2021 International Conference on Data Analytics for Business and Industry (ICDABI) | 978-1-6654-1656-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICDABI53623.2021.9655926

Time Framework: A Type Level and Algebra Driven Design Approach

Soufiane Maguerra
LIM/IOS, FSTM
Hassan II University of Casablanca
Mohammedia, Morocco
maguerra.soufiane@gmail.com

Azedine Boulmakoul
LIM/IOS, FSTM
Hassan II University of Casablanca
Mohammedia, Morocco
azedine.boulmakoul@gmail.com

Hassan Badir
IDS Team
Abdelmalek Essaâdi University
Tangier, Morocco
badir.hassan@uae.ac.ma

***Abstract*—Time is present in every real world data model. Literature on the subject includes several time models, algebras, formal logics, and ontologies. Furthermore, Functional Programming, Type-level Programming and Algebra Driven Design are valuable assets for constructing systems that are consistent, composable, extensible, highly polymorphic, and predictable. Nevertheless, there is a lack for a purely functional typed time framework that offers ways to construct type safe algebraic time structures and prove their algebraic laws. Our study answers this need by introducing a purely functional typed, highly extensible and polymorphic framework to encode time algebras. The framework can be used to build type safe, composable systems for processing time points and periods with laws as consistency proofs and compile-time optimisation rules. The language used to implement our library is Haskell, which is a purely functional, statically and strongly typed language. Additionally, the library includes dependent types, compile-time reflection, type families, and refined types. As for proofs, they are written as randomised property tests with QuickSpec, or enumerative property tests with SmallCheck. To demonstrate the utility of the framework, we provide two use cases that describe how to construct a time point and period algebraic structure, and how to assert that laws about them hold.**

***Index Terms*—functional programming, abstract algebra, logic, software libraries**

I. Introduction

Time is a valuable source of knowledge describing change, and is present in every data processing system. All over the world sensors and generators are producing instantaneously temporal data in massive manner. Each data has a particular nature and logic. In the past decades, researchers have studied the subject intensively and proposed different time models, formal logics, theories, and ontologies.

In addition, purely functional typed languages are leveraged to build formal programs that are composable, correct, safe, predictable, highly extensible and polymorphic. Also, Algebra Driven Design is an important approach to build systems supported by mathematical proofs to ensure consistency and a behaviour that is expected. Each of these approaches is invaluable to build software that have a mathematical foundation and are highly abstract, flexible, reliable, easy to understand and extend.

Despite the features stated above, no purely functional, typed framework to construct algebraic time structures exists in the literature. The objective of our study is to fill this gap with the following contribution:

- An original type-safe, purely functional, time framework [1] to construct, transform, and prove laws over algebraic time points and periods.

Our framework is a library that can be used to encode different kinds of time points and period algebras, depending on the users' needs. These algebras can be used to build type safe, composable, and consistent time-related systems. Our library is implemented using the purely functional, statically strongly typed language Haskell. The implementation includes powerful extensions such as types families to reach a higher polymorphism, singletons for dependent types, refinement types to add type-level predicates, data kinds for type safety, and Template Haskell for compile-time reflection. As for writing mathematical proofs, QuickSpec and Small Check are used to respectively write randomised and enumerative property tests. The utility of our framework is demonstrated in two use cases that describe the process of constructing time point and period algebraic structures, and how to use our predefined algebraic laws to ensure that they are consistent with a specific algebraic structure.

The remainder of this paper is organised as follows: Section 2 discusses related works, Section 3 describes our library, Section 4 provides use cases for it, and Section 5 serves as a conclusion and perspectives.

II. Literature

Research on the subject of time is divided into ontologies, algebras, logics, and time frameworks that focus on time points or periods. The scope of our study is limited to time frameworks but to have more context and understand our interest in the subject and the foundation of our study we still provide information about the other research fields.

