Time Hybrids a new Generic Theory of Reality

This document is about the *Time Hybrids* book of *Fred Van Oystaeyen*.

Fred is an outstanding mathematician in such fields as *noncommutative algebraic geometry*, *virtual topology* and *functor geometry*.

I dedicate this document to Fred.

Fred told me about an old Chinese legend about a poet who chiselled a small poem on a small stone with a small chisel and threw that stone into the sea, hoping that, one day, someone would read the poem.

Fred compared his book with that stone.

I hope that this document will, somehow, increase the number of people that read his book.

How to read this document

This document does not present its content in a linear way.

Sections contain hyperlinks to sections that can be read by need.

Introduction

Warning

This document, especially its introduction, is a highly opinionated one.

Many sentences start with "I tend to think of", emphasizing the fact that I may be wrong.

I hope that my opinion about the book is more or less compatible with the opinion of Fred.

Specification and Implementations

A *specification* consists of *declarations* of *features*.

Declarations come with *laws*.

Together, features and laws are called *requirements*.

Implementations of a specification consist of *definitions* of declared features.

Definitions come with *proofs* of laws.

Note that a specification with *less* requirements allows for *more* implementations, so it is important to keep requirements as minimal as possible.

In a way you can think of a specification as a *description*.

I tend to think of a description as an implementation that is an *informal specification*.

For example, the picture on The Treachery of Images is a description (pun intended!) of a well known description of a pipe.

Frankly, if you would never have seen a pipe before, would you be able to make a pipe *only* from this description?

Generic Theory

I tend to think of a *generic theory* as a *theory consisting of specifications of theories*, a *framework theory of theories* or *template theory of theories*, where *theories fit into as implementations*.

Generic Theory of Reality

The *Time Hybrids* book describes a *Generic Theory of Reality*.

Hence I tend to think of a generic theory of reality as a *theory consisting of specifications of theories of reality*, a *framework theory of theories of reality* or *template theory of theories of reality*, where *theories of reality fit into as implementations*.

Quantum Theory and Relativity Theory

It is generally agreed upon that *quantum theory* and *relativity theory* are two *fundamental theories of reality*.

A *unifying theory* for quantum theory and relativity theory is yet to be agreed upon.

I tend to think of the generic theory of reality of the book as a *partially unifying theory* for quantum theory and relativity theory, concentrating on *common requirements*.

The book states that various phenomena of quantum theory and relativity theory, which, until now, have been considered counter-intuitive, can, within its generic theory of reality, be viewed in a more intuitive way.

The book also provides some basic insight in *how* to fit quantum theory and relativity theory into its partially unifying generic theory of reality.

Future Research

I tend to think that, providing *all details* on how to fit quantum theory and relativity theory into the partially unifying generic theory of reality, and comparing it with *observed reality*, is a challenging topic for future research.

For example, would it not be nice to be able to show that, somehow, the theory of Stephen Wolfram and Co, explained in [https://www.ted.com/talks/stephen_wolfram_how_to_think_computationally_about_ai_the_universe_and_everything?language=en] can be seen as an implementation of the specification of Fred Van Oystaeyen?

If, one day, all those details would be provided, and if they all correspond to observed reality so far, then that would be an important scientific achievement.

If it is not possible to provide all those details, or if they do not all correspond to observed reality so far, then that would be useful as well.

The reason *why* may provide valuable insight into the generic theory of Fred and/or the theory of Stephen and/or quantum theory and/or relativity theory.

Reality and Compositionality

I tend to think of *compositionality* as an important aspect of reality.

Compositionality is about *components*.

Components can, starting from various *atomic components* be *composed* to *composed components* in various ways.

Category Theory

I tend to think of *category theory* as a *generic theory of mathematics*, a *theory consisting of specifications of theories of mathematics*, a *framework theory of theories of mathematics* or *template theory of theories of mathematics*, where *theories of mathematics fit into as implementations*.

Of course, a far as the book is concerned, the *theories of mathematics* involved are theories that are relevant for modeling reality.

Why Category Theory

Category theory is an *abstract* theory.

You could argue that using an abstract theory results in a steep learning curve.

But here is the thing: *abstraction* is about *simplification* and *simplicity is the ultimate sophistication*.

When I was a first year mathematics student, professor Gevers, my professor mechanics, said

- I am going to proceed slower in order to have covered more material at the end of the year.

Those were wise words.

Once you have built a solid foundation, agreed, first slowing down learning, you can, gradually, start accelerating, eventually speeding up learning.

Compositionality and Category Theory

Category theory is about *collections of objects* and *sets of morphisms*.

Collections are not necessarily sets.

Every morphism is one from a *source object* to a *target object*.

Category theory is compositional.

More precisely, compositionality of category theory is *sequential compositionality of morphisms*.

As is done in most documents about category theory, this document focuses on morphisms instead of on objects.

In general, it focusses on *pointfree concepts* rather than focussing on *pointful concepts*.

More precisely, it focusses on *pointfree, closed components* rather than on *pointful, open components*.

Programmatic notation

In this document I explain the content of the book using *programmatic notation*.

The programmatic notation defines a *domain specific language*, a.k.a. as *DSL*, that is a *library* that is written in the `Scala` *programming language*.

The DSL is a concise *formal language* that is syntactically verified by `Scala`'s powerful type system.

Both the DSL library and the `Scala` language are briefly explained by need.

The `Scala` code is not really idiomatic code.

It is code that, i.m.h.o, is very suitable for learning purposes.

The DSL is both one the *domain of the book*, *reality*, being part of *physisc*, and one for the its *foundations*, being part of *mathematics*.

For the domain of the book, it uses verbose notation that, more or less, corresponds to the one used in physics.

For its foundations, it uses verbose notation that, more or less, corresponds to the one used in mathematics.

In what follows `Scala` is not explicitly mentioned any more.

Programmatic notation for specifications

Specifications are, programmatically, denoted as

- *value classes*, and,
- *type classes*,
- *unary type constructor classes*,
- *binary type constructor classes*,
- ... .

More precisely, they are denoted as

- `trait`*s without corresponding parameter* for value classes, and,
- `trait`*s with corresponding parameter* for all other classes.

Programmatic notation for implementations

Implementations are, programmatically, denoted as

- `val`*s* for value classes, and,
- `given`*s* for all other classes.

Programmatic naming conventions

The programmatic notation uses names that are abbreviations using first letters of (sequences of (parts of)) words.

The programmatic notation uses names with letters of the Greek alphabet corresponding to first letters of words, according to the Romanization of Greek alphabet.

The programmatic notation uses *symbolic operator names between backticks*.

Analogy between Physics and Computer Science

Lets consider, for a moment, the analogy between *physics* and *computer science*, in as far as the goal of physics is to understand what *reality* is all about, and the goal of computer science is to understand what *software* is all about.

Agreed, understanding reality is so much more difficult than understanding software: software is something we invented ourselves, while reality is, afaik, still one great mystery.

It is now generally agreed upon that computer science benefits from category theory as a partially unifying theory of software theories into which *effectfree software theory* and *effectful software theory* fit as implementations.

Maybe, one day, it will be generally agreed upon that physics benefits from category theory as a partially unifying theory of physics theories into which *quantum theory* and *relativity theory* fit as implementations.

By the way, in a previous life, I was one of the researchers working on category theory as a partially unifying theory of effectfree software theory and effectful software theory using *monads*. I did most of my work as a "late at night hobby". I also worked two years at the University of Utrecht together with Erik Meijer, Graham Hutton and Doaitse Swierstra. I mainly wrote and teached courses, but I also did some research. Our group looked at monads as *computations* that are *computed*. Computations are generalizations of *expression* that are *evaluated*. Computations and expressions are *operational* components. Moreover, monads and expressions are *open*, pointful components.

Nowadays I look at *morphism* as *programs* that are run. Morphisms are generalizations of *functions*. Morphism and functions are *denotational* components. Moreover, morphism and functions are *closed*, pointfree components. You can look at my talk about *Program Description based Programming* on flatMap, 8-9 May 2019, Oslo Norway. I am refactoring the work presented in Olso, upgrading from `Scala2` to `Scala3`, and changing the paradigm name to *Program Specification based Programming* for reasons explained above.

In other words, I have been doig and I still am doing foundational work on software theories that is similar to the foundational work Fred is doing on physics theories.

Time Hybrids Domain: Specifications

Recently I attended a lecture of Fred on its book in Almeria. The lecture was accompanied with an informal paper.

