_1 = f(*_1)$ and $x_2 = f(*_2)$. Clearly, x_1 and x_2 are reachable from x. Thus, $\langle x \rangle_C$ is the union of $\langle x_1 \rangle_C$, $\langle x_2 \rangle_C$ and $\{x\}$ itself. Every pair of elements in $\langle x \rangle$ either has x, which is *par* so trivially in $\parallel$, or is a pair of $\langle x_1 \rangle_C \cup \langle x_2 \rangle_C$. All pairs of $\langle x \rangle_C$ are related in $\parallel$ if and only if all pairs of $\langle x_1 \rangle_C \cup \langle x_2 \rangle_C$ are. □

Parallelizability marks unrestricted types that can be used without errors. Gay and Hole's grammar is specific enough that unrestricted types are always parallelizable.

Lemma 5.20. *Every unrestricted type $\widetilde{\ }[T_1, ..., T_n]$ is parallelizable*

Proof. Let $T = \widetilde{\ }[\widetilde{T}]$, recall that $\delta(T)(*_1) = ?[\widetilde{T}].T$ and $\delta(T)(*_2) = ![\widetilde{T}].T$. The continuations are $\langle T \rangle_C = \{T,\ ?[\widetilde{T}].T,\ ![\widetilde{T}].T\}$. The top-level T is a *par* type, so pairs containing T are trivially in $\parallel$. The only other pair, $(?[\widetilde{T}].T,\ ![\widetilde{T}].T)$, has equal operation and arity. As all data transitions are the same (thus, bisimilar) for both types, $?[\widetilde{T}].T$ and $![\widetilde{T}].T$ are compatible. All pairs of $\langle T \rangle_C$ are compatible; consequently, T is parallelizable. □

This does not hold in $Type_V$. Vasconcelos made the type rules require that all continuations of an unrestricted type T must be equal to T. Our definition of parallelizability is similar, but loosens this constraint in two ways:

- We allow both input and output to appear in a single type, as long as the communicated data has the same type. A process can hold a copy of both channel ends anyway, so it makes sense to allow this from a single end.
- We allow a linear continuation of an unrestricted type if the *un* qualified version would have been valid, e.g., we consider $\mu X.un?\texttt{int}.?\texttt{int}.X$ valid because $\mu X.un?\texttt{int}.\textbf{un}?\texttt{int}.X$ is.

The semantics of unrestricted types are slightly different between Vasconcelos' original type system and our coalgebraic version. Still, we do not consider the difference a significant loss in generality. Rather, this shows that coalgebras allow our type system to be more flexible. It is possible to define parallelizability to exactly coincide with Vasconcelos' constraints, but then one loses the ability to also express bidirectional types within the same system.

5.4 Simulation and Subtyping

Where bisimilarity is a symmetric equivalence, simulation tells us when one state contains the behaviour of another. A good intuition is that simulation of F-coalgebras describes subtyping for session types, and indeed Theorem 6.4 will show that simulation of types guarantees substitutability in processes.

The order of related items matters for simulations. In general, continuations and input are covariant (equal transitions of a related pair are related in the same order) and output is contravariant (transitions are related in the opposite order).

The function $f_{\sqsubseteq} : Rel_X \to Rel_{FX}$ is defined by

$$
\begin{aligned}
f_{\sqsubseteq}(R) = \quad & \{\ ((com, in, n, f), (com, in, n, g)) \quad | \begin{array}{l} f(*)\ R\ g(*), \text{ and} \\ f(i)\ R\ g(i) \quad \text{for all } i \leq n \end{array} \} \\
\cup \quad & \{\ ((com, out, n, f), (com, out, n, g)) \quad | \begin{array}{l} f(*)\ R\ g(*), \text{ and} \\ g(i)\ R\ f(i) \quad \text{for all } i \leq n \end{array} \} \qquad (1a) \\
\cup \quad & \{\ ((branch, in, L_1, f), (branch, in, L_2, g)) \quad | \begin{array}{l} L_1 \subseteq L_2, \text{ and} \\ f(l)\ R\ g(l) \quad \text{for all } l \text{ in } L_1 \end{array} \} \\
\cup \quad & \{\ ((branch, out, L_1, f), (branch, out, L_2, g)) \quad | \begin{array}{l} L_2 \subseteq L_1, \text{ and} \\ f(l)\ R\ g(l) \quad \text{for all } l \text{ in } L_2 \end{array} \} \qquad (1b) \\
\cup \quad & \{\ ((end, f_\emptyset), (end, f_\emptyset))\ \} \\
\cup \quad & \{\ ((type, d, f_\emptyset), (type, d', f_\emptyset)) \quad | \ d \leq_D d'\ \} \qquad (1c) \\
\cup \quad & \{\ ((par, f), (par, g)) \quad | \begin{array}{l} f(*_1)\ R\ g(*_1), \quad f(*_2)\ R\ g(*_2), \text{ and} \\ \text{par}(f(*_1), f(*_2)) \quad \text{if and only if} \quad \text{par}(g(*_1), g(*_2)) \end{array} \} \\
\cup \quad & \{\ ((par, f), (par, g)) \quad | \begin{array}{l} f(*_1)\ R\ g(*_2), \quad f(*_2)\ R\ g(*_1), \text{ and} \\ \text{par}(f(*_1), f(*_2)) \quad \text{if and only if} \quad \text{par}(g(*_1), g(*_2)) \end{array} \} \qquad (1d)
\end{aligned}
$$

Part 1a: Communications are fairly straightforward, aside from the previous caveat on co- and contravariance.

Part 1b: Branches only have continuation types, but their labels may differ. A process using some type T with internal choice will only choose the options available in T. It doesn't matter if a subtype U has more choices, as long as it has all of the choices of T. The set of internal choices in the subtype can be expanded and external choice may be restricted.

Part 1c: The completed protocol can only simulate itself. Simulation of *type* states is exactly subtyping on their basic types.

Part 1d: In *par* states, we need to account for both orderings again, and we need to conserve parallelizability. Simulation should describe substitutability. Naturally, an invalid type can not be used where a valid type was expected. The other way around, substituting a parallelizable type for a non-parallelizable, could be allowed. Yet, any process that is valid for a channel with unrestricted, but non-parallelizable, type can never actually use said channel. Substituting its type is not particularly useful.

Definition 5.21. *A relation R is called a simulation if it is a postfixpoint of $cf_{\sqsubseteq} = (c^* \circ f_{\sqsubseteq})$. The greatest postfixpoint is the simulation relation $\sqsubseteq$.*

Theorem 5.22. *The simulation relation $\sqsubseteq$ is a preorder*

Lemma 5.23. *The simulation relation $\sqsubseteq$ is transitive*

Proof. Let $R = \sqsubseteq;\sqsubseteq$ be the self-composition of the simulation relation. By construction, every pair has the same properties, except for labels and basic data types. The subset relation $\subseteq$, subtyping $\leq_D$ and implications of parallelizabillity are all transitive. R is a simulation. □

Lemma 5.24. *The simulation relation $\sqsubseteq$ is reflexive*

Proof. Let $R = \{(x, x) \mid x \in X\}$ be the diagonal of X. By construction, every pair has the same operation, polarity, etc. Furthermore, the subset relation $\subseteq$ and subtyping $\leq_D$ are both reflexive, and types are trivially parallelizable if and only if they are parallelizable. R is a simulation. □

The implications of parallelizability in *par* states are important enough to be stated in a lemma.

Lemma 5.25. *Let x and y be some states of a coalgebra with $c(x) = (par, f)$ and $x \sqsubseteq y$. State x is parallelizable if and only if state y is parallelizable*

Proof. A trivial consequence of part 1d of the definition of simulation, using Lemma 5.19—which states that $\mathrm{par}(x)$ if and only if $\mathrm{par}(f(*_1), f(*_2))$. □

5.5 Composites

Bisimulation, simulation and duality share a lot between their definitions. In this subsection we are going to see how these relations compose with each other. Recall that stating "R is a bisimulation" is equivalent to "R is a subset of the bisimilarity relation", or "$x \sim y$ for all pairs (x, y) in R". The set $R_1 \,;\, R_2$ is the composition, defined as

$$R_1 \,;\, R_2 = \{(x, z) \mid x \; R_1 \; y \text{ and } y \; R_2 \; z\}$$

Dual types are opposite, and it makes sense that the opposite of an opposite is equivalent.

