channable

Generic Metadata Resolution

Victor Miraldo & Infra

January 6, 2023

Who is Channable?

Who is Channable?

SaaS company in Utrecht, with $\sim$250 employees.

Who is Channable?

SaaS company in Utrecht, with ∼250 employees.

We develop a tool that empowers advertisers and online businesses to manage, scale, and optimize their marketing.

Who is Channable?

SaaS company in Utrecht, with ~250 employees.

We develop a tool that empowers advertisers and online businesses to manage, scale, and optimize their marketing.

This tool is composed by a number of services in the backend, most of which are written in Haskell.

Haskell at Channable

- We have 8 (out of 12) services written in Haskell,

Haskell at Channable

- We have 8 (out of 12) services written in Haskell,
- using 770 source files (not counting tests),

Haskell at Channable

- We have 8 (out of 12) services written in Haskell,
- using 770 source files (not counting tests),
- which span around 150k lines,

Haskell at Channable

- We have 8 (out of 12) services written in Haskell,
- using 770 source files (not counting tests),
- which span around 150k lines,
- ...

Haskell at Channable

- We have 8 (out of 12) services written in Haskell,
- using 770 source files (not counting tests),
- which span around 150k lines,
- ...
- and *zero* uses of `Generic1`!

Today’s Agenda

Today's Agenda

Start using `Generic1` in Channable!

Today’s Agenda

Start using `Generic1` in Channable!

- A quick look at our notification service (aka *Megaphone*)

Today's Agenda

Start using `Generic1` in Channable!

- A quick look at our notification service (aka *Megaphone*)
- Metadata resolution

Today's Agenda

Start using `Generic1` in Channable!

- A quick look at our notification service (aka *Megaphone*)
- Metadata resolution
- Generic programming in 3 minutes

Today's Agenda

Start using `Generic1` in Channable!

- A quick look at our notification service (aka *Megaphone*)
- Metadata resolution
- Generic programming in 3 minutes
- *Generic* metadata resolution

Megaphone

Megaphone

Megaphone

- Receives events and computes notifications from them:
  - Notify the user?
  - Notify sales?
  - ???

Megaphone

- Receives events and computes notifications from them:
  - Notify the user?
  - Notify sales?
  - ???
- Dispatch notifications through their appropriate channel
  - Currently only e-mail

Megaphone

- Receives events and computes notifications from them:
  - Notify the user?
  - Notify sales?
  - ???
- Dispatch notifications through their appropriate channel
  - Currently only e-mail
- Started as two DB tables and a python script many years ago

Megaphone

- Receives events and computes notifications from them:
  - Notify the user?
  - Notify sales?
  - ???
- Dispatch notifications through their appropriate channel
  - Currently only e-mail
- Started as two DB tables and a python script many years ago
- Rewritten in Haskell circa 2020

Megaphone: A Sketch

```
forever $ do
  nextEvent <- readChan eventChan
  emails <- computeEmails nextEvent
  mapM_ (liftIO . sendMail) emails
```

Megaphone: A Sketch

```
forever $ do
  nextEvent <- readChan eventChan
  emails <- computeEmails nextEvent
  mapM_ (liftIO . sendMail) emails
```

In reality, more complex:

Megaphone: A Sketch

```
forever $ do
  nextEvent <- readChan eventChan
  emails <- computeEmails nextEvent
  mapM_ (liftIO . sendMail) emails
```

In reality, more complex:

- Generated emails sent *exactly* once

Megaphone: A Sketch

```
forever $ do
  nextEvent <- readChan eventChan
  emails <- computeEmails nextEvent
  mapM_ (liftIO . sendMail) emails
```

In reality, more complex:

- Generated emails sent *exactly* once
- No event is ever lost

channable

Megaphone: A Sketch

```
forever $ do
  nextEvent <- readChan eventChan
  emails <- computeEmails nextEvent
  mapM_ (liftIO . sendMail) emails
```

In reality, more complex:

- Generated emails sent *exactly* once
- No event is ever lost
- Even across restarts and crashes

Megaphone Events

Handles 27 different event types. For example:


```
{
    "id": "f8cedbb1-52f6-4d32-8aaf-f587ead057b0",
    "origin": "somewhere.com:4242",
    "payload": {
        "type": "cancellation",
        "value": {
            "company_id": 456,
            "reason": "missing_functionality",
            "additional": "I wish Channable would be rewritten in C, it has too few segfaults"
        }
    }
}
```

