Towards a more perfect union type

Functional pearl

Anonymous Author(s)

Abstract

We present a principled theoretical framework for dealing with union types, system, and show its work in practice on JSON data structures.

The framework poses union type inference as a problem of learning from multiple examples. Mathematical framework is quite generic, and easily extensible.

1 Introduction

Typing dynamic languages has been long considered a challenge [3]. The importance of the task grown with ubiquity cloud application programming interfaces (APIs) utilizing JavaScript object notation (JSON), where one needs to infer the structure having only a limited number of sample documents available.

Previous research have suggested it is possible to infer adequate type mappings from sample data [2, 10, 16].

In the present study, we expand on these results. We propose a framework for type systems in programming languages as learning algorithms, formulate it mathematically, and evaluate its performance on JSON API examples.

The proposed framework is grounded on mathematical theory, and complete typing relation. It is intended to add new features easily.

1.1 Related work

1.1.1 Union type providers

The earliest practical effort to apply union types to JSON inference to generate Haskell types[10]. It uses union type theory, but it also lacks an extensible theoretical framework.

F# type providers for JSON facilitate deriving a schema automatically; however, a type system is *ad-hoc*[16].

The other attempt to automatically infer schemas has been introduced in the PADS project [14]. Nevertheless, it has not specified a generalized type-system design methodology.

An approach presented with a program called [2] has been developed to derive types based on Markov chains. This approach approach requires considerable engineering time due to the implementation unit tests in a case-by-case mode, instead of formulating laws applying to all types. Moreover, this approach lacks sound underlying theory.

Therefore, we summarize that there are several previously introduced approaches that provide partially satisfactory results. In present study, we aim to expand these proposals to enable systematic addition of features, and automatic validation of types.

1.1.2 Frameworks for describing type systems

Type systems are commonly expressed as partial relation of *typing*. Their properties, such as subject reduction are also expressed relatively to the relation (also partial) of *reduction* within a term rewriting system.

General formulations have been introduced for the Damas-Milner type systems parameterized by constraints [19].

We are not aware of any attempts to formulate general laws that would apply to all existing union type systems. Moreover, to the best of our knowledge no previous formulation exists that consider complete relations or functions in order to provide consistent mathematical descriptions where terms stray beyond their desired types[1].

It is also worth noting that traditional Damas-Milner type disciplines embrace the laws of soundness, and subject-reduction. However these laws often prove too strict during type system extension, and are abandoned in practice of larger systems[17].

2 Motivation

2.1 Motivating examples

Here, we consider several examples paraphrased from JSON API descriptions. These describe types underlying the motivation for the present study:

1. Subsets of data within a single constructor:

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

```
{"example": [
  "amy@example.com"
  "robert@example.com"
  ]
}
```

```
example1a_values :: [Value]
example1a_values = String <$> [
    "amy@example.com"
  , "edward@example.com"
  ]
example1a_repr :: HType
example1a_repr = HRef "Email"
```

[1]Or at least beyond bottom expanding to *infamous undefined behaviour*[5].

Haskell'20, August, 2020, Saint Louis, USA
2020.

- *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 is represented as a String that contains a calendar date in the format "2019-03-03"

2. Optional fields:

- *The page size is equal to 100 by default* - it means we have a record {"page_size": 50} or an empty record that should be interpreted as default value {}

3. Variant fields:

- *Answer to a query is either a number of of registered objects, or String* "unavailable" - this is integer value (Int) or a String

4. Variant records:

- *Answer contains either a text message with an user id, or an error.* – That is can be represented as one of following options:

```
{"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}
```

5. Arrays corresponding to records[2]:

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

6. Maps of identical objects[3]:

```
{
    "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"
    }
}
```

[2]Which is considered a bad practice; however, it is part of real-life APIs. We may need to make it optional using the --array-records option.

[3]The example is taken from [2].

It should be noted that the last example presented above requires Haskell representation inference to be non-monotonic, as a dictionary with a single key would have an incompatible type:

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

It also suggests that a user might decide to explicitly add an evidence for one of alternative representations in the case when samples are insufficient. (like in case of a single element dictionary.)

2.2 Goal of inference

Given an undocumented (or incorrectly labelled) JSON API, we may need to read the input of Haskell encoding and avoid checking for the presence of *unexpected* format deviations. At the same time, we may decide to accept all known valid inputs outright so that we can use types[4] to ensure that the input is processed exhaustively.

Accordingly, we can assume that the smallest non-singleton set is a better approximation type than a singleton set. We call it *minimal containing set principle.*

Second we can prefer types that allow for fewer number of *degrees of freedom* compared with the others, while conforming to a commonly occuring structure. We denote it as an *information content principle.*

Given these principles, and examples of frequently occuring patterns, we can infer a reasonable *world of types* that can be used as approximations, instead of establishing this procedure in an ad-hoc manner. In this way,we can implement *type system engineering*, that allows deriving type system design directly from the information about data structures and the likelihood of their occurence.

3 Problem definition

3.1 Preliminaries

3.1.1 JSON values

As we focus on JSON, we utilize Haskell encoding of the JSON term for convenient reading[5]; specified as follows:

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

[4]Compiler feature of checking for unmatched cases.

[5]As used by Aeson[1] package.

```
  | Bool     Bool
  | Null
```

To incorporate both integers and exact decimal fractions[6] in the considered number representation, we employ decimal floating point[18]:

```
data Scientific =
  Scientific { coefficient     :: Integer
             , base10Exponent :: Int }
```

3.2 Defining type inference

3.2.1 Information in the type descriptions

If an inference fails, it is always possible to correct it by introducing an additional observation (example). To denote unification operation, or **information fusion** between two type descriptions, we use a Semigroup interface operation <> to merge types inferred from different observations.

```
class Semigroup ty where
  (<>) :: ty -> ty -> ty
```

We use neutral element of the Monoid to indicate a type corresponding to no observations:

```
class Semigroup ty
  => Monoid     ty
  where
   mempty :: ty
```

In other words, we can say that mempty corresponds to situation wher **no information was accepted** about a possible value (no term seen, not even a null). For example, an empty array [] can be referred to as an array type with mempty as an element type.

We describe the laws as QuickCheck [@quickcheck] properties so that unit testing can be implemented to detect obvious violations.

Neutral element of the Typelike monoid, mempty stands for **no information accepted** about possible value (no term seen, not even a null). For example an empty array [] could be typed as a array type with mempty as element type.

3.2.2 Beyond set

In the domain of permissive union types, a beyond set represents the case of **everything permitted** or a fully dynamic value, when we gather the information that permits every possible value inside a type. At the first reading, it may be deemed that a beyond set should comprise of only one single element – the top one.

However, since we defined **unification** operator <> as **information fusion**, we may encounter difficulties in assuring that no information has been lost during the unification[7].

Moreover, strict type systems usually specify more than one error value, as it should contain information about error messages and to keep track from where an error has been originated[8].

This observation lets us consider type inference as a **learning problem**, and allows finding the common ground between the dynamic and the static typing discipline.

The languages relying on the static type discipline usually consider beyond as a set of error messages, as a value should correspond to a statically assigned and a **narrow** type. In this setting mempty as a fully polymorphic type forall a. a.

Languages with dynamic type discipline will treat beyond as untyped, dynamic value, and mempty again is a fully unknown, polymorphic value (like a type of an element of an empty array)[9].