Let's start with the first sentences of the abstract of the paper.

We introduce a generic model for space-time where time is just a totally ordered set ordering the states of the universe at moments where over (not in) each state we define potentials or pre-things which are going to *evolve via correspondences between the momentary potentials to existing things. Existing takes time and observing takes more time.*

The following concepts are involved

- time moments,
- universe states,
- pre-things,
- space,

and,

- things.

Moreover

- pre-things are momentary (they do not really exist),
- things exist (which takes time) and,
- observing takes more time.

This document is work in progress.

For now only time moments, universe states and pre-things are dealt with.

Let's continue with another sentence of the content of the paper.

We can define a “place map”, p(t): PS(t)--->U(t) where some A(t) is taken to an element pA(t) of the nc-lattice L(t) giving the topology of U(t) such that p(t) respects the (inclusion) partial orders on PS(t) and U(t).

This sentence uses notation that needs some explanation.

- Time moments are denoted as *t*.
- The universe state at time moment *t* is denoted as *U(t)*.
- The collection of sets of pre-things at time moment *t* is denoted as *PS(t)*.
- Sets of pre-things are denoted as *A(t)*.
- The "place map" *p(t)* maps *A(t)* to *pA(t)*.
- The *non-commutative lattice* on *U(t)* is denoted as *L(t)*.

The non-commutative *L(t)* lattice defines a *virtual topology* on *U(t)*.

p(t) respects the non-commutative lattice order on *PS(t)* and the subset order on *U(t)*.

Note that the statements above can be seen as requirements.

In what follows we denote them using programmatic notation.

Time

```
package timehybrids.specification

import specification.{Arbitrary, Ordered}
```

```
trait Time[Moment: Arbitrary: Ordered]:

// ...
```

`Time` is a type class for parameter `Moment`.

The requirements for `Moment` to be a `Time` type are

- `Moment` is an `Arbitrary` type,
- `Moment` is a `Ordered` type.

`Arbitrary` is fully explained in Arbitrary.

`Ordered` is fully explained in Ordered.

A type *implicitly* denotes a *set* and a *value* of a type *implicitly* denotes an *element* of a set.

Recall that we are building a DSL for reality and its foundations.

The requirements above allow us to, programmatically,

- write statements involving arbitrary time moments,
- state that one time moment is before another one.

Time moments are also simply called *moments*.

Recall that the code is not really the most idiomatic one, it is idiomatic for the DSL we are defining.

Instead of using the *"is-a" Object Oriented Programming* idiom we use the *"has-a" Functional Programming* idiom.

In other words, instead of using *inheritance* we use *delegation*.

Moreover, delegation is *implicit*.

Delegates need to be defined *explicitly* by `summon`*ing* them.

They can then be `import`*ed* by need and made available as `given`*s* by need.

Agreed, all this is somewhat verbose, but, the beauty of programmatic notation like this comes from the combination of *compactness* and *conciseness* of `trait` *declarations* like

- `trait Time[Moment: Arbitrary: Ordered]`,

stating that, for `Moment` to be a `Time` type, `Moment` is required to be an `Arbitrary` type and an `Ordered` type.

```
  // ...

  val ma: Arbitrary[Moment] = summon[Arbitrary[Moment]]

  val mo: Ordered[Moment] = summon[Ordered[Moment]]
```

```
  // ...
```

`Time` related foundational delegates are defined.

```
  // ...

  val am: Moment = ma.arbitrary
```

`Time` related foundational members using members of `Time` related foundational delegates are defined.

You may wonder why similar members using `mo` are not defined.

This is because those members are `extension`s that are globally available.

Universe

```
package timehybrids.specification

import specification.{
  VirtualTopology,
  Sets,
  Category,
  ActingUponFunction,
  Functor
}

trait Universe[
    Set[_]: Sets,
    Morphism[_, _]: Category: ActingUponFunction,
    Moment: Time,
    State: [_] =>> VirtualTopology[Set, State]: [_] =>> Functor[
      [_, _] =>> Tuple2[Moment, Moment],
      Morphism,
      [_] =>> State
    ]
]:

  // ...
```

Using the `type` definitions below you can read the `Universe` definition above as

```
trait Universe[
    Set[_]: Sets,
    Morphism[_, _]: Category: ActingUponFunction,
    Moment: Time,
    State: [_] =>> VirtualTopology[Set, State]: [_] =>> Functor[
```

```
        [_, _] =>> MomentMorphism,
        Morphism,
        [_] =>> State
      ]
  ]:
```

`Universe` is a type class for parameter `State`.

`Universe` also has a foundational parameter `Set` that is required to be a `Sets` unary type constructor.

`Sets` is fully explained in Sets.

`Sets` *explicitly* denotes *the realm of all sets*.

Mathematically this is not a set.

Programmatically it is a constructive set (recall that, programmatically, a set is implicitly denoted by a type).

`Universe` also has a foundational parameter `Morphism` that is required to be a `Category` binary type constructor and a `ActingUponFunction` binary type constructor.

`Category` is fully explained in Category.

`ActingUponFunction` is fully explained in ActingUponFunction.

`Universe` also has a domain parameter `Moment` that is required to be a `Time` type.

Type `Tuple2[T, T]`, somewhat abusively, denotes an ordered set implicitly denoted by type `T`.

- A value `(l, r)`, somewhat abusively, denotes `{ l `<=` r } `=` { true }`.

Using the `type` definitions below the requirements for `State` to be a `Universe` type are

- `State` is a `VirtualTopology[Set, State]` type.
- `State` is a [ ] =>> State] type,

`VirtualTopology` is fully explained in VirtualTopology.

`Functor` is fully explained in Functor.

```
  // ...

  val cs: Sets[Set] = summon[Sets[Set]]

  val mc: Category[Morphism] = summon[Category[Morphism]]

  val mauf: ActingUponFunction[Morphism] =
summon[ActingUponFunction[Morphism]]

  // ...
```

Foundational delegates are defined.

```
// ...

val mt: Time[Moment] = summon[Time[Moment]]

// ...
```

`Time` related domain delegates are defined.

```
// ...

type MomentMorphism = Tuple2[Moment, Moment]

// ...
```

`Time` related domain types are defined.

```
// ...

// `Universe` related foundational delegates are defined.

val svt: VirtualTopology[Set, State] =
  summon[VirtualTopology[Set, State]]

val mmΦsm: Functor[[_, _] =>> MomentMorphism, Morphism, [_] =>> State] =
  summon[Functor[[_, _] =>> MomentMorphism, Morphism, [_] =>> State]]

// ...
```

`Universe` related foundational delegates are defined.

```
// ...

type StateMorphism = Morphism[State, State]

// ...
```

`Universe` related domain types are defined.

```
// ...

val mmφsm: Function[MomentMorphism, StateMorphism] = mmΦsm.φ

val svts: Function[Set[State], State] = svt.sup
```

```
  // ...
```

`Universe` related foundational members using members of `Universe` related foundational delegates are defined.

Composition2

```
package types

import specification.{Sets}

enum Composition2[Set[_]: Sets, Z]:
  case Atomic[Set[_]: Sets, Z](z: Z) extends Composition2[Set, Z]
  case Composed[Set[_]: Sets, Z](cs2: Set[Composition2[Set, Z]])
      extends Composition2[Set, Z]

import Composition2.{Composed}

def composition2[Set[_]: Sets, Z]
    : Set[Composition2[Set, Z]] => Composition2[Set, Z] =
  Composed.apply

def decomposition2[Set[_]: Sets, Z]
    : Composition2[Set, Z] => Set[Composition2[Set, Z]] =

  val sets: Sets[Set] = summon[Sets[Set]]

  import sets.{set2}

  c => set2 apply (c, c)
```

`Composition2` is an example of *structural compositionality*.

Note that the `Set[Composition2[Set, Z]]` is a `Set2[Composition2[Set, Z]]`.

