universität
innsbruck

Technical Report

From Induction to Coinduction

Andreas Forster, Hanna Köb (01518442, 01516395)
andreas.forster@student.uibk.ac.at, hanna.koeb@student.uibk.ac.at

28 February 2018

Supervisor: Vincent van Oostrom

Abstract

We explain the relation between induction and coinduction, the coinductively defined proof technique bisimulation and the duality between induction and coinduction through lattice theory and the concept of greatest and least fixed points.

Contents

1 Introduction

This report gives an overview as well as detailed information regarding induction and coinduction, especially their connection with one another. Furthermore, we explain bisimulation in concurrency, which makes use of the coinductive principle. While coinduction is still rather new to us, it is a continuously growing part of Computer Science, specifically Computational Logic. Due to working with potentially infinite processes becoming increasingly more common, it has become more and more important to study this particular concept, in order to be able to properly deal with infinite data and structures. Another defining trait of coinduction is its duality to induction.

Bisimulation, the technique to prove process equality, as well as definitions and explanations as to what a process mathematically is, are the topics of section 2.

In section 3, we show the differences between the two general principles of induction and coinduction, by introducing an inductive and a coinductive definition of a set.

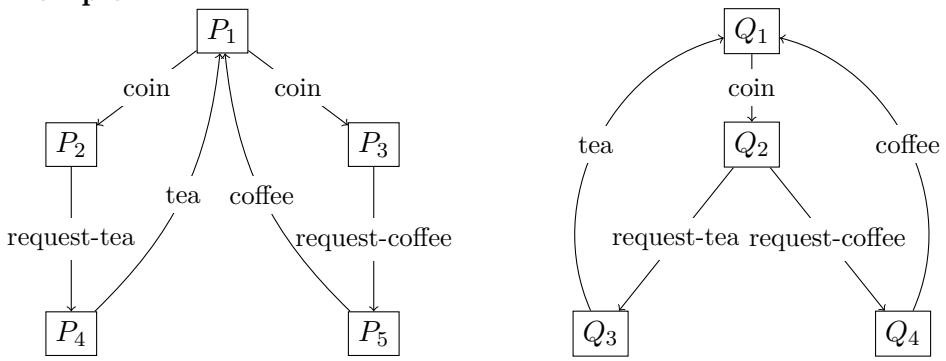
Lastly, we dive deeper into the duality between induction and coinduction and its usage in logic in computer science, in section 4.

When referencing more detailed information, it will be cited specifically. Otherwise, all information contained within this report is based on "Introduction to Bisimulation and Coinduction" [1] by Davide Sangiorgi.

2 Bisimulation

Comparing functions is common as you compare inputs and outputs. However, comparing processes is harder, because they potentially run infinitely long. Since we can not compare two infinite processes' outputs, we have to look at their structures and behavior to determine whether or not they are equal. Through bisimulation, which is coinductively defined, it is possible to check whether two processes behave equally, to check for instance, if two operating systems behave equally. To start this off, we bring an example of two infinitely running vending machines, to get a feeling of the differences between automata and processes.

Example 2.1.



In this example, we can see two slightly different vending machines. While the right vending machine behaves as expected, the left one lacks the option for the customer to be able to choose whether he wants coffee or tea, making the process non-deterministic. Recalling automata theory and interpreting these two vending machines as automata (seeing P_1 and Q_1 as *initial* states and P_4, P_5, Q_3 and Q_4 as *accepting* states), they are equal, since both accept the same language (in the case you are not familiar with automata theory, look up Chapter 3 in "Mathematical Foundations of Automata Theory"[1] from Jean-Èric Pin). However, seeing as their behavior is different, we need to look at them as something different, specifically, as processes. This shows us that bisimulation as a proof technique for process equality must differentiate between determinism and non-determinism.

Before further explaining bisimulation, we now need to mathematically define what a process is.

For our first attempt, we try interpreting a process as a function. In sequential languages, a program is a function which transforms inputs into outputs.

Example 2.2.

