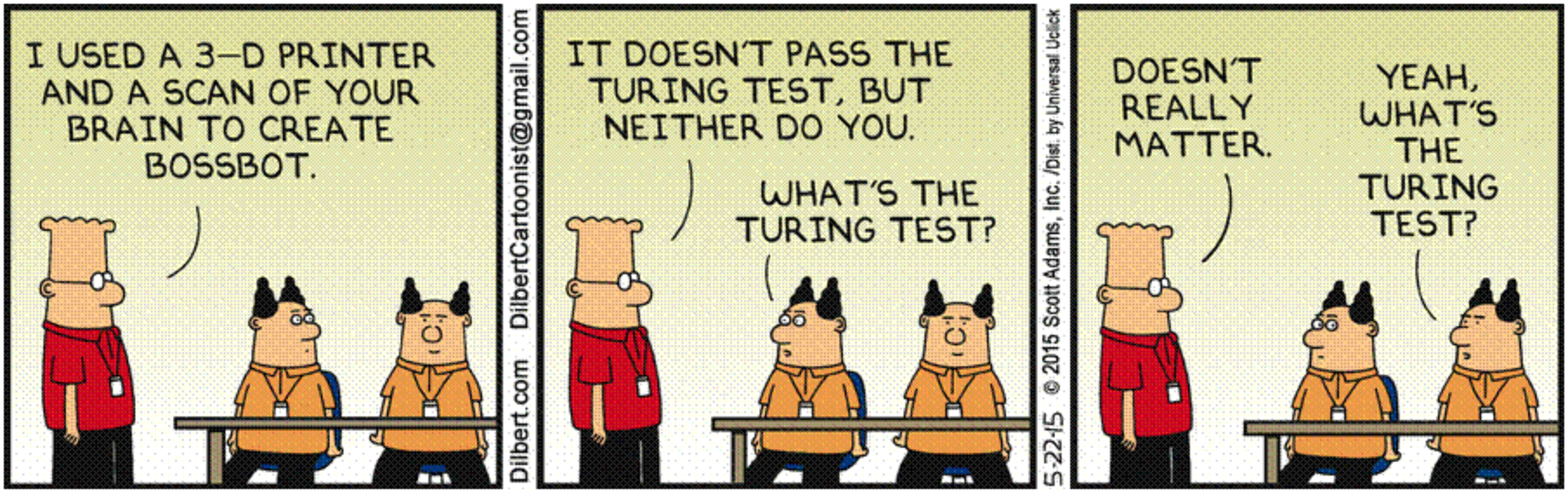
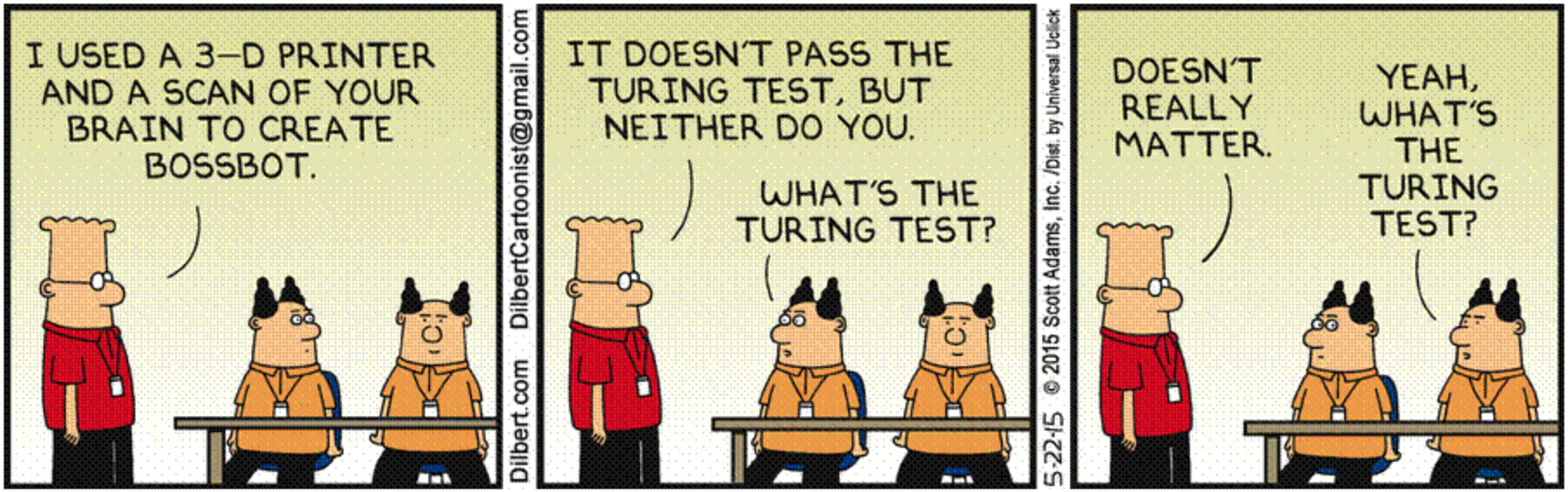
Haskell Scrap Your Boilerplate (SYB) – An Experience Report