Lemma 5.26 (Properties of duality)**.** *Let x, y and z be states of a coalgegra*

1. *$x \perp y$ and $y \perp z$ implies $x \sim y$*
2. *$x \perp y$ and $y \sim z$ implies $x \perp z$*

Proof. To prove the first, let R be the composition $\perp \,;\, \perp$ of the duality relation with itself. By definition of duality, all pairs $x \; R \; y$ must have the same operation, polarity (if $\overline{p_2} = p_1$ then $\overline{p_1} = p_2$), arity and labels. We can also infer that if their operation is *com*, then $\delta(x)(i) \sim \delta(y)(i)$ for all $i \leq \mathrm{ar}(x)$. As such, R can be expanded to a bisimulation. All pairs (x, y) for which there exists a z with $x \perp z \perp y$ are in R, implying $x \sim y$.

The second is done in a similar fashion. Let $R = \perp \,;\, \sim$. Bisimulation preserves all static properties, and implies that all pairwise transitions—including those of *com* states—must also be bisimilar. By transitivity, all data transitions of *com* states are bisimilar. Continuation states must, by construction, be related as $\delta(x)(\alpha) \perp \delta(y)(\alpha) \sim \delta(z)(\alpha)$, for any α, meaning $(\,\delta(x)(\alpha),\, \delta(z)(\alpha)\,)$ is in R. As such, R is a duality. □

Continuing, if two states are pairwise parallelizable, then two other, equivalent, states are logically still pairwise parallelizable.

Lemma 5.27. *Let x_1, x_2 and x_3 be three non-par states (i.e., $\mathrm{op}(x_i) \neq par$ for all x_i). Bisimilarity of $x_1 \sim x_2$ and compatibility of $x_2 \parallel x_3$ implies the compatibility of $x_1 \parallel x_3$.*

Proof. We merely consider *com*, *branch* and *end* states. Bisimilarity implies compatibility (as bisimulation, too, states that operation, labels and arity must match) and, when excluding *par*, compatibility is transitive. Therefore, $x_1 \sim x_2 \parallel x_3$ implies $x_1 \parallel x_3$. □

Lemma 5.28. *For bisimilar states $v_1 \sim w_1$ and $v_2 \sim w_2$, it holds that par(v_1, v_2) if and only if par(w_1, w_2)*

Proof. Let $V = \langle v_1 \rangle_C \cup \langle v_2 \rangle_C$ and $W = \langle w_1 \rangle_C \cup \langle w_2 \rangle_C$ be the unions of reachable states. Any pair which has at least one *par* state, is trivially parallelizable. Assume par(v_1, v_2) holds. For any pair (x, y) of W, where neither are *par* states, there must be a pair of bisimilar states (x', y') in V with $x' \parallel y'$. By the preceding lemma, $x \sim x' \parallel y'$ implies $x \parallel y'$. Using symmetry (Lemma 5.16), we get $y \sim y' \parallel x$, which implies $x \parallel y$. All pairs in W are compatible; thus par(w_1, w_2) holds. The proof of the reverse implication is symmetrical. □

Lemma 5.29. *par(x, y) if and only if par(y, x)*

Proof. A pair is pairwise parallelizable if all pairs in $\langle x \rangle_C \cup \langle y \rangle_C$ is compatible. Set union is symmetric, so $\langle x \rangle_C \cup \langle y \rangle_C = \langle y \rangle_C \cup \langle x \rangle_C$ □

Any preorder gives rise to an equivalence relation, by defining $x \approx y$ if $x \leq y$ and $y \leq x$. For simulation, this derived equivalence relation is exactly bisimulation. In other words, simulation is anti-symmetric up-to-bisimulation.

Theorem 5.30. *$x \sqsubseteq y$ and $y \sqsubseteq x$ if and only if $x \sim y$*

Proof. Let $R = \{(x, y) \mid x \sqsubseteq y \text{ and } y \sqsubseteq x\}$. By construction, all transition pairs must also be in R. Using the fact that $L_1 \subseteq L_2$ and $L_2 \subseteq L_1$ implies $L_1 = L_2$, we can see that R is a bisimulation.

Most of the definition of a bisimulation directly implies that it is also a simulation. The one non-trivial case is *par* types; let $c(x) = (par, f)$ and $c(y) = (par, g)$. $x \sim y$ implies either

$$\begin{matrix} f(*_1) \sim g(*_1) \\ f(*_2) \sim g(*_2) \end{matrix} \quad or \quad \begin{matrix} f(*_1) \sim g(*_2) \\ f(*_2) \sim g(*_1) \end{matrix}$$

The two preceding lemmata apply in both cases.

$$\text{par}(f(*_1), f(*_2)) \quad \text{if and only if} \quad \text{par}(g(*_1), g(*_2))$$

Thus, $\sim$ is also a simulation. Bisimilarity is symmetric, so we get simulations in both directions. □

Coalgebraic Simulation versus Type Simulation

Gay and Hole defined subtyping through type simulations. Their definition of type simulation is equivalent to a simulation of the coalgebra defined on the set $Type_{GH}$ of types specifically from their grammar.

Definition 5.31. *A relation $R \subseteq Type_{GH} \times Type_{GH}$ is a type simulation if $(t, u) \in R$ implies:*

1. *If $unfold(t) = \hat{}[t_1, ..., t_n]$ then $unfold(u) = \hat{}[u_1, ..., u_n]$ and for all $i \leq n$: t_i R u_i and u_i R t_i.*
2. *If $unfold(t) = ?[t_1, ..., t_n].s_1$ then $unfold(u) = ?[u_1, ..., u_n].s_2$, with s_1 R s_2 and for all $i \leq n$: t_i R u_i.*

3. *If* $unfold(t) = ![t_1, ..., t_n].S_1$ *then* $unfold(u) = ![u_1, ..., u_n].s_2$*, with* $s_1\ R\ s_2$ *and for all* $i \leq n$*:* $u_i\ R\ t_i$*.*

4. *If* $unfold(t) = \ \&\{l_1 : t_1, ..., l_n : t_n\}$ *then* $unfold(u) = \ \&\{l_1 : u_1, ..., l_m : u_m\}$*, with* $m \leq n$ *and for all* $i \leq m$*:* $t_i\ R\ u_i$*.*

5. *If* $unfold(t) = \ \oplus\{l_1 : t_1, ..., l_n : t_n\}$ *then* $unfold(u) = \ \oplus\{l_1 : u_1, ..., l_m : u_m\}$*, with* $n \leq m$ *and for all* $i \leq n$*:* $t_i\ R\ u_i$*.*

6. *If* $unfold(t) = end$ *then* $unfold(u) = end$

Theorem 5.32. *Any relation* $R \subseteq Type_{GH} \times Type_{GH}$ *is a type simulation if and only if* R *is a simulation of the coalgebra* $(Type, c_{Type})$*, under the assumption that* $\leq_D$ *relates all possible pairs* (d, d')

Proof. Notice the definition of a type simulation implies that all pairs have the same operation, polarity and arity, just like coalgebraic simulation. The restrictions on data- and continuation types are also equivalent. The statement $n \leq m$ exactly corresponds to $L_1 \subseteq L_2$, just like $m \leq n$ if and only if $L_2 \subseteq L_1$. The assumption on subtyping of basic types is needed, as the definition of type simulations always allows pairs of basic types to be related. Unrestricted types $t = \frown[t_1, ..., t_n]$ and corresponding $u = \frown[u_1, ..., u_n]$ are more involved: Recall that $c_{Type}(t)$ combines input $?[t_1, ..., t_n].t$ and output $![t_1, ..., t_n].t$. An output can never simulate an input, so t simulates u if and only if $?[t_1, ..., t_n].t \sqsubseteq ?[u_1, ..., u_n].u$ and $![t_1, ..., t_n].t \sqsubseteq ![u_1, ..., u_n].u$. Given that input is covariant and output is contravariant, $t_i \sqsubseteq u_i$ and $u_i \sqsubseteq t_i$ must both hold. □

5.6 Decidability of Coalgebraic Type Relations

In a practical type checker, we need an algorithm for determining whether two types are bisimilar (or dual, or whether one simulates the other). In this subsection we are going to proof there exists an algorithm that computes the answer in finite time for every valid input, i.e., that the problem is decidable.

A state of a coalgebra might be able to reach infinitely many other states, and an algorithm would need to check all of them. Luckily, this is a pathological case. Practical types are seldom irregularly infinite, and regular infinite behaviour can be represented using a finite amount of states, through recursion. We also need to assume subtyping on basic types is decidable.

Definition 5.33. *For any state* x *of a coalgebra* (X, c)*, the* generated coalgebra $\langle x \rangle$ *is the smallest subset of* X *which includes* x *and is closed under transitions.*

Because the generated coalgebra is closed under transitions, we can restrict c to elements of $\langle x \rangle$ and get a function $c_{\langle x \rangle} : \langle x \rangle \to F(\langle x \rangle)$. The set $\langle x \rangle$ and this restricted function $c_{\langle x \rangle}$ are a F-coalgebra themselves.