Program 1

```
1  x := 1
2  x := x + 1
```

Program 2

```
3  x := 2
```

Program 1 and 2 share the variable x. If we compute these functions sequentially, x will evaluate to 2, no matter if the order of code lines is $1, 2, 3$ or $3, 1, 2$. However, this attempt fails when trying to run this function concurrently due to the sequence of the order being non-deterministic. For example, if the code lines are executed in the order $1, 3, 2$ then x will evaluate to 3, which is incorrect.

[1] `https://www.irif.fr/~jep/PDF/MPRI/MPRI.pdf`

In conclusion, parallel programs can not be interpreted as functions and need to be seen as processes. Furthermore, in contrast to functions, processes can be non-deterministic and have no necessity for termination. Therefore, to express processes mathematically we introduce *Labelled Transition Systems.*

Definition 2.3 (LTS)**.** A Labelled Transition System is a triple $(Pr, Act, \rightarrow)$ where:

- Pr is a non-empty set (*domain* of the LTS)
- Act is the set of *actions (labels)*
- $\rightarrow \subseteq \mathcal{P}(Pr \times Act \times Pr)$ is the *transition relation*

Elements of Pr are called *states* or *processes.*
We write $P \xrightarrow{\mu} Q$ when $(P, \mu, Q) \in \rightarrow$ (Q is a μ-*derivative* of P).
If $P \xrightarrow{\mu_1} P_1 \dots P_{n1} \xrightarrow{\mu_n} P_n$ then P_n is a *multi-step derivative* of P.

Now that we know what processes are, we must find a way to determine whether or not processes are equal. As a method of proving this equality, we attempt isomorphism which is used to compare graphs. In graph theory, two structures are isomorphic, if a bijection can be established on their components (more details provided in Chapter 1 of "Graph Theory"[2] written by Keijo Ruohonen).

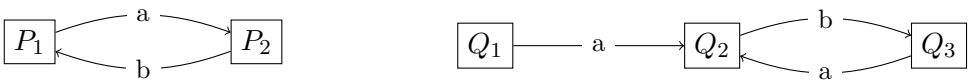


Figure 1: Example of a non-isomorphic LTS

While both processes in Figure 1 are essentially equal in their behavior, due to the same sequence ($ababa\dots$) in both cases, by using isomorphism, it fails immediately by the two processes having a different amount of nodes. Thus, isomorphism is **too strong** for the comparison of processes.

Another potential method for proving equality is trace equivalence, which is used to compare automata. In automata theory, two automata are equal if they accept the same language.

[2] `http://math.tut.fi/~ruohonen/GT_English.pdf`

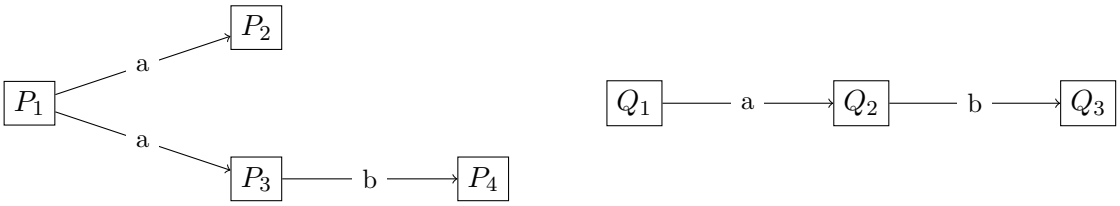


Figure 2: Example for trace equivalence

Both processes in Figure 2 are trace equivalent (seeing P_1 and Q_1 as *initial* states and P_2, P_3, P_4, Q_2 and Q_3 as *accepting* states) since a or ab is accepted for both, however the first process is non-deterministic, which means we cannot decide if P_1 chooses to go to P_2 or to P_3 with the input of a and in P_2 it is not possible anymore to reach P_4. Thus, trace equivalence is **too weak** for the comparison of processes.

Since isomorphism and trace equivalence do not work out for comparing processes properly, we further make use of *bisimulation*, which is "weaker" than isomorphism and "stronger" than trace equivalence.