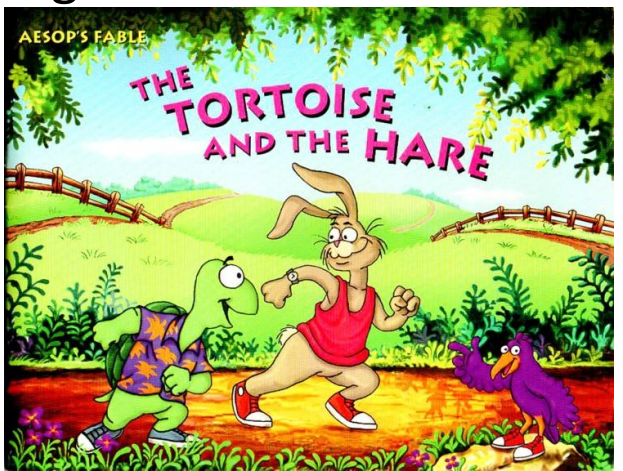
Agenda

- Wh* - Who, Why and When
- What
- How



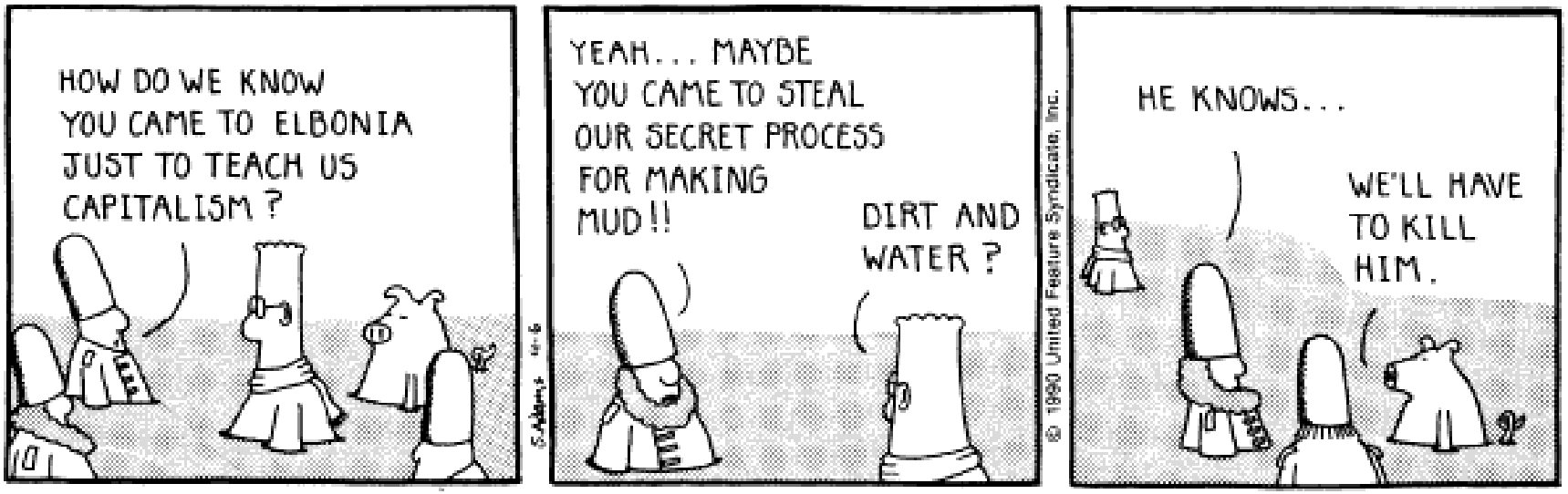
Who - Aesop's Tortoise

- Maths Rules
  - I remember back in the 'good' old bad days, in my Maths degree, learning Linear Algebra, Differential Equations, Central Limit theorem, which all involved mathematical proof.
  - But computing 101 was fully imperative with FORTRAN, COBOL.
  - And something felt broken - it was like a logical train wreck.
  - But Haskell led me to the way of constructive logic.
- "The unexamined life is not worth living" - Socrates
  - This is what learning means.
  - So I prepared myself for embarrassment and growth. ☺



Why Listen?

