Overloaded Record Fields for Haskell

Skills Matter — In The Brain

Adam Gundry

April 28, 2014 —

Once upon a time...

```
data Shape = Circle { centre :: Point, radius :: Int }
           | Rect   { centre :: Point, width :: Int, height :: Int }
```

```
centre :: Shape → Point
centre (Circle c _   ) = c
centre (Rect   c _ _) = c
```

```
radius :: Shape → Int
radius (Circle _ r   ) = r
radius (Rect   _ _ _) = error "?"
```

The way we live now

```
data Authorization
  = Authorization {
    _aut_id              :: AuthorizationID,
    _aut_organization_id :: OrganizationID,
    _aut_user_id         :: UserID,
    _aut_username        :: Username,
    _aut_password        :: Password,
    _aut_is_temporary    :: AuthIsTmp,
    _aut_key             :: AuthorizationKey,
    _aut_admin           :: Maybe AdministratorType,
    _aut_created_at      :: CreatedAt,
    _aut_updated_at      :: UpdatedAt }
```

```
data Avatar
  = Avatar {
    _avr_type            :: AvatarType,
    _avr_media_key       :: Maybe AvatarMasterKey,
    _avr_master_key      :: Maybe AvatarMasterKey,
    _avr_current_version :: AvatarVersion,
    _avr_history         :: AvatarMasterKey,
    _avr_master_s3uri    :: Maybe S3URI,
    _avr_viewable_s3uri  :: Maybe S3URI,
    _avr_filename        :: Maybe UploadFilename,
    _avr_magic_string    :: AvatarMagicString }
```

```
data Booking
  = Booking {
    _bkg_id                  :: BookingID,
    _bkg_user_id             :: UserID,
    _bkg_organization_id     :: OrganizationID,
    _bkg_policy_id           :: Maybe PolicyID,
    _bkg_tag                 :: Maybe Tag,
    _bkg_suspension_id       :: Maybe EventID,
    _bkg_trashing_id         :: Maybe EventID,
    _bkg_discarding_id       :: Maybe EventID,
    _bkg_device_id           :: DeviceID,
    _bkg_start_grace         :: Minutes,
    _bkg_starts_at           :: UTCTime,
    _bkg_duration            :: Minutes,
    _bkg_end_grace           :: Minutes,
    _bkg_reflection_user_id  :: Maybe UserID,
    _bkg_observer_id         :: Maybe UserID,
    _bkg_reflection_name     :: ReflectionName,
    _bkg_reflection_room     :: Room,
    _bkg_reflection_description :: ReflectionDescription,
    _bkg_reflection_id       :: Maybe ReflectionID,
    _bkg_booking_invitations :: BookingInvitation,
    _bkg_active              :: Acceptance,
    _bkg_created_at          :: CreatedAt,
    _bkg_updated_at          :: UpdatedAt }
```

```
data Channel
  = Channel {
    _chn_type            :: ChannelType,
    _chn_media_key       :: Maybe VideoMasterKey,
    _chn_filename        :: Maybe UploadFilename,
    _chn_streamable_videos :: StreamableVideo,
    _chn_prep_state      :: VideoPrepState,
    _chn_progress        :: Maybe Percent,
    _chn_eta             :: Maybe Miliseconds,
    _chn_diagnostic      :: Maybe Data.Text.Internal.Text }
```

```
data Comment
  = Comment {
    _cmt_id              :: CommentID,
    _cmt_parent_id       :: Maybe CommentID,
    _cmt_user_id         :: UserID,
    _cmt_organization_id :: OrganizationID,
    _cmt_policy_id       :: Maybe PolicyID,
    _cmt_tag             :: Maybe Tag,
    _cmt_suspension_id   :: Maybe EventID,
    _cmt_trashing_id     :: Maybe EventID,
    _cmt_discarding_id   :: Maybe EventID,
    _cmt_mark_inappropriate_id :: Maybe EventID,
    _cmt_body            :: CommentBody,
    _cmt_channel         :: Maybe Channel,
    _cmt_start           :: Maybe Miliseconds,
    _cmt_duration        :: Maybe Miliseconds,
    _cmt_created_at      :: CreatedAt,
    _cmt_updated_at      :: UpdatedAt }
```

```
data DatabaseSnapshot
  = DatabaseSnapshot {
    _dbs_started_at      :: Maybe StartedAt,
    _dbs_authorizations  :: Maybe Authorization,
    _dbs_bookings        :: Maybe Booking,
    _dbs_comments        :: Maybe Comment,
    _dbs_configurations  :: Maybe Configuration,
    _dbs_devices         :: Maybe Device,
    _dbs_device_reports  :: Maybe DeviceReport,
    _dbs_events          :: Maybe Event,
    _dbs_groups          :: Maybe Group,
    _dbs_invitations     :: Maybe Invitation,
    _dbs_reflections     :: Maybe Reflection,
    _dbs_settings        :: Maybe Setting,
    _dbs_shares          :: Maybe Share,
    _dbs_users           :: Maybe User,
    _dbs_videos          :: Maybe Video,
    _dbs_started_at      :: Maybe StartedAt,
    _dbs_authorizations  :: Maybe Authorization,
    _dbs_bookings        :: Maybe Booking,
    _dbs_comments        :: Maybe Comment,
    _dbs_configurations  :: Maybe Configuration,
    _dbs_devices         :: Maybe Device,
    _dbs_device_reports  :: Maybe DeviceReport,
    _dbs_events          :: Maybe Event,
    _dbs_groups          :: Maybe Group,
    _dbs_invitations     :: Maybe Invitation,
    _dbs_reflections     :: Maybe Reflection,
    _dbs_settings        :: Maybe Setting,
    _dbs_shares          :: Maybe Share,
    _dbs_users           :: Maybe User,
    _dbs_videos          :: Maybe Video
    _dbs_device_reports  :: Maybe DeviceReport,
    _dbs_events          :: Maybe Event,
    _dbs_groups          :: Maybe Group }
```