Definition 2.4 (Bisimulation and bisimilarity)**.** A process relation $\mathcal{R}$ is a *bisimulation* if whenever $(P, Q) \in \mathcal{R}$, for all μ we have:

1. for all P' with $P \xrightarrow{\mu} P'$, there is Q' such that $Q \xrightarrow{\mu} Q'$ and $(P', Q') \in \mathcal{R}$
2. the converse, on the transitions emanating from Q, i.e., for all Q' with $Q \xrightarrow{\mu} Q'$, there is P' such that $P \xrightarrow{\mu} P'$ and $(P', Q') \in \mathcal{R}$

Bisimilarity, written $\sim$, is the union of all bisimulations and therefore a bisimulation itself; thus $P \sim Q$ holds if there is a bisimulation $\mathcal{R}$ with $(P, Q) \in \mathcal{R}$

Example 2.5. We want to show that the two processes of Figure 1 are bisimilar. To show that, we need to find any bisimulation that fulfills the criteria of the Definition 2.4.

Iteration	$\mathcal{R}$	Bisimulation	Missing states
1	(P_1, Q_1)	✗	P_1, Q_2
2	$(P_1, Q_1), (P_2, Q_2)$	✗	Q_3
3	$(P_1, Q_1), (P_2, Q_2), (P_1, Q_3)$	✓	-

We start by placing the pair (P_1, Q_1) in $\mathcal{R}$. Afterwards, we checked if our current $\mathcal{R}$ follows the constraints. In the first iteration this is not the case, because not even one of the derivatives of P_1 or Q_1 are in $\mathcal{R}$. Therefore, we added (P_2, Q_2). Checking the constraints for (P_1, Q_1) again, we can see that P_1 and Q_1 have both the same transition a which leads P_1 to P_2 and Q_1 to Q_2. So we can say, that P_1 and Q_1 have the same behavior, which is what we wanted to achieve. But if we check for (P_2, Q_2) it fails again,

because P_2 leads to P_1 via the transition b and Q_2 leads to Q_3 via the transition b, but Q_3 is not in $\mathcal{R}$ so far. That is why we add (P_1, Q_3) to $\mathcal{R}$ and check again. Now all the criteria are fulfilled since P_1 and Q_3 have both the transition a, which leads P_1 to P_2 and Q_3 to Q_2 and (P_2, Q_2) is already in $\mathcal{R}$. Intuitively, it makes a lot of sense as P_1 behaves equally to Q_3. Therefore, as processes they are equal, but interpreting them as graphs they are not as they are obviously not isomorphic.

Example 2.6. Finding a bisimulation for the processes in Figure 2 is impossible, because the transition from P_1 to P_2 cannot be matched by Q_1.

To conclude this chapter, bisimiliarity and its proof techniques are not *inductive*, but *coinductive*. It is "circular", since we prove $(P, Q) \in \sim$ by showing that $(P, Q) \in \mathcal{R}$ and $\mathcal{R}$ is a bisimulation relation (relation that satisfies the same clauses as $\sim$).

3 Inductive defintion versus coinductive definition

Readers in Computer Science have seen a lot of inductive definitions. In Section 2 we claimed that bisimulation is a coinductively defined proof technique. But what does being *coinductively defined* stand for, especially in contrast to *inductively defined*? To demonstrate the difference, we introduce an inductive definition of finite traces as well as a coinductive definition of ω-traces.

3.1 Inductive definition of finite traces

Finite traces are akin to termination, so looking for finite traces in a LTS means looking for traces without cycles.

Definition 3.1 (Finite traces)**.**

$$\frac{P \text{ inactive}}{P \downarrow} \qquad\qquad \frac{P \xrightarrow{\mu} P' \qquad P' \downarrow}{P \downarrow}$$

For inductively defined sets the inference rules are read **top-down**, e.g. if P is inactive (has no outgoing transitions) P has a finite trace. The first rule is the base rule of the set, which includes all processes without any outgoing transition. The second rule is the step rule with which the composite elements of the set can be generated. So the process of generating elements with respect to an inductively defined set is **linear**: starting at a primitive base element and stepping forward to more complex composite objects. Moreover, $\downarrow$ is the **smallest** set of processes, that is **closed forward under the rules**[3].

