Estrutura de tempo: um nível de tipo e álgebra Abordagem de Design Orientado

Soufiane Maguerra
LIM/IOS, FSTM
Hassan II Universidade de Casablanca
Mohammedia, Marrocos
maguerra.soufiane@gmail.com

Azedine Boulmakoul
LIM/IOS, FSTM
Hassan II Universidade de Casablanca
Mohammedia, Marrocos
azedine.boulmakoul@gmail.com

Hassan Badir
Equipe IDS
Universidade Abdelmalek Essaadi
Tânger, Marrocos
badir.hassan@uae.ac.ma

Resumo—O tempo está presente em todos os modelos de dados do mundo real. A literatura sobre o assunto inclui vários modelos de tempo, álgebras, lógicas formais e ontologias. Além disso, a Programação Funcional, a Programação em Nível de Tipo e o Design Orientado à Álgebra são recursos valiosos para a construção de sistemas consistentes, combináveis, extensíveis, altamente polimórficos e previsíveis.

No entanto, há uma falta de uma estrutura de tempo tipada puramente funcional que ofereça maneiras de construir estruturas de tempo algébricas seguras e provar suas leis algébricas. Nosso estudo responde a essa necessidade introduzindo uma estrutura tipificada puramente funcional, altamente extensível e polimórfica para codificar álgebras de tempo. A estrutura pode ser usada para criar sistemas combináveis e de tipo seguro para processamento de pontos de tempo e períodos com leis como provas de consistência e regras de otimização de tempo de compilação. A linguagem utilizada para implementar nossa biblioteca é Haskell, que é uma linguagem puramente funcional, estaticamente e fortemente tipada. Além disso, a biblioteca inclui tipos dependentes, reflexão em tempo de compilação, famílias de tipos e tipos refinados. Quanto às provas, elas são escritas como testes de propriedades aleatórias com QuickSpec ou testes de propriedades enumerativas com SmallCheck. Para demonstrar a utilidade da estrutura, fornecemos dois casos de uso que descrevem como construir um ponto no tempo e uma estrutura algébrica de período e como afirmar que as leis sobre eles são válidas.

Termos do Índice—programação funcional, álgebra abstrata, lógica, bibliotecas de software

I. INTRODUÇÃO

O tempo é uma fonte valiosa de conhecimento que descreve a mudança e está presente em todos os sistemas de processamento de dados. Em todo o mundo, sensores e geradores estão produzindo dados temporais instantaneamente de maneira massiva. Cada dado tem uma natureza e uma lógica particulares. Nas últimas décadas, os pesquisadores estudaram o assunto intensamente e propuseram diferentes modelos de tempo, lógicas formais, teorias e ontologias.

Além disso, linguagens tipadas puramente funcionais são aproveitadas para criar programas formais que podem ser compostos, corretos, seguros, previsíveis, altamente extensíveis e polimórficos. Além disso, o Algebra Driven Design é uma abordagem importante para construir sistemas apoiados por provas matemáticas para garantir a consistência e um comportamento esperado. Cada uma dessas abordagens é valiosa para construir softwares que tenham uma base matemática e sejam altamente abstratos, flexíveis, confiáveis, fáceis de entender e estender.

Apesar dos recursos declarados acima, não existe uma estrutura tipificada puramente funcional para construir estruturas de tempo algébricas na literatura. O objetivo do nosso estudo é preencher essa lacuna com a seguinte contribuição: • Uma estrutura de tempo original e puramente funcional [1] para construir, transformar e provar leis sobre pontos e períodos de tempo algébricos.