PreThings

```
package timehybrids.specification

import types.{Composition2, composition2, decomposition2}

import specification.{
  Arbitrary,
  Ordered,
  Sets,
  Category,
  ActingUponFunction,
  Functor,
```

```
    Transformation
  }

  import implementation.{
    orderedCategory,
    functionCategory,
    functionValuedFunctor2
  }

  trait PreThings[
      Set[_],
      Morphism[_, _],
      Moment,
      State: [_] =>> Universe[Set, Morphism, Moment, State],
      PreObject: [_] =>> Arbitrary[
        Set[Set[Composition2[Set, PreObject]]]
      ]: [_] =>> Functor[
        [_, _] =>> Tuple2[Moment, Moment],
        Function,
        [_] =>> Set[Composition2[Set, PreObject]]
      ]: [_] =>> Transformation[
        [_, _] =>> Tuple2[Moment, Moment],
        Function,
        [_] =>> Set[
          Set[
            Composition2[Set, PreObject]
          ]
        ],
        [_] =>> Set[Set[Composition2[Set, PreObject]]]
      ]: [_] =>> Function[Set[Composition2[Set, PreObject]], State]
  ]:

    // ...
```

Using the `type` definitions below you can read the `PreThings` definition above as

```
  trait PreThings[
      Set[_],
      Morphism[_, _],
      Moment,
      State: [_] =>> Universe[Set, Morphism, Moment, State],
      PreObject: [_] =>> Arbitrary[
        PreInteractionsSet
      ]: [_] =>> Functor[
        [_, _] =>> MomentMorphism,
        Function,
        [_] =>> PreThingsSet
      ]: [_] =>> Transformation[
        [_, _] =>> MomentTransition,
        Function,
        [_] =>> Set2[PreThingsSet],
        [_] =>> PreInteractionsSet
```

```
    ]: [_] =>> Function[PreThingsSet, State]
  ]:
```

`PreThings` is a type class for parameter `PreObject`.

`PreThings` also has a foundational parameter `Set`.

`PreThings` also has a foundational parameter `Morphism`.

`PreThings` also has a domain parameter `Moment`.

`PreThings` also has a domain parameter `State` that is required to be a `Universe[Set, Morphism, Moment, State]` type.

Using the `type` definitions below the requirements for `PreObject` to be a `PreThings` type are

- `PreObject` is an `Arbitrary[PreInteractionsSet]` type.
- `PreObject` is a ☐ =>> PreThingsSet] type.
- `PreObject` is a `Transformation[ [_, _] =>> MomentMorphism, Function,` ☐ =>> Set2[PreThingsSet], ☐ =>> PreInteractionsSet] type.

`Transformation` is fully explained in Transformation.

`functionValuedFunctor2`, is fully explained in functionValuedFunctor2

`orderedCategory`, is fully explained in orderedCategory

`functionCategory`, is fully explained in functionCategory

```
  // ..

  val su: Universe[Set, Morphism, Moment, State] =
    summon[Universe[Set, Morphism, Moment, State]]

  // ...
```

`Universe` related domain delegates are defined.

```
  // ...

  import su.{cs}

  import cs.{Set2}

  type PreThing = Composition2[Set, PreObject]

  type PreThingsSet = Set[PreThing]
```

```
    type PreInteraction = Set2[PreThing]

    type PreInteractionsSet = Set[PreInteraction]

    // ...
```

`PreThings` related domain types are defined.

```
  import su.{MomentMorphism}

    val pisa: Arbitrary[PreInteractionsSet] =
      summon[Arbitrary[PreInteractionsSet]]

    val mmΦptsf: Functor[
      [_, _] =>> MomentMorphism,
      Function,
      [_] =>> PreThingsSet
    ] =
      summon[
        Functor[
          [_, _] =>> MomentMorphism,
          Function,
          [_] =>> PreThingsSet
        ]
      ]

    val ptss2Tpis: Transformation[
      [_, _] =>> MomentMorphism,
      Function,
      [_] =>> Set2[PreThingsSet],
      [_] =>> PreInteractionsSet
    ] = summon[
      Transformation[
        [_, _] =>> MomentMorphism,
        Function,
        [_] =>> Set2[PreThingsSet],
        [_] =>> PreInteractionsSet
      ]
    ]

    val ptsφs: Function[PreThingsSet, State] =
      summon[Function[PreThingsSet, State]]

    // ...
```

`PreThings` related foundational delegates are defined.

```
    // ...
```

```
    val apis: Set[PreInteraction] = pisa.arbitrary

    val mmφptsf: Function[
      MomentMorphism,
      Function[PreThingsSet, PreThingsSet]
    ] = mmΦptsf.φ

    val ptss2φtpis: Function[
      Set2[PreThingsSet],
      PreInteractionsSet
    ] = ptss2Tpis.τ

    // ...
```

`PreThings` related foundational members using members of `PreThings` related foundational delegates are defined.

```
    import su.{mc, mauf}

    given Sets[Set] = cs

    given Category[Morphism] = mc

    given ActingUponFunction[Morphism] = mauf
```

Foundational `given`s using `import`ed foundational delegates are defined.

```
    // ...

    import su.{mt, mmΦsm}

    import mt.{ma, mo}

    given Arbitrary[Moment] = ma

    given Ordered[Moment] = mo

    // ...
```

`Time` related foundational `given`s are defined.

```
    given Functor[[_, _] =>> MomentMorphism, Morphism, [_] =>> State] = mmΦsm

    // ...
```

`Universe` related foundational `given`s are defined.

```
// ...

given mmΦptss2f: Functor[
  [_, _] =>> MomentMorphism,
  Function,
  [_] =>> Set2[PreThingsSet]
] = functionValuedFunctor2[
  Set,
  [_, _] =>> MomentMorphism,
  [_] =>> PreThingsSet
]

// ...
```

`PreThings` related foundational `given`s are defined.

```
// ...

val mmφptss2f: Function[
  MomentMorphism,
  Function[
    Set2[PreThingsSet],
    Set2[PreThingsSet]
  ]
] = mmΦptss2f.φ

// ...
```

`PreThings` related foundational members using `PreThings` related foundational `given`s are defined

```
// ...

val pisφpts: Function[PreInteractionsSet, PreThingsSet] =
  pic =>
    for {
      pi <- pic
    } yield {
      composition2 apply pi
    }

val ptsφpis: Function[PreThingsSet, PreInteractionsSet] =
  pts =>
    for {
      pt <- pts
    } yield {
      decomposition2 apply pt
    }
```

```
    val mmφpisf: Function[
      MomentMorphism,
      Function[PreInteractionsSet, PreInteractionsSet]
    ] = mm => ptsφpis `o` mmφptsf(mm) `o` pisφpts

    // ...
```

Composed `PreThings` related foundational members using `PreThings` related foundational members using `PreThings` related foundational `given`s are defined.

The laws of `PreThingsRealityLaws`, `PreThingsRealityLaws` are fully explained in PreThingsRealityLaws.

Time Hybrids Domain: Laws

PreThingsLaws

Back to PreThings

```
    // ...

     // laws

    import specification.{Law}

    import implementation.{composedFunctor}

    trait PreThingsFunctorCompositionLaws[L[_]: Law]:

      given stΦptcf: Functor[Morphism, Function, [_] =>> PreThingsSet]

      val composition2: L[
        Functor[
          [_, _] =>> MomentMorphism,
          Function,
          [_] =>> PreThingsSet
        ]
      ] = {
        mmΦptsf
      } `=` {
        composedFunctor[
          [_, _] =>> MomentMorphism,
          Morphism,
          Function,
          [_] =>> State,
          [_] =>> PreThingsSet
        ]
      }

      // ...
```

This law refers to the following excerpts from the paper.

Thus the total order of Time is just the total order of the states of the universe and we may think of U as a book with pages U(t) indexed by elements of T.

The pages will be glued together by some maps f(t,t') for t<t' in T, where < is the order of T

We let s(t,t') denote a map from PS(t) to PS(t') which may be viewed as a correspondence from S(t) to S(t'), this mathematical concept is just a map from subsets of S(t) to subsets of S(t')

Although it is nowhere mentioned in the book or paper, this optional law relating *f(t,t')* and *s(t,t')* looks natural to me.

```
// ...

import implementation.{functionTargetOrdered, setOrdered}

trait PreThingsLaws[L[_]: Law]:

  val preInteractionAsPreThingComposition
      : L[PreInteraction => PreInteraction] = {
    decomposition2 `o` composition2
  } `=` {
    identity
  }

  val preThingAsPreInteractionDecomposition: L[PreThing => PreThing] = {
    composition2 `o` decomposition2
  } `=` {
    identity
  }

  // ...
```

This law refers to the following, slightly adapted, excerpt from the book (it is taken for granted in the paper).

In fact ,the difference between pre-things and pre-interactions is one of language only, we may just as well call a pre-thing A(t) a pre-interaction i(A(t),A(t)).

```
// ...

  val noPreThingFromNothing: MomentMorphism => L[PreThingsSet] =
    mm =>
      import cs.{set0}
      {
        mmφptsf(mm)(set0)
      } `=` {
        set0
      }
```

```
// ...
```

This law refers to the following, slightly adapted, excerpt from the paper.

