Towards a more perfect union type

Michał J. Gajda

Migamake Pte Ltd,
mjgajda@migamake.com
WWW company page: https://migamake.com

Abstract. We present a principled theoretical framework for dealing with union types, and show its work in practice on JSON data structures. The framework poses a union type inference as a learning problem from multiple examples. The categorical framework is generic, and easily extensible.

1 Introduction

Typing dynamic languages has been long considered a challenge [5]. The importance of the task grown with the 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 [6–8].

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

Union type providers The earliest practical effort to apply union types to JSON inference to generate Haskell types[6]. 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[8]. The other attempt to automatically infer schemas has been introduced in the PADS project [9]. Nevertheless, it has not specified a generalized type-system design methodology. One approach uses Markov chains to derive JSON types [7]. This 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 a sound underlying theory. Regular expression types were also used to type XML documents[10], which does not allow for selecting alternative representation. Therefore, we summarize that there are several previously introduced approaches that provide partially satisfactory results. In the present study, we aim to expand these proposals to enable systematic addition of features, and automatic validation of types.
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 [11]. It is also worth noting that traditional Damas-Milner type disciplines enjoy decidability, and embrace the laws of soundness, and subject-reduction. However these laws often prove too strict during type system extension, dependent type systems often abandon subject-reduction, and practical programming language type systems are either undecidable[12], or even sometimes even unsound[13].

Early approach approach used lattice structure on the types [14], which is more stringent that ours, since it requires idempotence of unification (as join operation), as well as complementary meet operation with the same properties. Semantic subtyping approach provides a characterization of set-based union, intersection, and complement types[15, 16], which allows model subtype containment on first order types and functions. This model relies on building a model using infinite sets in a set theory, and thus is not immediately apparent as implementable1. We are also not aware of a type inference framework that consistently and completely preserve information in face of inconsistencies nor errors, beyond using bottom and expanding to infamous undefined behaviour[17].

We propose a categorical and constructive framework that preserves the soundness in inference, while making compromises are allowed by consistent approximations. Indeed our experience is that most of the type system implementation may be generic.

2 Motivation

Here, we consider several examples paraphrased from JSON API descriptions. We provide these examples in the form of a few JSON objects, along with desired representation as Haskell data declaration.

1. Subsets of data within a single constructor:
   a. API argument is an email – it is a subset of valid String values, that can be validated on the client side.
   b. 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
   c. 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 {}

1 Since we are interested in engineering the type system further, a simple implementation serves as ultimate evidence of extensibility.
3. Variant fields: *Answer to a query is either a number of registered objects, or String "unavailable"*- this is integer value (Int) or a String (Int :|: 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:

```haskell
{-
  \{ "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 }
```

5. Arrays corresponding to records:

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

6. Maps of identical objects (example from [7]):

```haskell
{-
  \{ "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" \} }
-

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, f_54fced :: O_6408f5,
                          f_6c9589 :: O_6408f5 }

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

It also suggests that a user might decide to explicitly add evidence for one of alternative representations in the case when samples are insufficient. (like in case of a single element dictionary.)
2.1 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\(^2\) 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 a fewer number of degrees of freedom compared with the others, while conforming to a commonly occurring structure. We denote it as an information content principle.

Given these principles, and examples of frequently occurring 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 occurrence.

3 Problem definition

As we focus on JSON, we utilize Haskell encoding of the JSON term for convenient reading (from Aeson package [20]); specified as follows:

\[
\text{data Value} = \text{Object (Map String Value)} \mid \text{Array [Value]} \mid \text{Null} \\
\quad \mid \text{Number Scientific} \mid \text{String Text} \mid \text{Bool Bool}
\]

3.1 Defining type inference

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 \text{Semigroup} interface operation \(<\) to merge types inferred from different observations. If Typelike is semilattice, then \(<\) is meet operation (least upper bound). Note that this approach is dual to traditional unification that narrows down solutions and thus is join operation (greatest lower bound). We use neutral element of the \text{Monoid} to indicate a type corresponding to no observations.

\[^2\text{Compiler feature of checking for unmatched cases.}\]
class Semigroup ty where (<>): ty -> ty -> ty

In other words, we can say that mempty (or bottom) of the corresponds to situation where no information was accepted about a possible value (no term seen, not even a null). It is neutral element of Typelike. For example, an empty array [] can be referred to as an array type with mempty as an element type. This represents the view that <> always gathers more information about the type, as opposed to the traditional unification that always narrows down possible solutions. We describe the laws as QuickCheck [21] properties so that unit testing can be implemented to detect obvious violations.

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 (arriving at complete bounded semilattice), but this is to narrow for our purpose of monotonically gathering information

However, since we defined generalization operator <> as information fusion (corresponding to unification in categorically dual case of a strict type systems.), we may encounter difficulties in assuring that no information has been lost during the generalization. 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.

This observation lets us go well beyond typing statement of gradual type inference as discovery problem from partial information[22]. Here we 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 would be fully polymorphic type ∀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).

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<>a) for any value of a. In other words the beyond set is an attractor of <> on both sides. However we do not require idempotence

3 Examples will be provided later.
4 In this case: beyond (Error _) = True | otherwise = False.
5 May sound similar until we consider adding more information to the type.
6 So both for ∀a(a<>a) and ∀a.(a<>a), the result is kept in the beyond set.
of $\langle\rangle$, which is uniformly present in union type frameworks based on lattice[14] and set-based approaches[15]. Concerning union types, the key property of the beyond set, is that it is closed to information acquisition:

In this way, we can specify other elements of beyond set instead of a single top. When under strict type discipline, like that of Haskell[12], we seek to enable each element of the beyond set to contain at least one error message.

We abolish the semilattice requirement that has been conventionally assumed for type constraints [23], 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 5 in sec. 2, we need to perform non-monotonic step of choosing alternative representation after monotonic steps of merging all the information.

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[14]. 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 on ExampleMap in sec. 2, 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 [14, 23].

Now we are ready to present relation of typing and its laws. In order to preserve proper English word order, we state that $\text{ty} 
\text{\'Types\'} \text{val}$ instead of classical $\text{val:ty}$. 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. The minimal definition of typing inference relation and type checking relation is formulated as consistency between these two operations.