Nosso framework é uma biblioteca que pode ser usada para codificar diferentes tipos de pontos no tempo e álgebras de período, dependendo das necessidades dos usuários. Essas álgebras podem ser usadas para construir sistemas relacionados ao tempo consistentes, combináveis e seguros. Nossa biblioteca é implementada usando a linguagem Haskell puramente funcional e estaticamente fortemente tipada. A implementação inclui extensões poderosas, como famílias de tipos para alcançar um polimorfismo mais alto, singletons para tipos dependentes, tipos de refinamento para adicionar predicados de nível de tipo, tipos de dados para segurança de tipo e Template Haskell para reflexão em tempo de compilação. Quanto à escrita de provas matemáticas, QuickSpec e Small Check são usados para escrever, respectivamente, testes de propriedade aleatórios e er A utilidade de nossa estrutura é demonstrada em dois casos de uso que descrevem o processo de construção de estruturas algébricas de ponto e período no tempo e como usar nossas leis algébricas predefinidas para garantir que sejam consistentes com uma estrutura algébrica específica.

O restante deste documento está organizado da seguinte forma: a Seção 2 discute trabalhos relacionados, a Seção 3 descreve nossa biblioteca, a Seção 4 fornece casos de uso para ela e a Seção 5 serve como conclusão e perspectivas.

II. LITERATURA

A pesquisa sobre o assunto do tempo é dividida em ontologias, álgebras, lógicas e estruturas de tempo que se concentram em pontos ou períodos no tempo. O escopo de nosso estudo é limitado a prazos, mas para ter mais contexto e entender nosso interesse no assunto e a fundamentação de nosso estudo, ainda fornecemos informações sobre os outros campos de pesquisa.

A. Ontologias de tempo

Ontologias de tempo podem ser classificadas em ontologias de propósito geral, ou ontologias focadas em um domínio específico ou projetadas para fazer parte de um projeto particular. As ontologias BFO, DOLCE e GFO são ontologias de nível superior bem conhecidas. Além disso, OWL-Time, PSI-Time, Reusable Time e SWRL Temporal são todas ontologias de tempo focadas, cada uma com uma função específica.

978-1-6654-1656-6/21/$31,00 ©2021 IEEE

Utilitário. Como exemplo, PSI Time foi projetado para fazer parte do projeto Performance Simulation Initiative (PSI), e SWRL Temporal para modelar períodos de tempo em OWL. A lista de ontologias de tempo não se limita às menções acima, mais detalhes podem ser encontrados na revisão [2].

As ontologias são uma ótima ferramenta para modelar conhecimento, mas, em nossa opinião, são mais adequadas para aplicações web semânticas e bases de conhecimento. Em contraste, nossa estrutura é orientada para a construção de sistemas de uso geral, type-safe, puramente funcionais, orientados à álgebra e sistemas relacionados ao tempo.

B. Álgebras de tempo

Uma das álgebras de tempo mais conhecidas é a álgebra de intervalos de Allen [3]. Allen definiu um conjunto de relações de período de tempo e leis que os descrevem. Além disso, Allen e Hayes deram uma axiomatização lógica para essas relações em [4]. A álgebra de Allen é amplamente utilizada, serviu de base para muitos estudos e também se tornou uma ontologia [5]. Por esta razão, nosso trabalho incluiu as relações e leis do período de tempo de Allen.

As relações antes, depois e iguais são amplamente utilizadas para comparar pontos no tempo. Outros trabalhos forneceram mais relações ou extensões. Como [6] que adicionou granularidade e [7] que forneceu extensões difusas para as relações básicas. Nossa biblioteca inclui essas relações básicas, entre outras, e oferece a possibilidade de incluir outras relações necessárias.

Van Benthem conduziu um modelo teórico de estudo do tempo [8]. Ele estudou pontos de tempo e períodos e ofereceu relações e leis entre eles. Nosso trabalho é fortemente influenciado por sua pesquisa. Além disso, outro conjunto de operadores tem natureza lógica. Esses operadores afirmam a verdade de fatos temporais, por exemplo, existe, às vezes e sempre. Embora não façam parte de nossa biblioteca, eles podem ser incluídos em uma versão futura.

C. Prazos

Linguagens puramente funcionais são altamente compostas e possuem semântica denotacional, o que significa que possuem uma descrição matemática formal. Além disso, os tipos fornecem uma camada adicional de segurança e garantem que nosso programa se comporte conforme o esperado. Por essas razões, o escopo de nosso estudo está em bibliotecas puramente funcionais que são escritas em linguagem fortemente tipada estaticamente.

