Record Systems

Dylan Just

YOW LambdaJam 2015

Haskell

Elm

PureScript

OCaml

Idris

Pascal

Haskell

Elm

PureScript

OCaml

Idris

Pascal

Java classes

Scala classes

C structs

Database tables

Haskell

Elm

PureScript

Haskell

Elm

PureScript

Vinyl

Haskell

Elm

PureScript

Vinyl

What are records?

What can they do?

Structural typing

Row polymorphism

What is a record?

Record = Set of fields with

- name
- type
- value

Record = Set of fields with

- name
- type

type-level information

- value

value-level information

Record = Set of fields with

- name
- type
- value

Basic Operations

Declare Type

```
data Person = Person {name::String, age::Int}
```

haskell

Instantiate

```
data Person = Person {name::String, age::Int}

bob = Person {name="Bob", age=30}
```

haskell

Get

```
data Person = Person {name::String, age::Int}

bob = Person {name="Bob", age=30}

bobName = name bob
```

haskell

Set

```
data Person = Person {name::String, age::Int}

bob = Person {name="Bob", age=30}

bobName = name bob

olderBob = bob {age=31}
```

haskell

Haskell Problems

Namespacing

```
data Pos2D =
  Pos2D {x::Int, y::Int}
```

```
data Pos2D =
  Pos2D {x::Int, y::Int}

data Pos3D =
  Pos3D {x::Int, y::Int, z::Int}
```

```
data Pos2D =
  Pos2D {x::Int, y::Int}

data Pos3D =
  Pos3D {x::Int, y::Int, z::Int}
```

Compile error

```
Namespacing.hs:2:22:
    Multiple declarations of ‘x’
    Declared at: Namespacing.hs:1:22
                 Namespacing.hs:2:22

Namespacing.hs:2:32:
    Multiple declarations of ‘y’
    Declared at: Namespacing.hs:1:32
                 Namespacing.hs:2:32
```

Overlapping Record Names

```
data Key =
    StringKey  { key :: String }
  | NumericKey { key :: Int }
```

haskell

```haskell
data Key =
    StringKey  { key :: String }
  | NumericKey { key :: Int }
```

Compile error

haskell

Constructors StringKey and NumericKey give different types for field ‘key’

In the data declaration for ‘Key’

Partial Accessors

```
data Shape =
    Circle    {radius::Int}
  | Rectangle {width::Int, height::Int}
```

haskell

```haskell
data Shape =
    Circle    {radius::Int}
  | Rectangle {width::Int, height::Int}

r = Rectangle {width=3, height=4}
```

haskell

```
data Shape =
    Circle    {radius::Int}
  | Rectangle {width::Int, height::Int}

r = Rectangle {width=3, height=4}

r_radius = case r of (Circle {radius=r}) -> r
                     (Rectangle {}) -> -1

> r_radius
-1
```

haskell

```
data Shape =
    Circle    {radius::Int}
  | Rectangle {width::Int, height::Int}

r = Rectangle {width=3, height=4}

r_radius = radius r

> r_radius'
*** Exception: No match in record selector radius
```

Run-time error


wat

Haskell

vs

PureScript

Namespacing

```
data Pos2D =
  Pos2D {x::Int, y::Int}

data Pos3D =
  Pos3D {x::Int, y::Int, z::Int}
```

Compile error

```
data Pos2D =
  Pos2D {x::Number,y::Number}

data Pos3D =
  Pos3D {x::Number,y::Number,z::Number}
```

Overlapping Record Names

```
data Key =
    StringKey  { key :: String }
  | NumericKey { key :: Int }
```

Compile error

haskell

```
data Key =
    StringKey  { key :: String }
  | NumericKey { key :: Number }
```

Partial Accessors

```
data Shape =
    Circle    {radius::Int}
  | Rectangle {width::Int, height::Int}

r = Rectangle {width=3, height=4}

r_radius = radius r

> r_radius'
*** Exception: No match in record selector radius
```

