channable

File-Format Migrations

on the Type-Level!

Victor Miraldo

January 5, 2024

Who is Channable?

Who is Channable?

SaaS company in Utrecht, with ~300 employees.

Who is Channable?

SaaS company in Utrecht, with ~300 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 ~300 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 11 (out of 15) services written in Haskell,

Haskell at Channable

- We have 11 (out of 15) services written in Haskell,
- using about 1000 source files (not counting tests),

Haskell at Channable

- We have 11 (out of 15) services written in Haskell,
- using about 1000 source files (not counting tests),
- which `wc -l` to around 200k lines!

The Problem

Context

Per-user dynamic configuration, stored in a disk not too far away:

```
reasonable_programming_languages="functional"
type_system_and_eval=strong_lazy
```


Context

Per-user dynamic configuration, stored in a disk not too far away:

```
reasonable_programming_languages="functional"
type_system_and_eval=strong_lazy
```


Two services, A and B, rely on these configs

A Wild Change Appears

Now we want to split `type_system_and_eval` into two settings:

```
reasonable_programming_languages="functional"
type_system=strong
eval=lazy
```


A Wild Change Appears

Now we want to split `type_system_and_eval` into two settings:

```
reasonable_programming_languages="functional"
type_system=strong
eval=lazy
```

How do we update A and B to use this *new format*?

Potential Solutions

Potential Solutions

1. Pause A and B; change all user settings, deploy new versions of A and B.

Potential Solutions

1. Pause A and B; change all user settings, deploy new versions of A and B. Not great! Downtime, difficult to coordinate, prone to errors.

Potential Solutions

1. Pause A and B; change all user settings, deploy new versions of A and B. Not great! Downtime, difficult to coordinate, prone to errors.
2. Make versions of A and B that support both formats; read format `0`, write format `1`.

Potential Solutions

1. Pause A and B; change all user settings, deploy new versions of A and B. Not great! Downtime, difficult to coordinate, prone to errors.
2. Make versions of A and B that support both formats; read format `0`, write format `1`. Better… not ideal… Say A is in Haskell, B is in Python: need four parsers for your config file…

Potential Solutions

1. Pause A and B; change all user settings, deploy new versions of A and B. Not great! Downtime, difficult to coordinate, prone to errors.
2. Make versions of A and B that support both formats; read format `0`, write format `1`. Better… not ideal… Say A is in Haskell, B is in Python: need four parsers for your config file…
3. Serve the config file *in a given format*.

Potential Solutions

1. Pause A and B; change all user settings, deploy new versions of A and B. Not great! Downtime, difficult to coordinate, prone to errors.
2. Make versions of A and B that support both formats; read format `0`, write format `1`. Better… not ideal… Say A is in Haskell, B is in Python: need four parsers for your config file…
3. Serve the config file *in a given format*. Can update A independently: use config in format `1` while B can still use format `0`.

Potential Solutions

1. Pause A and B; change all user settings, deploy new versions of A and B. Not great! Downtime, difficult to coordinate, prone to errors.
2. Make versions of A and B that support both formats; read format `0`, write format `1`. Better… not ideal… Say A is in Haskell, B is in Python: need four parsers for your config file…
3. Serve the config file *in a given format*. Can update A independently: use config in format `1` while B can still use format `0`.

Going for the last option!

Recap

– User settings are seen as a *resource*

Recap

- User settings are seen as a *resource*
- The *format* of a resource changes over time

Recap

- User settings are seen as a *resource*
- The *format* of a resource changes over time
- The *format* depends on the service using the resource

Recap

- User settings are seen as a *resource*
- The *format* of a resource changes over time
- The *format* depends on the service using the resource
- Multiple services use resources concurrently

Resources and Formats

Resource

Information seen under different views (*formats*)

Resource

Information seen under different views (*formats*)

Loosely:

$$\mathit{Resource} = \mathbb{N} \to \{0,1\}^* + \bot$$

Takes the desired format; returns bytes of respective view, if any.

