Deriving a Relationship from a Single Example

Neil Mitchell
community.haskell.org/~ndm/derive

Haskell data type

- Haskell let’s us define data types:

```
data Language
    = Haskell [Extension] Compiler
     | Javascript
     | Cpp Version
```

Eq instance

- We can define equality on data types:

instance Eq Language where

$\text{Haskell } x_1\ x_2 \equiv \text{Haskell } y_1\ y_2 = x_1 \equiv y_1\ \&\&\ x_2 \equiv y_2$

$\text{Javascript} \equiv \text{Javascript} = \text{True}$

$\text{Cpp } x_1 \equiv \text{Cpp } y_1 = x_1 \equiv y_1$

$\_ \equiv \_ = \text{False}$

What is the relationship?

- Given a new data type, could you define equality on it?

- Could you precisely specify the relationship?
  - If so, in what formalism?

The relationship

```
List [Instance ["Eq"] "Eq" (List [App "InsDecl" (List [App "FunBind" (List [Concat (List [MapCtor (App
"Match" (List [App "Symbol" (List [String "=="]),List [App "PApp" (List [App "UnQual" (List [App "Ident"
(List [CtorName])]),MapField (App "PVar" (List [App "Ident" (List [Concat (List [String "x",ShowInt
FieldIndex])])]))])]),App "PApp" (List [App "UnQual" (List [App "Ident" (List [CtorName])]),MapField (App
"PVar" (List [App "Ident" (List [Concat (List [String "y",ShowInt FieldIndex])])]))])],App "Nothing" (List
[]),App "UnGuardedRhs" (List [Fold (App "InfixApp" (List [Head,App "QVarOp" (List [App "UnQual"
(List [App "Symbol" (List [String "&&"])])]),Tail])) (Concat (List [MapField (App "InfixApp" (List [App
"Var" (List [App "UnQual" (List [App "Ident" (List [Concat (List [String "x",ShowInt FieldIndex])])])]),App
"QVarOp" (List [App "UnQual" (List [App "Symbol" (List [String "=="])])]),App "Var" (List [App "UnQual"
(List [App "Ident" (List [Concat (List [String "y",ShowInt FieldIndex])])])])])),List [App "Con" (List [App
"UnQual" (List [App "Ident" (List [String "True"])])])]]))]),App "BDecls" (List [List []])])),List [App "Match"
(List [App "Symbol" (List [String "=="]),List [App "PWildCard" (List []),App "PWildCard" (List [])],App
"Nothing" (List []),App "GuardedRhss" (List [List [App "GuardedRhs" (List [List [App "Qualifier" (List
[App "InfixApp" (List [App "App" (List [App "Var" (List [App "UnQual" (List [App "Ident" (List [String
"length"])])]),App "List" (List [MapCtor (App "RecConstr" (List [App "UnQual" (List [App "Ident" (List
[CtorName])]),List []]))])]),App "QVarOp" (List [App "UnQual" (List [App "Symbol" (List [String
">"])])]),App "Lit" (List [App "Int" (List [Int 1])])])])],App "Con" (List [App "UnQual" (List [App "Ident" (List
[String "False"])])])])]]),App "BDecls" (List [List []])])]])])])])]
```

Can anyone spot the deliberate typo?

Relationship details

- To implement the relationship:
  – Input language/data type
  – Transformation language
  – Output language/data type

- Transformation could be Haskell?
- Others require a lot of learning

An easier way

- Write one example instance for a particular data type
- Derive the relationship *automatically*