```haskell
class Typelike ty => ty \text{\'Types\'} val where
  infer :: val -> ty
  check :: ty -> val -> Bool
```

First, we note that to describe no information, $\text{mempty}$ cannot correctly type any term. Second important rule of typing is that all terms are typed successfully by any value in the beyond set. Finally we state the most intuitive rule for typing: a type inferred from a term, must always be valid for that particular 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.) The minimal Typelike instance is the one that contains only $\text{mempty}$ corresponding to the case of no sample data received.

---

7 Which use Heyting algebras, which have more assumptions that the lattice approaches.

8 It should be noted that many but not all type constraints are semilattice. Please refer to the counting example below.
Perfect union type

\[
\begin{array}{c}
\text{Type}_1 \times \text{Type}_2 \\
\pi_1 \downarrow \\
\text{Type}_1 \\
\text{Value}_1 \\
\text{infer} \\
\end{array}
\quad
\begin{array}{c}
\text{Type}_1 \leftrightarrow \text{Type}_2 \\
\leftrightarrow \text{Type}_2 \\
\text{check value}_1 \\
\text{check with Type}_1 \\
\text{True} \\
\end{array}
\quad
\begin{array}{c}
\text{Type}_2 \\
\leftrightarrow \text{Type}_2 \\
\text{check value}_2 \\
\text{check with Type}_2 \\
\text{Value}_2 \\
\end{array}
\quad
\begin{array}{c}
\text{Type}_2 \\
\pi_2 \\
\end{array}
\]

Fig. 1: Categorical diagram for Typelike.

and a single beyond element for all values permitted. We will define it below as PresenceConstraint in sec. 3.3. 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 constraints 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.

3.2 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 \texttt{a}: false we can expect that in particular cases, we may obtain that \texttt{a}: true. After noting that \texttt{b}: 123, we expect that \texttt{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 the type system as an inference problem. As the purpose is to deliver the most descriptive types, we assume that we need to obtain a wider view rather than focusing on a free type and applying it to 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 in 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 rules of type design as short description as possible, inference must be a contravariant functor with regards to constructors. For example, if \texttt{AType x y types \{"a": X, "b": Y\}}, then \texttt{x} must type \texttt{X}, and \texttt{y} must type \texttt{Y}.

\footnote{The shortest one according to the information complexity principle.}
3.3 Constraint definition

Flat type constraints Let us first consider typing of flat type: String (similar treatment should be given to the Number.type.)

```haskell
data StringConstraint = SCDate | SCEmail | SCEnum (Set Text) | SCNever {- mempty -} | SCAny {- beyond -}

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

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

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 `Set.union` b) < 10 = SCEnum (a <> b)
  _ <> _ = SCAny
```

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 $\text{Set } T$ satisfies our notion of free type specified as follows:

```haskell
data FreeType a = FreeType { captured :: Set a } | Full

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, 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 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.