Resources: In Haskell

Families of Formats

Resources: In Haskell

Families of Formats

```
class Resource r where
  type Formats r :: '[ Type ]
```


Resources: In Haskell

Families of Formats

```
class All1 ValidFormat (Formats r) => Resource r where
  type Formats r :: '[ Type ]
```


Resources: In Haskell

Families of Formats

```
class All1 ValidFormat (Formats r) => Resource r where
  type Formats r :: '[ Type ]
```

```
type family All1 c xs where
  All1 c '[] = TypeError "Need at least one element"
  All1 c '[x] = ()
  All1 c (x ': xs) = (c x, All1 c xs)
```

Formats

Formats

The view over some information:

```
class (...) => ValidFormat (fmt :: Type) where
  type FormatBody fmt :: Type
  type FormatErrors fmt :: Type
```


Formats

The view over some information:

```
class (...) => ValidFormat (fmt :: Type) where
  type FormatBody fmt :: Type
  type FormatErrors fmt :: Type

  parser :: ByteString -> Either (FormatErrors fmt) (FormatBody fmt)
```


Formats: Example

Formats: Example

```
data Settings0 = Settings0 {
    reasonable_languages :: String,
    type_system_and_eval :: TSEOpts
  } deriving (...)

data F0
class ValidFormat F0 where
  type FormatBody F0 = Settings0
  type FormatErrors F0 = ()
```

Formats: Example

```
data Settings0 = Settings0 {
    reasonable_languages :: String,
    type_system_and_eval :: TSEOpts
  } deriving (...)

data F0
class ValidFormat F0 where
  type FormatBody F0 = Settings0
  type FormatErrors F0 = ()

  parser = parseFromJSON
```

Formats: Example

```
data Settings0 = Settings0 {
    reasonable_languages :: String,
    type_system_and_eval :: TSEOpts
  } deriving (...)

data F0
class ValidFormat F0 where
  type FormatBody F0 = Settings0
  type FormatErrors F0 = ()

  parser = parseFromJSON
```

```
data Settings1 = Settings1 {
    reasonable_languages :: String,
    type_system :: TSOpts,
    eval :: EOpts
  } deriving (...)

data F1
class ValidFormat F1 where
  type FormatBody F1 = Settings1
  type FormatErrors F1 = ()
```

Formats: Example

```
data Settings0 = Settings0 {
    reasonable_languages :: String,
    type_system_and_eval :: TSEOpts
  } deriving (...)

data F0
class ValidFormat F0 where
  type FormatBody F0 = Settings0
  type FormatErrors F0 = ()

  parser = parseFromJSON


data MySettings
class Resource MySettings where
  type Formats MySettings = '[ F0, F1 ]
```

```
data Settings1 = Settings1 {
    reasonable_languages :: String,
    type_system :: TSOpts,
    eval :: EOpts
  } deriving (...)

data F1
class ValidFormat F1 where
  type FormatBody F1 = Settings1
  type FormatErrors F1 = ()
```

Formats: Adding a new one.

Formats: Adding a new one.

Someone comes along and adds **F2**:

```
data F2
```


Formats: Adding a new one.

Someone comes along and adds **F2**:

```
data F2  -- new format, added without 'ValidFormat' instance !
```


Formats: Adding a new one.

Someone comes along and adds **F2**:

```
data F2  -- new format, added without 'ValidFormat' instance !!

instance Resource MySettings where
  type Formats MyFormats = '[ F0, F1, F2 ]
```


Formats: Adding a new one.

Someone comes along and adds **F2**:

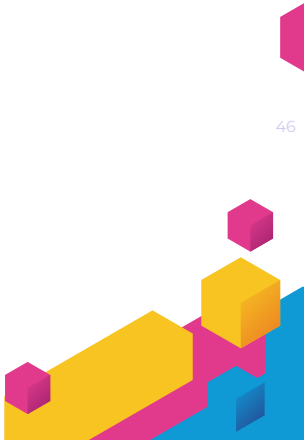
```
data F2  -- new format, added without 'ValidFormat' instance !!

instance Resource MySettings where
  type Formats MyFormats = '[ F0, F1, F2 ]
```

```
your/path/to/Module.hs:42:22: error:
    • No instance for (ValidFormat F2)
        arising from the superclasses of an instance declaration
    • In the instance declaration for Resource 'MySettings
   |
22 | instance Resource 'MySettings where
   |
```

Receiving Settings

Receiving settings from A or B: which format?