3.2 Coinductive definition of ω-traces

ω-traces are akin to non-termination. Therefore, we are seeking for cycles in a LTS since they express infinite traces.

[3] using the rules top-down (from premise to conclusion)

Definition 3.2 (ω-traces)**.**

$$\frac{P \xrightarrow{\mu} P' \qquad P' \restriction_\mu}{P \restriction_\mu}$$

In contrast to inductively defined sets, the inference rules in coinductively defined sets are read **bottom-up**, e.g. process P has an ω-trace under μ if it is possible to observe an infinite sequence of μ-transistions starting from P. Then $\restriction_\mu$ is the **largest** set of processes, that is **closed backward under the rule**[4].

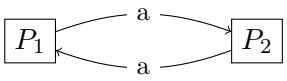


Figure 3: Example of a circular LTS

Example 3.3. To show that P_1 has an ω-trace under a in Figure 3, we need to find a P', which is a multistep-derivative of P_1 and has an ω-trace under a too. In our case we have found P_2, which fits in the rule. But to show that P_2 has an ω-trace under a, we need to find a P' again, which is our P_1. Here is the point where we meet this "circular behavior" of coinduction again since P_1 and P_2 mutually define each other.

It is noticeable, that in the coinductive definition the base rule is missing, which is one of the crucial differences to induction. In Section 3.1 we claimed, that induction is somehow "linear", because it has a base element or a "start point". The coinductive principle is "circular", it has a lack of a "start point".

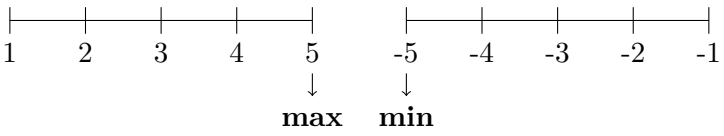
4 Duality between induction and coinduction

Now we want to further look at what coinduction is, especially in relation to induction. In the beginning, we stated that coinduction is dual to induction. Before going deeper, we need to look at what duality is, in general. To get a sense of it, we take De Morgan's law as a simple example of duality (for more details about De Morgan's law skim Chapter 7 in "Lecture Notes"[5] from the University of Manchester). De Morgan's law proves, that $a \wedge b$ equals $\neg(\neg a \vee \neg b)$. This works out since a is dual to $\neg a$, b is dual to $\neg b$ and $\wedge$ is dual to $\vee$. So $\neg a \vee \neg b$ is the dual of $a \wedge b$ and by negating $\neg a \vee \neg b$ it equals $a \wedge b$.
Thus we can imagine, that we can turn induction into coinduction and vice versa too, by using the dual structure, dual order and so on. This is useful in Computational Logic, since you can use either whenever needed by simply using the dual. For instance, in propositional logic it is sometimes desired to use CNF (e.g. for SAT Solving) and sometimes DNF (see Chapter 2 in "Logic and Proof"[6] from Lawrence C. Paulson for more information about CNF and DNF). And this is where the duality proven by De

[4] using the rules bottom-up (from conclusion to premise)
[5] `http://www.maths.manchester.ac.uk/~avb/0n1_pdf/0N1_All.pdf`
[6] `https://pdfs.semanticscholar.org/66c3/3a7190ecc6c1a61aed8c788b64ef735bbfeb.pdf`

Morgan comes into play, because it makes it possible to transform every propositional formula in a CNF or a DNF. And the same holds for induction/coinduction, in Section 3 it seemed natural to use induction for finite traces and coinduction for ω-traces.
To now express the duality between induction and coinduction, we use lattice theory (take a look at [2] for further reading), as it is an ordered structure, which can be turned upside down, and portrays greatest and least fixed-points, which are related to coinduction/induction (e.g. in Section 3 seeking for finite traces computes a least fixed-point and finding ω-traces computes a greatest fixed-point) and their duality very well.
But first we give you another example for a better understanding of duality.

Example 4.1.