A. Time Ontologies

Time Ontologies can be classified into general purpose ontologies, or ontologies focused on a specific domain or designed to be part of a particular project. The ontologies BFO, DOLCE, and GFO are well-known upper-level ontologies. In addition, OWL-Time, PSI-Time, Reusable Time, and SWRL Temporal are all focused time ontologies each having a specific utility. As an example, PSI Time was designed to be part of the Performance Simulation Initiative (PSI) project, and SWRL Temporal to model time periods in OWL. The list of time ontologies is not limited to the above mentions, more details can be found in the review [2].

Ontologies are a great tool to model knowledge but, in our opinion, they are more adequate for semantic web applications and knowledge bases. In contrast, our framework is oriented towards building general-purpose, type-safe, purely functional, algebra driven design, and time-related systems systems .

B. Time Algebras

One of the most known time algebras is Allen's interval algebra [3]. Allen defined a set of time period relations and laws describing them. Moreover, Allen and Hayes gave a logical axiomatisation for these relations in [4]. Allen's algebra is widely used, made the basis for a lot of studies, and made it as an ontology [5] too. For this reasons, our work included Allen's time period relations and laws.

The relations before, after, and equals are widely used to compare time points. Other works provided more relations or extensions. Such as [6] who added granularity, and [7] who provided a fuzzy extensions to the basic relations. Our library, includes these basic relations, among others, and offers the possibility to include other relations in need.

Van Benthem conducted a model theoretic study of time [8]. He studied both time points and periods and offered relations and laws between them. Our work is heavily influenced by his research. Moreover, another set of operators has a logical nature. These operators assert the truth of temporal facts, e. g., exists, sometimes, and always. Although not part of our library, they might be included in a future release.

C. Time Frameworks

Purely functional languages are highly composable and have denotational semantics that means they have a formal mathematical description. Moreover, types provide an additional layer of safety and ensure that our program behaves as expected. For these reasons, the scope of our study lays in purely functional libraries that are written in strongly statically typed language.

Time [9] is a general purpose Haskell date time library that provides predefined data structures and date time formats. The library inspired others such as HodaTime [10] and Chronos [11]. HodaTime tries to anticipate users' needs and be a batteries included package, while Chronos is more performance oriented. Another Haskell date time library is O'Clock [12]. The library is type-safe and provides a way to construct custom time units that are stored in the form of rationals. Timelike [13] is another Haskell library that provides type classes for date time arithmetics and can be used in conjunction with the other date time libraries.

As for time periods, Intervals [14] provides interval arithmetic but only for closed ones, and Interval Algebra [15] implements Allen's Algebra but is also only for open periods. Data Interval [16] overcomes the product type limit and offers functions for both open and closed intervals but the product type has a runtime representation, which can impose an additional memory footprint. Furthermore, these libraries heavily restrict the use of order relation to total orders.

Compared to the above, our library provides higher abstractions for both time points and periods. The library has a lightweight nature, only the minimum necessary is asked to construct time algebras that allow to build type-safe systems with tests to ensure that the algebraic structures are consistent. Moreover, all product types either open, closed, right closed, and left closed are accepted. The product types have a compile-time representation ensuring type safety and a low runtime memory footprint. No constraints are imposed on library users giving them the freedom of choosing the order and constraints they need for their algebra. Be it, a strict partial, partial, linear, or total order. All by allowing the possibility to define their own algebraic structures and integrate them with the existing ones.

III. Time Framework

Our framework is published in [1]. It publishes two modules Data and Test. In Data, constructors and observations over time points and periods can be found. As for Test, it contains type classes and law specs describing algebraic structures. The laws are written either in the form of randomised QuickCheck property tests or enumerative SmallCheck property tests. QuickCheck is used for universal quantified properties and SmallCheck for existential ones because randomised property tests are not meant for asserting existential properties. The next subsections explain how time points and periods are implemented and how the laws are written.