Receiving Settings

Receiving settings from A or B: which format? Whichever was sent! Receive a `(Int, ByteString)`,

Receiving Settings

Receiving settings from A or B: which format? Whichever was sent! Receive a **(Int, ByteString)**,

```
withFormat :: (Resource r)
           => Int
```


Receiving Settings

Receiving settings from A or B: which format? Whichever was sent! Receive a **(Int, ByteString)**,

```
withFormat :: (Resource r)
           => Int
           -> (forall n . (ValidFormat (Lookup n (Formats r))) => Proxy n -> a)
           -> a
withFormat fmtNum cont = ...
```

Receiving Settings

Receiving settings from A or B: which format? Whichever was sent! Receive a **`(Int, ByteString)`**,

```
withFormat :: (Resource r)
           => Int
           -> (forall n . (ValidFormat (Lookup n (Formats r))) => Proxy n -> a)
           -> a
withFormat fmtNum cont = ...
```

- Reify `fmtNum` to the type-level
- Lookup the correct **`ValidFormat`** instance

Checkpoint

Checkpoint

Cleared the problem: serve different views of the “same” information

Checkpoint

Cleared the problem: serve different views of the “same” information

To do so: Defined formats and resources

Checkpoint

Cleared the problem: serve different views of the “same” information

To do so: Defined formats and resources

Accept either format `F0` or `F1`

Checkpoint

Cleared the problem: serve different views of the “same” information

To do so: Defined formats and resources

Accept either format `F0` or `F1`

Next up: Ability to migrate between `F0` and `F1`

Migrations

Migrations

```
reasonable_programming_languages="functional"
type_system_and_eval=strong_and_lazy
```

Should become:

```
reasonable_programming_languages="functional"
type_system=strong
eval=lazy
```


Migrations

Example

```
splitSettings :: TSEOpts -> (TSOpts, EOpts)

upgrade :: FormatBody F0 -> FormatBody F1
upgrade Settings0 {..}
  = let (type_system, eval) = splitSettigs type_system_and_eval
     in Settings1 {..}
```

Migrations

Representing Migrations

```
data Migration (from :: Type) (to :: Type) where
  Migration
    :: { upstream   :: FormatBody from -> FormatBody to
       }
    -> Migration from to
```

Migrations

Representing Migrations

```
data Migration (from :: Type) (to :: Type) where
  Migration
    :: { upstream   :: FormatBody from -> Maybe (FormatBody to)
       }
    -> Migration from to
```

Migrations

Representing Migrations

```
data Migration (from :: Type) (to :: Type) where
  Migration
    :: { upstream   :: FormatBody from -> Maybe (FormatBody to)
       , downstream :: FormatBody to   -> Maybe (FormatBody from)
       }
    -> Migration from to
```

Migrations

Representing Migrations

```
data Migration (from :: Type) (to :: Type) where
  Migration
    :: { upstream   :: FormatBody from -> Maybe (FormatBody to)
       , downstream :: FormatBody to   -> Maybe (FormatBody from)
       }
    -> Migration from to
```

– Composable: `Migration b c -> Migration a b -> Migration a c`.

Migrations

Representing Migrations

```
data Migration (from :: Type) (to :: Type) where
  Migration
    :: { upstream   :: FormatBody from -> Maybe (FormatBody to)
       , downstream :: FormatBody to   -> Maybe (FormatBody from)
       }
    -> Migration from to
```

- Composable: `Migration b c -> Migration a b -> Migration a c`.
- Can represent *absence* of migrations:
  `Migration (const Nothing) (const Nothing)`.

Migrations

Representing Migrations

```
data Migration (from :: Type) (to :: Type) where
  Migration
    :: { upstream   :: FormatBody from -> Maybe (FormatBody to)
       , downstream :: FormatBody to   -> Maybe (FormatBody from)
       }
    -> Migration from to
```

- Composable: `Migration b c -> Migration a b -> Migration a c`.
- Can represent *absence* of migrations:
  `Migration (const Nothing) (const Nothing)`.
- Programmer has to be explicit.

