Fixing Records in Haskell

Neil Mitchell et al, ndmitchell.com
an in-your-face, glaring weakness telling you there is something wrong with Haskell
- Greg Weber

What is your least favorite thing about Haskell? Records are still tedious
- 2018 State of Haskell Survey

The record system is a continual source of pain
- Stephen Diehl

Haskell’s record system is a cruel joke
- Scrive

Records’ syntax sucks
- Bitcheese
myPerson.name

Which language is this?
It can be Haskell!

• Using record-dot-preprocessor
  – [github.com/ndmitchell/record-dot-preprocessor](https://github.com/ndmitchell/record-dot-preprocessor)
  – Available as a textual preprocessor and plugin

• Using DAML – a Haskell derivative
  – [daml.com](http://daml.com)

• If the latest GHC proposal gets accepted and implemented
  – [tinyurl.com/ghc-records](http://tinyurl.com/ghc-records)
Forbidden Questions (until later)
What I want to do

data Company = Company {
    name :: String,
    owner :: Person
}
data Person = Person {
    name :: String,
    age :: Int

ERROR: Multiple declarations of ‘name’
Automatic selectors

- Haskell helpfully generates

```haskell
name :: Company -> String
owner :: Company -> Person
name :: Person -> String
age :: Person -> Int
```

ERROR: Multiple declarations of ‘name’
What I actually do #1

data Company = Company {
    companyName :: String,
    companyOwner :: Person}
data Person = Person {
    personName :: String,
    personAge :: Int}

personName (companyOwner x)
What I actually do #2

import qualified Company(Company(..)) as C
import qualified Person(Person(..)) as P

P.name (C.owner x)
What I actually do #3

Especially when explaining this to Haskell beginners...
Especially experienced programmers...
With RecordDotSyntax

data Company = Company {
    name :: String,
    owner :: Person
}
data Person = Person {
    name :: String,
    age :: Int
}

x.owner.name
This change is a BIG deal

- DAML is a Haskell inspired DSL for smart contracts on a Distributed Ledger
  - Written by Digital Asset, a company that is hiring, that I used to work for: digitalasset.com
- Wanted to move from Haskell inspired to GHC based implementation
- Records stopped us, until we implemented this extension (in use ~18 months)
How does it work?

• Step 1: Don’t generate the selectors
  – Already part of the NoFieldSelectors proposal
  – But now how do I get at the fields?
  – Record puns to the rescue

``` scala
  case x of
    Company{owner} -> case owner of
      Person{name} -> name
```
Sugar that up #1

a.B.c  =>  case a of B{c} -> c

x.Company.owner.Person.name

- Ugly! Company should be inferred from the type of ‘a’.
Sugar that up #2

```
x.owner.name
a.b => getField a b

getField :: r -> String -> F r String

"b" :: String -- a value of type String
@"b" :: Label -- a type of kind Label
```
Implement that sugar

class HasField x r a | x r -> a where
  getField :: r -> a

instance HasField "name" Person String where
  getField Person{name} = name

x.owner.name

getField @"name" (getField @"owner" x)
Appreciate the Magic

- NoFieldSelectors
- HasField type class
- Automatic instances
- Minor syntax sugar

= records solved
Pairs of labels

instance (HasField l1 a b, HasField l2 b c) =>
  HasField (l1, l2) a c where
  getField = getField @l2 . getField @l1

• Since type is either a Label (lifted String) or
  pair (lifted pair)

getField @("owner", "name") x
Standalone selectors

• Old world
map name people

• New world
map (getField @"name") people
map (.name) people
Record Updates
Step 1: Make them work

\( a\{b=c\} \Rightarrow \text{setField } @"b" \ a \ c \)

class HasField x r a | x r -> a where
  setField :: r -> a -> r
Step 2: Multiple field updates

• a\{b=c, d=e\}

`setField @"d" (setField @"b" a c) e`

Real updates are more powerful.
Where did I cheat?
Type changing updates!

data Foo a = Foo {foo :: [a], bar :: Int}

(x :: Foo Int){foo = [True]} :: Foo Bool

setField :: Label -> r -> v -> F Label r v
Type inference issues

```plaintext
x{foo = [], bar = 2}
setField @"bar" (setField @"foo" x []) 2
:: Foo ???
```