Well-Typed

The problem

$$\textbf{data}\ \text{Person} = \text{Person}\ \{\mathit{name} :: \text{String}, \mathit{age} :: \text{Int}\}$$
$$\textbf{data}\ \text{Cat} \quad = \text{Cat} \quad\ \{\mathit{name} :: \text{String}\}$$

GHC says:

```
test.lhs:2:28:
    Multiple declarations of ‘name’
    Declared at: test.lhs:1:28
                 test.lhs:2:28
```

Disambiguation by prefix/suffix

```
data Person = Person { personName :: String, personAge :: Int }
data Cat    = Cat    { catName    :: String }
```

This works, but:

- it's verbose
- how do you keep track of the affixes?
- why must we tell the typechecker things it already knows?

Where we're going

Design goals for `OverloadedRecordFields`

Demo

How it works, more or less

Record update and lenses

Looking forward

Design goals for `OverloadedRecordFields`

- Use the same field in multiple records
- As simple as possible
  - No new syntax!
  - No anonymous/extensible records
- Interoperate with existing code
  - Data types declared in modules without the extension
  - Libraries need not force the extension on their users

Implementation

- Took 3 months over Summer 2013
- GHC's codebase is scary
- I couldn't have done it without help
- There will be bugs

Demo

Demo

```
GHCi, version 7.9.20140418: http://www.haskell.org/ghc/
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :set -XOverloadedRecordFields
Prelude> data Person = Person { name :: String }
Prelude> data Cat = Cat { name :: String }
Prelude> name (Person "Adam")
"Adam"
Prelude> name (Cat "Jeoffry")
"Jeoffry"
Prelude> :t name
name :: GHC.Records.Accessor t t1 "name" t2 => t t1 t2
```

How it works, more or less

data Person = Person $\{ \mathit{name} :: \mathsf{String}, \mathit{age} :: \mathsf{Int} \}$

$\mathit{name} :: r \, \{ \mathit{name} :: t \} \Rightarrow r \to t$
$\mathit{name} \approx \mathit{getField} \; (\mathit{proxy\#} :: \mathsf{Proxy\#} \; \texttt{"name"})$

- $r \, \{ \mathit{name} :: t \}$ is like a typeclass constraint
- "the type r has a field *name* of type t"
- Solved automatically when a suitable record field is in scope
- Actually uses a built-in magic typeclass Has r `"name"` t