- Who is this for?
  - Anybody who uses Haskell for ingesting data.
  - Some Haskell knowledge is assumed.
- The journey
  - It started with Template Haskell, which just produced a big ball of mud. (My fault, not the fault of TH.)
  - Then, on the journey, I discovered SYB (SPJ) and Oleg!

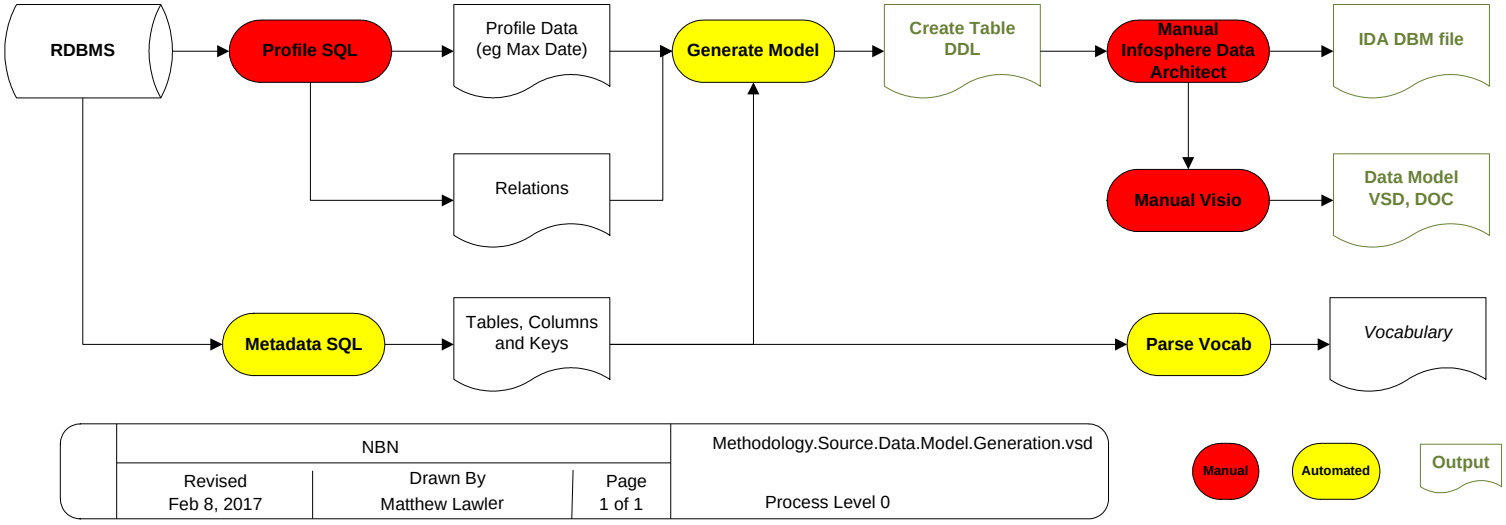


Why Listen, really ?

- When using Haskell and constantly changing loadable data types during design, this module uses a DRY approach.
- DRY = Don't repeat yourself
- WET = "write everything twice", "we enjoy typing" or "waste everyone's time".
- Simon Peyton-Jones etc. created SYB or Scrap Your Boilerplate is Data.Generics
- Oleg Kiselyov proposed an his ‘Impossible’ SYB.
- Finally, it uses 2 types of Property based testing.