Moreover the correspondence acting on the empty set is always the empty set; thus no pre-things arise as the result of a correspondence of the empty set! No pre-thing comes from nothing!

```
// ...

  val unionOfSingletonPreInteractions
      : Set2[PreThingsSet] => L[PreInteractionsSet] =
    ptss2 =>
      import cs.{tuple2, set1, set2, union}
      tuple2(ptss2) match
        case (lpts, rpts) =>
          {
            union {
              for {
                lpt <- lpts
                rpt <- rpts
              } yield {
                ptss2φtpis(set2(set1(lpt), set1(rpt)))
              }
            }
          } `=` {
            ptss2φtpis(ptss2)
          }

// ...
```

This law refers to the following excerpt from the paper, where [1] refers to the book.

The pre-interaction between A(t) and B(t) in S(t) is written as I(A,B)(t), in [1] I put I(A,B)(t) equal to v{ i(a(t),b(t)) for a(t) in A(t),b(t) in B(t) }.

```
// ...

  val naturePreservingPreInteraction: MomentMorphism => L[Boolean] =
    mm =>
      {
        { mmφpisf(mm) `o` ptss2φtpis } `<=` {
          ptss2φtpis `o` mmφptss2f(mm)
        }
      }
        `=` {
          true
        }
```

```
// ...
```

This law refers to the following excerpt from the paper (I changed < to <=).

However there is then a logical assumption, namely that s(t,t')(I(A,B)(t)) <= I(A,B)(t') for t<t', meaning that the correspondences s(t,t') do not change the nature of the later realisation as an interaction. Hence the s(t,t') respect the dichotomy between pre-interactions and other potentials we will call pre-objects, both together are pre-things.

This is. i.m.h.o., really the most fundamental law of the realm of pre-things.

```
// ...

trait PreThingsPlacesLaws[L[_]: Law]:

  val orderPreserving
      : PreThingsSet => PreThingsSet => L[Boolean] =
    lpts =>
      rpts =>
        import su.{svt}
        import svt.{`<=`}
        {
          lpts `<=` rpts `=` true
        } `=>` {
          ptsφs(lpts) `<=` ptsφs(rpts) `=` true
        }

// ...
```

This law refers to the following excerpt from the paper (already mentioned in the introduction).

We can define a "place map", p(t): PS(t)--->U(t) where some A(t) is taken to an element pA(t) of the nc-lattice L(t) giving the topology of U(t) such that p(t) respects the (inclusion) partial orders on PS(t) and U(t).

```
// ...

  val supremumOfAllSingletonPlaces: PreThingsSet => L[State] =
    pts =>
      import cs.{set1}
      import su.{svts}
      {
        svts {
          for {
            pt <- pts
          } yield ptsφs(set1(pt))
        }
      } `=` {
        ptsφs(pts)
```

```
        }

    // ...
```

This law refers to the following excerpt from the paper, where [2] refers to the "Virtual topology and functor geometry book" of Fred

In [2], I took pA(t)=V{ p({a(t)}), a(t) in A(t) } – we then say p is basic - which is harmless and seems logical for the notion of “place” yet we do not use that here.

```
    // ...

      val immobileAfter
          : MomentMorphism => L[Function[PreThingsSet, State]] =
        mm =>
          import su.{mmφsm}
          {
            ptsφs `o` mmφptsf(mm)
          } `=` {
            mmφsm(mm) `a` ptsφs
          }

      val immobileOnInterval: MomentMorphism => PreThingsSet => L[State] =
        case (bm, em) =>
          import cs.{Interval, interval, all}
          import su.{mmφsm}
          val mi: Interval[Moment] = interval apply ((bm, em))
          pts =>
            all apply {
              for {
                m <- mi
              } yield {
                {
                  (ptsφs `o` mmφptsf((bm, m)))(pts)
                } `=` {
                  (mmφsm((bm, m)) `a` ptsφs)(pts)
                }
              }
            }
```

This law refers to the following, slightly adapted, excerpt from the paper.

A string of correspondences, over an interval I=[t,t’], starting with A(t) then yields places p(A(t”)) with t” in I. If f(t,t”)p(A(t))=p(A(t”)) for all t” in I, then we say the pre-thing A is immobile on I.

`composedFunctor`, is fully explained in composedFunctor

`functionTargetOrdered`, is fully explained in functionCategory

`setOrdered`, is fully explained in functionValuedFunctor2

Back to PreThings

Mathematical Foundations Domain: Specifications

Types

Back to TripleLaws

Back to CompositionNaturalTransformation

Back to UnitNaturalTransformation

```
package types

type `o` = [G[_], F[_]] =>> [T] =>> G[F[T]]

type U = [T] =>> T
```

`` `o` `` defines unary type constructor composition.

`U` defines the unary type constructor unit.

Back to UnitNaturalTransformation

Back to CompositionNaturalTransformation

Back to TripleLaws

Arbitrary

Back to Time

```
package specification

trait Arbitrary[T]:

  // ...
```

`Arbitrary` is a type class for parameter `T`.

```
  // ...

  // declared

  def arbitrary: T
```

`Arbitrary` features are declared:

`Arbitrary` does not come with laws.

Recall that we are building a DSL for reality and its foundations.

The features above allow us to, programmatically,

- write statements involving *arbitrary elements*.

Note that `arbitrary` is declared as a `def`.

Two `arbitrary` values are not necessarily equal.

Back to Time

Ordered

Back to Time

Back to VirtualTopology

```
package specification

trait Ordered[T] extends Equality[T]:

  // ...
```

`Ordered` is a type class for parameter `T`.

`Equality` is fully explained in Equality.

```
  // ...

  // declared

  extension (lt: T) def `<`(rt: T): Boolean

  // ...
```

`Ordered` features are declared.

```
  // ...

  // defined

  extension (lt: T) def `<=`(rt: T): Boolean = lt `<` rt || lt `=` rt

  // ...
```

Extra `Ordered` features are defined.

Recall that we are building a DSL for reality and its foundations.

The features above allow us to, programmatically,

- state that *one element is less than another one*,
- state that *one element is less than or equal to another one*.

The laws of `Ordered`, `OrderedLaws` resp. `TotallyOrderedLaws` are fully explained in OrderedLaws, resp. TotallyOrderedLaws.

Back to VirtualTopology

Back to Time

Equality

Back to Ordered.

```
package specification

trait Equality[T]:

  // ...
```

`Equality` is a type class for parameter `T`.

```
  // ...

  // declared

  extension (lt: T) def `=`(rt: T): Boolean

  // ...
```

`Equality` features are declared.

Recall that we are building a DSL for reality and its foundations.

The features above allow us to, programmatically,

- state that *one element is equal to another one*.

The laws of `Equality`, `EqualityLaws` are fully explained in EqualityLaws.

Back to Ordered.

VirtualTopology

Back to Universe

```
package specification

trait VirtualTopology[Set[_]: Sets, T]
    extends Ordered[T],
      Meet[T],
      Join[T],
      Supremum[Set, T]
```

`VirtualTopology` is a type class for parameter `T`.

`Sets` is fully explained in Sets.

`Ordered` is fully explained in Ordered.

`Meet` is fully explained in Meet.

`Join` is fully explained in Join.

`Supremum` is fully explained in Supremum.

Back to Universe

Meet

Back to VirtualTopology

```
package specification

trait Meet[T: Ordered: Arbitrary]:

  // ...
```

`Meet` is a type class for parameter `T`.

```
  // ...

  // declared

  extension (lt: T) def ∧(rt: T): T

  // ...
```

`Meet` features are declared.

The laws of `Meet`, `MeetLaws` are fully explained in MeetLaws.

Back to VirtualTopology

Join

Back to VirtualTopology

```
package specification

trait Join[T: Ordered: Arbitrary]:

  // ...
```

`Join` is a type class for parameter `T`.

```
  // ...

  // declared

  extension (lt: T) def v(rt: T): T

  // ...
```

`Join` features are declared.

The laws of `Join`, `JoinLaws` are fully explained in JoinLaws.

Back to VirtualTopology

Supremum

Back to VirtualTopology

```
package specification

trait Supremum[Set[_]: Sets, T: Ordered: Arbitrary]:
```

`Supremum` is a type class for parameter `T`.

