DATA DEFINITION LANGUAGE IN HASKELL

for simple cross language data exchange

THE PROBLEM

Example: Web application

- Haskell server
- PureScript in the browser (client)

use same data structures for communication

ATTEMPT #1

Handwritten code in Haskell

```haskell
data FrameResult
  = FrameResult
  { frRenderTimes :: Vector Float
  , frImageWidth :: Int
  , frImageHeight :: Int
  }

data RenderJobResult
  = RenderJobResult FrameResult
  | RenderJobError String
```

let's define our data types

ATTEMPT #1

Serialization code in Haskell

```
instance ToJSON RenderJobResult where
  toJSON v = case v of
    RenderJobResult arg0 -> object
      [ "tag" .= ("RenderJobResult" :: Text)
      , "arg0" .= arg0 ]
    RenderJobError arg0 -> object
      [ "tag" .= ("RenderJobError" :: Text)
      , "arg0" .= arg0 ]

instance FromJSON RenderJobResult where
  parseJSON (Object obj) = do
    tag <- obj .: "tag"
    case tag :: Text of
      "RenderJobResult" -> RenderJobResult <$> obj .: "arg0"
      "RenderJobError" -> RenderJobError <$> obj .: "arg0"
  parseJSON _ = mzero
```

using Aeson JSON library

ATTEMPT #1

PureScript: Data definitions

```
data FrameResult
  = FrameResult
  { frRenderTimes :: Array Float
  , frImageWidth :: Int
  , frImageHeight :: Int
  }

data RenderJobResult
  = RenderJobResult FrameResult
  | RenderJobError String
```

ATTEMPT #1

PureScript: Serialization

```
instance encodeJsonRenderJobResult :: EncodeJson RenderJobResult wher
  encodeJson v = case v of
    RenderJobResult arg0 -> "tag" := "RenderJobResult" ~>
                            "arg0" := arg0 ~> jsonEmptyObject
    RenderJobError arg0 -> "tag" := "RenderJobError" ~>
                           "arg0" := arg0 ~> jsonEmptyObject

instance decodeJsonRenderJobResult :: DecodeJson RenderJobResult wher
  decodeJson json = do
    obj <- decodeJson json
    tag <- obj .? "tag"
    case tag of
      "RenderJobResult" -> RenderJobResult <$> obj .? "arg0"
      "RenderJobError" -> RenderJobError <$> obj .? "arg0"
```

using Argonaut JSON library

GENERATE CODE

Thrift, ProtoBuf, MessagePack, etc

Problem: no support for Algebraic Data Types!

Let's create an EDSL!

DATA DEFINITION EDSL

```
definitions = do
  -- test result
  data_ "FrameResult" $ do
    constR_ "FrameResult"
      [ "frRenderTimes" #:: Array Float
      , "frImageWidth"  #:: Int
      , "frImageHeight" #:: Int
      ]

  data_ "RenderJobResult" $ do
    const_ "RenderJobResult"  ["FrameResult"]
    const_ "RenderJobError"   [String]
```

do notation is perfect for this!

BUILTIN TYPES

String, Bool, Int, Float, Map, Array, Maybe, etc.

```
data Type
  = Int
  | Float
  | Bool
  | String
  -- higher order types
  | Array { type_ :: Type }
  | List { type_ :: Type }
  | Maybe { type_ :: Type }
  | Map { key_ :: Type, value_ :: Type }
  -- user defined
  | Data { name_ :: String }
```

DEFINITIONS

```
data DataDef = DataDef
  { dataName      :: String
  , constructors  :: [ConstructorDef]
  , instances     :: [Instance]
  }
data ConstructorDef = ConstructorDef
  { name          :: String
  , fields        :: [Field]
  }
data Field = Field
  { fieldName :: String
  , fieldType :: Type
  }
data Instance = Show | Eq | Ord
```