- No human need read or write that horrible slide

The particular data type

data Sample a = First | Second a a | Third a

instance Eq a ⇒ Eq (Sample a) where

First ≡ First = True

Second x_1 x_2 ≡ Second y_1 y_2 = $x_1 \equiv y_1$ && $x_1 \equiv y_2$ && True

Third x_1 ≡ Third y_1 = $x_1 \equiv y_1$ && True

_ ≡ _ = False

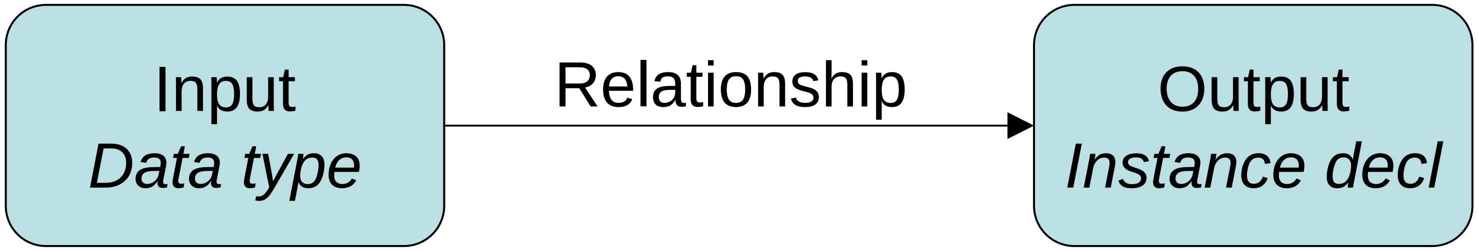
+ the Derive tool
= the relationship

The Derive tool

- Automatically generate instances for data types
  - Works via Template Haskell
  - Or via SYB
  - Or via Haskell-src-exts
- More instances = better
  - But more work for me…

Our Scheme

Our scheme



- Given 1 output for a particular input, derive the relationship

Restricted relationship (DSL)

- The relationship is a function
- But there are infinite functions, we can't write functions down easily…
- Instead have a DSL for the relationship
  - Tailored to each problem
  - Exactly the right expressive power

Our scheme (2)

data Input, Output, DSL

apply :: DSL $\rightarrow$ Input $\rightarrow$ Output

sample :: Input

derive :: Output $\rightarrow$ [DSL]

+ correctness
+ predictability

Correctness

- Derive must generate something consistent

$\forall o \in \text{Output}, d \in \text{derive } o, \text{apply } d \text{ sample} \equiv o$

Predictability

- The derive function is predictable if it does what the user expects
- Two DSL values are congruent if for all inputs they produce the same output
- All outputs from derive must be congruent

- But now the user needs to know/understand derive – not good!

Predictability (2)

- Stronger: Any possible result satisfying the correctness property is congruent

$\forall d_1, d_2$, apply d_1 sample $\equiv$ apply d_2 sample
$\Rightarrow d_1 \cong d_2$

- Predictability is *not* related to the derive function.

Instantiation of our scheme

- Input is data type descriptions
  - Using the haskell-src-exts data type
- Output is Haskell source code
  - Again using haskell-src-exts
- DSL is the relationship
  - Small functional language, with fold/map etc.
  - Plus functions over constructors/fields
  - And predictability proof

Bibtex Citations

Bibtex citations

- There are *many* Bibtex citation styles
  - All vary by where author name/year etc go
  - Implemented in Latex style files (ish)
    - I assume it's ugly – but don't actually know!

- Let's define a little DSL and prove it has the right properties
  - Illustrative of the paper

A citation type (Input)

```
data Input = Citation
    {year :: Int
    ,authors :: [(String,String)]}

Citation
    {year = 2009 -- Haskell considered evil
    ,authors = [(“Bjarne”,“Stroustrup”)
                ,(“James”,“Gosling”)]}
```

A little language (DSL)

```
data DSL1 = Str String
          | Year
          | Head DSL
          | AuthorFst
          | AuthorSnd
          | Authors String DSL

type DSL = [DSL1]
```

Bibtex apply

```
apply ds i = concatMap (`apply1` i) ds

apply1 :: DSL1 → Input → Output
apply1 (Str x) i = x
apply1 (Year x) i = show $ year i
apply1 (Head x) i = take 1 $ apply x i
apply1 (AuthorFst x) i = fst $ head $ authors i
apply1 (AuthorSnd x) i = snd $ head $ authors i
apply1 (Authors s x) i = intercalate s
    [apply x i{authors=[a]} | a ← authors i]
```

Some examples

- Stroustrup and Gosling 2009
  - [Authors “ and ” [AuthorSnd], Str “ ”, Year]
- B Stroustrup, J Gosling
  - [Authors “, ” [Head [AuthorFst], Str “ ”, AuthorSnd]]
- SG2009
  - [Authors “” [Head [AuthorSnd]], Year]

Challenge 1

- Stroustrup et al 2009

- Should omit “et al” if only 1 author
- Can this be defined in the DSL?