Time [9] é uma biblioteca Haskell de data e hora de uso geral que fornece estruturas de dados e formatos de data e hora predefinidos. A biblioteca inspirou outras como HodaTime [10] e Chronos [11]. O HodaTime tenta antecipar as necessidades dos usuários e ser um pacote de baterias incluídas, enquanto o Chronos é mais orientado para o desempenho. Outra biblioteca Haskell de data e hora é O'Clock [12]. A biblioteca é type-safe e fornece uma maneira de construir unidades de tempo personalizadas que são armazenadas na forma de racionais. Timelike [13] é outra biblioteca Haskell que fornece classes de tipo para aritmética de data e hora e pode ser usada em conjunto com outras bibliotecas de data e hora.

Quanto aos períodos de tempo, Intervals [14] fornece aritmética de intervalo, mas apenas para períodos fechados, e Interval Algebra [15] implementa a Álgebra de Allen, mas também é apenas para períodos abertos. Intervalo de dados [16] supera o limite do tipo de produto e oferece funções para intervalos abertos e fechados, mas o tipo de produto tem uma representação de tempo de execução, que pode impor um consumo de memória adicional. Além disso, essas bibliotecas restringem fortemente o uso da relação de pedidos para pedidos totais.

Em comparação com o anterior, nossa biblioteca fornece abstrações mais altas para pontos de tempo e períodos. A biblioteca tem uma natureza leve, apenas o mínimo necessário é solicitado para construir álgebras de tempo que permitem construir sistemas type-safe com testes para garantir que as estruturas algébricas sejam consistentes. Além disso, todos os tipos de produtos abertos, fechados, fechados à direita e fechados à esquerda são aceitos. Os tipos de produto têm uma representação de tempo de compilação garantindo a segurança do tipo e um baixo consumo de memória de tempo de execução. Nenhuma restrição é imposta aos usuários da biblioteca, dando-lhes a liberdade de escolher a ordem e as restrições necessárias para sua álgebra. Seja uma ordem estrita parcial, parcial, linear ou total. Tudo ao permitir a possibilidade de definir suas próprias estruturas algébricas e integrá-las com as existent uns.

III. ESTRUTURA DE TEMPO

Nosso framework está publicado em [1]. Publica dois módulos Data e Test. Em Dados, construtores e observações sobre pontos e períodos de tempo podem ser encontrados. Quanto ao Test, ele contém classes de tipo e especificações de lei que descrevem estruturas algébricas. As leis são escritas na forma de testes de propriedade QuickCheck aleatórios ou testes de propriedade enumerativos SmallCheck.
O QuickCheck é usado para propriedades quantificadas universais e o SmallCheck para propriedades existenciais porque os testes de propriedade aleatórios não se destinam a afirmar propriedades existenciais. As próximas subseções explicam como os pontos e períodos de tempo são implementados e como as leis são escritas.

A. Chronon

Em nossa biblioteca, os pontos de tempo nomeados como cronons são instâncias de uma família de tipos de dados. A família de tipos Chronon é polykinded. Isso significa que é possível ter instâncias do Chronon com um número diferente de variáveis de tipo, que podem variar de * a n. As observações são codificadas na forma de classes de tipo, que podem ser sobrecarregadas pelo usuário dependendo da instância do Chronon. Cada classe de tipo está relacionada a um conjunto de operações relativas a uma estrutura de dados de tempo específica.
O módulo Data.Chronon aproveita o polimorfismo ad-hoc de tipo e função. Consulte a listagem 1 para a implementação.

```
módulo Data.Chronon

família de dados Chronon :: k

-- | Observações classe
ChrononObs t onde
    (<) :: t -> t -> Bool

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

    -- | x está entre yz intermediação ::
    t -> t -> t -> Bool intermediação xyz = (y < x && x < z) || (z
    < x && ÿ x < y)
```

```
    (===) :: t -> t -> Bool

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


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

classe CyclicChronon t onde
    ciclo :: t -> t -> t -> Bool

class SynchronousChronon t onde síncrono :: t -> t
    -> Bool

classe ConcurrentChronon t onde
    concorrente :: t -> t -> Bool
```

Listagem 1: O Módulo Chronon.