**Presence and absence constraint** We call the degenerate case of Typelike a presence or absence constraint. It just checks that the type contains at least one observation of the input value, or no observations at all. It is important as it can be used to specify an element type of an empty array. After seeing true value we also expect false, so we can say that it is also basic constraint for boolean values. The same is valid for null values, as there is only one null value to ever observe.

type BoolConstraint = PresenceConstraint Bool

type NullConstraint = PresenceConstraint ()

data PresenceConstraint a = Present | Absent

**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, we denote it by :|:. In other words for Int :|: String type, we first control whether the value is an Int, 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:

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

data OurRecord = OurRecord { message, error :: Maybe String, code, uid :: Maybe Int }

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 Maybe field (on the toplevel, optionality is one), while the former definition has four Maybe fields (optionality is four). When we obtain more samples, the pattern emerges:
Type cost function Since we are interested in types with less complexity and less optionality, we will define cost function as follows:

```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
```

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 occurred (cost of `mempty` is 0), otherwise the cost is 1. Type cost should be non-negative, and non-decreasing when we add new observations to the type.

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, with the optionality cost being a sum of optionalities in its fields.

```haskell
data MappingConstraint = MappingNever -- mempty |
  MappingConstraint { keyConstraint :: StringConstraint ,
                     valueConstraint :: UnionType } |
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. Note that `RCTop` never actually occurs during inference. That is, we could have represented the `RecordConstraint` as a `Typelike` with an empty beyond set. The merging of constraints would be simply merging of all column constraints.

```haskell
data RecordConstraint =
  RCTop {- beyond -*} | RCBottom {- mempty -*} |
  RecordConstraint { fields :: HashMap Text UnionType }
```
instance RecordConstraint `Types` Object where
    infer = RecordConstraint . Map.fromList
             . fmap (second infer) . Map.toList
    check RecordConstraint {fields} obj =
        all (`elem` Map.keys fields) -- all object keys
        (Map.keys obj) -- present in type
        && and (Map.elems $ Map.intersectionWith -- values check
                check fields obj)
        && all isNullable (Map.elems $ fields `Map.difference` obj)
        -- absent values are nullable

Observing that the two abstract domains considered above are independent,
we can store the information about both options separately in a record\textsuperscript{10}. It
should be noted that this representation is similar to intersection type: any
value that satisfies ObjectConstraint must conform to both mappingCase, and
recordCase. Also this intersection approach in order to address alternative union
type representations benefits from principal type property, meaning that a prin-
cipal type is used to simply acquire the information corresponding to different
representations and handle it separately. Since we plan to choose only one repre-
sentation for the object, we can say that minimum cost of this type is a minimum
of component costs.

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

instance TypeCost ObjectConstraint where
    typeCost ObjectConstraint {..} = typeCost mappingCase
                                      `min` typeCost recordCase

Array constraint Similarly to the object type, ArrayConstraint is used to
simultaneously obtain information about all possible representations of an array,
differentiating between an array of the same elements, and 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 = ArrayNever -- mempty
    | ArrayConstraint { arrayCase :: UnionType, rowCase :: RowConstraint }

Semigroup operation just merges information on the components, and the
same is done when inferring types or checking them: 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.

\textsuperscript{10} The choice of representation will be explained later. Here we only consider acquiring
information about possible values.
instance ArrayConstraint `Types` Array where
  infer vs = ArrayConstraint {
    arrayCase = mconcat (infer <$> Foldable.toList vs)
    , rowCase = infer vs
  }

  check ArrayNever vs = False
  check ArrayConstraint {..} vs =
    and (check arrayCase <$> Foldable.toList vs)
    && check rowCase vs

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. Row constraint remains valid only if
both constraint describe record of the same length, otherwise we yield RowTop
to indicate that it is no longer valid. In other words, RowConstraint is a levitated semilattice[24][11] with a neutral element over the content type that is a list of
UnionType objects.

data RowConstraint = RowTop | RowNever | Row [UnionType]

Combining the 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[25]. Merging information with <> and mempty follow the pattern
above, by just lifting operations on the component.

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

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 anti-unification indepen-
dently over different dimensions.

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[12]. Since union
type is all about optionality, we need to sum all options from different alterna-
tives to obtain its typeCost.

instance UnionType `Types` Value where
  infer (Bool b) = mempty { unionBool = infer b }

---

[11] Levitated lattice is created by appending distinct bottom and top to a set that does not possess them by itself.
[12] 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.
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

Overlapping alternatives The essence of union type systems have long been dealing with the conflicting types provided 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: in example 5, a set of lists of values that may correspond to \texttt{Int}, \texttt{String}, or \texttt{null}, or a table that has the same (and predefined) type for each values; in example 6 A record of fixed names or the mapping from hash to a single object type.

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:

```json
{"history": [
  {"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:

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

We can add the auxiliary information about a number of samples observed, and the constraint will remain \texttt{Typelike} object. The \texttt{Counted} constraint counts the number of samples observed for the constraint inside so that we can decide on which alternative representation is best supported by evidence. It should be
noted that Counted constraint is the first example that does not correspond to a semilattice, that is \( a \triangleleft a \neq a \). This is natural for a Typelike object; it is not a type constraint in a conventional sense, just an accumulation of knowledge.

```haskell
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)
```

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

4 Finishing touches

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.