Run-time error

```
data Shape =
    Circle    {radius::Int}
  | Rectangle {width::Int, height::Int}

r = Rectangle {width=3, height=4}

r_radius = case r of (Circle {radius=rad}) -> rad
                     (Rectangle {}) -> -1

> r_radius
-1
```

```
data Shape =
    Circle    {radius::Number}
  | Rectangle {width::Number, height::Number}

r = Rectangle {width:3, height:4}

r_radius = case r of (Circle {radius=rad}) -> rad
                     (Rectangle {}) -> -1

> r_radius
-1
```

```
data Shape =
    Circle    {radius::Int}
  | Rectangle {width::Int, height::Int}

r = Rectangle {width=3, height=4}

r_radius = case r of (Circle r') -> r'
                     (Rectangle w' h') -> -1

> r_radius
-1
```

haskell

```
data Shape =
    Circle    {radius::Number}
  | Rectangle {width::Number, height::Number}

r = Rectangle {width:3, height:4}

r_radius = case r of (Circle c') -> c'.radius
                     (Rectangle r') -> -1

> r_radius
-1
```

```
r_radius = case r of (Circle r') -> r'
                     (Rectangle w' h') -> -1
```

haskell

---

purescript

```
r_radius = case r of (Circle c') -> c'.radius
                     (Rectangle r') -> -1
```

```haskell
data Shape =
    Circle    {radius::Int}
  | Rectangle {width::Int, height::Int}

c = Circle {radius=3}
c' = Circle 3
```

haskell

Fields are constructor parameters

```
data Shape =
    Circle    { radius::Int }
  | Rectangle { width::Int, height::Int }
```

haskell

Records are constructor parameters

```
data Shape =
    Circle    { radius::Number }
  | Rectangle { width::Number, height::Number }
```

```
data X = X { a :: String }
```

```
data X = X { a :: String }

data Y = Y String {a :: String}
```

```
data X = X { a :: String }

data Y = Y String {a :: String}

data Z = Z {a::String,b::Number} Number {c::String}
```

```
data X = X { a :: String }

data Y = Y String {a :: String}

data Z = Z {a::String,b::Number} Number {c::String}

type Q = {a::String, b::Int}
```

Tagged

```
data X = X { a :: String }

data Y = Y String {a :: String}

data Z = Z {a::String,b::Number} Number {c::String}
```

```
type Q = {a::String, b::Int}
```

Untagged

Tagged vs Untagged

Tagged

```
data Pos2D = Pos2D {x::Number,y::Number}
p = Pos2D {x:3,y:4}
```

Untagged

```
type Pos2D = {x::Number,y::Number}
p = {x:3,y:4}
```

Nominal Type

```
data Pos2D = Pos2D {x::Number,y::Number}
p = Pos2D {x:3,y:4}
```

Structural Type

```
type Pos2D = {x::Number,y::Number}
p = {x:3,y:4}
```

Nominal Typing

Two values are type-compatible iff
their declarations name the same type.

Structural Typing

Two values are type-compatible iff they exhibit the same structure.

Structural Typing

Extensible Records

Row Polymorphism

Record Subtyping

Extensible Records

Row Polymorphism

~~Record Subtyping~~

Extensible Records

- add fields
- remove fields

~~PureScript~~

Elm

Create

```
> bob  = {name="bob"}
{name="bob"} : {name:String}
```

Add Field

```
> bob  = {name="bob"}
{name="bob"} : {name:String}

> bob' = {bob | age=33}
{name="bob",age=33} : {name:String,age:number}
```

Remove Field

```
> bob  = {name="bob"}
{name="bob"} : {name:String}

> bob' = {bob | age=33}
{name="bob",age=33} : {name:String,age:number}

> bob'' = {bob' - age}
{name="bob"} : {name:String}
```