Solution

- Stroustrup et al 2008

```
[AuthorSnd]++ map f “ et al” ++[Str “ ”, Year]
    where
        f c = Head [Authors [c] []]
```

Challenge 2

- Give 2 congruent DSL's

Solutions

[Str “hello”] = [Str “he”, Str “llo”]

[Head [Str “”]] = [Str “”]

[Head [Head x]] = [Head x]

[Authors “” []] = [Str “”]

[Authors x [Authors y z]] = [Authors x z]

- Lot’s of congruent DSL’s

Challenge 3

- Come up with a sample input

- Needs to ensure the predictability property

$\forall d_1, d_2$, apply d_1 sample $\equiv$ apply d_2 sample
$\Rightarrow d_1 \cong d_2$

No solution!

- There is no possible sample which could work

derive “2009” =

[[Str “2009”]

,[Year]]

- Can’t tell what comes from where

Solution

- Give restrictions on the DSL
  - Aim to restrict to have only 1 meaning to each sample
  - Aim to give a natural/simple meaning

- Many possible design solutions
  - First thought: restricting Str?
  - Anyone any ideas?

Possible restrictions

- Restrict DSL
  - Head can only be applied to AuthorFst or AuthorSnd
  - Str cannot contain upper case or numbers

```
sample = Citation {Year = 2009
  , authors = [(“AMY”, “BALE”)
               ,(“CRAIG”, “DODDS”)]}
```

Previous examples simple

- BALE and DODDS 2009
- A BALE, C DODDS
- BD2009

- Can’t do the challenge 1 task

Bibtex summary

- Define a sensible looking DSL
- Restrict DSL (if necessary) while thinking about a sample
  - There is not always an obvious answer

- The derive in this restricted DSL is trivial
  - Challenge 4 ☺

Deriving Instances

Back to instances

data Sample a = First | Second a a | Third a

instance Eq a ⇒ Eq (Sample a) where

  First ≡ First = True

  Second x_1 x_2 ≡ Second y_1 y_2 = x_1 ≡ y_1 && x_1 ≡ y_2 && True

  Third x_1 ≡ Third y_1 = x_1 ≡ y_1 && True

  _ ≡ _ = False

- Given sensible restrictions, how do we derive?

What must derive do?

derive :: Output $\rightarrow$ [DSL]

- Be correct
- Terminate, ideally quickly
- Hope to find an answer if one exists

- The following implementation is just one possible version

Create guesses

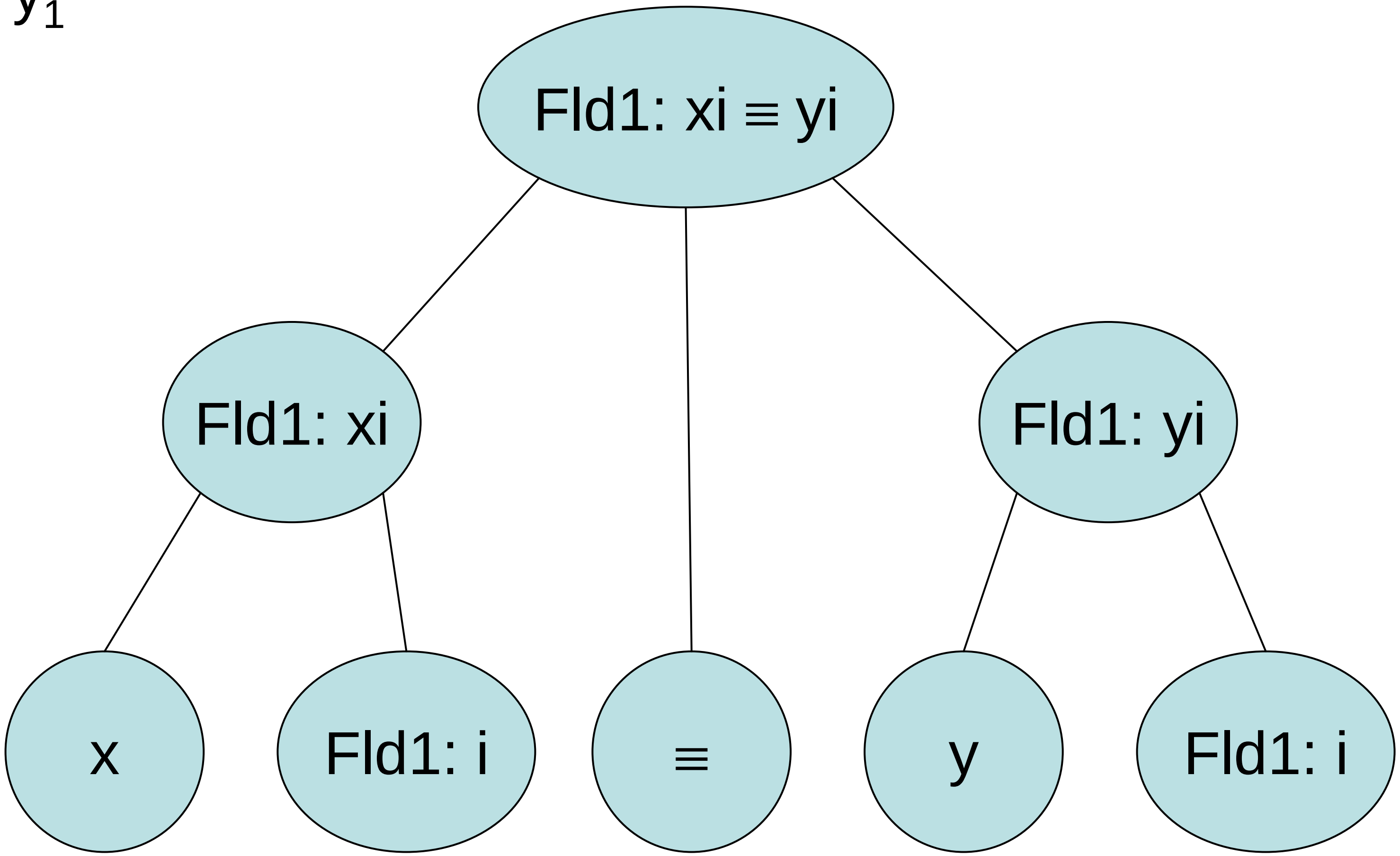
guess :: OutputFragment → [Guess]