A. Chronon

In our library, time points named as chronons are instance of a data type family. The *Chronon* type family is polykinded. This means it makes it possible to have instances of *Chronon* with a different number of type variables, which can range from * to n. The observations are encoded in the form of type classes, which can be overloaded by the user depending on the *Chronon* instance. Each type class is related to a set of operations relative to a specific time data structure. The *Data.Chronon* module leverages both type and function ad-hoc polymorphism. Please refer to the listing 1 for the implementation.

```
module Data.Chronon

data family Chronon :: k

-- | Observations
class ChrononObs t where
  (<) :: t -> t -> Bool

  (>) :: t -> t -> Bool
  x > y = y < x

  -- | x is between y z
  betweenness :: t -> t -> t -> Bool
  betweenness x y z = (y < x && x < z) || (z < x &&
  ↪  x < y)
  (===) :: t -> t -> Bool

  (<=) :: Eq t => t -> t -> Bool
  x <= y = x < y || x == y

  (>=) :: Eq t => t -> t -> Bool
  x >= y = y <= x

class CyclicChronon t where
  cycle :: t -> t -> t -> Bool

class SynchronousChronon t where
  synchronous :: t -> t -> Bool

class ConcurrentChronon t where
  concurrent :: t -> t -> Bool
```

Listing 1: The Chronon Module.

As for mathematical proofs, let us consider a Partial Cyclic Order described via a ternary relation that respects these laws:

- Cyclic: $\forall x\, y\, z.\ cycle\ x\ y\ z \Rightarrow cycle\ y\ z\ x$
- Asymmetric: $\forall x\, y\, z.\ cycle\ x\ y\ z \Rightarrow \neg\, cycle\ z\ y\ x$
- Transitive: $\forall x\, y\, z.\ cycle\ x\ y\ z \wedge cycle\ x\ z\ h \Rightarrow cycle\ x\ y\ h$

The listing 2 describes the PartialCyclicOrder and shows how the laws are encoded as QuickCheck properties.

```
class PartialCyclicOrder a where
  cycle :: a -> a -> a -> Bool

type Constraints t = (Arbitrary t,
↪  PartialCyclicOrder t, Show t)

prop_cyclic :: forall t. Constraints t => Property
prop_cyclic = forAll condition $ \(x, y, z) -> cycle
↪  y z x
  where condition :: Gen (t, t, t)
        condition = suchThat arbitrary $ \(x, y, z)
        ↪  -> cycle x y z

prop_asymmetric :: forall t. Constraints t =>
↪  Property
prop_asymmetric = forAll condition $
    \(x, y, z) -> not $ cycle z y x
  where condition :: Gen (t, t, t)
        condition = suchThat arbitrary $ \(x, y, z)
        ↪  -> cycle x y z

prop_transitive :: forall t. Constraints t =>
↪  Property
prop_transitive = forAll gen $ \(x, y, _, h) ->
↪  cycle x y h
  where gen :: Gen (t, t, t, t)
        gen = suchThat arbitrary $ \(x, y, z, h) ->
        ↪  cycle x y z && cycle x z h

--------------------------

laws :: forall t. Constraints t => [(String,
↪  Property)]
laws =
  [ ("Cyclic", prop_cyclic @t)
  , ("Asymmetric", prop_asymmetric @t)
  , ("Transitive", prop_transitive @t)
  ]
```

Listing 2: PartialCyclicOrder Type Class and Laws.

Another example is the existence of a beginning, described by the law:

$$\exists x.\forall y.\ \neg y < x \qquad (1)$$

The law contains an existential quantifier. SmallCheck is used to encode it, and the encoding can be seen in the listing 3.

```
type Constraints m t = (Serial m t, Begin t, Show t)

-- | begin Properties :
prop_begin :: forall m t. Constraints m t =>
↪  Property m
prop_begin = exists $ \(x :: t) -> forAll $ \(y ::
↪  t) -> not $ y < x

--------------------------------

laws :: forall m t. Constraints m t => [(String,
↪  Property m)]
laws = [ ("Begin", prop_begin @m @t) ]
```

Listing 3: Begin Laws.

