

Fields are Functions

Robert Ennals
Computer Laboratory, University of Cambridge
Robert.Ennals@cl.cam.ac.uk

ABSTRACT

The Object Oriented language community has long endorsed the concept of *properties* — object fields that are read and set through accessor functions.

In this paper, we show that the essence of properties can be captured in a functional setting by a decomposition of record syntax into functions. Moreover, we show that the existing concept of type classes can be used to provide extensible records while requiring minimal changes to a functional language.

While this paper is presented as an extension to the Haskell language, the ideas apply to many other languages.

1. INTRODUCTION

2. DESUGARING RECORDS

In Haskell, a programmer can define a record as follows:

```
data Car = Car {  
    passengers :: Int,  
    color :: Color,  
    owner :: Person}
```

Following the Haskell report, a Haskell compiler will reduce this to a tuple definition, and a set of selector functions:

```
data Car = Car Int Color Person  
  
passengers (Car p _ _) = p  
color (Car _ c _) = c  
owner (Car _ _ o) = o
```

The great thing about this is that record selection is just a function application. If I want to select the *owner* field for a *Car* I simply apply the *owner* function to it:

```
owner x
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2002 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

This is a bog-standard, normal function rather than a new construct. If I later redefine my *Car* type to not have an *owner* field, I can provide an *owner* function myself that computes the owner another way.

Unfortunately, this beauty does not extend to record update. A Haskell programmer would write the following code in order to update the color and owner fields of a *Car*:

```
x {color = Blue, owner = Rob}
```

This does **not** reduce to a function call. The Haskell report states that it should desugar to the following:

```
x (\(Car p _ _) -> (Car p Blue Rob))
```

This loses the lovely genericity that we had before. Unlike record selection syntax, record update syntax only works if *x* really is a record and really does have fields called *color* and *owner*. I can't remove these fields or replace them with getter and setter functions.

3. UPDATER FUNCTIONS

The obvious solution is to change our desugarer so that it generates field updater functions in addition to field selector functions. We will now generate the following code for our *Car* record:

```
data Car = Car Int Color Person  
  
passengers (Car p _ _) = p  
color (Car _ c _) = c  
owner (Car _ _ o) = o  
  
set_passengers p (Car _ c o) = Car p c o  
set_color c (Car p _ o) = Car p c o  
set_owner o (Car p c _) = Car p c o
```

Given these functions, we can now redefine our record update syntax so that it makes calls to these functions. The example from Section 2 now desugars to:

```
set_color Blue $ set_owner Rob $ x
```

I now get my abstraction property back. I can replace my *owner* or *color* field with a pair of functions that gets or sets instances of this property.

For example, I might wish to maintain an *isblack* field that is *True* if the car is black and *false* otherwise. I could do this as follows:

```
data Car = Car {passengers :: Int,
               color' :: Color,
               isblack :: Bool,
               owner :: Person}

color r = color' r

set_color Black r = r {color' = Black, isblack = True}
set_color c r = r {color' = c, isblack = False}
```

4. USING TYPE CLASSES

If record selectors and updaters are normal functions, then it makes sense that we should be able to put them in type classes, just like any other Haskell function.

4.1 Record Polymorphism

Type classes can be used to express record polymorphism—allowing us to write functions that can work with any type, provided that it has certain record fields.

For example, I can define a type class for all types that have an owner as follows:

```
class HasOwner a where
  owner :: a -> Person
  set_owner :: Person -> a -> a
```

When defining a new record, I can declare that the record type is an instance of the type class:

```
data CoffeeMug = CoffeeMug {owner :: Person,
                           size :: Int}
data Pen = Pen {owner :: Person, color :: Color}

instance HasOwner CoffeeMug
instance HasOwner Pen
```

4.2 Desugaring

When a record type is declared to be an instance of a type class, the names of the functions in the type class are checked against the names of the fields in the record. For every field name *f*, if the type class has a function called *f* then it will be implemented with a record selector function, as described in Section 2. Similarly, if the type class has a function called *set_f* then it will be implemented with a record updater function, as described in Section 3.

The code given above will thus desugar to the following:

```
data CoffeeMug = CoffeeMug Person Int
size (CoffeeMug _ s) = s
set_size s (CoffeeMug p _) = CoffeeMug p s

data Pen = Pen Person Color
color (Pen _ c) = c
set_color c (Pen p _) = Pen p c

instance HasOwner CoffeeMug where
  owner (CoffeeMug p _) = p
  set_owner p (CoffeeMug _ s) = CoffeeMug p s

instance HasOwner Pen where
  owner (Pen p _) = p
  set_owner p (Pen _ c) = Pen p c
```

Note that we do not generate top level owner and set_owner functions at the point at which the CoffeeMug and Pen record types are defined. If these declarations were generated, there would be a name clash with the names defined within the type class. The desugarer follows a simple strategy, generating accessor functions only if the function generated does not have a name that would clash with a function defined in a type class in scope.

5. MORE COMPLEX USES OF TYPE CLASSES

5.1 Other Type Class Functions

A type class may contain functions other than those that can be generated from record fields. In this case, the programmer will have to supply their own implementations for any additional functions. For example, the programmer might write the following:

```
class Vehicle a where
  passengers :: a -> Int
  set_passengers :: Int -> a -> a
  journeyTime :: a -> Location -> Location -> Time

data Car = Car {passengers :: Int, ... }

instance Vehicle Car where
  journeyTime = ...
```

The passengers functions are implemented automatically, but the journeyTime function must be implemented manually.

It should be stressed that the Vehicle class has been defined by the programmer. The desugarer never creates new type classes.

5.2 Read-Only Fields

It is not necessary to include both selector and updater functions in the same type class. A type class can include just a selector function, just an updater function, or both. This can be useful if writing a field does not make sense.

For example a Pogo Stick can only have one passenger - it thus does not make sense for a program to set its passengers field. We might thus wish to redefine our Vehicle class without the updater method:

```
class Vehicle a where
  passengers :: a -> Int
  journeyTime :: a -> Location -> Location -> Time

instance Vehicle PogoStick where
  passengers _ = 1
  journeyTime = ...

instance Vehicle Car where
  journeyTime = ...
```

5.3 Paramaterised Type Classes

A type class can include the type of a field as a parameter. For example, the programmer might use multiparameter type classes and functional dependencies to define the following class:

```
class LikeList l a | l -> a
```

```

where
  head :: 1 -> a
  tail :: 1 -> 1
  null :: 1 -> Bool

```

This could then be implemented by a simple list datatype:

```

data List a = Cons {head :: a, tail :: List a}
             | Null

```

```

instance LikeList (List a) a where
  null Null = True
  null _ = False

```

In this case, the types of the `head` and `tail` fields depend on the type of the list itself.

The desugarer is not be aware of any of the type issues involved here. It generates the same accessor function that it always dose. The only difference is that the type checker infers a more general type for them.

6. IMPERATIVE FIELDS

The records we have been decribing so far are immutable. We can read a field, and we can create a new record with a new value set for one of the fields. However we cannot update a field in place. For purely functional programming, that is exactly what we want, however some programming tasks require the use of imperative records that can me re-written with a side-effecting operation.

Fortunately, the framework described can also be used to implement imperative record fields:

The programmer can declare an imperative record as follows:

```

data ImpCar = ImpCar { gear :: RefProp Int,
                      speed :: RefProp Int}

```

where `RefProp` is defined as follows:

```

class ReadProperty p a
  readProp :: p -> IO a

```

```

class WriteProperty
  writeProp :: p -> a -> IO ()

```

```

class (ReadProperty p a, WriteProperty p a)
  => Property p a

```

```

data RefProf a = RefProp (IORef a)

```

```

instance Property (RefProp a) a where
  readProf (RefProp ref) = readIORef ref
  writeProp (RefProf ref) v = writeIORef ref v

```

At a later point, some imperative fields can be replaced with other implementations of the `Property` class. These other implementations may perform computations in order to work out the value of the property.

This solution is inspired by a proposal by Koen Claessen [?].

7. RELATED WORK

8. CONCLUSIONS

Acknowledgements