Row Polymorphism

Back to PureScript...

```
type House = {color::String,rooms::Number}
type Car = {color::String,model::String}
```

```
type House = {color::String,rooms::Number}
type Car = {color::String,model::String}

myhouse :: House
myhouse = {color:"pink",rooms:3}

mycar :: Car
mycar = {color:"red",model:"Corolla"}
```

```
type House = {color::String,rooms::Number}
type Car = {color::String,model::String}

myhouse = {color:"pink",rooms:3}
mycar = {color:"red",model:"Corolla"}
```

```
type House = {color::String,rooms::Number}
type Car = {color::String,model::String}

myhouse = {color:"pink",rooms:3}
mycar = {color:"red",model:"Corolla"}

getColor :: House -> String
getColor x = x.color

getColor' :: Car -> String
getColor' x = x.color
```

```
type House = {color::String,rooms::Number}
type Car = {color::String,model::String}

myhouse = {color:"pink",rooms:3}
mycar = {color:"red",model:"Corolla"}

getColor :: forall r. {color::String | r} -> String
getColor x = x.color
```

```
type House = {color::String,rooms::Number}
type Car = {color::String,model::String}

myhouse = {color:"pink",rooms:3}
mycar = {color:"red",model:"Corolla"}

getColor :: forall r. {color::String | r} -> String
getColor x = x.color

> getColor myhouse
"pink"
```

```
type House = {color::String,rooms::Number}
type Car = {color::String,model::String}

myhouse = {color:"pink",rooms:3}
mycar = {color:"red",model:"Corolla"}

getColor :: forall r. {color::String | r} -> String
getColor x = x.color

> getColor mycar
"red"
```

Required fields

```
getColor :: forall r. {color::String | r} -> String
```

Any other fields

Required fields

```
getColor :: forall r. {color::String | r} -> String
```

Polymorphic Row Variable

```
type House = {color::String,rooms::Number}
type Car = {color::String,model::String}

myhouse = {color:"pink",rooms:3}
mycar = {color:"red",model:"Corolla"}

getColor :: forall r. {color::String | r} -> String
getColor x = x.color
```

```
type House = {color::String,rooms::Number}
type Car = {color::String,model::String}

myhouse = {color:"pink",rooms:3}
mycar = {color:"red",model:"Corolla"}

lighten::forall r.{color::String|r} -> {color::String|r}
lighten x = x { color = "light " ++ x.color }
```

```
type House = {color::String,rooms::Number}
type Car = {color::String,model::String}

myhouse = {color:"pink",rooms:3}
mycar = {color:"red",model:"Corolla"}

lighten::forall r.{color::String|r} -> {color::String|r}
lighten x = x { color = "light " ++ x.color }

> getColor (lighten myhouse)
"light pink"
```

```
type House = {color::String,rooms::Number}
type Car = {color::String,model::String}

myhouse = {color:"pink",rooms:3}
mycar = {color:"red",model:"Corolla"}

lighten::forall r.{color::String|r} -> {color::String|r}
lighten x = x { color = "light " ++ x.color }

> getColor (lighten mycar)
"light red"
```

Haskell strikes back!

Type-level Strings

```haskell
data Person =
  Person {firstName::String, lastName::String}

bob :: Person
bob =
  Person {firstName="bob", lastName="cuttey"}

bobFirstName = firstName bob
bobLastName = lastName bob
```

haskell

```haskell
type Person =
  Map String String


bob' :: Person
bob' =
  fromList [("firstName", "bob"),
            ("lastName", "cuttey")]

bobFirstName' = bob' ! "firstName"
bobLastName' = bob' ! "lastName"
```

haskell

```
bob =
  Person {firstName="bob",
          lastName="cuttey"}


bob' =
  fromList [("firstName", "bob"),
            ("lastName", "cuttey")]
```

haskell

Field name

```
bob =
  Person {firstName="bob",
          lastName="cuttey"}
```