Migrations

Keeping Track of Migrations

Migrations

Keeping Track of Migrations

```
instance Resource 'MySettings where
  type Formats 'MySettings = [F0, F1, F2, F3]
```


Migrations

Keeping Track of Migrations

```
instance Resource 'MySettings where
  type Formats 'MySettings = [F0, F1, F2, F3]
```

Should be three migrations:

```
m01 :: Migration F0 F1
m12 :: Migration F1 F2
m23 :: Migration F2 F3
```

Migrations

Keeping Track of Migrations

```
instance Resource 'MySettings where
  type Formats 'MySettings = [F0, F1, F2, F3]
```

Should be three migrations:

```
m01 :: Migration F0 F1
m12 :: Migration F1 F2
m23 :: Migration F2 F3
```

In general, for a list of formats $[f_1, \cdots, f_n]$, we need:

$$(\texttt{Migration}\ f_1\ f_2,\ \cdots,\ \texttt{Migration}\ f_{n-1}\ f_n)$$

Migrations

Keeping Track of Migrations

Migrations

Keeping Track of Migrations

```
data Pairwise (f :: k -> k -> Type) (ns :: [k]) :: Type where
  PNil  :: Pairwise f '[k]
  (:::) :: f a b -> Pairwise f (b ': ks) -> Pairwise f (a ': b ': ks)
```


Migrations

Keeping Track of Migrations

```
data Pairwise (f :: k -> k -> Type) (ns :: [k]) :: Type where
  PNil  :: Pairwise f '[k]
  (:::) :: f a b -> Pairwise f (b ': ks) -> Pairwise f (a ': b ': ks)
class Resource r where
  type Formats r :: '[ Type ]

  migrations :: Pairwise Migration (Formats r)
```

Migrations

Computing a Migration Path

```
instance Resource 'MySettings where
  type Formats 'MySettings = [F0, F1, F2, F3]

  migrations = Migration u01 d10
             ::: Migration u12 d21
             ::: Migration u23 d32
             ::: PNil
```


Migrations

Computing a Migration Path

```
instance Resource 'MySettings where
  type Formats 'MySettings = [F0, F1, F2, F3]

  migrations = Migration u01 d10
             ::: Migration u12 d21
             ::: Migration u23 d32
             ::: PNil
```

Say A wants settings in format **F0**, but storage has **F2**.

Migrations

Computing a Migration Path

```
instance Resource 'MySettings where
  type Formats 'MySettings = [F0, F1, F2, F3]

  migrations = Migration u01 d10
             ::: Migration u12 d21
             ::: Migration u23 d32
             ::: PNil
```

Say A wants settings in format **F0**, but storage has **F2**. We can try!

```
p = d10 <=< d21 :: FormatBody F2 -> Maybe (FormatBody F0)
```

Migrations: Linear Paths

Computing a Migration Path

`foldSection`

Migrations: Linear Paths

Computing a Migration Path

```
foldSection  :: (Monoid a) => Int -> Int -> [a] -> a
```


Migrations: Linear Paths

Computing a Migration Path

```
foldSection  :: (Monoid a) => Int -> Int -> [a] -> a
foldSection offset len =
  foldr mappend mempty . take len . drop offset
```


Migrations: Linear Paths

Computing a Migration Path

```
foldSection  :: (Monoid a) => Int -> Int -> [a] -> a
foldSection offset len =
  foldr mappend mempty . take len . drop offset


foldSection 2 3 [10, 11, 12, 13, 14, 15] :: Sum Int
```

Migrations: Linear Paths

Computing a Migration Path

```
foldSection  :: (Monoid a) => Int -> Int -> [a] -> a
foldSection offset len =
  foldr mappend mempty . take len . drop offset


foldSection 2 3 [10, 11, 12, 13, 14, 15] :: Sum Int
  == foldr mappend mempty (take 3 (drop 2 [10, 11, 12, 13, 14, 15]))
```

Migrations: Linear Paths

Computing a Migration Path

```
foldSection  :: (Monoid a) => Int -> Int -> [a] -> a
foldSection offset len =
  foldr mappend mempty . take len . drop offset


foldSection 2 3 [10, 11, 12, 13, 14, 15] :: Sum Int
  == foldr mappend mempty (take 3 (drop 2 [10, 11, 12, 13, 14, 15]))
  == foldr mappend mempty (take 3 [12, 13, 14, 15])
```