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 program using our types must not have any assumptions about these values; however, at the same time it should be able to print them for debugging purposes.

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

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

---

13 If we detect a pattern too early, we risk to make the types too narrow to work with actual API responses.
types are referred to by name and possibly contain each other. To address these cases, we plan to show that the environment of Typelike objects is also Typelike, and that constraint generalization (anti-unification) can be extended in the same way.

It should be noted that many Typelike instances for non-simple types usually follow one the two patterns of (1) for 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 alternative representation. In both cases, Generic derivation procedure for the Monoid, Typelike, and TypeCost instances is possible\[26]. This allows us to design a type system by declaring datatypes themselves, and leave implementation to the compiler. Manual implementation would be only left for special cases, like StringConstraint and Counted constraint.

Finally we believe that we can explain duality of categorical framework of Typelike categories and use unification instead of generalization (anti-unification) as a type inference mechanism. The beyond set would then correspond to a set of error messages, and a result of the inference would represent a principal type in Damas-Milner sense.

5.1 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 followed a strictly constructive procedure, that can be implemented generically.

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. Knuth, D.E.: Literate programming. Comput. J. 27, 97–111 (1984). https: //doi.org/10.1093/comjnl/27.2.97.
2. Hidding, J.: EnTangleD: A bi-directional literate programming tool, https: //blog.esciencecenter.nl/entangled-174448f4b9f.
3. McFarlane, J.: Pandoc: A universal document converter, https://pandoc.org.
4. Mitchell, N.D.: GHCID - a new ghci based ide (ish), http://neilmitchell.blogspot.com/2014/09/ghcid-new-ghci-based-ide-ish.html.
5. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for javascript. In: Black, A.P. (ed.) ECOOP 2005 - object-oriented programming. pp. 428–452. Springer Berlin Heidelberg, Berlin, Heidelberg (2005).
6. Anonymous: JSON autotype: Presentation for Haskell.SG, https://engineers.sg/video/json-autotype-1-0-haskell-sg--429.
7. Siegel, D.: A first look at quicktype, https://blog.quicktype.io/first-look/.
8. Petricek, T., Guerra, G., Syme, D.: Types from Data: Making Structured Data First-Class Citizens in F#. SIGPLAN Not. 51, 477–490 (2016). https://doi.org/10.1145/2980983.2908115.
9. Mandelbaum, Y., Fisher, K., Walker, D., Fernandez, M., Gleyzer, A.: PADS/ML: A Functional Data Description Language. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 77–83. Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1190216.1190231.
10. Hosoya, H., Pierce, B.: XDuce: A typed xml processing language. (2000).
11. Sulzmann, M., Stuckey, P.J.: Hm(x) Type Inference is Clp(x) Solving. J. Funct. Program. 18, 251–283 (2008). https://doi.org/10.1017/S0956796807006569.
12. Peyton Jones, S.: Type inference as constraint solving: How ghc's type inference engine actually works, https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/, (2019).
13. Henry P. Tung and other contributors: https://github.com/microsoft/TypeScript/issues/9825.
14. Tiuryn, J.: Subtyping over a lattice (abstract). In: Gottlob, G., Leitsch, A., and Mundici, D. (eds.) Computational logic and proof theory. pp. 84–88. Springer Berlin Heidelberg, Berlin, Heidelberg (1997).
15. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping. In: Proceedings 17th annual ieee symposium on logic in computer science. pp. 137–146 (2002).
16. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM. 55, (2008). https://doi.org/10.1145/1391289.1391293.
17. Yodaiken, V.: C Standard undefined behaviour versus Wittgenstein, https://www.yodaiken.com/2018/05/20/depressing-and-faintly-terrifying-days-for-the-c-standard/.
18. Fairhead, H.: C Undefined Behavior - Depressing and Terrifying (Updated), https://www.yodaiken.com/2018/05/20/depressing-and-faintly-terrifying-days-for-the-c-standard/.
19. Cuq, P., Regehr, J.: Undefined behavior in 2017, https://blog.regehr.org/archives/1520.
20. O’Sullivan, B.: Aeson: Fast JSON parsing and generation, https://hackage.haskell.org/package/aeson.
21. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of haskell programs. SIGPLAN Not. 35, 268–279 (2000). https://doi.org/10.1145/357766.351266.
Perfect union type

22. Siek, J., Taha, W.: Gradual typing for objects. In: Proceedings of the 21st European Conference on Object-Oriented Programming. pp. 2–27. Springer-Verlag, Berlin, Heidelberg (2007).