Definition 5.34. *A finitely generated coalgebra* (X, c) *is a coalgebra such that* $\langle x \rangle$ *is a finite set for all* $x \in X$*.*

Indeed, expressions are finite, so their coalgebra must be finitely generated.

Lemma 5.35. *The coalgebra of types* $(Type, c_{Type})$ *is a finitely generated coalgebra.*

Proof. All elements T of $Type$ are finite strings. Types, except for unrestricted types, only transition to substrings of the unfolded type, and unfolding only replaces variables with the type-string itself. All substrings of a substring v are also substrings of the entire string uv. $\langle T \rangle$ can only contain substrings of T, so it is finite.

In an unrestricted type of the form $T = \widehat{\ }[\widetilde{T}]$, transitions are to input $?[\widetilde{T}].T$ or output $?[\widetilde{T}].T$, both of which can then only transition to substrings. Doubling a finite amount is still finite. Similarly, a type $un\ p$ transitions to $lin\ p$, but nothing else. Hence, $\langle un\ p \rangle = \{un\ p\} \cup \langle lin\ p \rangle$. The union of a single element and a finite set is again, finite. □

The algorithm to determine whether two types x and y are bisimilar creates a relation $R_0 = \{(x, y)\}$ and expands it until it is either a bisimulation or relates a pair which we know not to be bisimilar.

1. If R_i is a postfixpoint of $cf_\sim$, it must be a subset of the bisimilarity relation and all related pairs (including (x, y)) are bisimilar
2. If it is not, there must be some pair (a, b) for which either:
   - Some transitioned pair $(a', b') \notin R_i$: take $R_{i+1} = R_i \cup \{(a', b')\}$ and return to step 1, or
   - Some requirement not related to R was not met, which cannot be fixed and tells us that some required pair (if it wasn't required, it wouldn't have been added to R_i) can't be bisimilar, so x and y are not bisimilar

The algorithm for duality and simulation are the same, but with the functions $cf_\perp$ or $cf_\sqsubseteq$.

Theorem 5.36. *Bisimulation ($x \sim y$) and duality ($x \perp y$) are decidable when $\langle x \rangle$ and $\langle y \rangle$ are finite*

Proof. A relation R is a postfixpoint of $cf_\sim$ if $R \subseteq cf_\sim(R)$. Alternatively, if

$$(x, y) \in R \quad \text{implies} \quad (x, y) \in cf_\sim(R) \text{ for all } x \text{ and } y$$

By breaking down the composition, and using the definition of c^* as the inverse of c, this requirement becomes

$$(x, y) \in R \quad \text{implies} \quad (c(x), c(y)) \in f_\sim(R)$$

Which can be decided by checking all pairs in R have equal operation, polarity, etc, and all proper transition pairs also exist in R, all of which are decidable—either trivially or by the assumption that $\leq_D$ is decidable.

Given that, by assumption, there are only finitely many states to be transitioned to, the algorithm can only add finitely many pairs before reaching a relation that either is a bisimulation or can never be made into a bisimulation.

Duality is done almost the same, except the algorithm checks that polarities are opposite, and data states are bisimilar, which we've just proven to be decidable. □

The algorithm for parallelizability is straightforward. Generate all pairs of elements of $\langle x \rangle$ (or $\langle x \rangle \cup \langle y \rangle$ for pairwise parallelizability), which is guaranteed to be a finite set, and check whether they are compatible.

Theorem 5.37. *Parallelizability* ($par(x)$) *and pairwise parallelizability* ($par(x, y)$) *are decidable when* $\langle x \rangle$ *and* $\langle y \rangle$ *are finite*

Proof. We can infer that any states x and y are compatible ($x \parallel y$) if and only $(c(x), c(y)) \in f_{\parallel}(X)$ from the definition of compatibility. Such a check is decidable, by the same logic that made the bisimilarity checks decideable. There are finitely many states x or y in $\langle x \rangle$, so certainly in its subset $\langle x \rangle_C$; finitely many states lead to a finite amount of pairs; Checking a decidable property finite times is decidable. □

From this, we can expand onto our final decidable property, simulation.

Theorem 5.38. *Simulation* ($x \sqsubseteq y$) *is decidable when* $\langle x \rangle$ *and* $\langle y \rangle$ *are finite*

Proof. The algorithm, and decidability proof, is just like Theorem 5.36, using the fact that parallelizability, subtyping on basic types and the subset relation between finite sets are decidable. □

6 Type Rules

The most important part of a type system, besides the types themselves, is the type rules. The type rules define the semantics of a type by showing which processes are valid, and which are not, when a certain variable has that type.

Variables P, Q will refer to processes of the pi-calculus (as introduced in Section 2.3), x, y, z will range over channels and T, U, V are types, that is, elements of some F-coalgebra (X, c). Variables are associated with types in a context Γ.

$$\begin{aligned} \Gamma ::= \quad & \emptyset \\ | \quad & \Gamma, x : T \end{aligned}$$

A *context* is an unordered, finite set of pairs, that may have at most one pair (x, T) for each variable x. A context is thus isomorphic to a (partial) function from a finite set of variables to their types. We use Γ to denote this isomorphic function as well:

$$\Gamma(x) = T \quad \text{if } (x, T) \in \Gamma$$

The *domain* of a context is defined accordingly

$$dom(\Gamma) = \{x \mid (x, T) \in \Gamma \quad \text{for some } T \in X\}$$

We know *par* types are unrestricted, but they are not the only ones.

Definition 6.1. *A type is* unrestricted*, written* $un(T)$*, if its operation is par, end or type. A context is unrestricted, written* $un(\Gamma)$*, if all types in* Γ *are unrestricted. That is, if* $(x, T) \in \Gamma$ *implies* $un(T)$*. A type is* linear*, written* $lin(T)$*, if it is not unrestricted. A context is linear, if all its types are linear.*

$$\emptyset = \emptyset \circ \emptyset \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \mathrm{un}(T)}{\Gamma, x : T = (\Gamma_1, x : T) \circ (\Gamma_2, x : T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : T = (\Gamma_1, x : T) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : T = \Gamma_1 \circ (\Gamma_2, x : T)}$$

Figure 5: Context Split

Lemma 6.2. *Let T be a type with subtype $U \sqsubseteq T$:*

1. *un(T) if and only if un(U)*
2. *lin(T) if and only if lin(U)*

Proof. The two statements are equivalent, as $\mathrm{lin}(T)$ is defined as $\neg\mathrm{un}(T)$. Whether a type is unrestricted is purely determined by its operation. A subtype always has the same operation as the supertype; consequently, $\mathrm{un}(T)$ if and only if $\mathrm{un}(U)$. □

A context Γ may be split into two parts Γ_1 and Γ_2, such that the linear types are strictly divided between the Γ_1 and Γ_2, but unrestricted types are copied. Context split is a trinary relation, defined by the axioms in Fig. 5. We may write $\Gamma_1 \circ \Gamma_2$ to refer to a context Γ for which $\Gamma = \Gamma_1 \circ \Gamma_2$ is in the context split relation. Such a context does not necessarily exist for a given Γ_1 and Γ_2, so we will assume its existence when writing $\Gamma_1 \circ \Gamma_2$. Notice that the use of $\Gamma, x : T$ in the third rule of Fig. 5 carries the implicit assumption that x not in Γ. Otherwise, $\Gamma, x : T$ would have two pairs with x, which is not allowed in a context. If x not in Γ, then it can not be in Γ_2 either.