data Guess
    = Guess DSL
    | GuessCtr Int_0based DSL
    | GuessFld Int_1based DSL

- Guess bottom-up and combine

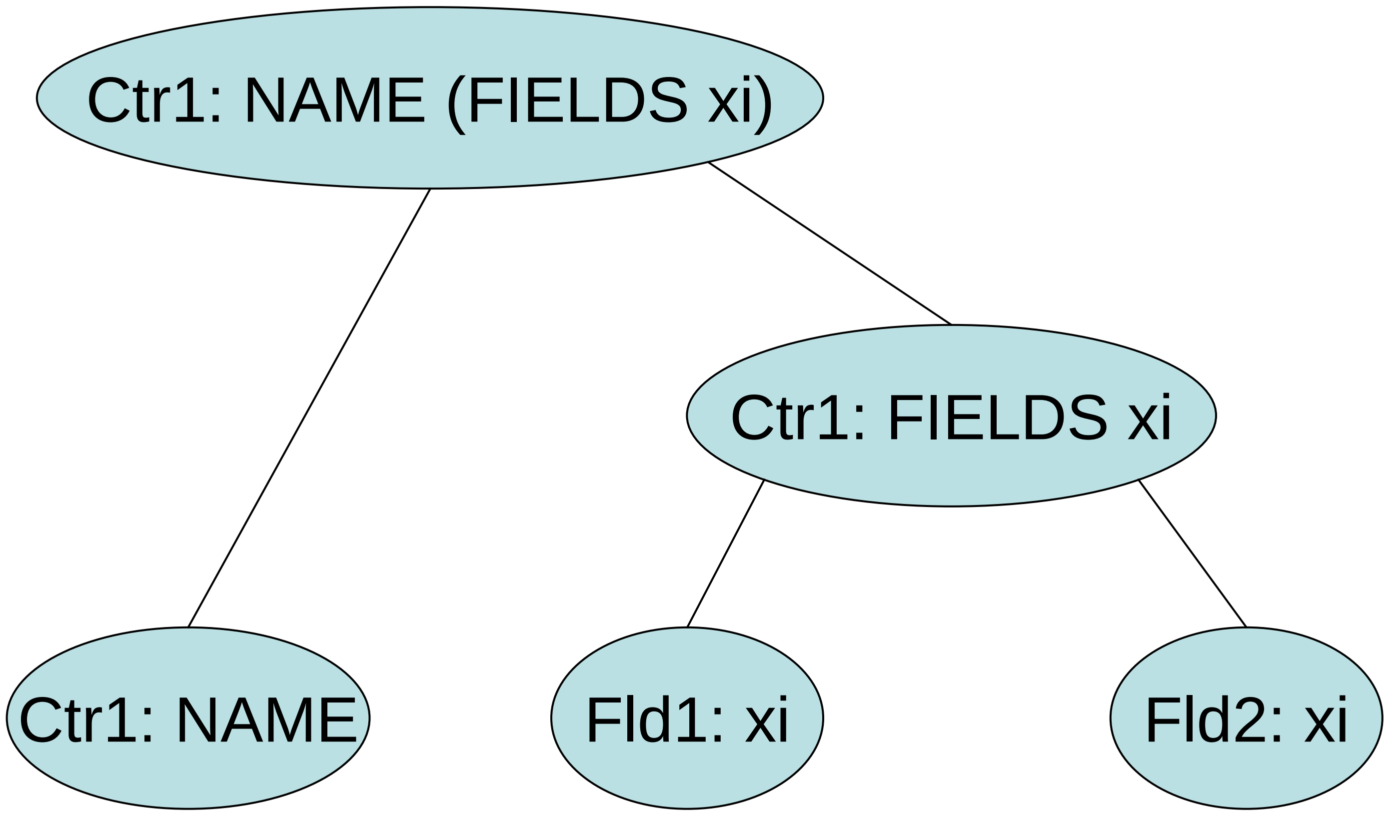
Examples

$x_1 \equiv y_1$



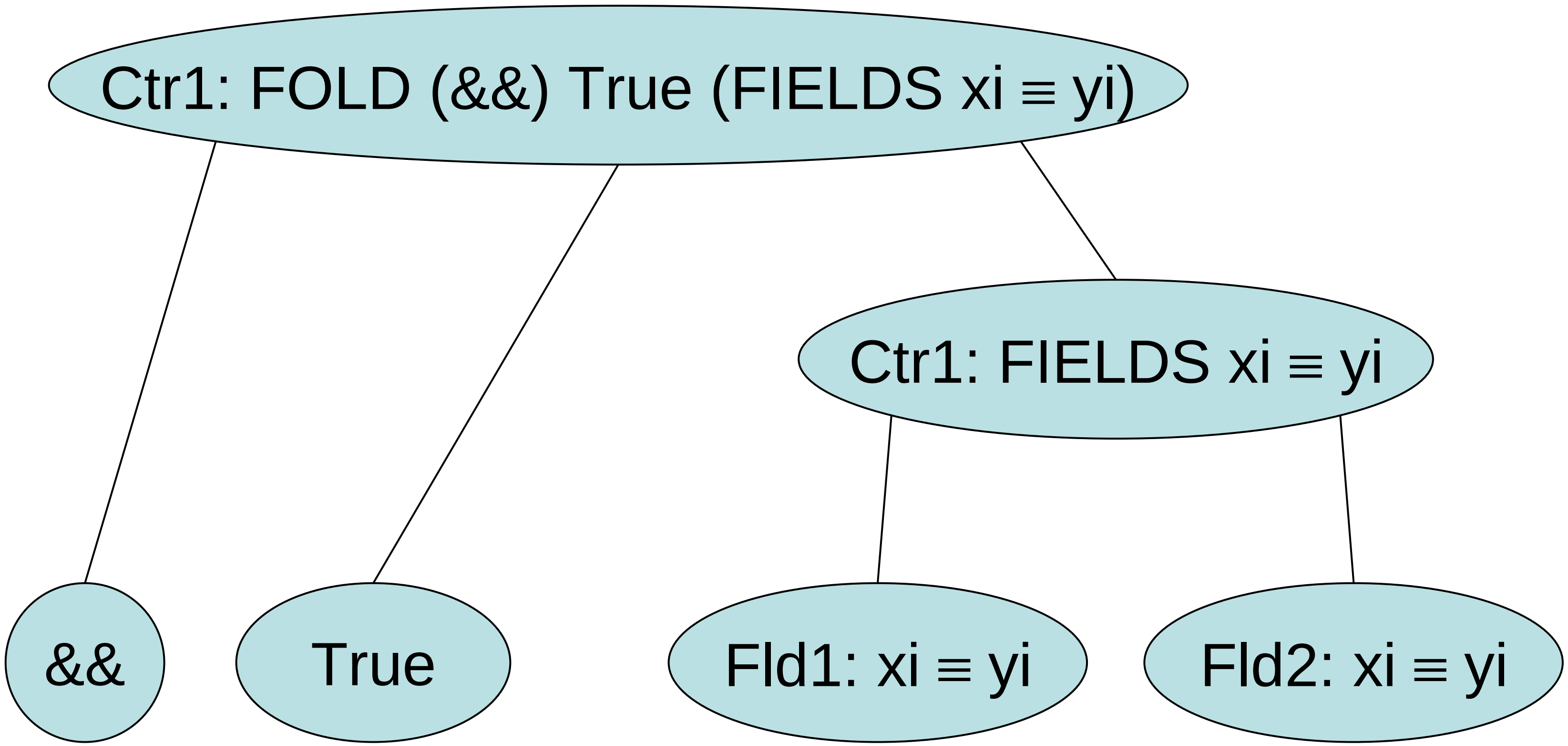
Examples

Second x_1 x_2



Examples

$x_1 \equiv y_1$ && $x_2 \equiv y_2$ && True



Guessing atoms - integers

- The number 2
  - – Might be the literal 2
  - – Might be the second field
  - – Might be the arity of constructor Second
  - – Might be the index of constructor Third
- Produce all these guesses

Guessing atoms - strings

- “Foo” – the literal string “Foo”
- “Second” – the name of Second
  - – not allowed to be a literal
- “Sample” – the name of the data type
  - – again, not allowed to be a literal

Application

- Given (a b)
  - Guess a, then b, then combine if consistent
- Guess x can be turned into GuessCtr i x
- x_1
  - Guess (Lit “x”) & GuessFld 1 FieldInd
  - GuessFld 1 (Lit “x” `Append` FieldInd)

Lists

- Can combine adjacent elements similar like we do for application
- Can lift a complete sequence:
  - [GuessFld 1 x, GuessFld 2 x] $\Rightarrow$ GuessCtr 1 (Fields x)