Migrations: Linear Paths

Computing a Migration Path

```
foldSection  :: (Monoid a) => Int -> Int -> [a] -> a
foldSection offset len =
  foldr mappend mempty . take len . drop offset


foldSection 2 3 [10, 11, 12, 13, 14, 15] :: Sum Int
  == foldr mappend mempty (take 3 (drop 2 [10, 11, 12, 13, 14, 15]))
  == foldr mappend mempty (take 3 [12, 13, 14, 15])
  == foldr mappend mempty [12, 13, 14]
```

Migrations: Linear Paths

Computing a Migration Path

```
foldSection  :: (Monoid a) => Int -> Int -> [a] -> a
foldSection offset len =
  foldr mappend mempty . take len . drop offset


foldSection 2 3 [10, 11, 12, 13, 14, 15] :: Sum Int
  == foldr mappend mempty (take 3 (drop 2 [10, 11, 12, 13, 14, 15]))
  == foldr mappend mempty (take 3 [12, 13, 14, 15])
  == foldr mappend mempty [12, 13, 14]
  == 12 + (13 + (14 + 0))
```

Migrations: Linear Paths

Computing a Migration Path

```
foldSection  :: (Monoid a) => Int -> Int -> [a] -> a
foldSection offset len =
  foldr mappend mempty . take len . drop offset


foldSection 2 3 [10, 11, 12, 13, 14, 15] :: Sum Int
  == foldr mappend mempty (take 3 (drop 2 [10, 11, 12, 13, 14, 15]))
  == foldr mappend mempty (take 3 [12, 13, 14, 15])
  == foldr mappend mempty [12, 13, 14]
  == 12 + (13 + (14 + 0))


p = let path :: Migration F0 F3
        path = pairwiseFoldrSection1 comp 0 2 (migrations @'MySettings)
    in downstream path
```

Migrations: Linear Paths

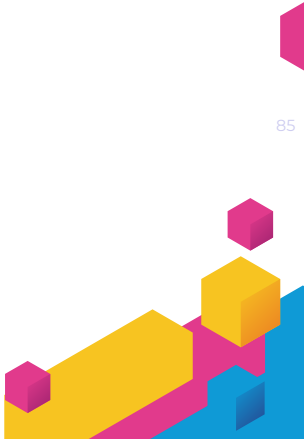
Computing a Migration Path: The Ugly Bit

```
pairwiseFoldSection
```


Migrations: Linear Paths

Computing a Migration Path: The Ugly Bit

```
pairwiseFoldSection1
```

Migrations: Linear Paths

Computing a Migration Path: The Ugly Bit

```
pairwiseFoldSection1
  :: (forall x y z. f x y -> f y z -> f x z)
```


Migrations: Linear Paths

Computing a Migration Path: The Ugly Bit

```
pairwiseFoldSection1
  :: (forall x y z. f x y -> f y z -> f x z)
  -> SNat offset
  -> SNat (S len)
```


Migrations: Linear Paths

Computing a Migration Path: The Ugly Bit

```
pairwiseFoldSection1
  :: (forall x y z. f x y -> f y z -> f x z)
  -> SNat offset
  -> SNat (S len)  -- Trick: non-empty section !
```

Migrations: Linear Paths

Computing a Migration Path: The Ugly Bit

```
pairwiseFoldSection1
  :: (forall x y z. f x y -> f y z -> f x z)
  -> SNat offset
  -> SNat (S len)  -- Trick: non-empty section !!
  -> Pairwise f ks
```

Migrations: Linear Paths

Computing a Migration Path: The Ugly Bit

```
pairwiseFoldSection1
  :: (forall x y z. f x y -> f y z -> f x z)
  -> SNat offset
  -> SNat (S len)  -- Trick: non-empty section !!
  -> Pairwise f ks
  -> f (Lookup ks offset) (Lookup ks (offset :+: S len))
```

Migrations: Linear Paths

Computing a Migration Path: The Ugly Bit

```
pairwiseFoldSection1
  :: (forall x y z. f x y -> f y z -> f x z)
  -> SNat offset
  -> SNat (S len)  -- Trick: non-empty section !!
  -> Pairwise f ks
  -> f (Lookup ks offset) (Lookup ks (offset :+: S len))
```

Not too different from:

```
foldSection :: (Monoid a) => Int -> Int -> [a] -> a
```

Migrations: Conclusions

Pros:

- Cannot forget a migraton.