The type system is defined by the rules given in Fig. 6. A process P is well-formed, under a specific context Γ, if there is some rule for which the premises (above the line) hold and the conclusion (below the line) is $\Gamma \vdash P$. Generally, such a rule has as a premise that another process is well-formed under a specific context. A complete proof of well-formedness is a tree of these rules. As T-Inact is the only rule that does not depend on another process, it forms the leaves of any inference tree. The type system guarantees, for well-formed processes:

- If the process terminates, then all linear sessions were completed
- If a process reads a value from a channel, the value has the type specified by the channel's type. If a process receives a label, it is one of the labels specified by the channel's type

T-Inact ensures that all linear channels in the context are interacted with until the type becomes unrestricted. If our context contains a variable x of type ?`int`, then the process is required to read an `int` from it:

$$x : ?\texttt{int} \quad \nvdash \quad \mathbf{0}$$

The same context would be correct for the process $x(z).\mathbf{0}$, which reads a value from x and binds it to z.

$$\frac{}{x : ?\texttt{int} \vdash \quad x(z).\mathbf{0}} \text{ T-In}$$

$$\frac{}{x : \mathit{end}, z : \texttt{int} \vdash \quad \mathbf{0}} \text{ T-Inact}$$

$$\frac{\mathrm{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad \frac{\Gamma \vdash P \qquad \mathrm{un}(\Gamma)}{\Gamma \vdash !P} \qquad \text{[T-Inact][T-Rep][T-Par]}$$

$$\frac{\Gamma, x : T, y : U \vdash P \qquad T \perp U}{\Gamma \vdash (\nu xy)P} \qquad \text{[T-Res]}$$

$$\frac{c(T) = (?, n, f) \qquad \Gamma,\, \widetilde{y} : \widetilde{U},\, x : f(*) \vdash P \qquad f(i) \sqsubseteq U_i \qquad \forall i \leq n}{\Gamma, x : T \vdash x(\widetilde{y}).P} \qquad \text{[T-In]}$$

$$\frac{c(T) = (!, n, f) \qquad \Gamma,\, x : f(*) \vdash P \qquad U_i \sqsubseteq f(i) \qquad \forall i \leq n}{\Gamma, x : T, \widetilde{y} : \widetilde{U} \vdash \overline{x}\langle\widetilde{y}\rangle.P} \qquad \text{[T-Out]}$$

$$\frac{c(T) = (\&, L_1, f) \qquad L_1 \subseteq L_2 \qquad \Gamma, x : f(l) \vdash P_l \qquad \forall l \in L_2}{\Gamma, x : T \vdash x \rhd \{l : P_l\}_{l \in L_2}} \qquad \text{[T-Branch]}$$

$$\frac{c(T) = (\oplus, L, f) \qquad \Gamma, x : f(l) \vdash P_l \qquad l \in L}{\Gamma, x : T \vdash x \lhd l.P_l} \qquad \text{[T-Sel]}$$

$$\frac{c(T) = (par, f) \qquad \mathrm{par}(T) \qquad \Gamma, x : f(*_i) \vdash P \qquad i \in \{1, 2\}}{\Gamma, x : T \vdash P} \qquad \text{[T-Unpack]}$$

Figure 6: Type Rules

Because *end* and `int` are both unrestricted types, the last context is unrestricted; thus T-Inact can be applied.

T-Par ensures unrestricted channels are copied and linear channels get distributed between the processes, through context split. Given that the process P is copied in replication $!P$, T-Rep prevents the use of linear channels in P. T-Res creates a channel by binding two covariables x and y, of dual type, together. Note that a scope restriction with one variable $(\nu xx)P$ is allowed, but the type of x would have to be dual to itself, like the bidirectional type $\widehat{\ }[\widetilde{T}]$.

Together, T-Par and T-Rep allow us to introduce new covariables, with new types, and distribute them. But, only unrestricted types may be copied.

$$\begin{aligned} v : \texttt{int} \quad &\vdash \quad (\nu xy)\, x(z).\mathbf{0} \mid \overline{y}\langle v\rangle.\mathbf{0} \\ x : un?\texttt{int} \quad &\vdash \quad x(z).\mathbf{0} \mid x(z).\mathbf{0} \\ x : ?\texttt{int} \quad &\nvdash \quad x(z).\mathbf{0} \mid x(z).\mathbf{0} \end{aligned}$$

Each action on a channel has its own rule: T-In handles input, binding the channel x to the continuation type and variables $\widetilde{y}$ to some supertype of the received types. T-Out handles output, which requires the sent variables to have a subtype of whatever the channel expects to send. T-Branch does external choice, where the process needs to offer at least all choices the type describes, coupled with processes that are correctly typed under the respective continuation types. T-Sel only has to check whether the single label that was chosen by the process was a valid option, and if the rest of the process is correct under the continuation type.

Whereas Vasconcelos' version of T-In, T-Out, T-Branch and T-Select apply to both linear and unrestricted types, our version requires a linear type. This means that they do not directly guarantee that unrestricted types don't change throughout the derivation. T-Unpack allows a *par* type that is parallelizable to be used as if it was one of the underlying types. The combination of these rules has the same guarantees (barring the caveats at the end of Section 5.3), but cleanly separates the semantics of interacting with a channel from those of usable unrestricted types.

We can actually create some interesting structures with *par*, that don't always have a syntactical equivalent. Let T_{end} be a type with $c(T_{end}) = (par, f)$ and $f(*_1) = f(*_2) = T_{end}$. It is a tape that can only reach itself, and a *par* type does not directly describe an action. T_end is just like an *end* type, but it can appear in non-trivial unrestricted types. For example, the type T, with $c(T) = (?, 1, f)$, $f(*) = T_{end}$ and $f(1) = \texttt{int}$:

$$? \xrightarrow{\texttt{int}} T_{end}$$

This type is parallelizable, so we can use it in a process.

$$x : T \quad \vdash \quad x(z).\mathbf{0}$$

It is also unrestricted, so it can be copied and shared between threads.

$$x : T \quad \vdash \quad x(z).\mathbf{0} \mid x(z).\mathbf{0} \mid x(z).\mathbf{0}$$

The same holds for the dual $\overline{T}$

$$x : \overline{T}, z_i : int \quad \vdash \quad \overline{x}\langle z_1\rangle.\mathbf{0} \mid \overline{x}\langle z_2\rangle.\mathbf{0} \mid \overline{x}\langle z_3\rangle.\mathbf{0}$$

Clearly, the channel can hold multiple values; yet, after a process interacts with its copy, it cannot use the channel again.

$$x : T \quad \not\vdash \quad x(z).x(z).\mathbf{0}$$

Such types might be interesting in combination with session delegation. A private session is established by receiving a channel from another shared (unrestricted) channel. By using a structure like T, each thread is guaranteed to establish at most one private session, but there can be many of such sessions in parallel threads.

In Section 5, we defined simulation through the intuition of subtyping. Subtyping is generally meant to describe a type (the subtype) that can used where another (the supertype) was expected. Theorem 6.4 shows that the rule

$$\frac{\Gamma, x : T \vdash P \qquad U \sqsubseteq T}{\Gamma, x : U \vdash P}$$

is admissible. An *admissible* rule is an extra rule that could be added to our set of rules, without changing which processes are accepted and which are not. Essentially, this means that the premises imply the conclusion in the original rules; the extra rule is just there for convenience.

Definition 6.3. *Let* Γ *and* Δ *be two contexts. We say* Δ simulates Γ *if their domains are equal and* $\Delta(x) \sqsubseteq \Gamma(x)$ *for every variable* x *in their domain.*

In other words, Δ simulates Γ if they contain the same variables and any type in Δ is a subtype of that variable's type in Γ.

Theorem 6.4. *Let* Γ *and* Δ *be two contexts, such that* Δ *simulates* Γ*. The judgement* $\Gamma \vdash P$ *implies* $\Delta \vdash P$

Proof. If $\Gamma \vdash P$ there must be a tree of inference rules, with $\Gamma \vdash P$ as the conclusion of the root, for which all premises hold. We will show that this tree can be translated into a valid inference tree for $\Delta \vdash P$, by induction on the structure of that tree.

- T-Inact forms the leaves of any inference tree, and thus the base case. Context Γ only contains unrestricted types. All subtypes of unrestricted types are unrestricted (see Lemma 6.2), so Δ only contains unrestricted types and $\Delta \vdash 0$ holds.
- T-Par Types are not changed in a context split, so when the same split (i.e., $\Delta = \Delta_1 \circ \Delta_2$ such that Δ_i and Γ_i have the same variables) is used, the context Δ_i simulates Γ_i. We can use the induction hypothesis for both premises.
- T-Rep Premise $\Delta \vdash P$ is exactly the hypothesis. $\mathrm{un}(\Delta)$ follows from the same reasoning as for T-Inact.
- T-Res We can choose to introduce the same types in the translated tree. The context $\Delta, x : T, y : U$ simulates $\Gamma, x : T, y : U$, so the hypothesis applies.
- T-In is more complicated. Our goal is to show all premises hold for $\Delta = \Delta_1, x : T'$

$$c(T') = (?, n, f') \qquad \Delta_1, x : f'(*), \widetilde{y} : \widetilde{U} \vdash P \qquad f'(i) \sqsubseteq U_i \quad \text{for all } i \leq n$$

We know they hold for the context $\Gamma = \Gamma_1, x : T$ in the original tree

$$c(T) = (?, n, f) \qquad \Gamma_1, x : f(*), \widetilde{y} : \widetilde{U} \vdash P \qquad f(i) \sqsubseteq U_i \quad \text{for all } i \leq n$$

We also know that type T' is a subtype of T. Therefore,

$$c(T') = (?, n, f') \qquad f'(*) \sqsubseteq f(*) \qquad f'(i) \sqsubseteq f(i) \quad \text{for all } i \leq n$$

Which tells us, by transitivity, that $f'(i) \sqsubseteq U_i$ for all $i \leq n$. By assumption, Δ simulates Γ; variable x was removed from both, so Δ_1 simulates Γ_1. Because $f'(*)$ is a subtype $f(*)$, context $\Delta_1, x : f'(*), \widetilde{y} : \widetilde{U}$ simulates $\Gamma_1, x : f(*), \widetilde{y} : \widetilde{U}$. The latter is defined, so neither x nor $\widetilde{y}$ are in Γ_1; By simulation, they cannot be in Δ_1, so the translated context is also defined. The final premise

$$\Delta_1, x : f'(*), \widetilde{y} : \widetilde{U} \vdash P$$

follows by induction, so all premises hold.