Different algebraic structures are part of the library including partial, linear, total, and cyclic. In addition, laws about relations such as identity, concurrency, betweenness, and synchronicity. Also, assumptions about the time structure such as the existence of a beginning or end. Given the user a wide choice with the possibility to define new ones.

B. Period

A period is an abstract data type (please see listing4) with two accessors *inf* to access the greatest lower bound, and *sup* to access the least upper bound. Each period can either be open, closed, right closed, or left closed. These product types are promoted and used as type arguments. Moreover, the singletons [17] library is used to introduce dependent types programming in Haskell. In particular, the library maps each promoted product type to a singleton runtime value that is used by the smart constructors to create the adequate products, i. e., *SClosed* is created and mapped to the *'Closed* type. Note that the singleton is used only when constructing the value. After that the information about the product type is only visible at compile time and this reduces the runtime memory footprint. There are two type of constructors, one for compile-time and the other for run-time validation. This gives the user freedom to choose the constructor suitable for his needs. Moreover, both constructors are implemented using the refined library [18]. Refined adds type level predicates to ensure that only valid periods are created, i. e., a period's supremum should not precede the infimum. As for compile-time validation, Template Haskell is used for reflection.

```
module Data.Period

data PeriodType = Open | Closed | RightClosed |
↪  LeftClosed

$(genSingletons [''PeriodType])

data Period (c :: PeriodType) t = Period { inf :: t,
↪  sup :: t }
```

```
-- | Exceptions
data InvalidPeriod = InvalidPeriod deriving (Show)

instance Exception InvalidPeriod where
  displayException _ = "Period Bounds are
  ↪  Invalid!!!"

-- | Refinement
data ValidPeriodBounds

instance ChrononObs t => Predicate
  ValidPeriodBounds (Period c t) where
  validate _ (Period x y)
    | (C.<) x y = Nothing
    | otherwise = throwRefineSomeException
        (typeRep (Proxy :: Proxy ValidPeriodBounds))
        (SomeException InvalidPeriod)

-- | Smart Constructor
period
  :: Predicate ValidPeriodBounds (Period c t)
  => SPeriodType c
  -> t
  -> t
  -> Either RefineException (Period c t)
period _ x y = (refine_ @ValidPeriodBounds) $ Period
↪  x y

periodTH
  :: (ChrononObs t, Lift (Period c t))
  => SPeriodType c
  -> t
  -> t
  -> Q (TExp (Period c t))
periodTH _ x y = (refineTH_ @ValidPeriodBounds) $
↪  Period x y
```

Listing 4: The Period Abstract Data Type.

Same as chronons, type classes describe periods' observations. There are three types of observations in our library: observations between periods of the same product type, observations between periods of different period types, and observations between periods and chronons. The listing 5 shows the observations between periods of the same period type, please refer to the source code for more details [1]. Moreover, intuitively periods can be seen as a set of chronons and we want to have that encoded in the library. In other terms, we want the period's time type parameter to be a chronon. To achieve this, we used a type constraint in the observations' type class. This way to be able to instantiate the periods' observations type class one has to have chronon observations defined for the period's time type parameter. Of course this assumes that a time data structure is defined by its observations.

```
-- | Period Observations
class ChrononObs t => PeriodObs (c :: PeriodType) t
↪  where
  (<) :: Period c t -> Period c t -> Bool

  (>) :: Period c t -> Period c t -> Bool
  x > y = y < x

  (==) :: Eq t => Period c t -> Period c t -> Bool
  x == y = (Pr.==) (inf x) (inf y) && (Pr.==) (sup
  ↪  x) (sup y)

  (===) :: Period c t -> Period c t -> Bool
  x === y = (C.===) (inf x) (inf y) && (C.===) (sup
  ↪  x) (sup y)

  starts :: Eq t => Period c t -> Period c t -> Bool
  starts x y = (Pr.==) (inf x) (inf y) && (C.<) (sup
  ↪  x) (sup y)

  finishes :: Eq t => Period c t -> Period c t ->
  ↪  Bool
  finishes x y = (C.<) (inf y) (inf x) && (Pr.==)
  ↪  (sup x) (sup y)

  during :: Period c t -> Period c t -> Bool
  during x y = (C.<) (inf y) (inf x) && (C.<) (sup
  ↪  x) (sup y)

  contains :: Period c t -> Period c t -> Bool
  contains x y = not $ during x y

  includedIn :: Eq t => Period c t -> Period c t ->
  ↪  Bool
  includedIn x y = during x y || x == y

  overlaps :: Period c t -> Period c t -> Bool
  overlaps x y = (C.<) (inf x) (inf y) && (C.<) (inf
  ↪  y) (sup x) && (C.<) (sup x) (sup y)

  meets :: Eq t => Period 'Closed t -> Period
  ↪  'Closed t -> Bool
  meets x y = (Pr.==) (sup x) (inf y)
```