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

In addition, the standard laws for a **commutative** Monoid, we state the new law for the beyond set: The beyond set is always **closed to information addition** by (<>a) or (a<>) for any value of a. In other words the beyond set is an attractor of <> on both sides.[10]

Concerning union types, the key property of the beyond set, is that it is closed to information acquisition:

```
beyond_is_closed ty1 ty2 = do
  beyond (ty1 :: ty) ==> beyond (ty1 <> ty2)
```

(We describe laws as QuickCheck properties~[4] so that unit testing can detect obvious violations.)

In this way, we can specify other elements of beyond set instead of a single top. When typing strict language, like Haskell, we seek to enable each element of the beyond set to contain at least one error message.[11]

It should be noted that here, we abolish the semilattice requirement that has been conventionally assumed for type constraints [20], as this requirement is valid only for strict type constraint inference, not for a more general type inference considered as a learning problem. As we observe in the example lst. 2.1, we need to perform non-monotonic inference when dealing with alternative representations.

When a specific instance of Typelike is also a semilattice (an idempotent semigroup), we will explicitly indicate if that is the case.

It is convenient validation when testing a recursive structure of the type.

Note that we abolish semilattice requirement that was traditionally assumed for type constraints here[21].

---

[6]JavaScript and JSON use a binary floating point instead; however we follow the representation selected by aeson library that parses JSON.

[7]Examples will be provided later.

[8]7

[9]May sound similar until we consider adding more information to the type.

[10]So both in forall a. (<> a) and ∀a.(a<>) the result is kept in the beyond set.

[11]It should be noted that many but not all type constraints are semilattice. Please refer to the counting example below.

That is because this requirement is valid only for strict type constraint inference, not for a more general type inference as a learning problem. As we saw in the example lst. 2.1, we need non-monotonic inference when dealing with alternative representations.

It should be noted that this approach significantly generalized the assumptions compared with a full lattice subtyping [20][21].

3.2.3 Typing relation and its laws

The minimal definition of typing inference relation and type checking relation can be formulated as follows:

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

Specifying the laws of typing is important, since we may need to consider separately the validity of a domain of types/type constraints, and that of the sound typing of the terms by these valid types.

First, we note that to describe *no information*, mempty cannot correctly type any term:

```
mempty_contains_no_terms term =
    check (mempty :: ty) term
      == False
```

Second important rule of typing is that all terms are typed successfully by any value in the beyond set.

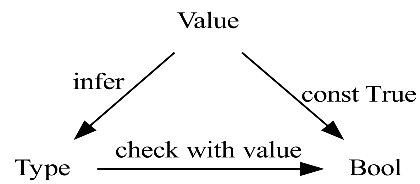
However, randomly drawing types for particular instances we might almost never get a type from the beyond set. In this case, we can use special generator called arbitraryBeyond that generates only the elements of the beyond set:

```
{-
beyond_contains_all_terms2 :: forall   ty term.
                              (Typelike ty
                              ,Types    ty term)
                            => term -> _
beyond_contains_all_terms2 term =
  forAll arbitraryBeyond $ (`check` term)
  -}
```

We state the most intuitive rule for typing: a type inferred from a term, must always be valid for that particular term.

```
inferred_type_contains_its_term ::
     forall ty         term.
            ty `Types` term
  =>                   term
  -> Bool
inferred_type_contains_its_term term =
  check ((infer:: term -> ty) term) (term :: term)
```

The law asserts that the following diagram commutes:



The last law states that the terms are correctly typechecked after adding more information into a single type. (For inference relation, it would be described as *principal type property*.)

```
fusion_keeps_terms v ty1 ty2 = do
  check ty1 v || check ty2 v ==>
    check (ty1 <> ty2) v
```

The minimal Typelike instance is the one that contains only mempty corresponding to the case of *no sample data received*, and a single beyond element for *all values permitted*. We will define it below as PresenceConstraint in sec. 3.5.

It should be noted that these laws are still compatible with the strict, static type discipline: namely the beyond set corresponds to a set of constrants with at least one type error, and a task of a compiler to prevent any program with the terms that type only to the beyond as a least upper bound.

3.3 Type engineering principles

Considering that we aim to infer a type from a finite number of samples, we are presented with a *learning problem*, so we need to use *prior* knowledge about the domain for inferring types.

Observing that a : false we can expect that in particular cases, we may obtain that a : true. After noting that b : 123, we expect that b : 100 would also be acceptable. It means that we need to consider a typing system to *learn a reasonable general class from few instances*. This motivates formulating type system as an inference problem.

As the purpose is to deliver the most descriptive[12] types, we assume that we need to obtain a wider view rather than focusing on a *free type* and applying it to a larger sets whenever it is deemed justified.

The other principle corresponds to **correct operation**. It implies that having operations regarded on types, we can find a minimal set of types that assure correct operation om the case of unexpected errors.

Indeed we want to apply this theory to infer a type definition from a finite set of examples. We also seek to generalize it to infinite types.

For this purpose, we set the following rules of type design:

[12] The shortest one accoding to the information complexity principle.

- type should have a finite description
- inference must be a contravariant functor with regards to constructors. For example, if {"a": X, "b": Y} that is typed by T x y, then X :: x and Y :: y must correspond to a valid typing.

3.3.1 Flat type constraints

Let us first consider typing of flat types: String and Number.

Constraints on number type First we infer the type description for integer valuess[13]:

```haskell
data IntConstraint = IntRange Int Int
                   | IntNever
                   | IntAny
  deriving (Show, Eq, Generic)

instance Semigroup IntConstraint where
  IntAny      <> _            = IntAny
  _           <> IntAny       = IntAny
  IntNever    <> a            = a
  a           <> IntNever     = a
  IntRange        a       b  <>
    IntRange        c       d =
      IntRange (min a c) (max b d)

instance Typelike IntConstraint where
  beyond = (==IntAny)

instance Monoid IntConstraint where
  mempty = IntNever

instance IntConstraint `Types` Int where
  infer              i = IntRange i i
  check  IntNever    _ = False
  check  IntAny      _ = True
  check (IntRange a b) i = a <= i && i <= b
```

JavaScript provides one number type that contains both Float and Int, so that the JSON values inherit this type:

```haskell
data NumberConstraint =
    NCInt
  | NCNever
  | NCFloat
  deriving(Eq,Show,Generic)

instance Semigroup NumberConstraint where
  NCFloat <> _       = NCFloat
  _       <> NCFloat = NCFloat
  NCNever <> a       = a
  a       <> NCNever = a
  NCInt   <> NCInt   = NCInt
```

[13] The implementation will make it optional with --infer-int-ranges.

```haskell
instance Typelike NumberConstraint where
  beyond = (==NCFloat)

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

instance Monoid NumberConstraint where
  mempty = NCNever
```

Constraints on string type

```haskell
data StringConstraint =
    SCDate
  | SCEmail
  | SCEnum  (Set Text)
  | SCNever
  | SCAny
  deriving(Eq, Show,Generic)

instance StringConstraint `Types` Text where
  infer (isValidDate  -> True) = SCDate
  infer (isValidEmail -> True) = SCEmail
  infer  _      = SCAny
  infer  value = SCEnum $ Set.singleton value

  check  SCDate     s = isValidDate  s
  check  SCEmail    s = isValidEmail s
  check (SCEnum vs) s = s `Set.member` vs
  check  SCNever    _ = False
  check  SCAny      _ = True
```

Then, whenever unifying the String constraint, the following code can be executed:

```haskell
instance Semigroup StringConstraint where
  SCNever    <>  a             = a
  a          <>  SCNever       = a
  SCAny      <>  _             = SCAny
  _          <>  SCAny         = SCAny
  SCDate     <>  SCDate        = SCDate
  SCEmail    <>  SCEmail       = SCEmail
  (SCEnum a) <> (SCEnum b)     |
        length (a <> b) < 10 = SCEnum (a <> b)
  _          <>  _             = SCAny

instance Monoid StringConstraint where
  mappend = (<>)
  mempty  = SCNever
```

```
instance Typelike StringConstraint where
  beyond  = (==SCAny)
```

3.4 Free union type

Before we endavour on finding type constraints for compound values (arrays and objects), it might be instructive to find a notion of *free type*, that is a type with no additional laws but the ones stated above.