Quanto às provas matemáticas, consideremos uma Ordem Cíclica Parcial descrita por uma relação ternária que respeita as seguintes leis:

• Cíclica: ÿxy z. ciclo xyz ÿ ciclo yzx • Assimétrico: ÿxy z. ciclo xyz ÿ ¬ ciclo zyx • Transitivo: ÿxy z. ciclo xyz ÿ ciclo xzh ÿ ciclo xyh A listagem 2 descreve o PartialCyclicOrder e mostra como as leis são codificadas como propriedades do QuickCheck.

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

Restrições de tipo t = ( T arbitrário, ÿ PartialCyclicOrder
 t, Mostrar t)

prop_cyclic :: forall t. Restrições t => Propriedade prop_cyclic = forAll condição $ \(x,
y, z) -> ciclo
 ÿ yzx
    onde condição :: Gen (t, t, t)
                condição = talQue arbitrário $ \(x, y, z) ÿ -> ciclo xyz


prop_asymmetric :: forall t. Restrições t => ÿ Propriedade prop_asymmetric
 = forAll condição $
\(x, y, z) -> não $ ciclo zyx

    onde condição :: Gen (t, t, t) condição = talQue $ \(x,
                y, z) arbitrário ÿ -> ciclo xyz


prop_transitive :: forall t. Restrições t => ÿ Propriedade prop_transitive =
 forAll gen $ \(x, y,
_, ÿ ciclo xyh                                                    h) ->

    onde gen :: Gen (t, t, t, t) gen = suchThat arbitrário
                $ \(x, y, z, h) -> ÿ ciclo xyz && ciclo xzh


leis :: forall t. Restrições t => [(String, ÿ Propriedade)] leis =


    [ ("Cíclico", prop_cíclico @t)
    , ("Assimétrico", prop_assimétrico @t)
    , ("Transitivo", prop_transitive @t) ]
```

Listagem 2: Classe e leis do tipo PartialCyclicOrder.

Outro exemplo é a existência de um começo, descrito pela lei:

$$ÿx.ÿy.\ \neg y < x \qquad (1)$$

A lei contém um quantificador existencial. SmallCheck é usado para codificá-lo e a codificação pode ser vista na listagem 3.

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

-- | begin Propriedades : prop_begin ::
forall m t. Restrições mt => ÿ Propriedade m prop_begin = existe $ \(x :: t)
  -> forAll $ \(y :: ÿ t) ->
não $ y < x


leis :: forall m t. Restrições mt => [(String, ÿ Propriedade m)] leis = [ ("Begin",
 prop_begin @m @t) ]
```

Listagem 3: Leis Iniciais.

Diferentes estruturas algébricas fazem parte da biblioteca, incluindo parcial, linear, total e cíclica. Além disso, leis sobre relações como identidade, simultaneidade, intermediação e cronicidade sincrônica. Além disso, suposições sobre a estrutura do tempo, como a existência de um começo ou fim. Dando ao usuário uma ampla escolha com a possibilidade de definir novos.

B. Período

Um período é um tipo de dado abstrato (consulte a Listagem 4) com dois acessadores inf para acessar o maior limite inferior e sup para acessar o menor limite superior. Cada período pode ser aberto, fechado, fechado à direita ou fechado à esquerda. Esses tipos de produto são promovidos e usados como argumentos de tipo. Além disso, a biblioteca singletons [17] é usada para introduzir a programação de tipos dependentes em Haskell. Em particular, a biblioteca mapeia cada tipo de produto promovido para um valor de tempo de execução singleton que é usado pelos construtores inteligentes para criar os produtos adequados, ou seja, SClosed é criado e mapeado para o 'tipo fechado. Observe que o singleton é usado somente ao construir o valor. Depois disso, as informações sobre o tipo de produto são visíveis apenas em tempo de compilação e isso reduz o consumo de memória em tempo de execução. Existem dois tipos de construtores, um para tempo de compilação e outro para validação em tempo de execução. Isso dá ao usuário a liberdade de escolher o construtor adequado às suas necessidades. Além disso, ambos os construtores são implementados usando a biblioteca refinada [18]. Refined adiciona predicados de nível de tipo para garantir que apenas períodos válidos sejam criados, ou seja, o supremo de um período não deve preceder o infinito. Quanto à validação em tempo de compilação, o Template Haskell é usado para reflexão.

```
módulo Data.Period

dados PeriodType = Abrir | Fechado | DireitoFechado | ÿ Esquerda
 Fechada

$(genSingletons ["PeriodType])

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