String

```
bob' =
  fromList [("firstName", "bob"),
            ("lastName", "cuttey")]
```

haskell

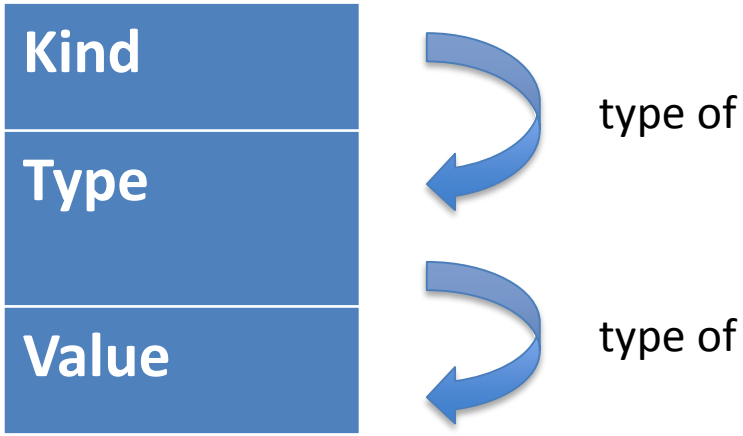
Type-level information

```
bob =
  Person {firstName="bob",
          lastName="cuttey"}
```

Value-level information

```
bob' =
  fromList [("firstName", "bob"),
            ("lastName", "cuttey")]
```

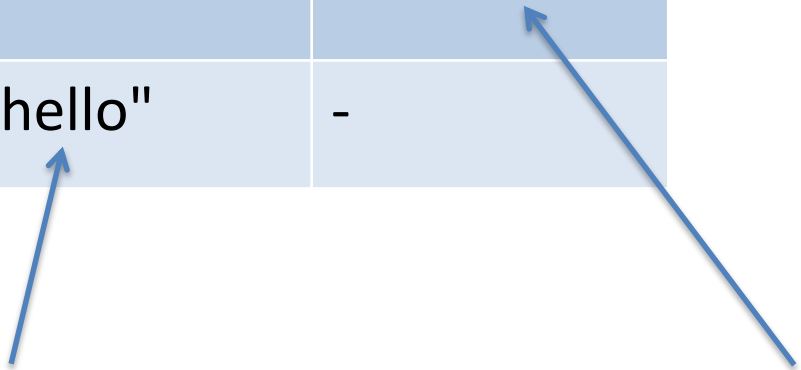
haskell


Kind
Type
Value
type of
type of

Kind	*
Type	String
Value	"hello"

Kind	*	Symbol
Type	String	"firstName"
Value	"hello"	-

Kind	*	Symbol
Type	String	"firstName"
Value	"hello"	-



Value-level String

Type-level String

```haskell
data Field (fieldName::Symbol) a = Field a


bob :: Field "firstName" String
bob = Field "bob"
```

haskell

```
data Field (fieldName::Symbol) a = Field a

                    name          type

bob :: Field "firstName" String
bob = Field "bob"
                    value
```

haskell

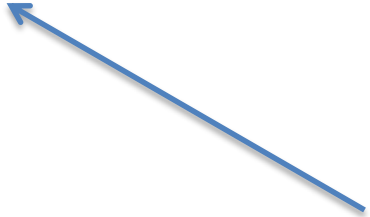
Vinyl

```haskell
{-# LANGUAGE DataKinds, KindSignatures TypeOperators,
GADTs, MultiParamTypeClasses #-}

import Data.Vinyl

get key rec = getField (rget key rec)
```

haskell

```
firstName = SField :: SField '("firstName", String)
lastName = SField :: SField '("lastName", String)
```