Given a term with arbitrary constructors we can infer a *free type* for every term set *T* as follows: For any *T* value type *SetT* satisfies our notion of *free type* specified as follows:

```
data FreeType a = FreeType { captured :: Set a }
                | Full
  deriving (Eq, Ord, Show, Generic)

instance (Ord a, Eq a) => Semigroup (FreeType a) where
  Full <> _     = Full
  _    <> Full = Full
  a    <> b    = FreeType
              $ (Set.union `on` captured) a b
instance (Ord a, Eq a) => Monoid (FreeType a) where
  mempty  = FreeType Set.empty

instance (Ord a, Eq a, Show a)
      => Typelike (FreeType a) where
  beyond    = (==Full)

instance (Ord      a
         ,Eq       a
         ,Show     a)
      =>  FreeType a `Types` a where
  infer                   = FreeType . Set.singleton
  check Full          _term = True
  check (FreeType s)  term = term `Set.member` s
```

This definition is deemed sound, and may be applicable to a finite sets of terms or values. For a set of values: ["yes", "no", "error"], we may reasonably consider that type is an appropriate approximation of C-style enumeration, or Haskell-style ADT without constructor arguments.

However, the deficiency of this notion of *free type* is that it does not allow generalizing in infinite and recursive domains! It only allows utilizing objects from the sample.

3.5 Presence and absence constraint

We call this useful case a *presence or absence constraint*:

```
type role PresenceConstraint nominal

data PresenceConstraint a =
    Present
  | Absent
  deriving (Eq, Show, Typeable, Generic)

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

instance Monoid (PresenceConstraint a) where
  mempty             = Absent

instance Typelike (PresenceConstraint a) where
  beyond     = (==Present)

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

Altough it does not seem useful in the context implying that we always have at least one input value, it is important as it can be used to specify an empty array (and therefore, an element type for which we observed no values).

After seeing true value we also expect false, so we can say that the basic constraint for a boolean value is its presence or absence.

```
type BoolConstraint = PresenceConstraint Bool
```

Note that booleans and null values are both denoted by this trivial PresenceConstraint constraint. The same is valid for null values, as there is only one null value.

```
type NullConstraint = PresenceConstraint ()
```

Note that we treat null as separate basic types, and postpone treatment of the union til later.

Variants Variants of two mutually exclusive types are also simple. They can be implement them with a type related to Either type that assumes these types are exclusive:

```
data a :|: b = AltLeft  a
             | AltRight b
  deriving (Show, Eq, Generic)

instance (FromJSON  a
         ,FromJSON       b)
      =>  FromJSON (a :|: b) where
  parseJSON a =  AltLeft  <$> decodeEither
             <|> AltRight <$> decodeEither
```

In other words for Int :|: String type, we first control whether the value is a String, and if this check fails, we attempt to parse it as String.

Variant records are slightly more complicated, as it may be unclear which typing is better to use:

```
{"message": "Where can I submit my proposal?",
    "uid" : 1014}
{"error"  : "Authorization failed",
   "code" : 401}
```

```
data OurRecord =
  OurRecord { message :: Maybe String
```

```
              , error   :: Maybe String
              , code    :: Maybe Int
              , uid     :: Maybe Int }
```

Or:

```
data OurRecord2 = Message { message :: String
                          , uid     :: Int }
                | Error   { error   :: String
                          , code    :: Int }
```

The best attempt here is to rely on the available examples being reasonably exhaustive. That is, we can estimate how many examples we have for each, and how many of them match. Then, we compare this number with type complexity (with options being more complex to process, because they need additional case expression.) In such cases, the latter definition has only one choice (optionality), but we only have two samples to begin with so we cannot be sure.

In the case of having more samples, the pattern emerges:

```
{"error"  : "Authorization failed",
   "code":  401}
{"message": "Where can I submit my proposal?",
   "uid" : 1014}
{"message": "Sent it to HotCRP",
   "uid" :   93}
{"message": "Thanks!",
   "uid" : 1014}
{"error"  : "Missing user",
   "code":  404}
```

Type cost function Since we are interested in types with less complexity and less optionality, we will define cost function as follows:

```
newtype TyCost = TyCost Int
  deriving (Eq, Ord, Show, Enum, Num)

class Typelike ty
   => TypeCost ty where
  typeCost :: ty -> TyCost
  typeCost a | a == mempty = 0
             | otherwise   = 1

instance Semigroup TyCost where (<>) = (+)

instance Monoid    TyCost where mempty = 0
```

When presented with several alternate representations from the same set of observations, we will use this function to select the least complex representation of the type. For flat constraints as above, we infer that they offer no optionality when no observations occured (cost of 0), otherwise the cost is 1.

Considering that types beyond are to be avoided, we can assign conceptual *infinity* to these values. For the implementation purposes we will represent it by the value so high, that is unlikely to ever occur in practical types, but still small enough that we can add it without checking for overflow.

```
inf :: TyCost
inf = 100000000
```

Type cost should be non-negative, and non-decreasing when we add new observations to the type.

3.5.1 Object constraint

To avoid information loss, a constraint for JSON object type is introduced in such a way to **simultaneously gather information** about representing it either as a Map, or a record.

The typing of Map would be specified as follows:

```
data MappingConstraint =
  MappingConstraint {
    keyConstraint   :: StringConstraint
  , valueConstraint :: UnionType
  }
  | MappingNever
  deriving (Eq, Show, Generic, Typeable)

instance Monoid MappingConstraint where
  mempty = MappingNever

instance Typelike MappingConstraint where
  beyond MappingNever = False
  beyond MappingConstraint {..} =
      beyond keyConstraint
    && beyond valueConstraint

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

instance MappingConstraint `Types`
         Object where
  infer obj =
    MappingConstraint
      (foldMap infer $ Map.keys obj)
      (foldMap infer            obj)
  check MappingNever              _  = False
  check MappingConstraint {..} obj =
      all (check keyConstraint)
          (Map.keys obj)
    && all (check valueConstraint)
          (Foldable.toList obj)
```

Cost of mapping representation is a sum of cost of its fields:

```haskell
instance TypeCost MappingConstraint where
  typeCost MappingNever          = 0
  typeCost MappingConstraint {..} =
      typeCost keyConstraint
    + typeCost valueConstraint
```

Separately, we acquire the information about a possible typing of a JSON object as a record of values:

```haskell
data RecordConstraint =
    RCTop
  | RCBottom
  | RecordConstraint {
      fields :: HashMap Text UnionType
    } deriving (Show,Eq,Generic, Typeable)

instance Typelike RecordConstraint where
  beyond = (==RCTop)

instance Semigroup  RecordConstraint where
  RCBottom <> a        = a
  a        <> RCBottom = a
  RCTop    <> _        = RCTop
  _        <> RCTop    = RCTop
  a        <> b        = RecordConstraint $
    Map.unionWith (<>) (fields a)
                       (fields b)

instance Monoid     RecordConstraint where
  mempty = RCBottom

instance RecordConstraint `Types` Object
  where
    infer =  RecordConstraint
           . Map.fromList
           . fmap (second infer)
           . Map.toList
    check RCTop    _ = True
    check RCBottom _ = False
    -- FIXME: treat extra keys!!!
    check rc obj
            | all (`elem` Map.keys (fields rc))
                          (Map.keys  obj)  =
      and $ Map.elems $
        Map.intersectionWith  check
                            (fields rc)
                             obj
    check _  _ = False

instance TypeCost RecordConstraint where
  typeCost RCBottom = 0
  typeCost RCTop    = inf
  typeCost RecordConstraint { fields } =
    Foldable.foldMap typeCost fields
```

Observing that the two abstract domains considered above are independent, we can store the information about both options separately in a record[14] as follows:

```haskell
data ObjectConstraint = ObjectConstraint {
    mappingCase :: MappingConstraint
  , recordCase  :: RecordConstraint
  }
  | ObjectNever
  deriving (Eq,Show,Generic)

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

instance Monoid ObjectConstraint where
  mempty = ObjectNever

instance Typelike ObjectConstraint where
  beyond ObjectNever            = False
  beyond ObjectConstraint {..} =
       beyond mappingCase
    && beyond recordCase

instance ObjectConstraint `Types` Object where
  infer v = ObjectConstraint (infer v)
                             (infer v)
  check ObjectNever           _ = False
  check ObjectConstraint {..} v =
       check mappingCase v
    && check recordCase  v
```

It should be noted that this representation is similar to *intersection type*: any value that satisfies ObjectConstraint must conform to both mappingCase, and recordCase.

It should be noted that this *intersection approach* to address alternative union type representations benefits from *principal type property*, meaning that a principal type is used to simply acquire the information corresponding to different representations and handle it separately.

Since we plan to choose only one representation for the object, we can say that minimum cost of this type is a minimum of component costs:

```haskell
instance TypeCost ObjectConstraint where
  typeCost ObjectNever          = 0
  typeCost ObjectConstraint {..} =
```

[14] Choice of representation will be explained later. Here we only consider acquiring the information about possible values.

```
    typeCost mappingCase `min`
    typeCost recordCase
```

3.5.2 Array constraint

Similarly to the object type, ArrayConstraint is used to simultaneously obtain information about all possible representations of an array, including the following:

- an array of the same elements;
- a row with the type depending on a column.

We need to acquire the information for both alternatives separately, and then, to measure a relative likelihood of either cases, before mapping the union type to Haskell declaration.

Here, we specify the records for two different possible representations:

```
data ArrayConstraint  = ArrayConstraint {
    arrayCase :: UnionType
  , rowCase   :: RowConstraint
  }
  | ArrayNever
  deriving (Show, Eq, Generic)

instance Monoid ArrayConstraint where
  mempty = ArrayNever

instance Typelike ArrayConstraint where
  beyond ArrayNever = False
  beyond ArrayConstraint {..} =
      beyond arrayCase
    && beyond rowCase

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

<<row-constraint>>

instance ArrayConstraint `Types` Array
  where
    infer vs =
      ArrayConstraint
        (mconcat (infer <$>
              Foldable.toList vs))
        (infer            vs)
    check ArrayNever          vs = False
    check ArrayConstraint {..} vs =
        and (check arrayCase <$>
                 Foldable.toList vs)
      && check rowCase    vs
```

For the arrays, we plan to again choose only one of possible representations, so the cost of optionality is the lesser of the costs of the representation-specific constraints:

```
instance TypeCost ArrayConstraint where
  typeCost ArrayNever = 0
  typeCost ArrayConstraint {..} =
    typeCost arrayCase `min`
    typeCost rowCase
```

3.5.3 Row constraint

A row constraint is valid only if there is the same number of entries in all rows, which is represented by escaping the beyond set whenever there is an uneven number of columns.

```
data RowConstraint =
    RowTop
  | RowNever
  | Row      [UnionType]
  deriving (Eq,Show,Generic)

instance Typelike RowConstraint where
  beyond = (==RowTop)

instance Monoid RowConstraint where
  mempty = RowNever

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

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

In other words, RowConstraint is a *levitated semilattice* with a neutral element [12] over the content type that is a list of UnionType objects.

The cost of the row constraint is inferred in a similar manner as the cost of the record constraint:

```
instance TypeCost RowConstraint where
  typeCost  RowNever   = 0
  typeCost  RowTop     = inf
  typeCost (Row cols) = foldMap typeCost cols
```

3.5.4 Combining the above into a union type

It should note that given the constraints for the different type constructors, the union type can be considered as mostly a generic Monoid instance[8]:

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

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
    }
```

The generic structure of union type can be explained by the fact that the information contained in each record field is *independent* from the information contained in other fields. It means that we perform unification independently over different dimensions.

```
instance Monoid UnionType where
  mempty = UnionType {
      unionNull = mempty
    , unionBool = mempty
    , unionNum  = mempty
    , unionStr  = mempty
    , unionObj  = mempty
    , unionArr  = mempty
    }
```

As we described previously, the beyond set may correspond to either **accepting any value** or to **accepting no more information**. Its definition should be no surprise:

```
instance Typelike UnionType where
  beyond UnionType {..} =
      beyond unionNull
   && beyond unionBool
   && beyond unionNum
   && beyond unionStr
   && beyond unionObj
   && beyond unionArr
```

Inference breaks down disjoint alternatives corresponding to different record fields, depending on the constructor of a given value.

It enables implementing a clear and efficient treatment of different alternatives separately[15].

```
instance UnionType `Types` Value where
  infer (Bool   b) = mempty { unionBool = infer b  }
  infer  Null      = mempty { unionNull = infer () }
  infer (Number n) = mempty { unionNum  = infer n  }
  infer (String s) = mempty { unionStr  = infer s  }
  infer (Object o) = mempty { unionObj  = infer o  }
  infer (Array  a) = mempty { unionArr  = infer a  }
  check UnionType { unionBool } (Bool   b) =
            check unionBool          b
  check UnionType { unionNull }  Null      =
            check unionNull         ()
  check UnionType { unionNum  } (Number n) =
            check unionNum             n
  check UnionType { unionStr  } (String s) =
            check unionStr            s
  check UnionType { unionObj  } (Object o) =
            check unionObj            o
  check UnionType { unionArr  } (Array  a) =
            check unionArr            a
```

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
```

3.5.5 Overlapping alternatives

The essence of union type systems have long been dealing with the conflicting types providen in the input.

Motivated by the examples above, we also aim to address conflicting alternative assignments.

It is apparent that examples 4. to 6. hint at more than one assignment:

5. A set of lists of values that may correspond to Int, String, or null, or a table that has the same (and predefined) type for each values.
6. A record of fixed names or the mapping from hash to a single object type.

[15] The question may arise: what is the *union type* without *set union*? When the sets are disjoint, we just put the values in different bins to enable easier handling.

3.5.6 Counting observations

In this section, we discuss how to gather information about the number of samples supporting each alternative type constraint. To explain this, the other example can be considered:

```
{"samples":
[{"error"  : "Authorization failed",
  "code"   :  401}
,{"message": "Where can I submit my proposal?",
    "uid" : 1014}
,{"message": "Sent it to HotCRP",
    "uid" :   93}
,{"message": "Thanks!",
    "uid" : 1014}
,{"error"  : "Authorization failed",
    "code":  401}
]}
```

First, we need to identify it as a list of similar elements. Second, we note, that there are multiple instances of each record example. We consider that the best approach would be to use the multisets of inferred records instead of normal sets. To find the best representation, we can a type complexity, and attempt to minimize the term.