  - [GuessCtr 0 x, GuessCtr 1 x, GuessCtr 2 x] $\Rightarrow$ Guess (Ctors x)

Special guesses

- Folds
  - Special hard-coded patterns are recognised
  - Turns into a fold, then normal guess on the arguments to the fold
- Vector application
  - haskell-src-exts has binary App nodes
  - Sometimes vector application is required, transform separately

Examples and Limitations

Module names

typename_Language =
    mkTyCon "ModuleName.Language“

- This doesn’t work as the input doesn’t contain the module name
  - Can always enrich the input
  - But might need a more complex sample

Infix constructors

show (Prefix a b) = [“Prefix”,show a,show b]

show (a :+: b) = [show a,“:+:”,show b]

- The input type doesn’t know about fixity
  - Could enrich the input type

Type-based derivations

- Some classes make choices based on the types of a constructors fields (i.e. Uniplate)
- The input doesn't have type information
  - If it did, a suitable sample would be huge
- Lack of type signatures means no -Wall
  - Some functions can be derived without their type sig, but not with

Variable naming

- Be careful when naming your variables

Second x y -- bad

Second x_1 x_2 -- good

- Think if you could come up with a simple pattern

Redundant fold terms

- Specify redundant fold units to make a pattern

$[0, x_1+x_2, x_1]$ -- bad

$[0, x_1+x_2+0, x_1+0]$ -- good

- Derive will usually optimise these bits away

The empty record

- The empty record match is incredibly useful

f (First{}) = …

f (Second{}) = …

f (Third{}) = …

Results

The results

- Our scheme is used in Derive
- Works (14)
  - ArbitraryOld, Arities, Binary, BinaryDefer, Bounded, Default, Enum, EnumCyclic, Eq, Monoid, NFData, Ord, PlateTypeable, Serial
- Partial (4)
  - Arbitrary, Data, DataAbstract, Read, Show

Main causes of failure

- Record based (5)
  - Update, Set, Ref, LazySet, Has
- Type based (6)
  - Uniplate, TTypeable, Traversable, PlateDirect, Functor, Foldable
- Other (3)
  - Is (type sig), Fold (type sig), Typeable (kind info)

Conclusion

- From a single example we can define a relationship
  - Which is correct and predictable
- Has been practically applied to instance generation (Derive tool)

cabal install derive