Towards a more perfect union type

Michał J. Gajda https://www.migamake.com

2022-07-28

Plan

- JSON

Plan

- JSON
- Typing dynamic languages

Plan

- JSON
- Typing dynamic languages
- Typelike framework

Plan

- JSON
- Typing dynamic languages
- Typelike framework
- Examples of implementation

Plan

- JSON
- Typing dynamic languages
- Typelike framework
- Examples of implementation
- Summary

Data format landscape

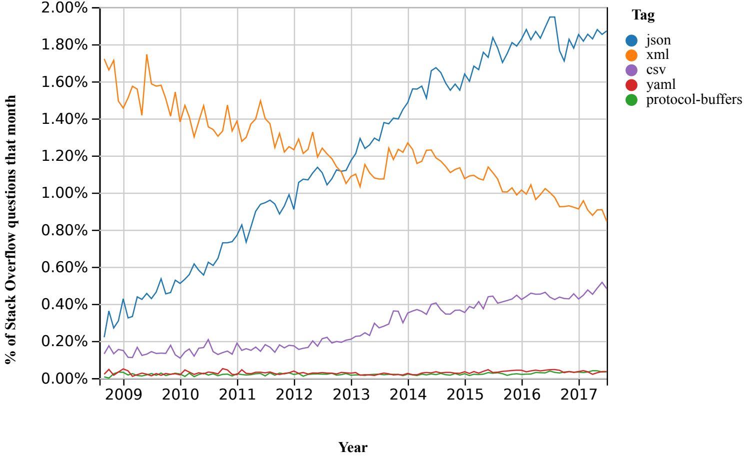


Figure 1: JSON growth

https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html

JSON

```
{"version": 1.0
,"author":  "Pole Anonymous"
,"sections": [
    "Introduction"
  , "Materials"
  , "Methods"
  , "Closure"
  ]}
```

JSON

```json
{"version": 1.0
,"author":  "Pole Anonymous"
,"sections": [
    "Introduction"
  , "Materials"
  , "Methods"
  , "Closure"
  ]}
```

```haskell
data Value = Object (Map String Value)
           | Array [Value]
           | Null
           | Number Scientific
           | String Text
           | Bool Bool
```

Related work

- quicktype
- F# type providers
- XDuce and Castagna framework

Key features

- Decidability
- Soundness
- Subject reduction

Motivation

Subsets of data within a single constructor:

- *API argument is an email* – it is a subset of valid `String` values that can be validated on the client-side.

Motivation

Subsets of data within a single constructor:

- *API argument is an email* – it is a subset of valid `String` values that can be validated on the client-side.
- *The page size determines the number of results to return (min: 10, max:10,000)* – it is also a subset of integer values (`Int`) between 10, and $10,000$

Motivation

Subsets of data within a single constructor:

- *API argument is an email* – it is a subset of valid `String` values that can be validated on the client-side.
- *The page size determines the number of results to return (min: 10, max:10,000)* – it is also a subset of integer values (`Int`) between 10, and $10,000$
- *The `date` field contains ISO8601 date* – a record field represented as a `String` that contains a calendar date in the format `"2019-03-03"`

Motivation

Subsets of data within a single constructor:

- *API argument is an email* – it is a subset of valid `String` values that can be validated on the client-side.
- *The page size determines the number of results to return (min: 10, max:10,000)* – it is also a subset of integer values (`Int`) between 10, and $10,000$
- *The `date` field contains ISO8601 date* – a record field represented as a `String` that contains a calendar date in the format `"2019-03-03"`

Motivation

Subsets of data within a single constructor:

- *API argument is an email* – it is a subset of valid `String` values that can be validated on the client-side.
- *The page size determines the number of results to return (min: 10, max:10,000)* – it is also a subset of integer values (`Int`) between 10, and $10,000$
- *The `date` field contains ISO8601 date* – a record field represented as a `String` that contains a calendar date in the format `"2019-03-03"`

```
newtype Example1a = Example Email
```