- T-Out The same argument as T-In, except that output is contravariant. For the same functions f and f', state $f(i)$ simulates $f'(i)$, instead of the other way around. Even so, the related premise $U_i \sqsubseteq f'(i)$ is also reversed, so the conclusion still stands.

- T-Branch The premises we would like to prove, for $\Delta = \Delta_1, x : T'$, are

$$c(T') = (\&, L_3, f') \qquad L_3 \subseteq L_2 \qquad \Delta_1, x : f'(l) \vdash P_l \quad \text{for all } l \in L_3$$

The original inference tree tells us, for $\Gamma = \Gamma_1, x : T$

$$c(T) = (\&, L_1, f) \qquad L_1 \subseteq L_2 \qquad \Gamma_1, x : f(l) \vdash P_l \quad \text{for all } l \in L_1$$

Furthermore, because $T' \sqsubseteq T$

$$c(T') = (\&, L_3, f') \qquad L_3 \subseteq L_1 \qquad f'(l) \sqsubseteq f(l) \quad \text{for all } l \in L_3$$

The premise $L_3 \subseteq L_2$ is a simply consequence of transitivity. Because L_3 is a subset of L_1, anything that holds for all elements in L_1 must hold for all elements in L_3.

$$\Gamma_1, x : f(l) \vdash P_l \quad \text{for all } l \in L_3$$

By the original context simulation, context $\Delta_1, f'(l)$ simulates $\Gamma_1, f(l)$ for all $l \in L_3$. Combined with the hypothesis, this implies

$$\Delta_1, x : f'(l) \vdash P_l \quad \text{for all } l \in L_3$$

Which was the last unproven premise; all premises of the translated rule hold.

- T-Sel This time our goal is to show, for $\Delta = \Delta_1, x : T'$

$$c(T') = (\oplus, L_2, f') \qquad \Delta_1, x : f'(l) \vdash P_l \qquad l \in L_2$$

given that, for $\Gamma = \Gamma_1, x : T$

$$c(T) = (\oplus, L_1, f) \qquad \Gamma_1, x : f(l) \vdash P_l \qquad l \in L_1$$

The simulation tells us

$$c(T') = (\oplus, L_2, f') \qquad L_1 \subseteq L_2 \qquad f'(l) \sqsubseteq f(l) \quad \text{for all } l \in L_1$$

L_1 is a subset of L_2, so $l \in L_1$ implies $l \in L_2$. The second premise follows from the hypothesis, as the translated context $\Delta_1, x : f'(l)$ simulates the original $\Gamma_1, x : f(l)$.

- T-Unpack We want to proof, for j in $\{1, 2\}$ and $\Delta = \Delta_1, x : T'$

$$c(T') = (par, f') \qquad \text{par}(T') \qquad \Delta_1, x : f'(*_j) \vdash P$$

given i in $\{1, 2\}$ and $\Gamma = \Gamma_1, x : T$

$$c(T) = (par, f) \qquad \text{par}(T) \qquad \Gamma_1, x : f(*_i) \vdash P$$

Simulation tells us $c(T') = (par, f')$ and that $f(*_i)$ is simulated either $f'(*_1)$ or $f'(*_2)$. Let j be such that

$$f'(*_j) \sqsubseteq f(*_i)$$

The parallelizability of $f'(*_j)$ is given by Lemma 5.25. The rest is straightforward application of the inductive hypothesis.

We have shown the inductive hypothesis to be valid for all rules of the inference tree, including the base case, so the hypothesis holds. □

From this, we can conclude that bisimulation implies type-equivalence.

Corollary 6.5. *For all types $T \sim U$, contexts Γ and processes P, it holds that $\Gamma, x : T \vdash P$ if and only if $\Gamma, x : U \vdash P$;*

Proof. By Lemma 5.30, bisimulation implies simulation in both directions ($T \sqsubseteq U$ and $U \sqsubseteq T$). The double implication then follows from Theorem 6.4. □

7 Algorithmic Type Checking

The type rules describe what well-formed processes look like, but do not directly give us a type checking algorithm. This is because, beforehand, we don't know:

1. Which type to introduce in reading (T-In) or scope restriction (T-Res)
2. How to split the context in T-Par
3. Which underlying type to use in T-Unpack

Rather than try to infer the introduced types, we augment the language of processes with type annotations. As long as we know one of the types introduced in scope restriction, we can create the other with the duality function, so the grammar of annotated processes is:

$$\begin{array}{rcl} P & ::= & \ldots \\ & | & (\nu xy : T)\, P \\ & | & x(y : T).P \end{array}$$

$$\Gamma \div \emptyset = \Gamma \qquad \frac{\Gamma_1 \div F = \Gamma_2, x : T \qquad \mathrm{un}(T)}{\Gamma_1 \div (F, x) = \Gamma_2} \qquad \frac{\Gamma_1 \div F = \Gamma_2 \qquad x \notin dom(\Gamma_2)}{\Gamma_1 \div (F, x) = \Gamma_2}$$

Figure 7: Context Difference

Of course, all other productions remain the same.

When checking parallel processes, we pass along the entire context to the first processes, keeping track of all linear variables used, and remove those from the context given to the second process.

The third problem, of unpacking unrestricted types, is addressed by assuming types are normalized. How this is done, and what normalized types are, will be covered after introducing the algorithmic version of the other rules.

The input for the algorithm is still a context Γ_1 and a process P, but it also outputs a different context Γ_2 and the set F of free, linear, variables in P. That is, free variables whose type in Γ_1 was linear.

$$\Gamma_1 \vdash P \ : \ \Gamma_2 \ ; \ F$$

The output context Γ_2 keeps track of the change in types of linear variables and should allow us to construct the input for potential parallel processes. This means any unrestricted variables, and linear variables not used in P, should have the same type in Γ_1 as in Γ_2. Furthermore, Γ_2 being unrestricted should imply that $\Gamma_1 \vdash P$ holds.

The context difference operator $\div$, which is described in Fig. 7, is used to remove variables of unrestricted type from a context; $\Gamma \div x$ is the context of all variable/type pairs in Γ minus a potential pair including x, but is only defined if x had an unrestricted type. The rules of the algorithm are given in Fig. 8.

A-Inact checks the completed process $\mathbf{0}$, but it does not depend on Γ. This means that

$$x : ?int \vdash \mathbf{0} : (x : ?\mathtt{int}); \emptyset$$

holds, even though $x : ?int \nvdash \mathbf{0}$. We will use the output context to tell whether a process was correct.

A-Par checks parallel processes in sequence, passing the entire context to the first process and removing the linear variables used from the context passed to the second process. A replicated process may not use linear variables, so A-Rep mandates that F is empty.

The rest of the rules are straightforward adoptions of their type rules. A-Res and A-Out describe the two situations where variables are bound, necessitating type annotations. Bound variables are also invisible to earlier parts of the process, so they are removed from the returned context and set of variables. Context difference ensures they are unrestricted at that point, otherwise the process would not be well-formed. The basic actions (A-In, A-Out, A-Branch and A-Sel) use a channel, so they add it to the set F.