`Sets` is fully explained in Sets.

```
  //

  val sup: Function[Set[T], T]

  // ...
```

Supremum features are declared.

The laws of Supremum, SupremumLaws are fully explained in SupremumLaws.

Back to VirtualTopology

Sets

Back to Universe

Back to VirtualTopology

Back to Supremum

```
package specification

trait Sets[Set[_]] extends MonadPlus[Set]:

  // ...
```

Sets is a unary type constructor class for parameter Set.

MonadPlus is fully explained in MonadPlus.

```
  // ...

  // types

  type Set0 = [Z] =>> Set[Z]

  type Set1 = [Z] =>> Set[Z]

  type Set2 = [Z] =>> Set[Z]

  type Interval = [Z] =>> Set[Z]

  // ...
```

Sets related types are defined.

```
  // ...

  // declared

  extension [Z](lc: Set[Z]) def `=s=`(rc: Set[Z]): Boolean

  extension [Z](lc: Set[Z]) def `<s<`(rc: Set[Z]): Boolean

  def tuple2[Z]: Set2[Z] => Tuple2[Z, Z]
```

```
def interval[Z: Ordered]: Option[Tuple2[Z, Z]] => Set[Z]

def all[Z, L[_]: Law]: Set[L[Z]] => L[Z]

// ...
```

`Sets` features are declared.

```
// ...

// defined

def set0[Z]: Set0[Z] = ζ

def set1[Z]: Z => Set1[Z] = ν

extension [Z](ls: Set[Z]) def ∪(rs: Set[Z]): Set[Z] = ls `+` rs

def set2[Z]: Tuple2[Z, Z] => Set2[Z] = (l, r) => set1(l) ∪ set1(r)

def union[Z]: Set[Set[Z]] => Set[Z] = μ

extension [Z](ls: Set[Z])
  def `<=s<=`(rs: Set[Z]): Boolean = ls `<s<` rs || ls `=s=` rs

// ...
```

`Sets` members are defined.

The laws of `Sets`, `SetsLaws`, are fully explained in SetsLaws.

Back to Supremum

Back to VirtualTopology

Back to Universe

Functor

Back to Universe

Back to ActingUpon

Back to Triple

Back to NaturalTransformation

Back to ActingUponNaturalTransformation

```
package specification

trait Functor[FBTC[_, _]: Category, TBTC[_, _]: Category, UTC[_]]:
```

`Functor` is a unary type constructor class for parameter `UTC`.

`Functor` has two parameters `FBTC` and `TBTC` that are required to be `Category` binary type constructors.

`Category` is fully explained in Category.

```
  // ...

  // declared

  def φ[Z, Y]: Function[FBTC[Z, Y], TBTC[UTC[Z], UTC[Y]]]
```

`Functor` features are declared.

The laws of `Functor`, `FunctorLaws` are fully defined in FunctorLaws.

Back to ActingUponNaturalTransformation

Back to NaturalTransformation

Back to Triple

Back to ActingUpon

Back to Universe

MonadPlus

Back to Sets

```
package specification

trait MonadPlus[UTC[_]] extends Monad[UTC], Plus[UTC]:

  // ...
```

`MonadPlus` is a unary type constructor class for `UTC`.

`Monad` is fully explained in Monad.

`Plus` is fully explained in Plus.

The laws of `MonadPlus`, `MonadPlusLaws`, are fully explained in MonadPlusLaws.

Back to Sets

Monad

Back to MonadPlus

```
package specification

trait Monad[UTC[_]] extends Triple[Function, UTC]:

  // ...
```

`Monad` is a unary type constructor class for `UTC`.

`Triple` is fully explained in Triple.

```
  // ...

  // defined

  extension [Z, Y](utcz: UTC[Z])
    def map(zφy: Function[Z, Y]): UTC[Y] = φ(zφy)(utcz)

  extension [Z, Y](utcz: UTC[Z])
    def flatMap(zφutcy: Function[Z, UTC[Y]]): UTC[Y] = μ(utcz map zφutcy)

  // ...
```

`Monad` members are defined.

`map` and `flatMap` support *powerful `for`-iteration notation*.

See flatmapthatshit.

Back to MonadPlus

Plus

Back to MonadPlus

```
package specification

trait Plus[UTC[_]] extends UtcComposition[UTC], UtcUnit[UTC]:

  // ...
```

`Plus` is a unary type constructor class for `UTC`.

`UtcComposition` is fully explained in UtcComposition.

`UtcUnit` is fully explained in UtcUnit.

The laws of `Plus`, `PlusLaws` are fully explained in PlusLaws.

Back to MonadPlus

Triple

Back to Monad

```
package specification

trait Triple[BTC[_, _]: Category, UTC[_]]
    extends Functor[BTC, BTC, UTC],
      CompositionNaturalTransformation[BTC, UTC],
      UnitNaturalTransformation[BTC, UTC]:

  // ...
```

`Triple` is a unary type constructor class for `UTC`.

`Triple` has a parameter `BTC` that is required to be a `Category` binary type constructor.

`Category` is fully explained in Category.

`Functor` is fully explained in Functor.

`CompositionNaturalTransformation` is fully explained in CompositionNaturalTransformation.

`UnitNaturalTransformation` is fully explained in UnitNaturalTransformation.

```
  // ...

  // delegates

  val c: Category[BTC] = summon[Category[BTC]]

  // ...
```

`Triple` delegates are defined.

The laws of `Triple`, `TripleLaws` are fully explained in TripleLaws.

Back to Monad

UtcComposition

Back to Plus

```
package specification

trait UtcComposition[UTC[_]]:

  // ...
```

`UtcComposition` is a unary type constructor class for `UTC`.

```
  // ...

  // declared

  extension [Z](lutc: UTC[Z]) def `+`(rutc: UTC[Z]): UTC[Z]

  // ...
```

`UtcComposition` features are declared.

The laws of `UtcComposition`, `UtcCompositionLaws` are fully explained in UtcCompositionLaws.

Back to Plus

UtcUnit

Back to Plus

```
package specification

trait UtcUnit[UTC[_]]:

  // ...
```

`UtcUnit` is a unary type constructor class for `UTC`.

```
  // ...

  // declared

  def ζ[Z]: UTC[Z]
```

`UtcUnit` features are declared.

Back to Plus

Category

Back to Universe

Back to Functor

Back to ActingUpon

Back to Triple

Back to NaturalTransformation

Back to ActingUponNaturalTransformation

```
package specification

trait Category[BTC[_, _]] extends BtcComposition[BTC], BtcUnit[BTC]:
  // ...
```

`Category` is a binary type constructor class for parameter `BTC`.

`BtcComposition` is fully explained in BtcComposition

`BtcUnit` is fully explained in BtcUnit

The laws of `Category`, `CategoryLaws` are fully defined in CategoryLaws.

Back to ActingUponNaturalTransformation

Back to NaturalTransformation

Back to Triple

Back to ActingUpon

Back to Functor

Back to Universe

BtcComposition

Back to Category

```
package specification

trait BtcComposition[BTC[_, _]]:
```

`BtcComposition` is a binary type constructor class for parameter `BTC`.

```
  // declared

  extension [Z, Y, X](yμx: BTC[Y, X]) def `o`(zμy: BTC[Z, Y]): BTC[Z, X]
```

`BtcComposition` features are declared.

The laws of `BtcComposition`, `BtcCompositionLaws` are fully defined in BtcCompositionLaws.

Back to Category

BtcUnit

Back to Category

```
package specification

trait BtcUnit[BTC[_, _]]:

  // ...
```

`BtcUnit` is a binary type constructor class for parameter `BTC`.

```
  // ...

  // declared

  def ι[Z]: BTC[Z, Z]
```

`BtcUnit` features are declared.

Back to Category

ActingUponFunction

Back to Universe

```
package specification

type ActingUponFunction = [BTC[_, _]] =>> ActingUpon[Function, BTC]
```

`ActingUpon` is fully explained in ActingUpon.

Back to Universe

ActingUpon

Back to ActingUponFunction

Back to ActingUponNaturalTransformation

```
package specification

trait ActingUpon[LBTC[_, _], RBTC[_, _]: Category]:

  // ...
```

`ActingUpon` is a binary type constructor class for `LBTC`.

`ActingUpon` has a parameter that is required to be a `Category`binary type constructor.