There are complex solutions, but...
Powerful idea

Complex and rarely used feature
Easily emulated

let Foo{..} = x in Foo{foo=[], bar=2, ...}
Deep updates still suck

• Set the age of the owner to 42

```plaintext
x{owner = x.owner{age=42}}
```

Repeated owner twice. Gets much worse as we nest further.
Deep updates fixed

- Set the age of the owner to 42

$$x\{\text{owner.age} = 42\}$$

```regex
setField @("owner","age") x 42
```
Field modification still sucks

- Increment the age of the owner

```
x{owner.age = x.owner.age + 1}
```

Not terrible, but not beautiful.
Field modification fixed

- Increment the age of the owner

\[
x\{\text{owner.age + 1}\}
\]

\[
\text{modifyField @("owner","age") x (+ (1))}
\]
Field modification with lambda

- Do something weird

\[
x\{\text{owner}.\text{age} & \ \land \ i \rightarrow \ \text{floor} \ (i \ * \ 57) + 21\}
\]

modifyField @("owner","age") x (\(\land \ (i \rightarrow \ ...)\))

Data.Function.(\&) = flip (\$)
Is modifyField expensive?

-- Traversing the structure twice is bad (maybe?)
modifyField @l x f =
    setField @l x $ f $ getField @l x

instance HasField x r a | x r -> a where
    hasField :: r -> (a, a -> r)

modifyField @l x f = u $ f v
    where (v, u) = hasField @l x
HasField FAQ

• Can I define my own HasField instance, e.g. to pretend my structure has a virtual field
  – Yes, you can. Let’s not do one for Map though, please...

• Can I access non-exported fields now?
  – No. HasField is magic. GHC manufactures it locally only if the field/constructor are in scope.
Hmm, DuplicateRecordFields?

• An extension in GHC that let’s you write:
  name (owner c :: Person)

• name’s arg must be a locally known type:
  – f c = name (owner (c :: Company)) -- bad
  – f c = name (owner c :: Person) -- good
  – f (p :: Person) = name p -- bad

• We use real constraints for better power
Did you just reinvent lenses?

- There’s definitely overlap!
- Lenses are record fields as first-class values, which is awesome. Powerful. Scary. These records are concrete.

- It does conflict with the lens `c^\cdot\text{companyOwner.personName}` style.
Remember the original motivation

For the domain of DAML, lens is not a feasible solution.
# Syntactic extensions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.lbl</td>
<td>getField &quot;lbl&quot; e</td>
</tr>
<tr>
<td>e{lbl = val}</td>
<td>setField &quot;lbl&quot; e val</td>
</tr>
<tr>
<td>(.lbl)</td>
<td>(\lambda x \rightarrow x.lbl)</td>
</tr>
<tr>
<td>e{lbl1.lbl2 = val}</td>
<td>e{lbl1 = (e.lbl1){lbl2 = val}}</td>
</tr>
<tr>
<td>e{lbl * val}</td>
<td>e{lbl = e.lbl * val}</td>
</tr>
<tr>
<td>e{lbl1.lbl2}</td>
<td>e{lbl1.lbl2 = lbl2}</td>
</tr>
</tbody>
</table>
## Combinations

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.lbl1.lbl2</td>
<td>(e.lbl1).lbl2</td>
</tr>
<tr>
<td>(.lbl1.lbl2)</td>
<td>(\x -&gt; x.lbl1.lbl2)</td>
</tr>
<tr>
<td>e.lbl1{lbl2 = val}</td>
<td>(e.lbl1){lbl2 = val}</td>
</tr>
<tr>
<td>e{lbl1 = val}.lbl2</td>
<td>(e{lbl1 = val}).lbl2</td>
</tr>
<tr>
<td>e{lbl1.lbl2 * val}</td>
<td>e{lbl1.lbl2 = e.lbl1.lbl2 * val}</td>
</tr>
<tr>
<td>e{lbl1 = val1, lbl2 = val2}</td>
<td>(e{lbl1 = val1}){lbl2 = val2}</td>
</tr>
<tr>
<td>e{lbl1.lbl2, ..}</td>
<td>e{lbl2=lbl1.lbl2, ..}</td>
</tr>
</tbody>
</table>

myPerson.name

Coming to a GHC near you! (Maybe)