Next step is to detect the similarities between type descriptions introduced for different parts of the term:

```
{"samples"      :  [...],
 "last_message" : {"message": "Thanks!",
                   "uid" : 1014}
}
```

We can add the auxiliary information about a number of samples observed, and the constraint remains a Typelike object:

```haskell
data Counted a =
  Counted { count      :: Int
          , constraint :: a
          } deriving (Eq, Show, Generic)

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

instance Monoid  a
      => Monoid (Counted a) where
  mempty = Counted 0 mempty

instance Typelike          a
      => Typelike (Counted a) where
  beyond Counted {..} = beyond constraint

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

instance TypeCost           ty
      => TypeCost (Counted ty) where
  typeCost (Counted _ ty) = typeCost ty
```

We can interconnect Counted as parametric functor to select constraints to track auxiliary information.

It should be noted that Counted constraint is the first example that does not correspond to a semilattice, that is a<>a/=a.

This is because it is a Typelike object; however, it is not a type constraint in a conventional sense. Instead it counts the number of samples observed for the constraint inside so that we can decide on which alternative representation is best supported by evidence.

Therefore, at each step, we may need to maintain a **cardinality** of each possible value, and being provided with sufficient number of samples, we may attempt to detect.[16]

To preserve efficiency, we may need to merge whenever the number of alternatives in a multiset crosses the threshold.[17] We can attempt to narrow strings only in the cases when cardinality crosses the threshold.[18]

4 Selecting representations

4.1 Specifying heuristics to achieve better types

The final touch would be to perform the post-processing of an assigned type before generating it to make it more resilient to common uncertainties.

It should be noted that these assumptions may bypass the defined least-upper-bound criterion specified in the initial part of the paper; however, they prove to work well in practice.

4.1.1 Promoting empty type

If we have no observations corresponding to an array type, it can be inconvenient to disallow an array to contain any values at all. Therefore, we introduce a non-monotonic step of converting the mempty into a final Typelike object aiming to introduce a representation allowing the occurrence of any Value in the input. That still preserves the validity of the typing.

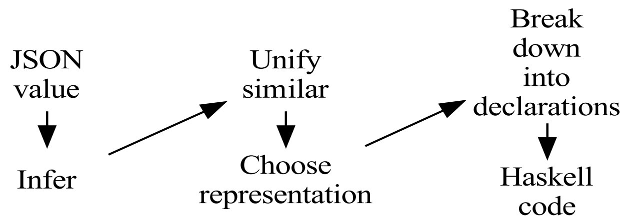
We note that the proposed program must not have any assumptions about these values; however, at the same time it should be able to print them for debugging purposes.

[16] If we detect a pattern too early, we risk to make the types too narrow to work with actual API answers.

[17] Option --max-alternative-constructors=N.

[18] Option --min-enumeration-cardinality.

4.2 Overall processing scheme



4.3 Simplification by identifying unification candidates

In most JSON documents, we observe that the same object can be described in different parts of sample data structures. Due to this reason, we compare the sets of labels assigned to all objects and propose to unify those that have more than 60% of identical labels.

For transparency, the identified candidates are logged for each user, and a user can also indicate them explicitly instead of relying on automation.

We conclude that this allows considerably decreasing the complexity of types and makes the output less redundant.

5 Future work

5.1 Scaling to type environments

In the present paper, we only discuss typing of tree-like values. However, it is natural to scale this approach to multiple types in APIs, in which different types are referred to by name and possibly contain each other.

To address these cases, we will show that the environment of Typelike objects is also Typelike, and that constraint unification can be extended in the same way.

5.2 Generic derivation of Typelike

It should be noted that Typelike instances for non-simple types usually follow one the two patterns:

1. for typing terms that have a finite sum of disjoint constructors, we bin this information by each constructor during the inference
2. for typing terms with multiple alternative representations, we infer all constraints separately for each representation by applying a different inference algorithm to the same term

In both cases, the derivation procedure of the Monoid, and Typelike instances is the same.

It allows using GHC Generics[13] to specify standard implementations for most of the boilerplate code.

It means that we only have to manually define the following:

- new constraint data types[19],
- inference from constructors (case 1), as well as providing the entirety of handling alternative constraints until we select representations.

[19]In many cases one can also rely on a generic constraint representation derived from Generic representation type Rep, that is when inference is mutually exclusive by term type constructors.

5.3 Conclusion

In the present study, we aimed to derive the types that were valid with respect to the provided specification, thereby obtaining the information from the input in most comprehensive way.

We defined type inference as representation learning and type system engineering as a meta-learning problem in which the **priors corresponding to the data structure induced typing rules**.

We also formulated the **union type discipline** as manipulation of Typelike commutative monoids, that represented knowledge about the data structure.

In addition, we proposed a union type system engineering methodology that was logically justified by a theoretical criteria. We demonstrated that it was capable of consistently explaining the decisions made in practice.

We consider that this kind of *formally justified type system engineering* can become widely used in practice, replacing *ad-hoc* approaches in the future.

The proposed approach may be used to underlie the way towards formal construction and derivation of type systems based on the specification of value domains and design constraints.

Bibliography

[1] Aeson: Fast JSON parsing and generation: 2011. *https://hackage.haskell.org/package/aeson*.

[2] A first look at quicktype: 2017. *https://blog.quicktype.io/first-look/*.

[3] Anderson, C. et al. 2005. Towards type inference for javascript. *ECOOP 2005 - object-oriented programming* (Berlin, Heidelberg, 2005), 428–452.

[4] Claessen, K. and Hughes, J. 2000. QuickCheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.* 35, 9 (Sep. 2000), 268–279. DOI:https://doi.org/10.1145/357766.351266.

[5] C Standard undefined behaviour versus Wittgenstein: *https://www.yodaiken.com/2018/05/20/depressing-and-faintly-terrifying-days-for-the-c-standard/*.

[6] C Undefined Behavior - Depressing and Terrifying (Updated): 2018. *https://www.yodaiken.com/2018/05/20/depressing-and-faintly-terrifying-days-for-the-c-standard/*.

[7] EnTangleD: A bi-directional literate programming tool: 2019. *https://blog.esciencecenter.nl/entangled-1744448f4b9f*.

[8] Generics example: Creating monoid instances: 2012. *https://www.yesodweb.com/blog/2012/10/generic-monoid*.