Motivation: optional fields

The page size is equal to 100 by default

```
{}
{"page_size": 50}
```

Motivation: optional fields

The page size is equal to 100 by default

```
{}
{"page_size": 50}
```

```
newtype Example2 = Example2 { page_size :: Maybe Int }
```

Motivation: variant records

Answer contains either a text message with a user identifier or an error.

```
{"message" : "Where can I submit my proposal?",
 "uid"     : 1014}
{"message" : "Submit it to HotCRP",
 "uid"     :  317}
{"error"   : "Authorization failed",
 "code"    :  401}
{"error"   : "User not found",
 "code"    :  404}
```

```
data Example4 = Message { message :: String, uid  :: Int }
              | Error   { error   :: String, code :: Int }
```

Motivation: alternative objects

Answer to a query is either a number of registered objects, or an identifier of a singleton object.

```
newtype Example3 = Example3 (String :|: Int)
```

Motivation: array of records

```
[ [1, "Nick",    null        ]
, [2, "George", "2019-04-11"]
, [3, "Olivia", "1984-05-03"] ]
```

```
data Examples = Examples [Example]

data Example5 = Example5 { col1 :: Int
                         , col2 :: String
                         , col3 :: Maybe Date }
```

Motivation: maps

Example of map of identical objects:

```
{   "6408f5": { "size":       969709
              , "height":    510599
              , "difficulty": 866429.732
              , "previous": "54fced" },
    "54fced": { "size":       991394
              , "height":    510598
              , "difficulty": 866429.823
              , "previous": "6c9589" },
    "6c9589": { "size":       990527
              , "height":     510597
              , "difficulty": 866429.931
              , "previous":  "51a0cb"
              }
}
```

Motivation: **objects** vs maps

```
data Example = Example { f_6408f5 :: O_6408f5
                       , f_54fced :: O_6408f5
                       , f_6c9589 :: O_6408f5 }
data O_6408f5 = O_6408f5 {
         size       :: Int
       , height     :: Int
       , difficulty :: Double
       , previous   :: String }
```

Motivation: objects vs **maps**

```
data ExampleMap = ExampleMap (Map Hex ExampleElt)
data ExampleElt = ExampleElt {
        size       :: Int
      , height     :: Int
      , difficulty :: Double
      , previous   :: String }
```

Goals

- detect unexpected format deviations
- detect need for program updates
- minimal containing set
- information content
- correct operation
- inference as contravariant functor

Type inference

Information fusion

- unification

```
class Semigroup ty              where (<>)   :: ty -> ty ->
class Semigroup ty => Monoid ty where mempty :: ty
```

Type inference

Information fusion

- unification
- or anti-unification

```
class Semigroup ty                  where (<>)   :: ty -> ty ->
class Semigroup ty => Monoid ty where mempty :: ty
```

Beyond set

```
class (Monoid t, Eq t, Show t) => Typelike t where beyond :
```

The `beyond` set is always **closed to information addition** by `(<>a)` or `(a<>)` for any value of `a`, or **submonoid**.

Do not require *idempotence*, nor *commutativity* of `<>`.

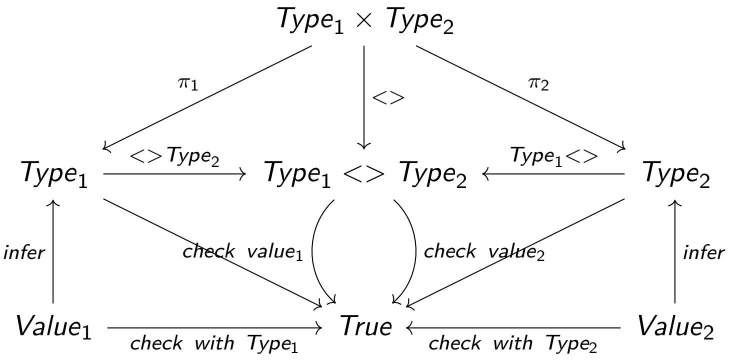
Typelike

```
class Typelike ty => ty `Types` val where
   infer ::       val -> ty
   check :: ty -> val -> Bool
```