`Category` is fully explained in Category.

```
  // ...

  // declared

  def actionFunctor[Z]: Functor[RBTC, Function, [Y] =>> LBTC[Z, Y]]

  // ...
```

`ActingUpon` features are declared.

```
  // ...

  // defined

  extension [Z, Y, X](lyμx: RBTC[Y, X])
    def `a`(rzμy: LBTC[Z, Y]): LBTC[Z, X] = actionFunctor.φ(lyμx)(rzμy)

  // ...
```

`ActingUpon` members are defined.

`Functor` is fully explained in Functor.

Back to ActingUponNaturalTransformation

Back to ActingUponFunction

CompositionNaturalTransformation

Back to Triple

```
package specification
```

```
import types.{`o`}

trait CompositionNaturalTransformation[
    BTC[_, _]: Category,
    UTC[_]: [_[_]] =>> Functor[BTC, BTC, UTC]
]:

  // ...
```

`` `o` `` is fully explained in Types.

`CompositionNaturalTransformation` is a unary type constructor class for `UTC`.

```
  // ...

  // declared

  val compositionNaturalTransformation: NaturalTransformation[
    BTC,
    BTC,
    UTC `o` UTC,
    UTC
  ]

  // ...
```

`CompositionNaturalTransformation` features are declared.

`NaturalTransformation` is fully explained in NaturalTransformation

```
  // ...

  // defined

  def μ[Z]: BTC[(UTC `o` UTC)[Z], UTC[Z]] =
compositionNaturalTransformation.τ
```

`CompositionNaturalTransformation` members are defined.

Back to Triple

UnitNaturalTransformation

Back to Triple

```
package specification

import types.{U}

trait UnitNaturalTransformation[
    BTC[_, _]: Category,
    UTC[_]: [_[_]] =>> Functor[BTC, BTC, UTC]
]:
```

`U` is fully explained in Types.

`UnitNaturalTransformation` is a unary type constructor class for `UTC`.

```
  // ...

  // declared

  val unitNaturalTransformation: NaturalTransformation[BTC, BTC, U, UTC]

  // ...
```

`UnitNaturalTransformation` features are declared.

`NaturalTransformation` is fully explained in NaturalTransformation

```
  // ...

  // defined

  def ν[Z]: BTC[U[Z], UTC[Z]] = unitNaturalTransformation.τ
```

`UnitNaturalTransformation` members are defined.

Back to Triple

NaturalTransformation

Back to CompositionNaturalTransformation

Back to UnitNaturalTransformation

```
package specification

trait NaturalTransformation[
    FBTC[_, _],
    TBTC[_, _]: Category,
    FUTC[_]: [_[_]] =>> Functor[FBTC, TBTC, FUTC],
```

```
    TUTC[_]: [_[_]] =>> Functor[FBTC, TBTC, TUTC]
] extends Transformation[FBTC, TBTC, FUTC, TUTC]:

  // ...
```

`NaturalTransformation` is a value class.

`NaturalTransformation` has two parameters, `FBTC` and `TBTC` that are required to be `Category` binary type constructors.

`NaturalTransformation` has two parameters, `FUTC` and `TUTC` that are required to be `Functor` unary type constructors.

`Category` is fully explained in Category

`Functor` is fully explained in Functor

`Transformation` is fully explained in Transformation

The laws of `NaturalTransformation`, `NaturalTransformationLaws` are fully defined in NaturalTransformationLaws.

Back to UnitNaturalTransformation

Back to CompositionNaturalTransformation

ActingUponNaturalTransformation

```
package specification

trait ActingUponNaturalTransformation[
    FBTC[_, _]: Category: [_[_, _]] =>> ActingUpon[FBTC, TBTC],
    TBTC[_, _]: Category,
    FUTC[_]: [_[_]] =>> Functor[TBTC, FBTC, FUTC],
    TUTC[_]: [_[_]] =>> Functor[TBTC, TBTC, TUTC]
] extends Transformation[TBTC, FBTC, FUTC, TUTC]:

  // ...
```

`ActingUponNaturalTransformation` is a value class.

`ActingUponNaturalTransformation` has a parameter, `FBTC` that is required to be `Category` binary type constructor and an `ActingUpon` binary type constructor.

`ActingUponNaturalTransformation` has a parameter, `TBTC` that is required to be `Category` binary type constructor

`NaturalTransformation` has two parameters, `FUTC` and `TUTC` that are required to be `Functor` unary type constructors.

`Category` is fully explained in Category

`ActingUpon` is fully explained in ActingUpon

`Functor` is fully explained in Functor

`Transformation` is fully explained in Transformation

The laws of `ActedUpoNaturalTransformation`, `ActingUponNaturalTransformationLaws` are fully defined in ActingUponNaturalTransformationLaws.

Transformation

Back to PreThings

Back to NaturalTransformation

Back to ActingUponNaturalTransformation

```
package specification

trait Transformation[FBTC[_, _], TBTC[_, _], FUTC[_], TUTC[_]]:
```

`Transformation` is a value class.

```
  // declared

  def τ[Z]: TBTC[FUTC[Z], TUTC[Z]]
```

`Transformation` features are declared.

Back to ActingUponNaturalTransformation

Back to NaturalTransformation

Back to PreThings

Mathematical Foundations Domain: Laws

Law

Back to OrderedLaws

Back to TotallyOrderedLaws

Back to EqualityLaws

```
package specification

trait Law[UTC[_]]:
```

```
// ...
```

Law is a unary type constructor class for parameter UTC.

```
// ...

// declared

extension [Z, Y](lly: UTC[Y]) def `=>`(rlz: UTC[Z]): UTC[Z]

extension [Z](l: Z) def `=`(r: Z): UTC[Z]

extension [Z](ll: UTC[Z]) def `&`(rl: UTC[Z]): UTC[Z]

extension [Z](ll: UTC[Z]) def `|`(rl: UTC[Z]): UTC[Z]
```

Law features are declared.

Laws are *conditional equational* laws with *conjunction* and *disjunction*.

Back to EqualityLaws

Back to TotallyOrderedLaws

Back to OrderedLaws

OrderedLaws

Back to Ordered

```
// ...

// laws

trait OrderedLaws[L[_]: Law]:

  val reflexive: T => L[Boolean] = t =>
    {
      t `<=` t
    } `=` {
      true
    }

  val antiSymmetric: T => T => L[Boolean] = lt =>
    rt =>
      {
        lt `<=` rt `=` true `&` rt `<=` lt `=` true
      } `=>` {
        lt `=` rt `=` true
```

```
        }

      val transitive: T => T => T => L[Boolean] = lt =>
        mm =>
          rt =>
            {
              lt `<=` mm `=` true `&` mm `<=` rt `=` true
            } `=>` {
              lt `<=` rt `=` true
            }
```

Hopefully the laws do not surprise you.

Anyway, see, for example, Partially ordered sets.

`Law` is fully explained in Law.

Back to Ordered

TotallyOrderedLaws

Back to Ordered

```
    // ...

    // laws

    trait TotallyOrderedLaws[L[_]: Law]:

      val stronglyConnected: T => T => L[Boolean] = lt =>
        rt =>
          {
            lt `<=` rt `|` rt `<=` lt
          } `=` {
            true
          }
```

Hopefully the laws do not surprise you.

Anyway, see, for example, Total order for more details.

`Law` is fully explained in Law.

Back to Ordered

EqualityLaws

Back to Equality

```scala
// ...

// laws

trait EqualityLaws[L[_]: Law]:

  val reflexive: T => L[Boolean] = t =>
    {
      t `=` t
    } `=` {
      true
    }

  val symmetric: T => T => L[Boolean] = lt =>
    rt =>
      {
        lt `=` rt `=` true
      } `=>` {
        rt `=` lt `=` true
      }

  val transitive: T => T => T => L[Boolean] = lt =>
    mm =>
      rt =>
        {
          lt `=` mm `=` true `&` mm `=` rt `=` true
        } `=>` {
          lt `=` rt `=` true
        }
```

Hopefully the laws do not surprise you.

`Law` is fully explained in Law.

Back to Equality

MeetLaws

Back to Meet

```scala
// ...

// laws

trait MeetLaws[L[_]: Law]:

  val at = summon[Arbitrary[T]].arbitrary

  val greatestSmallerThanBoth: T => T => L[Boolean] =
    lt =>
      rt =>
```

```
          {
            ((lt ∧ rt) `<=` lt) `=` true `&` ((lt ∧ rt) `<=` rt) `=` true
          } `&` {
            (at `<=` lt) `=` true `&` (at `<=` rt) `=` true
          } `=>` {
            at `<=` (lt ∧ rt) `=` true
          }
```

Hopefully the laws do not surprise you.