Migrations: Conclusions

Pros:

- Cannot forget a migraton. Even if it's `(const Nothing)`.

Migrations: Conclusions

Pros:

- Cannot forget a migraton. Even if it's `(const Nothing)`.
- Centralized handling of formats:

Migrations: Conclusions

Pros:

- Cannot forget a migraton. Even if it's `(const Nothing)`.
- Centralized handling of formats: A and B can request different formats; helpful for no-downtime deploys.

Migrations: Conclusions

Pros:

- Cannot forget a migraton. Even if it's `(const Nothing)`.
- Centralized handling of formats: A and B can request different formats; helpful for no-downtime deploys.
- Handling of formats is independent of the `Resource` *r*

Migrations: Conclusions

Pros:

- Cannot forget a migraton. Even if it's `(const Nothing)`.
- Centralized handling of formats: A and B can request different formats; helpful for no-downtime deploys.
- Handling of formats is independent of the `Resource` *r*

Cons:

- Non-trivial amount of type-level programming.

Migrations: Conclusions

Pros:

- Cannot forget a migraton. Even if it's `(const Nothing)`.
- Centralized handling of formats: A and B can request different formats; helpful for no-downtime deploys.
- Handling of formats is independent of the `Resource` *r*

Cons:

- Non-trivial amount of type-level programming.
- Only support linear migration paths

Migrations: Conclusions

Pros:

- Cannot forget a migraton. Even if it's `(const Nothing)`.
- Centralized handling of formats: A and B can request different formats; helpful for no-downtime deploys.
- Handling of formats is independent of the `Resource` *r*

Cons:

- Non-trivial amount of type-level programming.
- Only support linear migration paths (exercise to the reader?)

channable

File-Format Migrations

on the Type-Level!

Victor Miraldo

January 5, 2024

Things I didn’t show you!

The `All1` class

Remember definition of **`Resource`**:

```
class All1 ValidFormat (Formats r) => Resource r where
  type Formats r :: '[ Type ]

type family All1 c xs where
  All1 c '[] = TypeError "Need at least one element"
  All1 c '[x] = ()
  All1 c (x ': xs) = (c x, All1 c xs)
```

LIES!

The `All1` class

For `withFormat`, need *witnesses*:

```
class All c xs => All1 c xs where
  witness :: SNat m -> (c (Lookup xs m) => r) -> r
```


The `All1` class

For `withFormat`, need *witnesses*:

```
class All c xs => All1 c xs where
  witness :: SNat m -> (c (Lookup xs m) => r) -> r
```

Then,

`withFormat`

The `All1` class

For `withFormat`, need *witnesses*:

```
class All c xs => All1 c xs where
  witness :: SNat m -> (c (Lookup xs m) => r) -> r
```

Then,

```
withFormat
  :: Resource r
  => Int
```

The `All1` class

For `withFormat`, need *witnesses*:

```
class All c xs => All1 c xs where
  witness :: SNat m -> (c (Lookup xs m) => r) -> r
```

Then,

```
withFormat
  :: Resource r
  => Int
  -> (forall n. (ValidFormat (Lookup (Formats r) n)) => SNat n -> res)
  -> res
withFormat format act
```

The `All1` class

For `withFormat`, need *witnesses*:

```
class All c xs => All1 c xs where
  witness :: SNat m -> (c (Lookup xs m) => r) -> r
```

Then,

```
withFormat
  :: Resource r
  => Int
  -> (forall n. (ValidFormat (Lookup (Formats r) n)) => SNat n -> res)
  -> res
withFormat format act  =
  withSomeSNat (fromIntegral format) (\sn ->
    witness @ValidFormat @(Formats r) sn (act sn))
```