Megaphone Events

Handles 27 different event types. For example:

```
{
    "id": "f8cedbb1-52f6-4d32-8aaf-f587ead057b0",
    "origin": "somewhere.com:4242",
    "payload": {
        "type": "cancellation",
        "value": {
            "company_id": 456,
            "reason": "missing_functionality",
            "additional": "I wish Channable would be rewritten in C, it has too few segfaults"
        }
    }
}
```

```
data Payload =
  ...
  | CustomerCancellationEvent CompanyId CancellationReason AdditionalInfo
```

Megaphone Events

Handles 27 different event types. For example:


```
{
    "id": "f8cedbb1-52f6-4d32-8aaf-f587ead057b0",
    "origin": "somewhere.com:4242",
    "payload": {
        "type": "cancellation",
        "value": {
            "company_id": 456,
            "reason": "missing_functionality",
            "additional": "I wish Channable would be rewritten in C, it has too few segfaults"
        }
    }
}
```


Resolved inside `computeEmails`

```
data Payload =
  ...
  | CustomerCancellationEvent CompanyId CancellationReason AdditionalInfo
```

Megaphone: The Task

We had:

```
computeEmails :: Event -> Db.Session [Message]
computeEmails (Event id orig payload) = case payload of
  CustomerCancellationEvent cId reason info -> do
    cInfo <- getCompanyInfo cId
    ...
  ...
```

Megaphone: The Task

We had:

```
computeEmails :: Event -> Db.Session [Message]
computeEmails (Event id orig payload) = case payload of
  CustomerCancellationEvent cId reason info -> do
    cInfo <- getCompanyInfo cId
    ...
  ...
```


We want:

```
computeEmails = fmap (concatMap pureComputeMessage) . resolvePayload
```

channable

Megaphone: The Task

We had:

```
computeEmails :: Event -> Db.Session [Message]
computeEmails (Event id orig payload) = case payload of
  CustomerCancellationEvent cId reason info -> do
    cInfo <- getCompanyInfo cId
    ...
  ...
```


We want:

```
computeEmails = fmap (concatMap pureComputeMessage) . resolvePayload
```

Can’t change **Event**: full backwards compatibility.

Megaphone: Metadata Resolution

Would like two types:

```
data UnresolvedPayload =
  ...
  | CustomerCancellationEvent CompanyId CancellationReason AdditionalInfo
```

Megaphone: Metadata Resolution

Would like two types:

```
data UnresolvedPayload =
  ...
  | CustomerCancellationEvent CompanyId CancellationReason AdditionalInfo

data ResolvedPayload =
  ...
  | CustomerCancellationEvent (CompanyId, CompanyInfo) CancellationReason AdditionalInfo
```

Megaphone: Metadata Resolution

Would like two types:

```
data UnresolvedPayload =
  ...
  | CustomerCancellationEvent CompanyId CancellationReason AdditionalInfo


data ResolvedPayload =
  ...
  | CustomerCancellationEvent (CompanyId, CompanyInfo) CancellationReason AdditionalInfo
```

Don’t want to duplicate 27 constructors (… and growing)

channable

Megaphone: Events and Metainformation

One teaspoon of `-XDataKinds` and a pinch of `-XGADTs`:

```
data MetaStatus = Resolved | Unresolved
```

Megaphone: Events and Metainformation

One teaspoon of `-XDataKinds` and a pinch of `-XGADTs`:

```
data MetaStatus = Resolved | Unresolved

data Meta unr f :: MetaStatus -> Type where
  UnresolvedMeta :: unr -> Meta unr f 'Unresolved
```

Megaphone: Events and Metainformation

One teaspoon of `-XDataKinds` and a pinch of `-XGADTs`:

```
data MetaStatus = Resolved | Unresolved

data Meta unr f :: MetaStatus -> Type where
  UnresolvedMeta :: unr -> Meta unr f 'Unresolved
  ResolvedMeta :: unr -> ResolverResult unr f -> Meta unr f 'Resolved
```


channable

Megaphone: Events and Metainformation

One teaspoon of `-XDataKinds` and a pinch of `-XGADTs`:

```haskell
data MetaStatus = Resolved | Unresolved

data Meta unr f :: MetaStatus -> Type where
  UnresolvedMeta :: unr -> Meta unr f 'Unresolved
  ResolvedMeta :: unr -> ResolverResult unr f -> Meta unr f 'Resolved

class (Monad m) => Resolver m unr f where
  type ResolverResult unr f :: Type
  resolve :: unr -> m (ResolverResult unr f)
```