Take, for example, the process

$$x : ?\mathtt{int}, \ y : ?\mathtt{int} \quad \vdash \quad x(z_1).\mathbf{0} \mid y(z_2).\mathbf{0}$$

The variables are split correctly, and both split contexts are unrestricted when the process is completed, thus its well-formed. An algorithmic derivation is

[A-Inact][A-Rep]

$$\Gamma \vdash \mathbf{0} : \Gamma; \emptyset \qquad \frac{\Gamma_1 \vdash P : \Gamma_2; F \qquad F = \emptyset}{\Gamma_1 \vdash\, !P : \Gamma_2; \emptyset}$$

[A-Par]

$$\frac{\Gamma \vdash P \ : \ \Gamma_1 \ ; \ F_1 \qquad \Gamma_1 \div F_1 \vdash Q \ : \ \Gamma_2 \ ; \ F_2}{\Gamma \vdash P \mid Q \ : \ \Gamma_2 \ ; \ F_1 \cup F_2}$$

[A-Res]

$$\frac{\Gamma_1, x : T, y : \overline{T} \vdash P : \Gamma_2; F}{\Gamma_1 \vdash (\nu xy : T)P \ ; \ \Gamma_2 \div (x, y) \ : \ F - \{x, y\}}$$

[A-In]

$$\frac{c(T) = (?, n, f) \qquad f(i) \sqsubseteq U_i \qquad \Gamma_1,\, \widetilde{y} : \widetilde{U},\, x : f(*) \vdash P \ : \ \Gamma_2 \ ; \ F \qquad \forall i \leq n}{\Gamma_1, x : T \vdash x(\widetilde{y} : \widetilde{U}).P \ : \ \Gamma_2 \div \widetilde{y} \ ; \ (F \cup \{x\}) - \widetilde{y}}$$

[A-Out]

$$\frac{c(T) = (!, n, f) \qquad U_i \sqsubseteq f(i) \qquad \Gamma_1,\, x : f(*) \vdash P \ : \ \Gamma_2 \ ; \ F \qquad \forall i \leq n}{\Gamma_1, x : T, \widetilde{y} : \widetilde{U} \vdash \overline{x}\langle\widetilde{y}\rangle.P \ : \ \Gamma_2 \ ; \ F \cup \{x\}}$$

[A-Branch]

$$\frac{c(T) = (\&, L_1, f) \qquad L_1 \subseteq L_2 \qquad \Gamma_1, x : f(l) \vdash P_l \ : \ \Gamma_2 \ ; \ F \qquad \forall l \in L_1}{\Gamma_1, x : T \vdash x \rhd \{l : P_l\}_{l \in L_2} \ : \ \Gamma_2 \ ; \ F \cup \{x\}}$$

[A-Sel]

$$\frac{c(T) = (\oplus, L, f) \qquad \Gamma_1, x : f(l) \vdash P_l \ : \ \Gamma_2 \ ; \ F \qquad l \in L}{\Gamma_1, x : T \vdash x \lhd l.P_l \ : \ \Gamma_2 \ ; \ F \cup \{x\}}$$

[A-Unpack]

$$\frac{c(T) = (par, f) \qquad \mathrm{par}(T) \qquad \Gamma_1, x : next(T, P) \vdash P : \Gamma_2; F}{\Gamma_1, x : T \vdash P \ : \ (\Gamma_2 \div x), x : T \ ; \ F - x}$$

Figure 8: Algorithmic Type Checking Rules

$$x : ?\texttt{int},\ y : ?\texttt{int} \ \vdash\ x(z_1).\mathbf{0} \mid y(z_2).\mathbf{0} \ : (y : end) \ ; \{y\}$$

$$x : ?\texttt{int},\ y : ?\texttt{int} \ \vdash\ x(z_1).\mathbf{0} \ : (x : end, y : ?\texttt{int}) \ ; \{x\}$$
$$\Gamma_1 = x : end,\ y : ?\texttt{int},\ z_1 : \texttt{int} \ \vdash\ \mathbf{0} \ : (x : end,\ y : ?\texttt{int},\ z_1 : \texttt{int}) \ ; \emptyset$$

$$y : ?\texttt{int} \ \vdash\ y(z_2).\mathbf{0} \ : (y : end) \ ; \{y\}$$
$$y : \ end,\ z_2 : \texttt{int} \ \vdash\ \mathbf{0} \ : (y : \ end,\ z_2 : \texttt{int}) \ ; \emptyset$$

Because the entire context was passed to the first process, Γ_1 is not unrestricted. Naively copying the unrestricted constraint from T-Inact to A-Inact would have caused the algorithm to wrongly reject this process.

On the other hand, if the left process did not complete the linear session, then the context difference would not have been defined. Take one such process:

$$x : ?\texttt{int}.?\texttt{int},\ y : ?\texttt{int} \quad \not\vdash \quad x(z_1).\mathbf{0} \mid y(z_2).\mathbf{0}$$

The result of type checking the left part is

$$x : ?\texttt{int}.?\texttt{int},\ y : ?\texttt{int} \quad \vdash \quad x(z_1).\mathbf{0} \ : \quad (x : ?\texttt{int}, y : ?\texttt{int}) \quad ; \{x\}$$

Because x was used, the context passed to $y(z_2).\mathbf{0}$ is

$$(x : ?\texttt{int}, y : ?\texttt{int}) \div \{x\}$$

But, x has a linear type, so the context difference is undefined. This breaks the implicit assumption, so the algorithm rejects this input entirely. The process was indeed not well-formed, and no further parallel processes could fix it; the rejection is expected.

If the variable is unrestricted, A-Unpack first unpacks it into $next(T, P)$. This is a (partial) function from an unrestricted type and the process being type checked to one of the underlying types. This function will be defined in the next subsection.

Although x could be free in P, it obviously does not have a linear type in Γ_1, so we don't return it. We also need to pass along the original type in the output context for any further parallel processes. At the same time, we need to make sure that a type that is unpacked is actually interacted with until it becomes unrestricted again; hence we use context difference.

To illustrate why this is needed, let x be a variable of type

$$T = \mu X.\mathit{un}?\texttt{int}.!\texttt{int}.X$$

Type T is unrestricted, and parallelizable, but its continuation $!\texttt{int}.T$ is linear. Any process reading an $\texttt{int}$ from x must at some later point write another $\texttt{int}$, it cannot just read and immediately terminate. A simple process using x wrongly would be

$$x : T \quad \not\vdash \quad x(z).\mathbf{0}$$

The algorithm applied to the unpacked type yields

$$x : ?\texttt{int}.!\texttt{int}.T \ \vdash\ x(z).\mathbf{0} \ : (x : !\texttt{int}.T) \ ; x$$
$$x : !\texttt{int}.T \ \vdash\ \mathbf{0} \ : (x : !\texttt{int}.T) \ ; x$$

The type of x became linear after unpacking, and was still linear at the final A-Inact application. This tells us the process was not well-formed and cannot be fixed by parallel processes, so should be rejected. Indeed, the context difference

$$(x : !\texttt{int}.T) \div \{x\}$$

is not defined, breaking the implicit assumption in A-Unpack.

7.1 Normalization and Unpacking

The structural approach to unrestricted types is powerful, and easy to reason about, but also allows for a lot of complex, deeply nested, structures. At the same time, parallelizability heavily limits which types are actually viable.

When compatible types (directly describing an action) can only differ in polarity, and there are only two polarities, there are at most two compatible actions. Compatible states with the same action can still have different continuations, but we can delay choosing which path to follow after said action. A single *par* type can wrap both actions; thus there is no need to have a *par* type transition directly to another *par* type. A notable exception is the type that describes no actions at all, we choose to represent that case by a type transitioning only to itself.

Definition 7.1. *A type T is* normalized *if $c(T) = (par, f)$ implies:*

- *If $(c \circ f)(*_1) = (par, g)$, then $f(*_1) \sim f(*_2) \sim T$*
- *Otherwise, either*
  - *$f(*_1) \sim f(*_2)$, or*
  - *If $(pol \circ f)(*_1) = p$, then $(pol \circ f)(*_2) = \overline{p}$*

A coalgebra (X, c) is normalized if all types $x \in X$ are normalized.

Our motivation gives an informal description of an algorithm for normalization. Luckily, most types are already normalized.