Anyway, see, for example, Meet.

Back to Meet

JoinLaws

```
// ...

// laws

trait JoinLaws[L[_]: Law]:

  val at = summon[Arbitrary[T]].arbitrary

  val smallestGreaterThanBoth: T => T => L[Boolean] =
    lt =>
      rt =>
        {
          (lt `<=` (lt ∨ rt)) `=` true `&` (rt `<=` (lt ∨ rt)) `=` true
        } `&` {
          (lt `<=` at) `=` true `&` (rt `<=` at) `=` true
        } `=>` {
          (lt ∨ rt) `<=` at `=` true
        }
```

Hopefully the laws do not surprise you.

Anyway, see, for example, Join.

Back to Join

SupremumLaws

Back to Supremum

```
// ...

// laws

trait SupremumLaws[L[_]: Law]:
```

```
    val sets: Sets[Set] = summon[Sets[Set]]

    import sets.{map, all}

    val at = summon[Arbitrary[T]].arbitrary

    val smallestGreaterThanAll: Set[T] => L[Boolean] =
      ts =>
        {
          all apply {
            for {
              t <- ts
            } yield t `<=` sup(ts) `=` true
          }
        } `&` {
          all apply {
            for {
              t <- ts
            } yield t `<=` at `=` true
          }
        } `=>` {
          sup(ts) `<=` at `=` true
        }

    given join: Join[T]

    val joinAsSupremum: Tuple2[T, T] => L[T] =
      val sets = summon[Sets[Set]]
      import sets.{set2}
      (lt, rt) =>
        {
          sup(set2(lt, rt))
        } `=` {
          lt v rt
        }
```

Hopefully the laws do not surprise you.

Anyway, see, for example, Supremum.

Back to Supremum

SetsLaws

Back to Sets

```
  // ...

  // laws
```

```
    trait SetsLaws[L[_]: Law]:

      trait Set2Related:

        def unordered[Z]: Tuple2[Z, Z] => L[Set2[Z]] =
          (l, r) =>
            {
              set2(l, r)
            } `=` {
              set2(r, l)
            }

        import implementation.{functionCategory}

        def iso2[Z]: L[Function[Tuple2[Z, Z], Tuple2[Z, Z]]] = {
          identity[Tuple2[Z, Z]]
        } `=` {
          tuple2 `o` set2
        }

        def osi2[Z]: L[Function[Set2[Z], Set2[Z]]] = {
          identity[Set2[Z]]
        } `=` {
          set2 `o` tuple2
        }

      trait EqualityRelated:

        def reflexive[Z]: Set[Z] => L[Boolean] = s =>
          {
            s `=s=` s
          } `=` {
            true
          }

        def symmetric[Z]: Set[Z] => Set[Z] => L[Boolean] =
          l =>
            r =>
              {
                l `=s=` r `=` true
              } `=>` {
                r `=s=` l `=` true
              }

        def transitive[Z]: Set[Z] => Set[Z] => Set[Z] => L[Boolean] =
          l =>
            m =>
              r =>
                {
                  { l `=s=` m `=` true } `&` { m `=s=` r `=` true }
                } `=>` {
                  l `=s=` r `=` true
                }
```

```
  trait OrderedRelated:

    def reflexive[Z]: Set[Z] => L[Boolean] = s =>
      {
        s `<=s<=` s
      } `=` {
        true
      }

    def antiSymmetric[Z]: Set[Z] => Set[Z] => L[Boolean] =
      l =>
        r =>
          {
            { l `<=s<=` r } `=` true `&` { r `<=s<=` l } `=` true
          } `=>` {
            l `=s=` r `=` true
          }

    def transitive[Z]: Set[Z] => Set[Z] => Set[Z] => L[Boolean] =
      l =>
        m =>
          r =>
            {
              { l `<=s<=` m } `=` true `&` { m `<=s<=` r } `=` true
            } `=>` {
              { l `<=s<=` r } `=` true
            }
```

Hopefully the laws do not surprise you.

`Law` is fully explained in Law.

Back to Sets

FunctorLaws

Back to Functor

```
// ...

// laws

trait FunctorLaws[L[_]: Law]:

  def identity[Z]: L[TBTC[UTC[Z], UTC[Z]]] =
    val fbtc = summon[Category[FBTC]]
    val tbtc = summon[Category[TBTC]]
    φ(fbtc.ι[Z]) `=` tbtc.ι[UTC[Z]]

  def composition[Z, Y, X]
      : FBTC[Z, Y] => FBTC[Y, X] => L[TBTC[UTC[Z], UTC[X]]] =
    fzμy =>
```

```
        fyμx =>
          {
            φ(fyμx `o` fzμy)
          } `=` {
            φ(fyμx) `o` φ(fzμy)
          }
```

Hopefully the laws do not surprise you.

Anyway, see, for example, Functor.

Back to Functor

MonadPlusLaws

Back to MonadPlus

```
// ...

// laws

trait MonadPlusLaws[L[_]: Law]:

  def mappingOverZero[Z, Y]: Function[Z, Y] => L[UTC[Y]] =
    zφy =>
      {
        for {
          z <- ζ[Z]
        } yield {
          zφy(z)
        }
      } `=` {
        ζ
      }

  def mappingOverPlus[Z, Y]: Function[Z, Y] => UTC[Z] => UTC[Z] =>
L[UTC[Y]] =
    zφy =>
      lutc =>
        rutc =>
          {
            for {
              z <- lutc `+` rutc
            } yield {
              zφy(z)
            }
          } `=` {
            {
              for {
                lz <- lutc
              } yield {
                zφy(lz)
```

```
              }
            } `+` {
              for {
                lz <- rutc
              } yield {
                zφy(lz)
              }
            }
          }

    def flatMappingWithIdentityOverZero[Z]: L[UTC[Z]] = {
      for {
        utcz <- ζ[UTC[Z]]
        z <- utcz
      } yield {
        identity(z)
      }
    } `=` {
      ζ
    }

    def flatMappingWithPlus[Z]: UTC[UTC[Z]] => UTC[UTC[Z]] =>
L[UTC[UTC[Z]]] =
      lutcutcz =>
        rutcutcz =>
          {
            for {
              lutcz <- lutcutcz
              rutcz <- rutcutcz
            } yield {
              lutcz `+` rutcz
            }
          } `=` {
            lutcutcz `+` rutcutcz
          }
```

Hopefully the laws do not surprise you.

Anyway, see, for example, Monad

Back to MonadPlus

PlusLaws

Back to Plus

```
// ...

// laws

trait PlusLaws[L[_]: Law]:
```

```
    def leftZero[Z]: UTC[Z] => L[UTC[Z]] =
      utc =>
        {
          ζ `+` utc
        } `=` {
          utc
        }

    def rightZero[Z]: UTC[Z] => L[UTC[Z]] =
      utc =>
        {
          utc `+` ζ
        } `=` {
          utc
        }
```

Hopefully the laws do not surprise you.

Anyway, see, for example, Monad

Back to Plus

TripleLaws

Back to Triple

```
// ...

// laws

import specification.{Law}

trait TripleLaws[L[_]: Law]:

  import types.{`o`}

  import c.{ι}

  def associativity[Z]: L[BTC[(UTC `o` UTC `o` UTC)[Z], UTC[Z]]] = {
    μ `o` μ
  } `=` {
    μ `o` φ(μ)
  }

  def leftIdentity[Z]: L[BTC[UTC[Z], UTC[Z]]] = {
    μ `o` ν
  } `=` {
    ι
  }

  def rightIdentity[Z]: L[BTC[UTC[Z], UTC[Z]]] = {
    μ `o` φ(ν)
```

```
    } `=` {
      ι
    }
```

`o` is fully explained in Types.

Hopefully the requirements do not surprise you.

Anyway, see, for example, Monad (category theory) for more details.

Back to Triple

UtcCompositionLaws

Back to UtcComposition

```
 // ...

// laws

trait UtcCompositionLaws[L[_]: Law]:

  def associativity[Z]: UTC[Z] => UTC[Z] => UTC[Z] => L[UTC[Z]] =
    lutc =>
      mutc =>
        rutc =>
          {
            (lutc `+` mutc) `+` rutc
          } `=` {
            lutc `+` (mutc `+` rutc)
          }
```

Back to UtcComposition

CategoryLaws

Back to Category

```
// ...

// laws

trait CategoryLaws[L[_]: Law]:

  def leftIdentity[Z, Y]: BTC[Z, Y] => L[BTC[Z, Y]] = zμy =>
    {
      ι `o` zμy
    } `=` {
      zμy
    }
```

```
    def rightIdentity[Z, Y]: BTC[Z, Y] => L[BTC[Z, Y]] = zμy =>
      {
        zμy `o` ι
      } `=` {
        zμy
      }
```

Hopefully the laws do not surprise you.