Megaphone: Events and Metainformation

Now, we get to:

```
data PayloadWithMeta (r :: MetaStatus) =
  ...
  | CustomerCancellationEvent
      (Meta CompanyId GetCompanyInfo r)
      CancellationReason
      AdditionalInfo
```

Megaphone: Events and Metainformation

Now, we get to:

```
data PayloadWithMeta (r :: MetaStatus) =
  ...
  | CustomerCancellationEvent
      (Meta CompanyId GetCompanyInfo r)
      CancellationReason
      AdditionalInfo

instance Resolver Db.Session CompanyId GetCompanyInfo where
  type ResolverResult CompanyId GetCompanyInfo = CompanyInfo
  resolve = doSomeQueriesHere
```


channable

Megaphone: Checkpoint

Megaphone: Checkpoint

- $1 + 1$ *gratis:* `PayloadWithMeta`!

channable

Megaphone: Checkpoint

- $1 + 1$ *gratis:* `PayloadWithMeta`!
- Fully backwards compatible. Old `Payload` is isomorphic to `PayloadWithMeta 'Unresolved`.

Megaphone: Checkpoint

- $1 + 1$ *gratis:* `PayloadWithMeta`!
- Fully backwards compatible. Old `Payload` is isomorphic to `PayloadWithMeta 'Unresolved`.
- Prevent duplication: only 10 resolvers needed.

Megaphone: Checkpoint

- $1 + 1$ *gratis:* `PayloadWithMeta`!
- Fully backwards compatible. Old `Payload` is isomorphic to `PayloadWithMeta 'Unresolved`.
- Prevent duplication: only 10 resolvers needed.

Megaphone: Checkpoint

- $1 + 1$ *gratis:* `PayloadWithMeta`!
- Fully backwards compatible. Old `Payload` is isomorphic to `PayloadWithMeta 'Unresolved`.
- Prevent duplication: only 10 resolvers needed.

Still need:

```
resolvePayload :: PayloadWithMeta 'Unresolved -> Db.Session (PayloadWithMeta 'Resolved)
```

Generic Programming

Representing Datatypes Uniformly

The core idea behind generic programming is that we can represent some datatypes uniformly,

Representing Datatypes Uniformly

The core idea behind generic programming is that we can represent some datatypes uniformly, enabling writing programs over that representation.

Representing Datatypes Uniformly

The core idea behind generic programming is that we can represent some datatypes uniformly, enabling writing programs over that representation.

For example, the type:

```haskell
data Tree a = Leaf a | Node Int (Tree a) (Tree a)
```

Representing Datatypes Uniformly

The core idea behind generic programming is that we can represent some datatypes uniformly, enabling writing programs over that representation.

For example, the type:

```haskell
data Tree a = Leaf a | Node Int (Tree a) (Tree a)
```

is isomorphic to the type:

```haskell
type UTree a = Either a (Int, Tree a, Tree a)
```

Representing Datatypes Uniformly

Almost...

```
UTree Int == Either Int (Int, Tree Int, Tree Int)
```

Lost information about which `Int` corresponds to the parameter a.

Representing Datatypes Uniformly

Almost...

```
UTree Int == Either Int (Int, Tree Int, Tree Int)
```

Lost information about which `Int` corresponds to the parameter a.

Solution? Move to kind `* -> *`:

```
type UTree = Id :+: (Const Int :*: (Tree :*: Tree))

data (f :*: g) x = f x :*: g x
data (f :+: g) x = L1 (f x) | R1 (g x)
data Const a x = Const a
data Id x = Id x
```

Representing Datatypes Uniformly

To and from uniform representations:

```
to :: Tree a -> UTree a
to (Leaf i) = L1 (Id i)
to (Node n t u) = R1 (Const n :*: (t :*: u))

from :: UTree a -> Tree a
from = ...
```

The `sumNodes` Function

```
sumNodes :: Tree a -> Int
sumNodes (Leaf _) = 0
sumNotes (Node n t u) = n + sumNodes t + sumNodes u
```