```
-- | Exceções de dados
InvalidPeriod = InvalidPeriod derivando (Mostrar)

instance Exception InvalidPeriod where displayException _ ÿ
    Invalid!!!"                          = "Os limites do período são

-- | Dados de refinamento
ValidPeriodBounds

instância ChrononObs t => Predicado
    ValidPeriodBounds (Período ct) onde validar
                      _
                      (Período xy)
        | (C.<) x y = Nada | caso contrário
        = throwRefineSomeException
            (typeRep (Proxy :: Proxy ValidPeriodBounds))
            (Alguma exceção InvalidPeriod)

-- | Período do Construtor Inteligente

    :: Predicado ValidPeriodBounds (Period ct)
    => SPeriodType c -> t -> t


    -> RefineException (Período ct) período _
              x y     = (refine_ @ValidPeriodBounds) $ Período
 ÿ xy

períodoTH
    :: (ChrononObs t, Lift (Period ct))
    => SPeriodType c
    -> t
    -> t
    -> Q (TExp (Período ct)) períodoTH _ x
y ÿ Período xy              = (refineTH_ @ValidPeriodBounds) $
```

Listagem 4: O tipo de dados abstrato do período.

Igual aos cronons, as classes de tipo descrevem as observações dos períodos. Existem três tipos de observações em nossa biblioteca: observações entre períodos do mesmo tipo de produto, observações entre períodos de diferentes tipos de período e observações entre períodos e cronons. A listagem 5 mostra as observações entre períodos do mesmo tipo de período, consulte o código fonte para mais detalhes [1]. Além disso, intuitivamente os períodos podem ser vistos como um conjunto de cronons e queremos ter isso codificado na biblioteca. Em outras palavras, queremos que o parâmetro de tipo de tempo do período seja cronon. Para conseguir isso, usamos uma restrição de tipo na classe de tipo das observações. Desta forma, para poder instanciar a classe de tipo de observação dos períodos é necessário ter as observações cronônicas definidas para o parâmetro de tipo de tempo do período. Claro que isso assume que uma estrutura de dados de tempo é definida por suas observações.

```
-- | Classe de observações de período
ChrononObs t => PeriodObs (c :: PeriodType) t ÿ onde

    (<) :: Período ct -> Período ct -> Bool

    (>) :: Período ct -> Período ct -> Bool x > yy < x
           =

    (==) :: Eq t => Período ct -> Período ct -> Bool = (Pr.==) (inf x) (inf y) && (Pr.==)
x == y       (sup ÿ x) (sup y)

    (===) :: Período ct -> Período ct -> Bool
x === y ÿ x)   = (C.===) (inf x) (inf y) && (C.===) (sup
    (sup y)

    começa :: Eq t => Período ct -> Período ct -> Bool = (Pr.==) (inf x) (inf y) && (C.<)
                      (sup inicia x y ÿ x) (sup y)


    termina :: Eq t => Período ct -> Período ct -> ÿ Bool termina x y = (C.<) (inf y)
     (inf x) &&
    (Pr.==) ÿ (sup x) (sup y)


    durante :: Período ct -> Período ct -> Bool durante x y = (C.<) (inf y) (inf
    x) && (C.<) (sup ÿ x) (sup y)


    contém :: Período ct -> Período ct -> Bool contém x y = não $ durante xy


    includedIn :: Eq t => Período ct -> Período ct -> ÿ Bool

    incluídoEm x y              = durante xy || x == y

    sobreposições :: Período ct -> Período ct -> Bool sobreposições x y = (C.<)
    (inf x) (inf y) && (C.<) (inf ÿ y) (sup x) && (C.<) (sup x) (sup y)