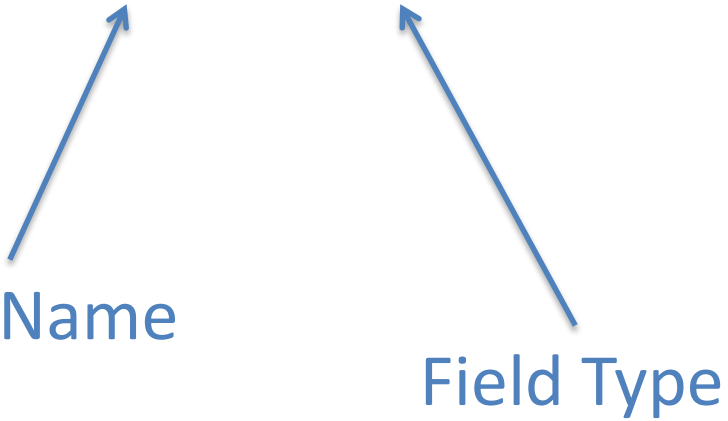
SField Constructor



SField Type Constructor

haskell

```
firstName = SField :: SField '("firstName", String)
lastName = SField :: SField '("lastName", String)
```



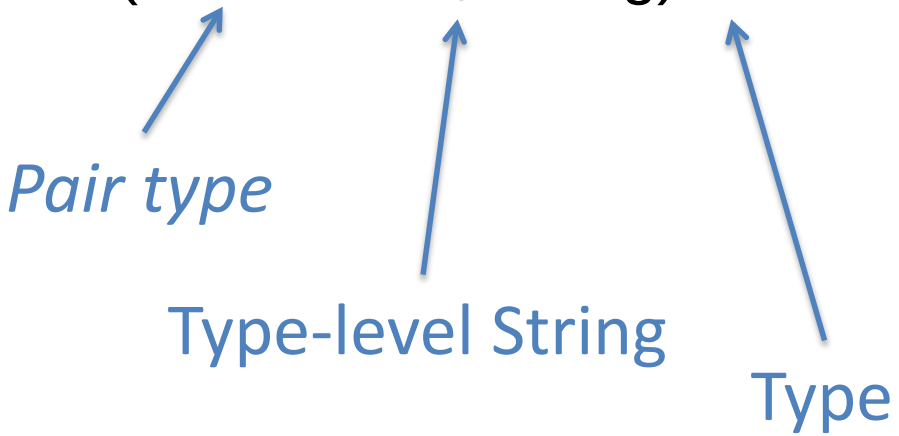
haskell

```
firstName = SField :: SField '("firstName", String)
lastName = SField :: SField '("lastName", String)
```

Pair type

haskell

```
firstName = SField :: SField '("firstName", String)
lastName = SField :: SField '("lastName", String)
```