The left number line contains the numbers 1 to 5 and the right number line contains the dual numbers to $1, \ldots, 5$, which are $-1, \ldots, -5$. Number 5 is the **maximum** of the left number line with -5 being its dual element, which is the **minimum** of the dual number line. Informally, we can say, that if an element has a certain property in a structure (e.g. lattice) then the dual element has the dual property in the dual structure.

To make the duality between coinduction and induction even clearer, we will now dive into lattice theory. Because inductively defined sets relate to least fixed points, whereas coinductively defined sets relate to greatest fixed points, we can imagine with the example 4.1 given above, that coinduction can be turned upside down to induction and vice versa. Through the duality of maximum and minimum in example 4.1 we can imagine that this property also holds for the greatest and least fixed point. A least fixed point in a certain lattice will be a greatest fixed point in the dual lattice. To explain this further, we will follow this up with some formal definitions.

Definition 4.2 (Poset)**.** A *partially ordered set* (or *poset*) is a non-empty set equipped with a relation on its elements that is reflexive, transitive, and antisymmetric.

We usually indicate the partial order relation of a poset by $\leq$.

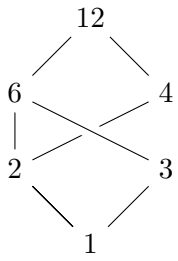


Figure 4: Divisibility relation of 12

Example 4.3. The divisibility relation of 12 in Figure 4 is a primitive example of a poset.

Definition 4.4 (Complete lattice)**.** A *complete lattice* is a poset in which all subsets have a unique supremum (also called a least upper bound or *join*) and a unique infimum (also called a greatest lower bound or *meet*).

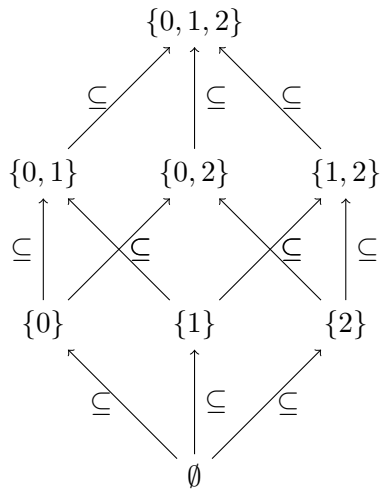


Figure 5: Example of a powerset lattice

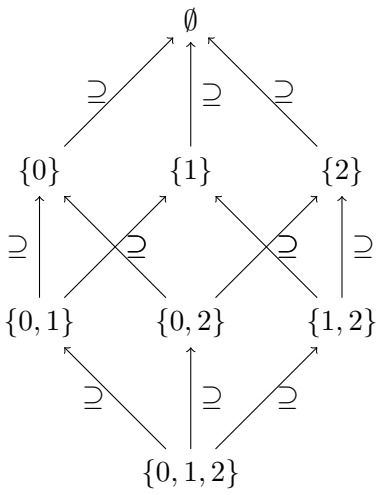


Figure 6: Powerset lattice, which is dual to lattice in Figure 5

Example 4.5. We built the powerset lattice of the set $\{0, 1, 2\}$ in Figure 5, with $\{0\}$ being a superset of $\emptyset$ and $\{1, 2\}$ being the superset of $\{1\}$, $\{2\}$ and $\emptyset$, etc. In this case the join is given by set union $\cup$ and the meet by set intersection $\cap$. Therefore, the *join* (also supremum or least upper bound) evaluates to $\{0, 1, 2\}$ and the *meet* (also infimum or greatest lower bound) to $\emptyset$.

Definition 4.6 (Endofunction)**.** An *endofunction* on a set L is a function from L to itself.

Definition 4.7. Let F be an endofunction on a poset L.

- F is *monotone* if $x \leq y$ implies $F(x) \leq F(y)$, for all x, y.
- An element x of the poset is a *pre-fixed point* of F if $F(x) \leq x$. Dually, a *post-fixed point* of F is an element x with $x \leq F(x)$.
- A *fixed point* of F is an element x that is both a pre-fixed point and a post-fixed point, that is, $F(x) = x$. In the set of fixed points of F, the least element and the greatest element are respectively called the *least fixed point* of F and the *greatest fixed point* of F.