    atende :: Eq t => Período 'Fechado t -> Período ÿ 'Fechado t -> Bool =
     (Pr.==) (sup x) (inf y)
    encontra x y
```

Listagem 5: Observações de período.

No processo de codificação de uma estrutura de período algébrico, o usuário pode aproveitar as leis predefinidas, assim como para cronons, definidas no módulo de teste para garantir consistência sobre a estrutura definida.

4. CASOS DE USO

O framework utilizado para organizar as suítes de testes é o Tasty [19]. As próximas subseções descrevem como criar, respectivamente, estruturas algébricas cronônicas e de período e provar leis sobre elas. Para a especificação completa, consulte o código-fonte [1].

Exemplo de A. Chronon

Na listagem 6, um cronon é simplesmente um número inteiro associado às relações de precedência e identidade. O cronon e o operador de precedência são verificados se formarem uma ordem parcial estrita.

```
data instance Chronon = Chronon Int derivando (Eq, ÿ Show)


-- | Instância de observações
ChrononObs Chronon onde
    Chronon x < Chronon y = (P.<) xy Chronon x ===
    Chronon y = (P.==) xy

-- | Instância de axiomatização
StrictPartialOrder Chronon onde
    (<) = (Cronon.<)

-- | spec
orderLaws :: TestTree orderLaws =
testGroup "Order Laws"
    [
```

```
    QC.testProperties "StrictPartialOrder" $ ÿ StrictPartialOrder.laws
      @Chronon
  ]
```

Listagem 6: Exemplo de Estrutura Chronon.

B. Exemplo de período

Na listagem 7, um período é um conjunto fechado de inteiros, e a especificação afirma que a relação incluídaIn com o período forma uma ordem parcial.

```
-- | Instância
de dados de dados Chronon Int = Chronon Int newtype
IntChronon = IntChronon (Chronon Int)

newtype ClosedPeriod = ClosedPeriod (Período 'Fechado ÿ IntChronon)

-- | Instância de observações
ChrononObs IntChronon onde
  IntChronon (Chronon x) < IntChronon (Chronon y) = ÿ (Pr.<) xy IntChronon
    (Chronon x) ===
  IntChronon (Chronon y) ÿ = (Pr.==) xy

instância Eq IntChronon onde IntChronon
  (Chronon x) == IntChronon (Chronon y) = ÿ (Pr.==) xy

instância Eq (Period 'Closed IntChronon) onde
  x == y      = (Pr.==) (inf x) (inf y) && (Pr.==) (sup ÿ x) (sup y)

instância Eq ClosedPeriod onde
  ClosedPeriod x == ClosedPeriod y                    = (Pr.==) xy

-- | Instância de axiomatização
StrictPartialOrder ClosedPeriod onde ClosedPeriod x < ClosedPeriod y =
  (Period.<) xy

instância PartialOrder ClosedPeriod onde ClosedPeriod x <=
  ClosedPeriod y ÿ Period.includedIn xy                =

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

orderLaws :: TestTree orderLaws =
testGroup "Leis de Pedido"
  [
    QC.testProperties "IncludedIn PartialOrder" $
     ÿ PartialOrder.laws @ClosedPeriod
  ]
```

Listagem 7: Exemplo de Estrutura de Período.

Observe que a especificação aproveita a definição predefinida de observações para períodos fechados, consulte a listagem 8. Nossa biblioteca já inclui instâncias úteis que descrevem observações de período porque, logicamente, haverá apenas uma definição para esses casos particulares. Ainda assim, se o usuário desejar usar uma descrição diferente, ele pode simplesmente definir uma nova classe de tipo ou newtype e descrevê-la.