The Generic `sumNodes` Function

Why not define:

```
class GSum (t :: * -> *) where
  gsum :: t x -> Int
```

The Generic `sumNodes` Function

Why not define:

```
class GSum (t :: * -> *) where
  gsum :: t x -> Int

instance (GSum t, GSum u) => GSum (t :*: u) where
  gsum (x :*: y) = gsum x + gsum y

instance (GSum t, GSum u) => GSum (t :+: u) where
  gsum (L1 x) = gsum x
  gsum (R1 y) = gsum y

instance GSum (Const Int) where
  gsum (Const n) = n

instance GSum f where
  gsum _ = 0
```

channable

The Generic `sumNodes` Function

We could now define:

```haskell
sumNodes :: Tree a -> Int
sumNodes = gsum . to
```

The Generic `sumNodes` Function

We could now define:

```haskell
sumNodes :: Tree a -> Int
sumNodes = gsum . to
```

We could do even better:

```haskell
sumNodes :: (Generic a) => a -> Int
sumNodes = gsum . to

class Generic a where
  type Rep a :: * -> *
  to :: a -> Rep a
  from :: Rep a -> a
```

The Generic `sumNodes` Function

We could now define:

```haskell
sumNodes :: Tree a -> Int
sumNodes = gsum . to
```

We could do even better:

```haskell
sumNodes :: (Generic a) => a -> Int
sumNodes = gsum . to
```

```haskell
class Generic a where
  type Rep a :: * -> *
  to :: a -> Rep a
  from :: Rep a -> a
```

```haskell
instance Generic Tree where
  type Rep Tree = UTree
  to = to
  from = from
```


channable

Generic Programming

The main take-away is:

- write functions by induction on structure of datatypes.

Generic Programming

The main take-away is:

- write functions by induction on structure of datatypes.

Multiple libraries: `GHC.Generics`, `generics-sop`, `generics-simplistic`, ...

Generic Programming

The main take-away is:

- write functions by induction on structure of datatypes.

Multiple libraries: `GHC.Generics`, `generics-sop`, `generics-simplistic`, ...

Interested? Some references at the end!

Back to Megaphone

Megaphone: Finishing up

Megaphone: Finishing up

Used `GHC.Generics` to write:

```
genericResolve :: (Generic1 f , ...) => f 'Unresolved -> m (f 'Resolved)
```

Megaphone: Finishing up

Used `GHC.Generics` to write:

```
genericResolve :: (Generic1 f , ...) => f 'Unresolved -> m (f 'Resolved)
```

Before, only had: `Event` → `Db.Session [Message]`.

channable

Megaphone: Finishing up

Used `GHC.Generics` to write:

```
genericResolve :: (Generic1 f , ...) => f 'Unresolved -> m (f 'Resolved)
```

Before, only had: `Event` → `Db.Session [Message]`.

Now, that arrow is factored into:

Conclusion

channable

Conclusion

- Decomposing software is orders of magnitude *harder than composing* software.

Conclusion

- Decomposing software is orders of magnitude *harder than composing* software.
- A strong type system helped a lot!

Conclusion

- Decomposing software is orders of magnitude *harder than composing* software.
- A strong type system helped a lot!
- Generic programming was paramount:

Conclusion

- Decomposing software is orders of magnitude *harder than composing* software.
- A strong type system helped a lot!
- Generic programming was paramount:
  - Some events had nested occurences of metadata.

Conclusion

- Decomposing software is orders of magnitude *harder than composing* software.
- A strong type system helped a lot!
- Generic programming was paramount:
  - Some events had nested occurences of metadata.
  - Very easy to add new events.

Conclusion

- Decomposing software is orders of magnitude *harder than composing* software.
- A strong type system helped a lot!
- Generic programming was paramount:
  - Some events had nested occurences of metadata.
  - Very easy to add new events.
  - Moderately easy to add new resolvers.

Conclusion

- Decomposing software is orders of magnitude *harder than composing* software.
- A strong type system helped a lot!
- Generic programming was paramount:
  - Some events had nested occurences of metadata.
  - Very easy to add new events.
  - Moderately easy to add new resolvers.
- Decomposing the pure parts will be much easier.

Bibliography

- Haskell Wiki: GHC.Generics, 2022.
  `https://wiki.haskell.org/GHC.Generics`.
- Edsko de Vries and Andres Löh.
  True sums of products.
  WGP, 2014.
  `http://edsko.net/pubs/TrueSumsOfProducts.pdf`.
- Alejandro Serrano and Victor Cacciari Miraldo.
  Generic programming of all kinds.
  Haskell Symposyum, 2018.
  `https://victorcmiraldo.github.io/data/hask2018_draft.pdf`.

channable

channable

Generic Metadata Resolution

Victor Miraldo & Infra

January 6, 2023