Anyway, see, for example, Category.

Back to Category

BtcCompositionLaws

Back to BtcComposition

```
  // ...

  // laws

  trait CompositionLaws[L[_]: Law]:

    def associativity[Z, Y, X, W]
        : BTC[X, W] => BTC[Y, X] => BTC[Z, Y] => L[BTC[Z, W]] =
      xμw =>
        yμx =>
          zμy =>
            {
              (xμw `o` yμx) `o` zμy
            } `=` {
              xμw `o` (yμx `o` zμy)
            }
```

Hopefully the laws do not surprise you.

Anyway, see, for example, Category.

Back to BtcComposition

NaturalTransformationLaws

Back to NaturalTransformation

```
  // laws

  trait EqualityNaturalTransformationLaws[L[_]: Law](
      transformation: Transformation[FBTC, TBTC, FUTC, TUTC]
  ):
```

```
    val futc = summon[Functor[FBTC, TBTC, FUTC]]

    val tutc = summon[Functor[FBTC, TBTC, TUTC]]

    def natural[Z, Y](using
        equality: Equality[TBTC[FUTC[Z], TUTC[Y]]]
    ): FBTC[Z, Y] => L[Boolean] =
      fzφy =>
        {
          {
            transformation.τ `o` futc.φ(fzφy)
          } `=` {
            tutc.φ(fzφy) `o` transformation.τ
          }
        } `=` {
          true
        }

  trait OrderedNaturalTransformationLaws[L[_]: Law](
      transformation: Transformation[FBTC, TBTC, FUTC, TUTC]
  ):

    val futc = summon[Functor[FBTC, TBTC, FUTC]]

    val tutc = summon[Functor[FBTC, TBTC, TUTC]]

    def natural[Z, Y](using
        ordered: Ordered[TBTC[FUTC[Z], TUTC[Y]]]
    ): FBTC[Z, Y] => L[Boolean] =
      fzφy =>
        {
          {
            transformation.τ `o` futc.φ(fzφy)
          }
        } `<=` {
          {
            tutc.φ(fzφy) `o` transformation.τ
          }
        } `=` {
          true
        }
```

Hopefully the laws do not surprise you.

Anyway, see, for example, Natural Transformation.

Back to BtcComposition

Back to NaturalTransformation

ActingUponNaturalTransformationLaws

Back to ActingUponNaturalTransformation

```
  // ...

  // laws

  trait ActingUponNaturalTransformationLaws[L[_]: Law](
      transformation: NaturalTransformation[RBTC, LBTC, FUTC, TUTC]
  ):

    val futc = summon[Functor[RBTC, LBTC, FUTC]]

    val tutc = summon[Functor[RBTC, RBTC, TUTC]]

    def natural[Z, Y]: RBTC[Z, Y] => L[LBTC[FUTC[Z], TUTC[Y]]] =
      fzφy =>
        {
          transformation.τ `o` futc.φ(fzφy)
        } `=` {
          tutc.φ(fzφy) `a` transformation.τ
        }
```

Back to ActingUponNaturalTransformation

Mathematical Foundations Domain: Implementations

functionValuedFunctor2

Back to PreThings

```
package implementation

import specification.{Sets, Category, Functor}

given functionValuedFunctor2[
    Set[_]: Sets,
    BTC[_, _]: Category,
    UTC1[_]: [_[_]] =>> Functor[BTC, Function, UTC1]
]: Functor[
  BTC,
  Function,
  [Z] =>> Set[UTC1[Z]]
] with

  val sets: Sets[Set] = summon[Sets[Set]]

  val btcToFunctionFunctor = summon[Functor[BTC, Function, UTC1]]

  import types.{`o`}
```

```
  import sets.{Set2, tuple2, set2}

  type UTC2 = [Z] =>> (Set2 `o` UTC1)[Z]
  def φ[Z, Y]: BTC[Z, Y] => Function[UTC2[Z], UTC2[Y]] =
    zμy =>
      val zφy = btcToFunctionFunctor.φ(zμy)
      d =>
        tuple2(d) match
          case (l, r) => set2(zφy(l), zφy(r))
```

Back to PreThings

orderedCategory

Back to PreThings

```
package implementation

import specification.{Arbitrary, Ordered, Sets, Category}

given orderedCategory[Set[_]: Sets, T: Arbitrary: Ordered]
    : Category[[_, _] =>> Tuple2[T, T]] with

  type BTC = [_, _] =>> Tuple2[T, T]

  extension [Z, Y, X](yμx: BTC[Y, X])
    def `o`(zμy: BTC[Z, Y]): BTC[Z, X] =
      (yμx, zμy) match
        case ((llt, lrt), (rlt, rrt)) =>
          require(llt `<=` lrt && lrt == rlt && rlt `<=` rrt)
          (llt, rrt)

  def ι[Z]: BTC[Z, Z] =
    val at = summon[Arbitrary[T]].arbitrary
    (at, at)
```

Back to PreThings

functionCategory

Back to PreThings

```
package implementation

import specification.{Category, ActingUponFunction, Functor}

given functionCategory: Category[Function] with

  type BTC = [Z, Y] =>> Function[Z, Y]
```

```
  extension [Z, Y, X](yμx: BTC[Y, X])
    def `o`(zμy: BTC[Z, Y]): BTC[Z, X] = z => yμx(zμy(z))

  def ι[Z]: BTC[Z, Z] = z => z

given functionFunctionActingUpon: ActingUponFunction[Function] with

  type BTC = [Z, Y] =>> Function[Z, Y]

  def actionFunctor[Z]: Functor[BTC, Function, [Y] =>> Function[Z, Y]] =
    new:
      def φ[Y, X]: Function[
        BTC[Y, X],
        Function[Function[Z, Y], Function[Z, X]]
      ] = yμx => zμy => yμx `o` zμy
```

Back to PreThings

composedFunctor

Back to PreThingsLaws

```
package implementation

import specification.{Category, Functor}

import types.{`o`}

given composedFunctor[
    FBTC[_, _]: Category,
    MBTC[_, _]: Category,
    TBTC[_, _]: Category,
    F2MUTC[_]: [_[_]] =>> Functor[FBTC, MBTC, F2MUTC],
    M2TUTC[_]: [_[_]] =>> Functor[MBTC, TBTC, M2TUTC]
]: Functor[FBTC, TBTC, M2TUTC `o` F2MUTC] with

  type UTC = [Z] =>> (M2TUTC `o` F2MUTC)[Z]

  def φ[Z, Y]: Function[FBTC[Z, Y], TBTC[UTC[Z], UTC[Y]]] =
    summon[Functor[MBTC, TBTC, M2TUTC]].φ `o`
      summon[Functor[FBTC, MBTC, F2MUTC]].φ
```

Back to PreThingsLaws

functionTargetOrdered

Back to PreThingsLaws

```
package implementation

import specification.{Arbitrary, Ordered}

given functionTargetOrdered[Z: Arbitrary, Y: Ordered]: Ordered[Function[Z, Y]]
with

  val az: Z = summon[Arbitrary[Z]].arbitrary

  val ordered: Ordered[Y] = summon[Ordered[Y]]

  import ordered.{`=` => `=t=`, `<` => `<t<`}

  type T = Function[Z, Y]

  extension (lt: T) def `=`(rt: T): Boolean = lt(az) `=t=` rt(az)

  extension (lt: T) def `<`(rt: T): Boolean = lt(az) `<t<` rt(az)
```

Back to PreThingsLaws

setOrdered

Back to PreThingsLaws

```
package implementation

import specification.{Sets, Ordered}

given setOrdered[Z, Set[_]: Sets]: Ordered[Set[Z]] with

  val sets: Sets[Set] = summon[Sets[Set]]

  import sets.{`=s=`, `<s<`}

  type T = [Z] =>> Set[Z]

  extension (lt: T[Z]) def `=`(rt: T[Z]): Boolean = lt `=s=` rt

  extension (lt: T[Z]) def `<`(rt: T[Z]): Boolean = lt `<s<` rt
```

Back to PreThingsLaws