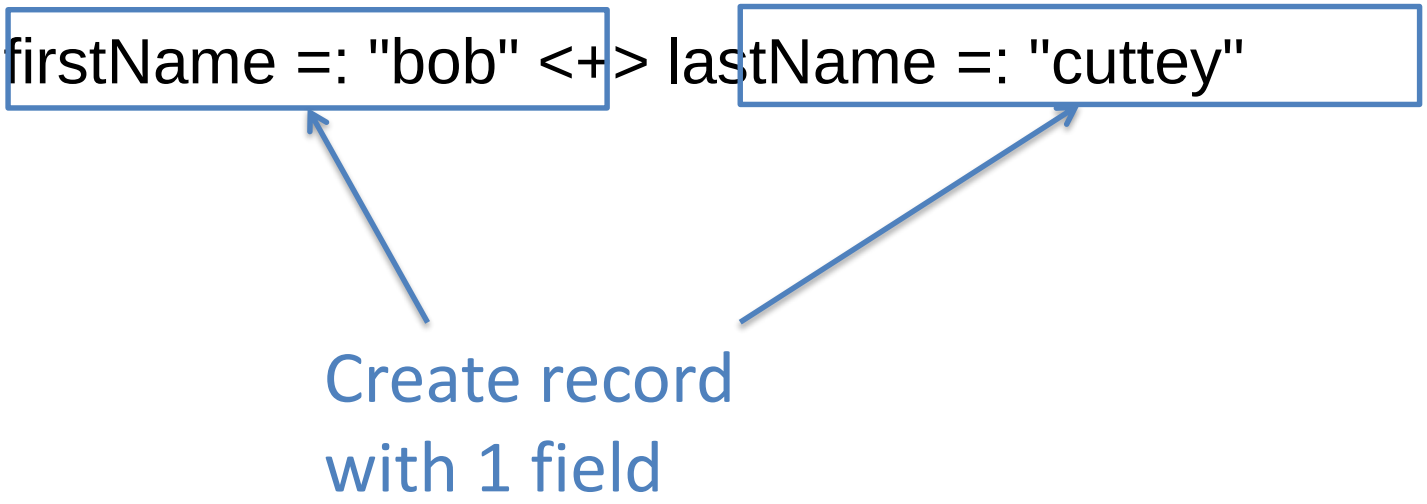


haskell

```haskell
firstName = SField :: SField '("firstName", String)
lastName = SField :: SField '("lastName", String)

bob = firstName =: "bob" <+> lastName =: "cuttey"
```

haskell

```
firstName = SField :: SField '("firstName", String)
lastName = SField :: SField '("lastName", String)

bob = firstName =: "bob" <+> lastName =: "cuttey"
```



haskell

```
firstName = SField :: SField '("firstName", String)
lastName = SField :: SField '("lastName", String)

bob = firstName =: "bob" <+> lastName =: "cuttey"
```

Append records

haskell

```
firstName = SField :: SField '("firstName", String)
lastName = SField :: SField '("lastName", String)

bob = firstName =: "bob" <+> lastName =: "cuttey"

bobFirstName = get firstName bob
bobLastName = get lastName bob
```

haskell

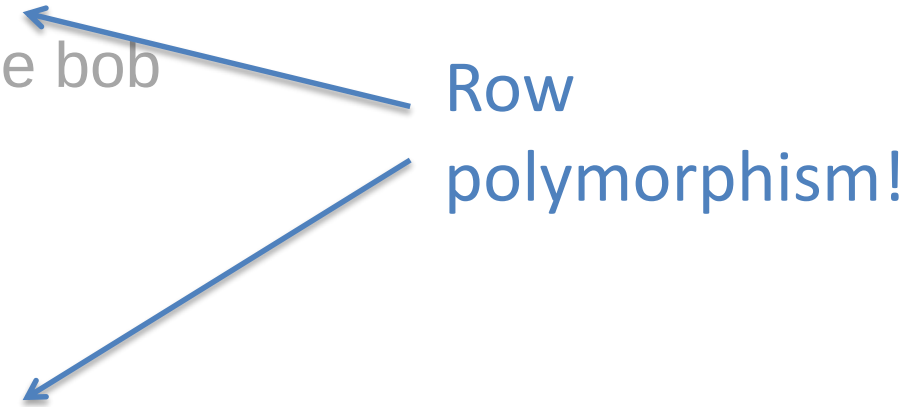
```haskell
firstName = SField :: SField '("firstName", String)
lastName = SField :: SField '("lastName", String)

bob = firstName =: "bob" <+> lastName =: "cuttey"

bobFirstName = get firstName bob
bobLastName = get lastName bob

dog = firstName =: "doggie"

dogFirstName = get firstName dog
```

haskell

```haskell
firstName = SField :: SField '("firstName", String)
lastName = SField :: SField '("lastName", String)

bob = firstName =: "bob" <+> lastName =: "cuttey"

bobFirstName = get firstName bob
bobLastName = get lastName bob

dog = firstName =: "doggie"

dogFirstName = get firstName dog
```

haskell

```
firstName = SField :: SField '("firstName", String)
lastName = SField :: SField '("lastName", String)

bob = firstName =: "bob" <+> lastName =: "cuttey"

bobFirstName = get firstName bob
bobLastName = get lastName bob

dog = firstName =: "doggie"

dogFirstName = get firstName dog
```



haskell

In conclusion...

Record = {(Name, Type, Value)}

Define record type

Instantiate type

Get field

Set field

Add field

Remove field

Structural typing

- Extensible records
- Row Polymorphism

Built-in record systems:

Elm > PureScript > Haskell

Vinyl library

Cheers!

Slides and code at
https://github.com/techtangents/recordsystemstalk