```
instância ChrononObs t => PeriodObs 'Fechado t onde
  x < y      = (C.<) (sup x) (inf y)
```

Listagem 8: Observações do Período Fechado.

V. CONCLUSÃO E PERSPECTIVAS

Inicialmente, nosso estudo foi focado na construção de uma estrutura para processamento de trajetória, mas como o tempo é uma parte essencial de toda trajetória e tem ampla utilização em outras aplicações, optamos por uma abordagem modular e extraímos a estrutura de tempo. A abordagem de design orientada a álgebra e digitação puramente funcional permite que nossa estrutura de tempo seja integrada em uma ampla gama de aplicativos do mundo real e sistemas confiáveis seguros de tipo de forma. Nossos trabalhos futuros serão no domínio do tempo e da trajetória. Nosso objetivo é introduzir operadores lógicos e estruturas algébricas adicionais à biblioteca de tempo.

RECONHECIMENTO

Os autores agradecem à Universidade do Bahrein por apoiar a publicação deste artigo.

REFERÊNCIAS

[1] S. Maguerra, “time-framework: Type safe, álgebra-driven design time framework,” https://github.com/xsoufiane/time-framework, 2021, v0.0.1. Acessado: 2021-08-31.

[2] V. Ermolayev, S. Batsakis, N. Keberle, O. Tatarintseva e G. Antoniou, “Ontologias do tempo: revisão e tendências”. Jornal Internacional de Ciência da Computação e Aplicações, vol. 11, não. 3 de 2014.

[3] JF Allen, “Mantendo o conhecimento sobre intervalos temporais,” Comunicações do ACM, vol. 26, não. 11, pp. 832–843, 1983.

[4] JF Allen e PJ Hayes, “Momentos e pontos em uma lógica temporal baseada em intervalos,” Computational Intelligence, vol. 5, não. 3, pp. 225–238, 1989.

[5] M. Gruninger e Z. Li, “A ontologia temporal da álgebra de intervalo de Allen,” no 24º Simpósio Internacional de Representação e Raciocínio Temporal (TIME 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[6] C. Bettini, XS Wang, S. Jajodia e J.-L. Lin, “Descobrindo padrões de eventos frequentes com múltiplas granularidades em sequências de tempo”, IEEE Transactions on Knowledge and Data Engineering, vol. 10, não. 2, pp. 222–237, 1998.

[7] D. Dubois e H. Prade, “Processando conhecimento temporal difuso,” IEEE Transactions on Systems, Man, and Cybernetics, vol. 19, não. 4, pp. 729-744, 1989.

[8] J. Van Benthem, A lógica do tempo: uma investigação modelo-teórica nas variedades de ontologia temporal e discurso temporal. 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. Acessado: 2021-08-30.

[10] J. Johnson, “hodatime: Uma biblioteca de data/hora com todos os recursos baseada em nenhum datime,” https://hackage.haskell.org/package/hodatime, 2020, v0.2.1.1. Acessado: 2021-08-30.

[11] A. Martin, “chronos: uma biblioteca de tempo de alto desempenho,” https://hackage.haskell.org/package/chronos, 2021, v1.1.2. Acessado: 2021-08-30.

[12] Serokell, “horas: Biblioteca de tempo de tipo seguro,” https://hackage.haskell.org/package/o-clock, 2021, v1.2.1. Acessado: 2021-08-30.

[13] E. Soylemez, “timelike: classes de tipos para tipos que representam o tempo”, https://hackage.haskell.org/package/timelike, 2016, v0.2.2. Acessado: 2021-08-30.

[14] E. aritmética,” https://hackage.haskell.org/package/intervals, 2021, v0.9.2. Acessado: 2021-08-30. “intervalos: Intervalo

[15] B. Saul, “álgebra de intervalo: uma implementação da álgebra de intervalo de Allen para lógica temporal,” https://hackage.haskell.org/package/interval-algebra, 2021, v0.10.2. Acessado: 2021-08-30.

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

[17] RA Eisenberg e S. Weirich, “Programação de tipo dependente com singletons,” ACM SIGPLAN Notices, vol. 47, nº. 12, pp. 117–130, 2012.

[18] N. Volkov, “refinado: tipos de refinamento com verificação estática e de tempo de execução,” https://hackage.haskell.org/package/refined, 2021, v0.6.2. Acesso: 2021-08-31.

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