Example 4.8. Let F be a monotone endofunction on our powerset lattice in Figure 5. F is defined via the following inference rules:

$$F(X) = \frac{0 \in X}{0 \in F(X)}, \frac{1 \in X}{1 \in F(X)}, \frac{2 \in X}{}$$

Then the values computed by applying F on our powerset lattice are:

Points	Pre-fixed point	Post-fixed point	(Greatest/Least) fixed point
$F(\emptyset) = \emptyset$	✓	✓	**least fixed point**
$F(\{0\}) = \{0\}$	✓	✓	fixed point
$F(\{1\}) = \{1\}$	✓	✓	fixed point
$F(\{2\}) = \emptyset$	✓	✗	-
$F(\{0,1\}) = \{0,1\}$	✓	✓	**greatest fixed point**
$F(\{0,2\}) = \{0\}$	✓	✗	-
$F(\{1,2\}) = \{1\}$	✓	✗	-
$F(\{0,1,2\}) = \{0,1\}$	✓	✗	-

Now if we compute the values for the dual powerset lattice visualized in Figure 6 with the dual monotone endofunction F' it is noticeable that $\emptyset$ is a least fixed point in the first lattice and the dual element of $\emptyset$ being $\{0,1,2\}$, is a greatest fixed point in the second (dual) lattice. This holds for all points, e.g. the dual elements of pre-fixed points are post-fixed points in the dual lattice and vice versa.

$$F'(X) = \frac{0 \in X}{0 \in F(X)}, \frac{1 \in X}{1 \in F(X)}, \frac{}{2 \in F(X)}$$

Points	Pre-fixed point	Post-fixed point	(Greatest/Least) fixed point
$F(\{0,1,2\}) = \{0,1,2\}$	✓	✓	**greatest fixed point**
$F(\{1,2\}) = \{1,2\}$	✓	✓	fixed point
$F(\{0,2\}) = \{0,2\}$	✓	✓	fixed point
$F(\{0,1\}) = \{0,1,2\}$	✗	✓	-
$F(\{2\}) = \{2\}$	✓	✓	**least fixed point**
$F(\{1\}) = \{1,2\}$	✗	✓	-
$F(\{0\}) = \{0,2\}$	✗	✓	-
$F(\emptyset) = \{2\}$	✗	✓	-

Definition 4.9 (Fixed-point theorem)**.** On a complete lattice, a monotone endofunction has a *complete lattice of fixed points*. In particular the *greatest fixed point* of the function is the *join of all its post-fixed points*, and the *least fixed point* is the *meet of all its pre-fixed points*.

Definition 4.10 (Sets inductively and coinductively defined by F)**.** For a complete lattice L whose points are sets (as in the complete lattices obtained by the powerset construction), and an endofunction F on L, the sets

$$F_{ind} \stackrel{\text{def}}{=} \cap\{x \mid F(x) \leq x\}$$
$$F_{coind} \stackrel{\text{def}}{=} \cup\{x \mid x \leq F(x)\}$$

(the meet of the pre-fixed points, and the join of the post-fixed points) are, respectively, the sets *inductively defined by F* and *coinductively defined by F*.

According to definition 4.10 we can get the least fixed point by computing the meet of all pre-fixed points. Dually, we get the greatest fixed point by computing the join of all post-fixed points. In LICS[7] it is desirable to compute the greatest/least fixed point iteratively (starting from the bottom/top element). An example for an inductive (iterativ) computation is resolution, which computes a least fixed point and an example for a coinductive (iterativ) computation is the minimization of automata, which computes a greatest fixed point. To make this possible, we introduce continuity and cocontinutiy.

Definition 4.11 (Continuity and cocontinuity)**.** An endofunction on a complete lattice is:

- *continuous* if for all sequences $\alpha_0, \alpha_1, \ldots$ of increasing points in the lattice we have $F(\cup_i \alpha_i) = \cup_i F(\alpha_i)$.
- *cocontinuous* if for all sequences $\alpha_0, \alpha_1, \ldots$ of decreasing points in the lattice we have $F(\cap_i \alpha_i) = \cap_i F(\alpha_i)$.