[9] GHCID - a new ghci based ide (ish): 2014. *http://neilmitchell.blogspot.com/2014/09/ghcid-new-ghci-based-ide-ish.html*.
[10] JSON autotype: Presentation for Haskell.SG: 2015. *https://engineers.sg/video/json-autotype-1-0-haskell-sg--429*.
[11] Knuth, D.E. 1984. Literate programming. *Comput. J.* 27, 2 (May 1984), 97–111. DOI:https://doi.org/10.1093/comjnl/27.2.97.
[12] Lattices: Fine-grained library for constructing and manipulating lattices: 2017. *http://hackage.haskell.org/package/lattices-2.0.2/docs/Algebra-Lattice-Levitated.html*.
[13] Magalhães, J.P. et al. 2010. A generic deriving mechanism for haskell. *SIGPLAN Not.* 45, 11 (Sep. 2010), 37–48. DOI:https://doi.org/10.1145/2088456.1863529.
[14] Mandelbaum, Y. et al. 2007. PADS/ML: A Functional Data Description Language. *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2007), 77–83.
[15] Pandoc: A universal document converter: 2000. *https://pandoc.org*.
[16] Petricek, T. et al. 2016. Types from Data: Making Structured Data First-Class Citizens in F#. *SIGPLAN Not.* 51, 6 (Jun. 2016), 477–490. DOI:https://doi.org/10.1145/2980983.2908115.
[17] Peyton Jones, S. 2019. Type inference as constraint solving: How ghc's type inference engine actually works. Zurihac keynote talk.
[18] scientific:Numbers represented using scientific notation: *https://hackage.haskell.org/package/scientific-0.3.6.2*.
[19] Sulzmann, M. and Stuckey, P.j. 2008. Hm(x) Type Inference is Clp(x) Solving. *J. Funct. Program.* 18, 2 (Mar. 2008), 251–283. DOI:https://doi.org/10.1017/S0956796807006569.
[20] Tiuryn, J. 1992. Subtype inequalities. *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science.* (1992), 308–315.
[21] Tiuryn, J. 1997. Subtyping over a lattice (abstract). *Computational logic and proof theory* (Berlin, Heidelberg, 1997), 84–88.
[22] Undefined behavior in 2017: 2017. *https://blog.regehr.org/archives/1520*.

Appendix: definition module headers

```
{-# language AllowAmbiguousTypes    #-}
{-# language DeriveGeneric          #-}
{-# language DuplicateRecordFields  #-}
{-# language FlexibleInstances      #-}
{-# language GeneralizedNewtypeDeriving #-}
{-# language MultiParamTypeClasses  #-}
{-# language NamedFieldPuns         #-}
{-# language PartialTypeSignatures  #-}
{-# language ScopedTypeVariables    #-}
{-# language TypeOperators          #-}
{-# language RoleAnnotations        #-}
{-# language ViewPatterns           #-}
{-# language RecordWildCards        #-}
{-# language OverloadedStrings      #-}
{-# ghc_options -Wno-orphans        #-}
module Unions where

import           Control.Arrow(second)
import           Data.Aeson
import           Data.Maybe(isJust,catMaybes)
import qualified Data.Foldable as Foldable
import           Data.Function(on)
import           Data.Text(Text)
import qualified Data.Text as Text
import qualified Data.Text.Encoding  as Text
import qualified Text.Email.Validate(isValid)
import qualified Data.Set  as Set
import           Data.Set(Set)
import           Data.Scientific
import           Data.String
--import           Data.List(sortBy)
import qualified Data.HashMap.Strict as Map
import           Data.HashMap.Strict(HashMap)
import           GHC.Generics(Generic)
import           Data.Hashable
import           Data.Typeable
import Data.Time.Format(iso8601DateFormat,parseTimeM,default
import Data.Time.Calendar(Day)


<<freetype>>
<<typelike>>
<<basic-constraints>>
<<array-constraint>>
<<object-constraint>>
<<presence-absence-constraints>>
<<union-type-instance>>
<<type>>
<<counted>>
<<typecost>>
<<representation>>

<<missing>>
```