The record update problem

Haskell's traditional record update syntax is clumsy but powerful

- Type-changing update
- Update multiple fields at once:

$$\textbf{data}\ \text{Pair}\ a = \text{Pair}\ \{x :: a, y :: a\}$$

$$\begin{aligned} &\mathit{foo} :: \text{Pair Char} \to \text{Pair Bool} \\ &\mathit{foo}\ r = r\ \{x = \text{True}, y = \mathit{isDigit}\ (y\ r)\} \end{aligned}$$

$$\mathit{bar}\ r = r\ \{x = \text{True}\}\ \{y = \mathit{isDigit}\ (y\ r)\}$$

Record update for overloaded fields

- Don't try to be clever!
- Require a type signature to resolve ambiguity
- At least this is simple and backwards-compatible

$$foo :: \text{Pair Char} \to \text{Pair Bool}$$
$$foo\ r = r\ \{x = \text{True}, y = isDigit\ (y\ r)\}$$

$$foo\ r = (r :: \text{Pair Char})\ \{x = \text{True}, y = isDigit\ (y\ r)\}$$

$$foo\ r = r\ \{x = \text{True}, y = isDigit\ (y\ r)\} :: \text{Pair Bool}$$

Record update the ugly way

```
type family UpdTy r (n :: Symbol) t :: *
setField :: Proxy# n → r → t → UpdTy r n t
```

Instead of

```
baz r = r {age = 30}
```

we can write

```
baz r = setField (proxy# :: Proxy# "age") r (30 :: Int)
```

Yuk!

Lenses to the rescue

A lens combines a getter and setter for a field:

```
data Lens r a
get :: Lens r a →        r → a
set :: Lens r a → a → r → r
```

- A record field corresponds not just to a getter, but to a lens!
- Lens libraries provide combinators for working with lenses
- But which lens library should we pick?

Abstraction over lens libraries

```
name :: Accessor p r "name" t ⇒ p r t
name = field (proxy# :: Proxy# "name")
```

- Pick $p = (\rightarrow)$ to get back selector functions

```
name (Person "Adam" 26) :: String
```

- Or $p =$ Lens

```
set age 27 (Person "Adam" 26)
```

- Lens libraries can give their own instances of Accessor

Warts and all

- Record projections must be brought into scope somehow
- Type inference error messages
- Cannot overload higher-rank fields
- Multiple field update
- van Laarhoven lenses require a wrapper type

The future?

- `OverloadedRecordFields` in HEAD Real Soon Now™
- Gather feedback from users, tweak design, fix some bugs
- Projected to be released in GHC 7.10
- Syntax for projections: perhaps *rec*#*x* instead of *x* *rec*?
- `OverloadedDataConstructors`?
- More coherent story about special-purpose constraint solving

Thanks and acknowledgments

- Google Summer of Code
- Simon Peyton Jones
- Edward Kmett
- Many more...

Here be dragons

The Has typeclass

```
type family FldTy (r :: *) (n :: Symbol) :: *
class t ~ FldTy r n => Has r (n :: Symbol) t where
  getField :: Proxy# n -> r -> t
```

```
type instance FldTy Person "name" = String
instance t ~ String => Has Person "name" t where
  getField _ = name
```

The Upd typeclass

```
type family UpdTy (r :: *) (n :: Symbol) (t :: *) :: *
class (Has r n (FldTy r n), t ~ UpdTy r n (FldTy r n))
    => Upd r (n :: Symbol) t where
  setField :: Proxy# n -> r -> t -> UpdTy r n t
```

```
type instance UpdTy Person "name" t = Person
instance t ~ String => UpdTy Person "name" t where
  setField _ (Person _ a) n = Person n a
```

Accessor

```
class Accessor (p :: * → * → *) r (n :: Symbol) t where
  accessField :: Proxy# n
              → (Has r n t ⇒ r → t)
              → (forall t'.Upd r n t' ⇒ r → t' → UpdTy r n t')
              → p r t
instance Has r n t ⇒ Accessor (→) r n t where
  accessField _ getter _ = getter
```

```
field :: Accessor p r n t ⇒ Proxy# n → p r t
field z = accessField z (getField z) (setField z)
```