Listing 5: Period Observations.

In the encoding process of an algebraic period structure, the user can leverage the predefined laws, same as for chronons, defined in the Test module to ensure consistency over the defined structure.

IV. Use Cases

The framework used to organise the test suites is Tasty [19]. The next subsections describe how to create respectively chronon and period algebraic structures and prove laws about them. For the full spec, Please refer to the source code [1].

A. Chronon Example

In the listing 6, a chronon is simply an integer associated with the *precedence* and *identity* relations. The chronon and *precedence* operator are checked if they form a strict partial order.

```
data instance Chronon = Chronon Int deriving (Eq,
↪  Show)

-- | Observations
instance ChrononObs Chronon where
  Chronon x < Chronon y = (P.<) x y
  Chronon x === Chronon y = (P.==) x y

-- | Axiomatization
instance StrictPartialOrder Chronon where
  (<) = (Chronon.<)

-- | spec
orderLaws :: TestTree
orderLaws = testGroup "Order Laws"
  [
    QC.testProperties "StrictPartialOrder" $
    ↪  StrictPartialOrder.laws @Chronon
  ]
```

Listing 6: Example of a Chronon Structure.

B. Period Example

In the listing 7, a period is a closed set of integers, and the spec asserts that the relation *includedIn* with the period form a partial order.

```
-- | Data
data instance Chronon Int = Chronon Int
newtype IntChronon = IntChronon (Chronon Int)

newtype ClosedPeriod = ClosedPeriod (Period 'Closed
↪  IntChronon)

-- | Observations
instance ChrononObs IntChronon where
  IntChronon (Chronon x) < IntChronon (Chronon y) =
  ↪  (Pr.<) x y
  IntChronon (Chronon x) === IntChronon (Chronon y)
  ↪  = (Pr.==) x y

instance Eq IntChronon where
  IntChronon (Chronon x) == IntChronon (Chronon y) =
  ↪  (Pr.==) x y

instance Eq (Period 'Closed IntChronon) where
  x == y = (Pr.==) (inf x) (inf y) && (Pr.==) (sup
  ↪  x) (sup y)

instance Eq ClosedPeriod where
  ClosedPeriod x == ClosedPeriod y = (Pr.==) x y

-- | Axiomatization
instance StrictPartialOrder ClosedPeriod where
  ClosedPeriod x < ClosedPeriod y = (Period.<) x y

instance PartialOrder ClosedPeriod where
  ClosedPeriod x <= ClosedPeriod y =
  ↪  Period.includedIn x y

-- | spec
spec :: TestTree
spec = testGroup "Closed Period Spec" [ orderLaws ]

orderLaws :: TestTree
orderLaws = testGroup "Order Laws"
  [
    QC.testProperties "IncludedIn PartialOrder" $
    ↪  PartialOrder.laws @ClosedPeriod
  ]
```

Listing 7: Example of a Period Structure.

Note that the spec leverages the predefined definition of observations for closed periods, please see listing 8. Our library already includes useful instances describing period observations because logically there will be only one definition for those particular cases. Still, if the user wishes to use a different description, then he can simply define a new type class or *newtype* and describe it.

```
instance ChrononObs t => PeriodObs 'Closed t where
  x < y = (C.<) (sup x) (inf y)
```

Listing 8: Closed Period Observations.