Lemma 7.2. *The coalgebra of types $(Type, c_T)$ is normalized*

Proof. The only non-trivial cases are unrestricted types $T_1 = un\ p$ and $T_2 = \widehat{\ }[\widetilde{T}]$. Transitions of the first are defined as $\delta(T_1)(*_1) = \delta(T_1)(*_2) = lin\ p$. A linear type is not *par*, and equal types are always bisimilar. The second type gives $\delta(T_2)(*_1) = ?[\widetilde{T}].T_2$ and $\delta(T_2)(*_2) = ![\widetilde{T}].T_2$. Reading and writing are both linear and they have opposite polarity. All types $un\ p$ and $\widehat{\ }[\widetilde{T}]$ are normalized. □

Assuming a normalized type T, $next(T, P)$ is defined by Fig. 9. Remember that $\text{pol} : X \to P$ is a part of every coalgebra, mapping each state to either *in* or *out*. We use the process in question to determine which polarity it needs. For example, if a process does a read, it should get a type that describes an input. Normalization tells us that types are either bisimilar, or that they have different polarities. Technically, this means $next(T, P)$ is not entirely deterministic. But, as bisimilar types are equivalent for the type system, it does not matter.

$$\frac{(\text{pol} \circ f)(*_i) = in \qquad P = x(\widetilde{y} : \widetilde{U}).Q}{next(T, P) = f(*_i)} \qquad \frac{(\text{pol} \circ f)(*_i) = in \qquad P = x \rhd \{l : P_l\}_{l \in L}}{next(T, P) = f(*_i)}$$

$$\frac{(\text{pol} \circ f)(*_i) = out \qquad P = \overline{x}\langle \widetilde{y} \rangle.Q}{next(f, P) = f(*_i)} \qquad \frac{(\text{pol} \circ f)(*_i) = out \qquad P = x \lhd l.P_l}{next(f, P) = f(*_i)}$$

Figure 9: Algorithmic Unpacking; let $i \in 1, 2$ and $f = \delta(T)$

Lemma 7.3. *The result of $next(T, P)$, when defined, is unique up to bisimilarity, for any normalized type T and process P.*

Proof. Let $T_1 = \delta(T)(*_1)$ and $T_2 = \delta(T)(*_2)$, $next(T, P)$ must be either of these. If $T_1 \sim T_2$, then it doesn't matter which we return, both options are bisimilar. If they are not bisimilar, the process directs which rule we can use. This rule requires the returned type to have a specific polarity, normalization tells us that T_1 and T_2 cannot have the same polarity, so only one can be $next(T, P)$. □

A crucial property of *next* is that it will not return another *par* type. The only normalized type with another underlying par type, is the type that only transitions to itself. This type does not describe any action, so we do not want to unpack it.

Lemma 7.4. *Let T be any normalized type and let P be a process, $T_n = next(T, P)$ implies $c(T_n) \neq (par, g)$ for any g.*

Proof. Polarity is only defined on *com* and *branch* states, and all rules implicitly require it to be defined. □

Theorem 7.5. *The type checking algorithm terminates in finite time for every input, assuming a finitely generated, normalized coalgebra.*

Proof. The input of the algorithm is a finite context Γ and a process, a finite expression, P. Just like a proof of well-formedness is a tree of type rules, an execution of the algorithm is a tree of algorithmic rules. For any non-Unpack node in the tree, the rule removes some element from the process(es) to be recursively type checked. The process of any such node is thus strictly larger than the concatenation of all its childrens' processes. Because a process is a finite expression, one can only remove finitely many elements; hence, there can only be finitely many of these non-Unpack nodes in the tree.

For example, T-Par checks a process $P \mid Q$. Its children check P and Q, and $length(PQ) < length(P \mid Q)$, in terms of their string concatenation.

A-Unpack does not change the process, but, because of Lemma 7.4, we can only use A-Unpack once per variable, before we need another, non-Unpack nodes to make its type unrestricted again. There are finitely many variables in a context and finitely many non-Unpack nodes in the tree, so there are finitely many A-Unpack nodes as well.

All of the non-recursive premises are decidable (see Lemma 5.38 for a decidability proof for simulation). As such, a finite tree corresponds to an execution that finishes in finite time. □

7.2 Algorithmic Soundness

Algorithmic type checking should allow us to determine whether a process is well-formed according to the type rules. The previous section showed that the algorithmic rules do not reject every process with unused linear variables. Thus, we add the proviso that the returned context must be unrestricted. Our type system and algorithm was inspired by [Vas12]. The theorem and lemmata presented here follow a similar structure to the algorithmic correctness proof of that work, but our rules and representation of types are different. Here too, we will be assuming types are normalized and finitely generated.

In the process of creating the algorithm, we modified the language of processes to include type annotations. To go back to the language our type rules are defined on, we can erase those annotations. Anything but scope restriction and input is unchanged by type erasure.

$$\begin{aligned} erase((\nu xy : T).Q) &= (\nu xy).erase(Q) \\ erase(x(\widetilde{y} : \widetilde{T}).Q) &= x(\widetilde{y}).erase(Q) \\ erase(P \mid Q) &= erase(P) \mid erase(Q) \\ erase(\overline{x}\langle\widetilde{y}\rangle.Q) &= \overline{x}\langle\widetilde{y}\rangle.erase(Q) \\ &\dots \end{aligned}$$

We would like to show our algorithm is complete, that it will accept every correct process. Sadly, it is not. There is a slight edge case where linear *branch* types end in a different unrestricted type

$$T = \& \begin{cases} a : \mu X.un?\texttt{int}.X \\ b : \mu X.un?\texttt{bool}.X \end{cases}$$

Both continuations are unrestricted, so $x : T \vdash x \triangleright \{a : \mathbf{0}, b : \mathbf{0}\}$. A-Branch requires all branches to return the same context. Yet, in this case, $\Gamma_2(x)$ will be $\mu X.un?\texttt{int}.X$ or $\mu X.un?\texttt{bool}.X$, depending on which branch was checked.

Although incomplete, the algorithm is not useless, for it is still sound. That is, if the algorithm accepts a process, it is guaranteed to be well-formed.

Theorem 7.6. $\Gamma_1 \vdash P : \Gamma_2 \,;\, F$ *and* $un(\Gamma_2)$ *implies* $\Gamma_1 \vdash erase(P)$

The rest of the current section will be proving this theorem. Let us begin by stating exactly what the algorithm outputs. Let $\Gamma \setminus F$ be the biggest subset of Γ without (x, T) for any x in F. That is, just like the context difference, but defined even if the removed types are linear. Let $\mathcal{L}(\Gamma)$ be the biggest subset of Γ with $\text{lin}(\mathcal{L}(\Gamma))$, i.e., including only linear types.

Lemma 7.7. *Properties of context difference Let* $\Gamma \div F = \Delta$

- $\Delta = \Gamma \setminus F$
- $\mathcal{L}(\Gamma) = \mathcal{L}(\Delta)$
- *If* $x : T \in \Gamma$ *and* $x \in L$, *then* $un(T)$ *and* $x \notin dom(\Delta)$

Proof. Our definition of a context, and context difference, is the same as in [Vas12]. This lemma is Lemma 8.1 of that work.[3] □

The above lemma lets us derive other, interesting properties, such as $F \cap dom(\mathcal{L}(\Gamma)) = \emptyset$, $(\Gamma \div F_1) \setminus F_2 = (\Gamma \setminus F_2) \div F_1$ and $(\Gamma, x : T) \div F = \Delta, x : T$ when $x \notin F$.

We define $\mathcal{U}(\Gamma)$ as the biggest subset of Γ containing only unrestricted types. Clearly, $\Gamma = \mathcal{L}(\Gamma) \cup \mathcal{U}(\Gamma)$.

Lemma 7.8 (Algorithmic monotonicity)**.** *If* $\Gamma_1 \vdash P : \Gamma_2; F$*, then*

1. $F \subseteq dom(\Gamma_1)$
2. $\Gamma_2 \setminus F \subseteq \Gamma_1$*, and*
3. $\mathcal{U}(\Gamma_2) \setminus F = \mathcal{U}(\Gamma_1)$

Proof. The proof is an induction on the structure of the execution tree, using the properties of context difference. We illustrate the case for A-Unpack. Suppose $\Gamma_1 = \Delta_1, x : T$, $\Gamma_2 = (\Delta_2 \div x), x : T$, and

$$\Delta_1, x : next(T, P) \vdash P : \Delta_2 \; ; \; F$$

By induction,

$$\begin{aligned} &F \subseteq dom(\Delta_1) \cup \{x\} \\ &\Delta_2 \setminus F \subseteq \Delta_1, x : next(T, P) \\ &\mathcal{U}(\Delta_2) \setminus F = \mathcal{U}(\Delta_1, x : next(T, P)) \end{aligned}$$

Now, $dom(\Gamma_1) = dom(\Delta_1) \cup \{x\}$, so $F \subseteq dom(\Gamma_1)$. Consequently, $F - x \subseteq dom(\Gamma_1)$. To proof the second, we use $\Gamma_1(x) = \Gamma_2(x) = T$. Clearly, $(\Delta_2 \div x) \setminus F \subseteq \Delta_1$. Because we add the same pair $x : T$ to both contexts, $[(\Delta_2 \div x), x : T] \setminus F' \subseteq \Delta_1, x : T$ still holds for $F' = F - x$. Finally, type T is definitely unrestricted, so $x : T \in \mathcal{U}(\Gamma_1)$. Using the fact that the same type was added to both contexts again, $\mathcal{U}[(\Delta_2 \div x), x : T] \setminus (F - x) = \mathcal{U}(\Delta_1, x : T)$. The other cases are proven similarly. □

In our proof of soundness, we need algorithmic linear strengthening. In A-Par the entire context is passed along to the first process, but the type rules require a strict split of linear variables. Linear variables that are not part of F (thus, not referenced in the process) should be safe to remove from the context used for typing that process.