23. Tiuryn, J.: Subtype inequalities. [1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science. 308–315 (1992).

24. Grenrus, O.: Lattices: Fine-grained library for constructing and manipulating lattices, http://hackage.haskell.org/package/lattices-2.0.2/docs/Algebra-Lattice-Levitated.html.

25. Snoyman, M.: Generics example: Creating monoid instances, https://www.yesodweb.com/blog/2012/10/generic-monoid.

26. Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for haskell. SIGPLAN Not. 45, 37–48 (2010). https://doi.org/10.1145/2088456.1863529.

27. Michal J. Gajda, D.K.: Fast XML/HTML tools for Haskell: XML Typelift and improved Xeno, (2020).

**Appendix: definition module headers**

```haskell
{-# language AllowAmbiguousTypes #-}
{-# language DeriveGeneric #-}
{-# language DuplicateRecordFields #-}
{-# language FlexibleInstances #-}
{-# language GeneralizedNewtypeDeriving #-}
{-# language MultiParamTypeClasses #-}
{-# language NamedFieldPuns #-}
{-# language PartialTypeSignatures #-}
{-# language ScopedTypeVariables #-}
{-# language TypeOperators #-}
{-# language RoleAnnotations #-}
{-# language ViewPatterns #-}
{-# options_ghc -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 as IsValid
import qualified Data.Set as Set
import Data.Set (Set)
import Data.Scientific
```
import Data.String
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, defaultTimeLocale)
import Data.Time.Calendar (Day)
import Missing

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

Appendix: test suite

{-# language FlexibleInstances  #-}
{-# language Rank2Types  #-}
{-# language MultiParamTypeClasses #-}
{-# language MultiWayIf  #-}
{-# language NamedFieldPuns  #-}
{-# language ScopedTypeVariables  #-}
{-# language StandaloneDeriving  #-}
{-# language TemplateHaskell  #-}
{-# language TypeOperators  #-}
{-# language TypeApplications  #-}
{-# language TupleSections  #-}
{-# language UndecidableInstances  #-}
{-# language AllowAmbiguousTypes  #-}
{-# language OverloadedStrings  #-}
{-# language ViewPatterns  #-}
{-# options_ghc -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

instance ArbitraryBeyond a
  => ArbitraryBeyond (Counted a) where
    arbitraryBeyond = Counted <$> LessArbitrary.choose (0, 10000)

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

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
  {-
  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
,ArbitraryBeyond 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
, ArbitraryBeyond 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)
,"fusion keeps terms"
,property $
fusion_keeps_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
    thenStr = putStrLn $ "*** Representation test " <> name <> " succeeded."
    return True
  elseStr = 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:
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:

```haskell
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:

```haskell
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:

```haskell
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[27].

```haskell
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:

```haskell
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:

```haskell
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 polymorphic for ease of use:

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

### Code for selecting representation

Below is the code to select Haskell type representation. To convert union type discipline to strict Haskell type representations, we need to join the options to get the actual representation:

```haskell
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 = []
```haskell
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)
```

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

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

```haskell
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:

```haskell
instance Ord Value where
  compare = compare `on` hash
```
For validation of dates and emails, we import functions from Hackage:

```haskell
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

instance (Hashable k
  ,Hashable v)
  => Hashable (HashMap k v) where
hashWithSalt s = hashWithSalt s
  . Foldable.toList

instance Hashable v
  => Hashable (V.Vector v) where
hashWithSalt s = hashWithSalt s
  . Foldable.toList

-- instance Hashable Scientific where
-- instance Hashable Value where

Then we put all the missing code in the module:
```

```haskell
{-# 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  #-}
```
{-# options_ghc -Wno-orphans #-}
module Missing 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 qualified Data.Vector as V
import qualified Data.HashMap.Strict as Map
import GHC.Generics (Generic)
import Data.Hashable
import Data.Typeable
import Data.Time.Format (iso8601DateFormat,parseTimeM,defaultTimeLocale)
import Data.Time.Calendar (Day)

<<missing>>