Our SNat Datatype

Peano naturals are easier to work with.

```
data N = Z | S N

data SNat (n :: N) where
  ZZ :: SNat Z
  SS :: SNat n -> SNat (S n)
```


A Natural Challenge

We wanted `GHC.TypeNats.SNat`, but writing an eliminator is not simple:

```
natElim :: f 0 -> (forall k . SNat k -> f (k + 1)) -> SNat n -> f n
```


A Natural Challenge

We wanted `GHC.TypeNats.SNat`, but writing an eliminator is not simple:

```
natElim :: f 0 -> (forall k . SNat k -> f (k + 1)) -> SNat n -> f n
natElim base ind = aux base ind . unsafeCoerce
```


A Natural Challenge

We wanted **GHC.TypeNats.SNat**, but writing an eliminator is not simple:

```
natElim :: f 0 -> (forall k . SNat k -> f (k + 1)) -> SNat n -> f n
natElim base ind = aux base ind . unsafeCoerce  -- UnsafeSNat is not exposed :/
  where
    aux b i (UUnsafeSNat 0) = unsafeCoerce b
    aux b i (UUnsafeSNat n) = unsafeCoerce $ i $ unsafeCoerce (USNat n)

newtype USNat n = UUnsafeSNat Natural -- made to mirror SNat
```

A Natural Challenge

We wanted `GHC.TypeNats.SNat`, but writing an eliminator is not simple:

```
natElim :: f 0 -> (forall k . SNat k -> f (k + 1)) -> SNat n -> f n
natElim base ind = aux base ind . unsafeCoerce  -- UnsafeSNat is not exposed :/
  where
    aux b i (UUnsafeSNat 0) = unsafeCoerce b
    aux b i (UUnsafeSNat n) = unsafeCoerce $ i $ unsafeCoerce (USNat n)

newtype USNat n = UUnsafeSNat Natural -- made to mirror SNat
```

We couldn't escape the `unsafeCoerce` though...

A Natural Challenge

Had to use `unsafeCoerce` when reifying a run-time **`Int8`** to tye type-level.

```
withSomeMySNat :: forall rep (r :: TYPE rep). Int8 -> (forall n. SNat n -> r) -> r
withSomeMySNat n k = k @(Any @N) (unsafeMySNat n)
 where
  unsafeMySNat :: Int8 -> SNat (Any @N)
  unsafeMySNat 0 = unsafeCoerce ZZ
  unsafeMySNat sn = unsafeCoerce (SS (unsafeMySNat (sn - 1)))
```

A Natural Challenge

Had to use `unsafeCoerce` when reifying a run-time **`Int8`** to tye type-level.

```
withSomeMySNat :: forall rep (r :: TYPE rep). Int8 -> (forall n. SNat n -> r) -> r
withSomeMySNat n k = k @(Any @N) (unsafeMySNat n)
 where
  unsafeMySNat :: Int8 -> SNat (Any @N)
  unsafeMySNat 0 = unsafeCoerce ZZ
  unsafeMySNat sn = unsafeCoerce (SS (unsafeMySNat (sn - 1)))
```

Like **`GHC.TypeNats`**`.withSomeSNat`, needs `NOINLINE`; We can't use **`GHC.TypeNats`** yet, still in `base-4.17`.

A Natural Challenge

Had to use `unsafeCoerce` when reifying a run-time **`Int8`** to tye type-level.

```
withSomeMySNat :: forall rep (r :: TYPE rep). Int8 -> (forall n. SNat n -> r) -> r
withSomeMySNat n k = k @(Any @N) (unsafeMySNat n)
 where
  unsafeMySNat :: Int8 -> SNat (Any @N)
  unsafeMySNat 0 = unsafeCoerce ZZ
  unsafeMySNat sn = unsafeCoerce (SS (unsafeMySNat (sn - 1)))
```

Like **`GHC.TypeNats`**`.withSomeSNat`, needs `NOINLINE`; We can't use **`GHC.TypeNats`** yet, still in `base-4.17`.

No big deal in practice: format integers quite small (< 10).

channable

File-Format Migrations

on the Type-Level!

Victor Miraldo

January 5, 2024