V. Conclusion and Perspectives

Initially, our study was focused on building a framework for trajectory processing but since time is an essential part of every trajectory and has wide use in other applications we decided for a modular approach and extract the time framework. The purely functional typed, and algebra driven design approach enables our time framework to be integrated in a wide range of real world application, and form type safe reliable systems. Our future works will be in both in the time and trajectory domain. We aim to introduce additional logical operators and algebraic structures to the time library.

Acknowledgment

The authors thank the University of Bahrain for supporting the publication of this paper.

References

[1] S. Maguerra, "time-framework : Type safe, algebra driven design time framework," https://github.com/xsoufiane/time-framework, 2021, v0.0.1. Accessed: 2021-08-31.
[2] V. Ermolayev, S. Batsakis, N. Keberle, O. Tatarintseva, and G. Antoniou, "Ontologies of time: Review and trends." *International Journal of Computer Science & Applications*, vol. 11, no. 3, 2014.
[3] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.
[4] J. F. Allen and P. J. Hayes, "Moments and points in an interval-based temporal logic," *Computational Intelligence*, vol. 5, no. 3, pp. 225–238, 1989.
[5] M. Grüninger and Z. Li, "The time ontology of allen's interval algebra," in *24th International Symposium on Temporal Representation and Reasoning (TIME 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
[6] C. Bettini, X. S. Wang, S. Jajodia, and J.-L. Lin, "Discovering frequent event patterns with multiple granularities in time sequences," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10, no. 2, pp. 222–237, 1998.
[7] D. Dubois and H. Prade, "Processing fuzzy temporal knowledge," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 19, no. 4, pp. 729–744, 1989.
[8] J. Van Benthem, *The logic of time: a model-theoretic investigation into the varieties of temporal ontology and temporal discourse*. Springer Science & Business Media, 2013, vol. 156.
[9] A. Yakeley, "Time: A time library," https://hackage.haskell.org/package/time-1.12, 2021, v1.12. Accessed: 2021-08-30.
[10] J. Johnson, "hodatime: A fully featured date/time library based on nodatime," https://hackage.haskell.org/package/hodatime, 2020, v0.2.1.1. Accessed: 2021-08-30.
[11] A. Martin, "chronos: A high-performance time library," https://hackage.haskell.org/package/chronos, 2021, v1.1.2. Accessed: 2021-08-30.
[12] Serokell, "o-clock: Type-safe time library," https://hackage.haskell.org/package/o-clock, 2021, v1.2.1. Accessed: 2021-08-30.
[13] E. Söylemez, "timelike: Type classes for types representing time," https://hackage.haskell.org/package/timelike, 2016, v0.2.2. Accessed: 2021-08-30.
[14] E. Kmett, "intervals: Interval arithmetic," https://hackage.haskell.org/package/intervals, 2021, v0.9.2. Accessed: 2021-08-30.

[15] B. Saul, “interval-algebra: An implementation of allen’s interval algebra for temporal logic,” https://hackage.haskell.org/package/interval-algebra, 2021, v0.10.2. Accessed: 2021-08-30.

[16] M. Sakai, “data-interval: Interval datatype, interval arithmetic and interval-based containers,” https://hackage.haskell.org/package/data-interval, 2021, v2.1.0. Accessed: 2021-08-30.

[17] R. A. Eisenberg and S. Weirich, “Dependently typed programming with singletons,” *ACM SIGPLAN Notices*, vol. 47, no. 12, pp. 117–130, 2012.

[18] N. Volkov, “refined: Refinement types with static and runtime checking,” https://hackage.haskell.org/package/refined, 2021, v0.6.2. Accessed: 2021-08-31.

[19] R. Cheplyaka, “tasty: Modern and extensible testing framework,” https://hackage.haskell.org/package/tasty, 2021, v1.4.2. Accessed: 2021-08-31.