Lemma 7.9 (Algorithmic linear strengthening)**.** *If* $\Gamma_1 \vdash P : \Gamma_2, x : T \; ; \; F$*, with* $lin(T)$ *and* $x \notin F$*, then* $\Gamma_1 = \Gamma_3, x : T$ *and* $\Gamma_3 \vdash P : \Gamma_2 \; ; \; F$

Proof. The first consequent, $\Gamma_1(x) = \Gamma_2(x)$, follows directly from monotonicity. The rest requires an inductive analysis on the structure of the execution tree. Let us detail two cases, the rest are done in a similar fashion.

When the root of the execution tree is A-Par, suppose that

$$\Gamma, x : T \vdash P \mid Q : (\Gamma_2, x : T) \; ; \; F_1 \cup F_2$$

[3] Note that where we use F for a set of free, linear variables, the referenced paper uses L

Which tells us, by premise of the rule

$$\Gamma, x : T \vdash P : (\Gamma_1, x : U) \; ; \; F_1 \qquad \text{and} \qquad (\Gamma_1, x : U) \div F_1 \vdash Q : (\Gamma_2, x : T) \; ; \; F_2$$

Because $x \notin F_1 \cup F_2$, it is not in one of the constituent sets either. By the first consequent of the current lemma, $T = U$. Induction on P tells us $\Gamma \vdash P : \Gamma_1 \; ; \; F$ (1). We can also infer from $x \notin F_1$ that $(\Gamma, x : T) \div F_1 = (\Gamma \div F_1), x : T$; hence, by induction on Q, we know $\Gamma_1 \div F_1 \vdash Q : \Gamma_2 \; ; \; F_2$. Combined with (1), this gives the desired result.

When the root is A-In, suppose

$$\Gamma_1, x : T \vdash z(\widetilde{y} : \widetilde{U}).P : \Gamma_2, x : T \; ; \; (F \cup x) - \widetilde{y}$$

The rule specifies that $\Gamma_2, x : T = (\Gamma_3, x : T) \div \widetilde{y}$, for some Γ_3, so x must be distinct from every y_i. As $x \notin (F \cup \{z\}) - \widetilde{y}$, we also know $z \neq x$. The premise of the rule tells us

$$\Gamma_1, x : T, \widetilde{y} : \widetilde{U} \vdash P : \Gamma_3, x : T \; ; \; F$$

The inductive hypothesis implies

$$\Gamma_1, \widetilde{y} : \widetilde{U} \vdash P : \Gamma_3 \; ; \; F$$

From $x \neq y_i$, we can infer $\Gamma_3 \div \widetilde{y} = \Gamma_2$; the desired result follows directly.

$$\Gamma_1 \vdash z(\widetilde{y} : \widetilde{U}).P : \Gamma_2 \; ; \; (F \cup x) - \widetilde{y}$$

□

Finally, we can proof the leading theorem, that $\Gamma_1 \vdash P : \Gamma_2 \; ; \; F$ and $\text{un}(\Gamma_2)$ implies $\Gamma \vdash \mathit{erase}(P)$.

Proof. Cases other than A-Par are proven with a straightforward induction; let us illustrate the procedure with A-Branch. Let $c(T) = (\&, L_1, f)$ and suppose that

$$\Gamma_1, x : T \vdash x \rhd \{l : P_l\}_{l \in L_2} : \Gamma_2 \; ; \; F \cup \{x\}$$

The premise says that $\Gamma_1, x : f(l) \vdash P_l : \Gamma_2 \; ; \; F$ for every $l \in L_1$. By induction, $\Gamma_1, x : \delta(T)(l) \vdash \mathit{erase}(P_l)$ for the same $l \in L_1$. The result is directly implied by T-Branch.

In the case of A-Unpack, suppose that $\Gamma_1, x : T \vdash P : (\Gamma_2 \div x), x : T \; ; \; F - x$. Context difference only removes unrestricted types, so Γ_2 is unrestricted because $(\Gamma_2 \div x), x : T$ is. By induction, $\Gamma_1, x : \mathit{next}(T, P) \vdash \mathit{erase}(P)$. The result of $\mathit{next}(T, P)$ is guaranteed to be $f(*_1)$ or $f(*_2)$. The desired result, $\Gamma_1, x : T \vdash \mathit{erase}(P)$, follows directly.

Let us elaborate on A-Par. Suppose that $\Gamma \vdash P \mid Q : \Gamma_2 \; ; \; F_1 \cup F_2$. We know that both processes are accepted, as $\Gamma \vdash P : \Gamma_1 \; ; \; F_1$ and $\Gamma_1 \div F_1 \vdash Q : \Gamma_2 \; ; \; F_2$. Due to the properties of context difference, $F_1 \cap \mathit{dom}(\mathcal{L}(\Gamma_1)) = \emptyset$. Otherwise, $\Gamma_1 \div F_1$ would not be defined. Let $\Gamma = \Delta_1, \mathcal{L}(\Gamma_1)$; all elements $x \in \mathcal{L}(\Gamma_1)$ are obviously linear and $x \notin F$. We can apply strengthening to P, resulting in $\Delta_1 \vdash P : \mathcal{U}(\Gamma_1) \; ; \; F$. The returned $\mathcal{U}(\Gamma_1)$ is trivially unrestricted, so induction yields $\Delta_1 \vdash \mathit{erase}(P)$. Let $\Delta_2 = \Gamma_1 \div F_1$; induction on Q tells us $\Delta_2 \vdash \mathit{erase}(Q)$. Finally, we need to show that $\Gamma = \Delta_1 \circ \Delta_2$. Due to monotonocity, $\mathcal{U}(\Gamma_1) \setminus F_1 = \mathcal{U}(\Gamma)$. We know $\Gamma_1 \div F_1$ is defined as Δ_1, so $\mathcal{U}(\Delta_1) = \mathcal{U}(\Gamma)$. From the construction of $\Gamma = \Delta_1, \mathcal{L}(\Gamma_1)$, we can infer $\mathcal{U}(\Delta_1) = \mathcal{U}(\Gamma) = \mathcal{U}(\Delta_2)$. Also, we know Δ_1 has all of the linear variables of Γ not in Γ_1, whereas $\mathcal{L}(\Delta_2) = \mathcal{L}(\Gamma_1)$. Wrapping up, the two contexts have the same unrestricted variables as themselves, and as Γ, and their linear variables partition the linear variables of Γ, so $\Delta_1 \circ \Delta_2 = \Gamma$. We can conclude $\Gamma \vdash \mathit{erase}(P \mid Q)$

□

8 Conclusions and Further Research

The goal of this thesis was to define a session type system, independent of syntax, such that any type system could get type equivalence and subtyping as an instance of our generic system.

The coalgebraic approach proposed does indeed abstract away from the syntax of types, and we've shown how a coalgebra can be defined on expressions of the two different systems presented in [GH05] and [Vas12].

Semantically, however, our system is fairly rigid. It assumes synchronization signals as a language primitive and provides no way to check more advanced properties like deadlock freedom. Label Dependent Session Types, as introduced in [TV19], gets rid of the first assumption by using dependent typing, it could be interesting to explore a dependently typed version of our coalgebraic system. Eliminating deadlocks seems impossible when reasoning purely about binary session types, so further work would be needed to adopt the functor to multiparty session types.

Linear and unrestricted types are currently very strictly separated. A channel end may either never be copied, or copied infinitely many times. It could be interesting to explore a version of our system which can express that a channel may be a copied a given, finite, amount of times.

Ultimately, we would like to treat processes as coalgebras as well. Given the similarities between session types and processes, such a functor for processes could be the same or very similar to the functor for types. The question then becomes how to represent the context mapping names to types, but it might be possible to represent well-formedness of processes as some form of simulation.

References

[Awo10] Steve Awodey. *Category Theory, 2nd ed.* Oxford University Press, June 2010.

[Bar84] H. P. Barendregt. The lambda calculus. its syntax and semantics. *Studies in logic and foundations of mathematics*, 103, 1984.

[GH05] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf*, 42(2/3):191–225, 2005.

[GK09] Nicola Gambino and Joachim Kock. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society*, 154, 06 2009.

[GTV20] Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. Duality of session types: The final cut. *Electronic Proceedings in Theoretical Computer Science*, 314:23–33, Apr 2020.

[H+16] H. Hüttel et al. Foundations of session types and behavioural contracts. *ACM Computing Surveys*, 49(1), April 2016.

[Rut19] Jan Rutten. *The Method of Coalgebra: exercises in coinduction.* CWI, Amsterdam, February 2019. *Available at* http://persistent-identifier.org/?identifier=urn:nbn:nl:ui:18-28550.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[TV19] Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[Vas12] Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.