Appendix: test suite

```
{-# language FlexibleInstances      #-}
{-# language Rank2Types             #-}
{-# language MultiParamTypeClasses  #-}
{-# language NamedFieldPuns         #-}
{-# language ScopedTypeVariables    #-}
{-# language StandaloneDeriving     #-}
{-# language TemplateHaskell        #-}
{-# language TypeOperators          #-}
```

```haskell
{-# language TypeApplications      #-}
{-# language TupleSections         #-}
{-# language UndecidableInstances  #-}
{-# language AllowAmbiguousTypes   #-}
{-# language OverloadedStrings     #-}
{-# language ViewPatterns          #-}
{-# ghc_options -Wno-orphans       #-}
module Main where

import qualified Data.Set              as Set
import qualified Data.Text             as Text
import qualified Data.ByteString.Char8 as BS
import Control.Monad(when)
import Data.FileEmbed
import Data.Maybe
import Data.Scientific
import Data.Aeson
import Data.Proxy
import Data.Typeable
import Test.Hspec
import Test.Hspec.QuickCheck
import Test.QuickCheck
import Test.Validity.Shrinking.Property
import           Test.Validity.Utils(nameOf)
import qualified GHC.Generics as Generic
import           Test.QuickCheck.Classes
import           System.Exit(exitFailure)

import Test.Arbitrary
import Test.LessArbitrary as LessArbitrary
import Unions

instance Arbitrary Value where
  arbitrary = fasterArbitrary

instance LessArbitrary Value where
  lessArbitrary = cheap $$$? genericLessArbitrary
    where
      cheap = LessArbitrary.oneof [
                pure        Null
              , Bool   <$> lessArbitrary
              , Number <$> lessArbitrary
              ]

instance LessArbitrary          a
      => LessArbitrary (Counted a) where

instance LessArbitrary a
      => Arbitrary     (Counted a) where
  arbitrary = fasterArbitrary

instance Arbitrary Object where
  arbitrary = fasterArbitrary

instance Arbitrary Array where
  arbitrary = fasterArbitrary

class Typelike         ty
   => ArbitraryBeyond ty where
  arbitraryBeyond :: CostGen ty

instance ArbitraryBeyond (PresenceConstraint a) where
  arbitraryBeyond = pure Present

instance ArbitraryBeyond StringConstraint where
  arbitraryBeyond = pure SCAny

instance ArbitraryBeyond IntConstraint where
  arbitraryBeyond = pure IntAny

instance ArbitraryBeyond NumberConstraint where
  arbitraryBeyond = pure NCFloat

instance ArbitraryBeyond RowConstraint where
  arbitraryBeyond = pure RowTop

instance ArbitraryBeyond RecordConstraint where
  arbitraryBeyond = pure RCTop

instance ArbitraryBeyond MappingConstraint where
  arbitraryBeyond =
    MappingConstraint <$$$> arbitraryBeyond
                      <*>   arbitraryBeyond

instance (Ord                      a
         ,Show                     a
         )
    => ArbitraryBeyond (FreeType a) where
  arbitraryBeyond = pure Full

instance ArbitraryBeyond ObjectConstraint where
  arbitraryBeyond = do
    ObjectConstraint <$$$> arbitraryBeyond
                     <*>   arbitraryBeyond

instance ArbitraryBeyond ArrayConstraint where
  arbitraryBeyond = do
    ArrayConstraint <$$$> arbitraryBeyond
                    <*>   arbitraryBeyond

instance ArbitraryBeyond UnionType where
  arbitraryBeyond =
    UnionType       <$$$> arbitraryBeyond
                    <*>   arbitraryBeyond
                    <*>   arbitraryBeyond
                    <*>   arbitraryBeyond
                    <*>   arbitraryBeyond
                    <*>   arbitraryBeyond
```

```haskell

arbitraryBeyondSpec :: forall        ty.
                      (ArbitraryBeyond ty
                      ,Typelike        ty)
                    => Spec
arbitraryBeyondSpec =
  prop "arbitrarybeyond returns terms beyond" $
    (beyond <$> (arbitraryBeyond :: CostGen ty))

instance LessArbitrary Text.Text where
  lessArbitrary = Text.pack <$> lessArbitrary

instance Arbitrary Text.Text where
  arbitrary = Text.pack <$> arbitrary

instance Arbitrary Scientific where
  arbitrary = scientific <$> arbitrary
                         <*> arbitrary

instance (LessArbitrary          a
         ,Ord                    a)
      =>  LessArbitrary (FreeType a) where

instance Arbitrary (FreeType Value) where
  arbitrary = fasterArbitrary
  {-shrink  Full            = []
  shrink (FreeType elts) = map FreeType
                         $ shrink elts-}

instance (Ord              v
         ,Show             v)
      => TypeCost (FreeType v) where
  typeCost  Full         = inf
  typeCost (FreeType s) = TyCost $ Set.size s

{-
instance (Eq                      a
         ,Ord                     a
         ,GenUnchecked            a
         ,LessArbitrary           a
         ,LessArbitrary (FreeType a)
         ,Arbitrary     (FreeType a))
      =>  GenUnchecked  (FreeType a) where
  genUnchecked    = fasterArbitrary
  shrinkUnchecked Full                   = []
  shrinkUnchecked FreeType { captured } =
      map (FreeType . Set.fromList)
    $ shrinkUnchecked
    $ Set.toList captured

instance Validity (FreeType a) where
  validate _ = validate True
 -}

instance LessArbitrary (PresenceConstraint a) where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     (PresenceConstraint a) where
  arbitrary = fasterArbitrary

instance LessArbitrary IntConstraint where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     IntConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary NumberConstraint where
  lessArbitrary = genericLessArbitrary

instance Arbitrary     NumberConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary StringConstraint where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     StringConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary ObjectConstraint where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     ObjectConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary RecordConstraint where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     RecordConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary ArrayConstraint where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     ArrayConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary RowConstraint where
  lessArbitrary = genericLessArbitraryMonoid

instance Arbitrary     RowConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary MappingConstraint where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     MappingConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary UnionType where
  lessArbitrary = genericLessArbitraryMonoid

instance Arbitrary     UnionType where
  arbitrary = fasterArbitrary

{-
```

```
instance GenUnchecked UnionType where
  genUnchecked    = arbitrary
  shrinkUnchecked = shrink
 -}

{-
instance Validity UnionType where
  validate _ = validate True-}

shrinkSpec :: forall    a.
             (Arbitrary a
             ,Typeable  a
             ,Show      a
             ,Eq        a
             )
           => Spec
shrinkSpec = prop ("shrink on " <> nameOf @a)
           $ doesNotShrinkToItself arbitrary (shrink :: a -> [a])

allSpec :: forall         ty v.
           (Typeable        ty
            ,Arbitrary      ty
            ,Show           ty
            ,Types          ty v
            ,ArbitraryBeyond ty
            ,Arbitrary         v
            ,Show              v
            ) => Spec
allSpec = describe (nameOf @ty) $ do
  arbitraryBeyondSpec @ty
  shrinkSpec    @ty

<<typelike-spec>>
<<types-spec>>
<<typecost-laws>>

main :: IO ()
main  = do
  exitFailure
  putStrLn "NumberConstraint"
  {-sample $ arbitrary @Value
  sample $ arbitrary @NullConstraint
  sample $ arbitrary @NumberConstraint
  sample $ arbitrary @RowConstraint
  sample $ arbitrary @RecordConstraint
  sample $ arbitrary @ArrayConstraint
  sample $ arbitrary @MappingConstraint
  sample $ arbitrary @ObjectConstraint-}

  lawsCheckMany
    [typesSpec (Proxy :: Proxy (FreeType Value) )
               (Proxy :: Proxy Value      ) True
    ,typesSpec (Proxy :: Proxy NumberConstraint )
               (Proxy :: Proxy Scientific) True
    ,typesSpec (Proxy :: Proxy StringConstraint )
               (Proxy :: Proxy Text.Text ) True
    ,typesSpec (Proxy :: Proxy BoolConstraint   )
               (Proxy :: Proxy Bool       ) True
    ,typesSpec (Proxy :: Proxy NullConstraint   )
               (Proxy :: Proxy ()         ) True
    ,typesSpec (Proxy :: Proxy RowConstraint    )
               (Proxy :: Proxy Array      ) True
    ,typesSpec (Proxy :: Proxy ArrayConstraint  )
               (Proxy :: Proxy Array      ) True
    ,typesSpec (Proxy :: Proxy MappingConstraint)
               (Proxy :: Proxy Object     ) True
    ,typesSpec (Proxy :: Proxy RecordConstraint )
               (Proxy :: Proxy Object     ) True
    ,typesSpec (Proxy :: Proxy ObjectConstraint )
               (Proxy :: Proxy Object     ) True
    ,typesSpec (Proxy :: Proxy UnionType        )
               (Proxy :: Proxy Value      ) True
    ,typesSpec (Proxy :: Proxy (Counted NumberConstraint))
               (Proxy :: Proxy Scientific      ) False
    ]
  representationSpec

typesSpec :: (Typeable  ty
             ,Typeable     term
             ,Monoid    ty
             ,Arbitrary ty
             ,Arbitrary    term
             ,Show      ty
             ,Show         term
             ,Eq        ty
             ,Eq           term
             ,Typelike  ty
             ,Types     ty term
             ,TypeCost  ty
             )
          =>  Proxy     ty
          ->  Proxy        term
          ->  Bool -- idempotent?
          -> (String, [Laws])
typesSpec (tyProxy   :: Proxy ty)
          (termProxy :: Proxy term) isIdem =
  (nameOf @ty <> " types " <> nameOf @term, [
      arbitraryLaws          tyProxy
    , eqLaws                 tyProxy
    , monoidLaws             tyProxy
    , commutativeMonoidLaws tyProxy
    , typeCostLaws           tyProxy
    , typelikeLaws           tyProxy
    , arbitraryLaws                  termProxy
    , eqLaws                         termProxy
    , typesLaws              tyProxy termProxy
    ]<>idem)
  where
```

```
    idem | isIdem    = [idempotentSemigroupLaws tyProxy]
         | otherwise = []

typesLaws :: (           ty `Types` term
             ,Arbitrary ty
             ,Arbitrary            term
             ,Show       ty
             ,Show                 term
             )
          => Proxy ty
          -> Proxy term
          -> Laws
typesLaws (_ :: Proxy ty) (_ :: Proxy term) =
  Laws "Types" [("mempty contains no terms"
                ,property $
                  mempty_contains_no_terms        @ty @term)
               ,("beyond contains all terms"
                ,property $
                  beyond_contains_all_terms       @ty @term)
               ,("inferred type contains its term"
                ,property $
                  inferred_type_contains_its_term @ty @term)
               ]

<<representation-examples>>

representationTest :: String -> [Value] -> HType -> IO Bool
representationTest name values repr = do
    if foundRepr == repr
      then do
        putStrLn $ "*** Representation test " <> name <> " succeeded."
        return True
      else do
        putStrLn $ "*** Representation test " <> name <> " failed: "
        putStrLn $ "Values        : " <> show values
        putStrLn $ "Inferred type : " <> show inferredType
        putStrLn $ "Representation: " <> show foundRepr
        putStrLn $ "Expected      : " <> show repr
        return False
  where
    foundRepr :: HType
    foundRepr = toHType inferredType
    inferredType :: UnionType
    inferredType = foldMap infer values

readJSON :: HasCallStack
         => BS.ByteString -> Value
readJSON = fromMaybe ("Error reading JSON file")
         . decodeStrict
         . BS.unlines
         . filter notComment
         .
         BS.lines
  where
    notComment (BS.isPrefixOf "//" -> True) = False
    notComment _                            = True

representationSpec :: IO ()
representationSpec  = do
  b <- sequence
    [representationTest "1a" example1a_values example1a_repr
    ,representationTest "1b" example1b_values example1b_repr
    ,representationTest "1c" example1c_values example1c_repr
    ,representationTest "2"  example2_values  example2_repr
    ,representationTest "3"  example3_values  example3_repr
    ,representationTest "4"  example4_values  example4_repr
    ,representationTest "5"  example5_values  example5_repr
    ,representationTest "6"  example6_values  example6_repr]
  when (not $ and b) $
    exitFailure
```