Laws of Typelike

$$
\begin{array}{llcllcl}
 & & & \text{check} & \text{mempty} & v & = \textbf{False} \\
\text{beyond} & t & & \Rightarrow \text{check} & t & v & = \textbf{True} \\
\text{check} & t_1 & v & \Rightarrow \text{check} & (t_1 \diamond t_2) & v & = \textbf{True} \\
\text{check} & t_2 & v & \Rightarrow \text{check} & (t_1 \diamond t_2) & v & = \textbf{True} \\
 & & & \text{check} & (\text{infer } v) & v & = \textbf{False} \\
 & & & & t_1 \diamond (t_2 \diamond t_3) & & = t_1 \diamond (t_2 \diamond t_3) \\
 & & & & \text{mempty} \diamond t & & = t \\
 & & & & t \diamond \text{mempty} & & = t
\end{array}
$$

Typelike


$Type_1 \times Type_2$
π_1
$<>$
π_2
$Type_1$
$<> Type_2$
$Type_1 <> Type_2$
$Type_1 <>$
$Type_2$
$infer$
$check\ \ value_1$
$check\ \ value_2$
$infer$
$Value_1$
$True$
$Value_2$
$check\ \ with\ \ Type_1$
$check\ \ with\ \ Type_2$

Presence and absence constraint

```
data PresenceConstraint a = Present -- beyond
                          | Absent  -- mempty

instance Semigroup (PresenceConstraint a) where
  Absent  <> a       = a
  a       <> Absent  = a
  Present <> Present = Present

instance PresenceConstraint a `Types` a where
  infer _          = Present
  check Present _  = True
  check Absent  _  = False
```

Flat type constraints

```haskell
data NumberConstraint = NCInt
                      | NCNever -- mempty
                      | NCFloat -- beyond

instance Semigroup NumberConstraint where
  NCInt   <> NCInt   = NCInt
  NCFloat <> _       = NCFloat -- beyond
  _       <> NCFloat = NCFloat -- beyond
  NCNever <> a       = a       -- mempty
  a       <> NCNever = a       -- mempty

instance NumberConstraint `Types` Scientific where
  infer sci | base10Exponent sci >= 0 = NCInt
  infer _                             = NCFloat
  check NCInt   sci = base10Exponent sci >= 0
  check NCFloat _   = True
  check NCNever _   = False
```

Cost of optionality

```haskell
class Typelike ty => TypeCost ty where
  typeCost ::  ty -> TyCost
  typeCost a = if a == mempty then 0 else 1

instance Semigroup TyCost where (<>)   = (+)
instance Monoid    TyCost where mempty = 0

newtype TyCost = TyCost Int
```

Mapping constraint

```haskell
data MappingConstraint =
    MappingNever -- mempty
  | MappingConstraint { keyConstraint
                           :: StringConstraint
                      , valueConstraint
                           :: UnionType        }
```

Mapping constraint 2

```
instance Semigroup   MappingConstraint where
  MappingNever <> a = a
  a <> MappingNever = a
  a <> b = MappingConstraint {
      keyConstraint   =
        ((<>) `on` keyConstraint  ) a b
    , valueConstraint =
        ((<>) `on` valueConstraint) a b
    }
```

Record constraint

```haskell
data RecordConstraint =
    RCTop    {- beyond -}
  | RCBottom {- mempty -}
  | RecordConstraint { fields :: HashMap Text UnionType }

instance Semigroup   RecordConstraint where
  RecordConstraint     a  <>
    RecordConstraint     b = RecordConstraint $
         Map.intersectionWith (<>) a b
      <> (makeNullable <$> mapXor  a b)
```

RecordConstraint 2

```
instance RecordConstraint `Types` Object where
    infer = RecordConstraint     . Map.fromList
          . fmap (second infer) . Map.toList
    check RecordConstraint {fields} obj =
         all (`elem` Map.keys fields)
                    (Map.keys  obj)
      && and (Map.elems $ Map.intersectionWith
                                  check fields obj)
      && all isNullable (Map.elems
                           $ fields `Map.difference` obj)
         -- absent values are nullable
```