OPERATORS

```
type DDef = Writer ([DataDef],[String])
type CDef = Writer ([ConstructorDef],[Instance])

data_ :: String -> CDef () -> DDef ()
data_ n l = tell ([let (c,i) =
  execWriter l in DataDef n c i],mempty)
const_ :: String -> [Type] -> CDef ()
const_ n t = tell ([ConstructorDef n
                    [Field a b | Field a b <- map toField t]]
                   ,mempty)
constR_ :: String -> [Field] -> CDef ()
constR_ n t = tell ([ConstructorDef n
                    [Field a b | Field a b <- map toField t]]
                    ,mempty)
(#::) :: String -> Type -> Field
a #:: b = Field a b
```

CODE GENERATOR

EDE is a templating language written in Haskell

- Write templates for languages
- Dump DataDef list into an EDE template
- Easy to pass data: just derive Generic for DataDef

CODE GENERATOR

Haskell data definitions

```
{% for t in dataAndType %}
{% case t.value | constType %}
{% when "DataDef" %}
data {{ t.value.dataName }}{% for c in t.value.constructors %}
{% if c.value.fields | hasFieldNames %}
  {% if c.first %}={% else %}|{% endif %} {{ c.value.name }}
{% for f in c.value.fields %}{% if f.first %}  { {%else%}  , {%endif%
{% endfor %}
  }
{% else %}
  {% if c.first %}={% else %}|{% endif %} {{ c.value.name }}{% for f
  deriving (Show, Eq, Ord)
{% when "TypeAlias" %}
type {{ t.value.aliasName }} = {{ t.value.aliasType | hsType }}
{% endcase %}
```

Template Haskell! :)

CODE GENERATOR

Aeson: ToJSON & FromJSON instances

```
{% for t in definitions %}
instance ToJSON {{ t.value.dataName }} where
  toJSON v = case v of{% for c in t.value.constructors %}{% if c.valu
    {{ c.value.name }}{..} -> object
      [ "tag" .= ("{{ c.value.name }}" :: Text){% for f in c.value.fi
      , "{{ f.value.fieldName }}" .= {{ f.value.fieldName }}{% endfor
      ]{% else %}
    {{ c.value.name }}{% for f in c.value.fields %} arg{{ f.index0 }}
instance FromJSON {{ t.value.dataName }} where
  parseJSON (Object obj) = do
    tag <- obj .: "tag"
    case tag :: Text of{% for c in t.value.constructors %}{% if c.val
      "{{ c.value.name }}" -> do{% for f in c.value.fields %}
        {{ f.value.fieldName }} <- obj .: "{{ f.value.fieldName }}"{%
        pure $ {{ c.value.name }}{% for f in c.value.fields %}
          {% if f.first %}{ {% else %}, {%endif%}{{ f.value.fieldName
```

CODE GENERATOR: JAVA

Algebraic Data Type as Classes

```
public class RenderJobResult {
  public enum Tag {
    RenderJobResult,
    RenderJobError
  }
  public Tag tag;

  public class RenderJobResult_ extends RenderJobResult {
    public FrameResult _0;
    public RenderJobResult_() { tag = RenderJobResult.Tag.RenderJobRe
  }
  public class RenderJobError_ extends RenderJobResult {
    public String _0;
    public RenderJobError_() { tag = RenderJobResult.Tag.RenderJobErr
  }
}
```

CODE GENERATOR: JAVA JSON

No type class support: no branching on types

```
public class JSON {
  // JSON deserializer
  public enum Type {
    Int,
    Float,
    String,
    Array_Float,
    FrameResult,
    RenderJobResult
  }
```

Do it explicitly!

CODE GENERATOR: JAVA JSON

```
public static Object fromJSON(Type type, Object rawObj) throws JSONEx
    switch (type) {
      case Float: return ((Number)rawObj).floatValue();
      case Array_Float: {
        JSONArray obj = (JSONArray)rawObj;
        ArrayList<float> v = new ArrayList<float> ();
        for (int i = 0; i < obj.length(); i++) {
          v.add((Float)fromJSON (Type.Float, obj.get(i)));
        }
        return v;
      }
      case FrameResult: {
        JSONObject obj = (JSONObject)rawObj;
        String tag = obj.getString("tag");
        switch (tag) {
          case "FrameResult": {
```

Generated from template

THAT'S ALL

Check it out on GitHub.

QUESTIONS?