Appendix: package dependencies

```
name: union-types
version: '0.1.0.0'
category: Web
author: Anonymous
maintainer: example@example.com
license: BSD-3
extra-source-files:
- CHANGELOG.md
- README.md
dependencies:
- base
- aeson
- containers
- text
- hspec
- QuickCheck
- unordered-containers
- scientific
- hspec
- QuickCheck
- validity
- vector
- unordered-containers
- scientific
- genvalidity
- genvalidity-hspec
- genvalidity-property
- time
- email-validate
- generic-arbitrary
- mtl
- hashable
library:
  source-dirs: src
  exposed-modules:
```

```
  - Unions
tests:
  spec:
    main: Spec.hs
    source-dirs:
      - test/lib
      - test/spec
    dependencies:
      - union-types
      - mtl
      - random
      - transformers
      - hashable
      - quickcheck-classes
      - file-embed
      - bytestring
  less-arbitrary:
    main: LessArbitrary.hs
    source-dirs:
      - test/lib
      - test/less
    dependencies:
      - union-types
      - mtl
      - random
      - transformers
      - hashable
      - quickcheck-classes
      - quickcheck-instances
```

Appendix: representation of generated Haskell types

We will not delve here into identifier conversion between JSON and Haskell, so it suffices that we have an abstract datatypes for Haskell type and constructor identifiers:

```
newtype HConsId  = HConsId String
  deriving (Eq,Ord,Show,Generic,IsString)
newtype HFieldId = HFieldId String
  deriving (Eq,Ord,Show,Generic,IsString)
newtype HTypeId  = HTypeId String
  deriving (Eq,Ord,Show,Generic,IsString)
```

For each single type we will either describe its exact representation or reference to the other definition by name:

```
data HType =
    HRef HTypeId
  | HApp HTypeId [HType]
  | HADT [HCons]
  deriving (Eq, Ord, Show, Generic)
```

For syntactic convenience, we will allow string literals to denote type references:

```
instance IsString HType where
  fromString = HRef . fromString
```

When we define a single constructor, we allow field and constructor names to be empty strings (""), assuming that the relevant identifiers will be put there by post-processing that will pick names using types of fields and their containers[???].

```
data HCons = HCons {
               name :: HConsId
             , args :: [(HFieldId, HType)]
             }
  deriving (Eq, Ord, Show, Generic)
```

At some stage we want to split representation into individually named declarations, and then we use environment of defined types, with an explicitly named toplevel type:

```
data HTypeEnv = HTypeEnv {
    toplevel :: HTypeId
  , env      :: HashMap HTypeId HType
  }
```

When checking for validity of types and type environments, we might need a list of predefined identifiers that are imported:

```
predefinedHTypes :: [HType]
predefinedHTypes = [
    "Data.Aeson.Value"
  , "()"
  , "Double"
  , "String"
  , "Int"
  , "Date" -- actually: "Data.Time.CalendarDay"
  , "Email" -- actually: "Data.Email"
  ]
```

Consider that we also have an htop value that represents any possible JSON value. It is polimorphic for ease of use:

```
htop :: IsString s => s
htop = "Data.Aeson.Value"
```

5.4 Code for selecting representation

Below is the code to select representation, as described in sec. 4.

To convert union type discipline to strict Haskell type representations, we need to join the options to get the actual representation:

```
toHType :: ToHType ty => ty -> HType
toHType  = joinAlts . toHTypes

joinAlts     :: [HType] -> HType
joinAlts []   =  htop -- promotion of empty type
joinAlts alts =  foldr1 joinPair alts
  where
    joinPair a b = HApp ":|:" [a, b]
```

Considering the assembly of UnionType, we join all the options, and convert nullable types to Maybe types

```
instance ToHType UnionType where
  toHTypes UnionType {..} =
      prependNullable unionNull opts
    where
      opts = concat [toHTypes unionBool
                    ,toHTypes unionStr
                    ,toHTypes unionNum
                    ,toHTypes unionArr
                    ,toHTypes unionObj]

prependNullable :: PresenceConstraint a -> [HType] -> [HType]
prependNullable Present tys = [HApp "Maybe" [joinAlts tys]]
prependNullable Absent  tys =               tys
```

The type class returns a list of mutually exclusive type representations:

```
class Typelike ty
   => ToHType  ty where
  toHTypes :: ty -> [HType]
```

Conversion of flat types is quite straightforward:

```
instance ToHType BoolConstraint where
  toHTypes Absent  = []
  toHTypes Present = ["Bool"]
instance ToHType NumberConstraint where
  toHTypes NCNever = []
  toHTypes NCFloat = ["Double"]
  toHTypes NCInt   = ["Int"]
instance ToHType StringConstraint where
  toHTypes  SCAny      = ["String"]
  toHTypes  SCEmail    = ["Email"]
  toHTypes  SCDate     = ["Date"]
  toHTypes (SCEnum es) = [HADT $
                          mkCons <$> Set.toList es
                         ]
    where
      mkCons = (`HCons` [])
             .  HConsId
             .  Text.unpack
  toHTypes  SCNever    = []
```

For array and object types we pick the representation which presents the lowest cost of optionality:

```
instance ToHType ObjectConstraint where
  toHTypes ObjectNever            = []
  toHTypes ObjectConstraint {..} =
    if typeCost recordCase <= typeCost mappingCase
      then toHTypes recordCase
      else toHTypes mappingCase

instance ToHType RecordConstraint where
  toHTypes  RCBottom = []
  toHTypes  RCTop    = [htop] -- should never happen
  toHTypes (RecordConstraint fields) =
      [HADT
          [HCons "" $ fmap convert $ Map.toList fields]
      ]
    where
      convert (k,v) = (HFieldId $ Text.unpack k
                      ,toHType v)

instance ToHType MappingConstraint where
  toHTypes MappingNever = []
  toHTypes MappingConstraint {..} =
    [HApp "Map" [toHType keyConstraint
                ,toHType valueConstraint
                ]]

instance ToHType RowConstraint where
  toHTypes  RowNever   = []
  toHTypes  RowTop     = [htop]
  toHTypes (Row cols) =
    [HADT
        [HCons "" $ fmap (\ut -> ("", toHType ut)) cols]
    ]

instance ToHType ArrayConstraint where
  toHTypes ArrayNever            = []
  toHTypes ArrayConstraint {..} =
    if   typeCost arrayCase <= typeCost rowCase
      -- || count <= 3
      then [toHType arrayCase]
      else [toHType rowCase  ]
```

Appendix: Missing pieces of code

In order to represent FreeType for the Value, we need to add Ord instance for it:

```
instance Ord       Value where
  compare = compare `on` hash
```

For validation of dates and emails, we import functions from Hackage:

```
isValidDate :: Text -> Bool
isValidDate = isJust
            . parseDate
            . Text.unpack
  where
    parseDate :: String -> Maybe Day
    parseDate  = parseTimeM True
                   defaultTimeLocale $
                   iso8601DateFormat Nothing

isValidEmail :: Text -> Bool
isValidEmail = Text.Email.Validate.isValid
           . Text.encodeUtf8
```

Appendix: Damas-Milner as Typelike