Object constraint

```
data ObjectConstraint = ObjectNever -- mempty
  |  ObjectConstraint { mappingCase :: MappingConstraint
                      , recordCase  :: RecordConstraint }

instance Semigroup ObjectConstraint where
  a <> b = ObjectConstraint {
             mappingCase = ((<>) `on` mappingCase) a b
           , recordCase  = ((<>) `on` recordCase ) a b
           }

instance ObjectConstraint `Types` Object where
  infer v = ObjectConstraint (infer v) (infer v)
```

Array constraint

```haskell
data ArrayConstraint =
     ArrayNever -- mempty
   | ArrayConstraint { rowCase   :: RowConstraint,
                     , arrayCase :: UnionType }

instance Semigroup ArrayConstraint where
  a1 <> a2 =
    ArrayConstraint {
      rowCase   = ((<>) `on` rowCase  ) a1 a2
    , arrayCase = ((<>) `on` arrayCase) a1 a2
    }

instance ArrayConstraint `Types` Array where
    infer vs = ArrayConstraint {
        rowCase = infer vs
      , arrayCase = mconcat (infer <$> Foldable.toList vs)
      }
```

Row constraint

```
data RowConstraint = RowTop | RowNever | Row [UnionType]

instance Semigroup RowConstraint where
  Row bs <> Row cs | length bs /= length cs = RowTop
  Row bs <> Row cs = Row $ zipWith (<>) bs cs

instance RowConstraint `Types` Array where
  infer = Row
        . Foldable.toList
        . fmap infer
  check (Row rs) vs
    | length rs == length vs =
      and $
        zipWith check                    rs
                     (Foldable.toList vs)
```

Union type

```
data UnionType = UnionType {
    unionNull :: NullConstraint
  , unionBool :: BoolConstraint
  , unionNum  :: NumberConstraint
  , unionStr  :: StringConstraint
  , unionArr  :: ArrayConstraint
  , unionObj  :: ObjectConstraint }
```

Union type 2

```haskell
instance Semigroup UnionType where
  u1 <> u2 =
    UnionType {
      unionNull = ((<>) `on` unionNull) u1 u2
    , unionBool = ((<>) `on` unionBool) u1 u2
    , unionNum  = ((<>) `on` unionNum ) u1 u2
    , unionStr  = ((<>) `on` unionStr ) u1 u2
    , unionObj  = ((<>) `on` unionObj ) u1 u2
    , unionArr  = ((<>) `on` unionArr ) u1 u2
    }
```

Union type 3

```haskell
-- Since union type is all about optionality,
-- we need to sum all options from different alternatives:
instance TypeCost UnionType where
  typeCost UnionType {..} = typeCost unionBool
      + typeCost unionNull + typeCost unionNum
      + typeCost unionStr  + typeCost unionObj
      + typeCost unionArr
```

Counting observations

```
data Counted a = Counted { count :: Int, constraint :: a }

instance Semigroup a => Semigroup (Counted a) where
  a <> b = Counted (count      a +  count      b)
                   (constraint a <> constraint b)

instance           ty  `Types` term
     => (Counted ty) `Types` term where
  infer term = Counted 1 $ infer term
  check (Counted _ ty) term = check ty term
```

Summary

- Monoid-based exposition of type inference

Summary

- Monoid-based exposition of type inference
- Typelike as non-lossy learning

Summary

- Monoid-based exposition of type inference
- Typelike as non-lossy learning
- Generic monoid suffices

Summary

- Monoid-based exposition of type inference
- Typelike as non-lossy learning
- Generic monoid suffices
- Easy extensibility

Summary

- Monoid-based exposition of type inference
- Typelike as non-lossy learning
- Generic monoid suffices
- Easy extensibility
- Liberal laws

Summary

- Monoid-based exposition of type inference
- Typelike as non-lossy learning
- Generic monoid suffices
- Easy extensibility
- Liberal laws
- Next version of `json-autotype`