If an endofunction F on a complete lattice is *continuous*, then we have: $lfp(F) = F^{\cup\omega}(\bot)$ and if an endofunction F on a complete lattice is *cocontinuos*, we have: $gfp(F) = F^{\cap\omega}(\top)$, for which $\bot$ is the bottom element and $\top$ the top element and $\cup\omega$ or $\cap\omega$ means the union/intersetion of the values computed by applying F on the bottom/top element arbitrary often.

Example 4.12. The powerset lattice in Figure 5 is continuous as for all sequences of increasing points ($\emptyset$, {0}, {0,1}, ...) the condition from definition 4.11 hold. So we are able to compute the least fixed point iteratively starting with the bottom element:

$$F(\emptyset) \cup F(F(\emptyset)) \cup F(F(F(\emptyset))) \cup \ldots = \emptyset \cup \emptyset \cup \emptyset \cup \ldots = \emptyset$$

Since the powerset lattice in Figure 6 is dual to the powerset lattice in Figure 5, the powerset lattice in Figure 6 has to be cocontinuous. And therefore we are able to compute the greatest fixed point iteratively starting with the top element:

$$F(\{0,1,2\}) \cap F(F(\{0,1,2\})) \cap F(F(F(\{0,1,2\}))) \cap \ldots = \{0,1,2\} \cap \{0,1,2\} \cap \{0,1,2\} \cap \ldots = \{0,1,2\}$$

4.1 Bisimilarity as fixed point

To see how bisimulation and its proof method fit the *coinductive scheme*, consider the function $F_\sim : \mathcal{P}(Pr \times Pr) \to \mathcal{P}(Pr \times Pr)$ defined by:
$F_\sim(\mathcal{R})$ is the set of all pairs (P, Q) such that:

[7] Logic In Computer Science

1. for all P' with $P \xrightarrow{\mu} P'$, there is Q' such that $Q \xrightarrow{\mu} Q'$ and $(P', Q') \in \mathcal{R}$

2. for all Q' with $Q \xrightarrow{\mu} Q'$, there is P' such that $P \xrightarrow{\mu} P'$ and $(P', Q') \in \mathcal{R}$

Then $\sim$ *is the greatest fixed point* of $F_\sim$ and $\sim$ is the largest relation $\mathcal{R}$ such that $\mathcal{R} \subseteq F_\sim(\mathcal{R})$; thus $\mathcal{R} \subseteq \sim$ for all $\mathcal{R}$ with $\mathcal{R} \subseteq F_\sim(\mathcal{R})$.

5 Conclusion

To sum up, coinduction and coinductively defined proof techniques such as bisimulation are effective and becoming increasingly more important since they are very handy for dealing with potentially infinite data (i.e. processes). While still being a fairly new field of research, coinduction is more easily understandable due to its duality to induction, which lets readers understand both, by simply understanding one of them as well as duality. The latter as well as the close connection for practical use of both induction and coinduction in relation to each other is explained in our report through lattice theory and least and greatest fixed points.
Furthermore throughout the report, we explained that an inductive definition describes the construction for generating elements (closure forward), while a coinductive definition describes the deconstruction for decomposing elements (closure backwards). In induction, rules are strict and straight forward and therefore we look for the universe, which is the smallest in which such rules live. In contrary to that, coinduction seeks the largest universe. Seeing as bisimulation compares processes, we mathematically defined processes and showed why common mathematical methods for comparing graphs and automata (isomorphism, trace equivalence) do not fit the requirements for comparing processes. For the future, coinduction will only get more important. Since the current research is somewhat limited, it can be expected to be researched even more.

References

[1] D. Sangiorgi. Introduction to Bisimulation and Coinduction. *Cambridge University Press*, New York, NY, USA, 2012.

[2] B.A. Davey and H.A. Priestley. Introduction to Lattices and Order. *Cambridge University Press*, New York, NY, USA, 1990.