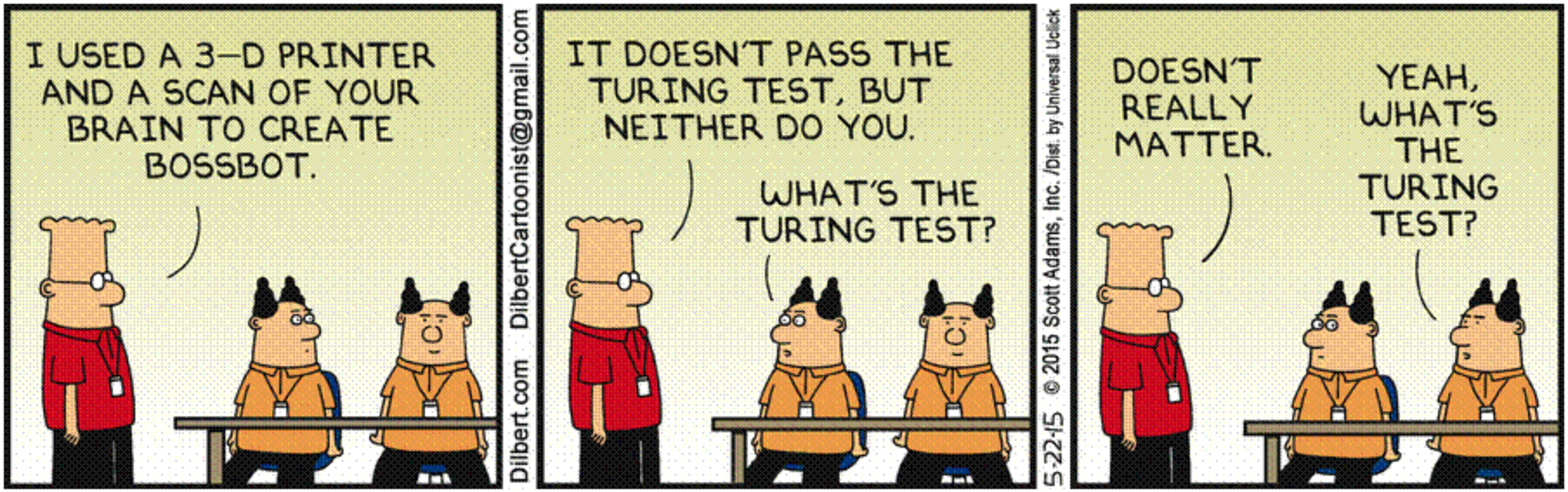
When and Where?

- The specific use case was loading in database metadata that could be used to generate DDL and SQL.
- As the only user of this module, it is not designed for general use.
- On Github at: https://github.com/lawlermj1/DBGlossary
- module DBCommon.Boilerplate
- The IO is very simple, consisting of CSV file handing.
- DB Meta -> CSV -> HS Type -> HS -> SQL -> DB



Agenda

- Wh* - Who, Why and When
- What
- How



What – main class functions

```
gdefaultU :: Data a => a -> a
-- Generates a default from a type
-- U means ‘a’ type is undefined, no need to have a value
-- U means that it uses Oleg’s impossible SYB

gshowQ :: Data a => a -> String
-- Encoder - Takes a value and turns it into a String
-- Q means this uses the SYB module

gloadU :: Data a => a -> [String] -> a
-- Decoder - Reads a CSV line
```

What – Usage example

```
--  example ADT – Algebraic Data Type
data DBCommonFile = DBCommonFile { dBCommonFileType :: FileType
      , dBCommonFileName :: String
      , dBCommonFileIsIn :: Bool
      , dBCommonFileCount :: Int
} deriving (  Eq, Ord, Typeable, Data  )

--  Impossible SYB default function
dBCommonFileDefault = gdefaultU ( undefined::DBCommonFile )

--  Impossible SYB load function
dBCommonFileLoad = gloadU ( undefined::DBCommonFile )

--  SYB show function
instance Show DBCommonFile where show = gshowQ
```

What - Inverse Function Property

-- An **invariant** is a object **property** which remains unchanged when a function is applied to the object.

-- For any type, composing the class function and its inverse should equal the identity function.

-- gread and gshow are both SYB functions

```
prop_inverse_gread ::  ( Data a, Eq a ) => a -> Bool
prop_inverse_gread a
    = ( fst.head.gread.gshow ) a == id a
```

What - Equivalent Function Property

-- For any type, the equivalent class functions should produce equal results. Here there are SYB (uses gmapQ) and Impossible SYB (uses gunfold)

-- gADT means generic Algebraic Data Type

-- D means Defined (uses gmapQ) and U means Undefined (uses gunfold)

-- 2 intermediate forms are DTTree and [Nameda] list

-- Four properties used are:

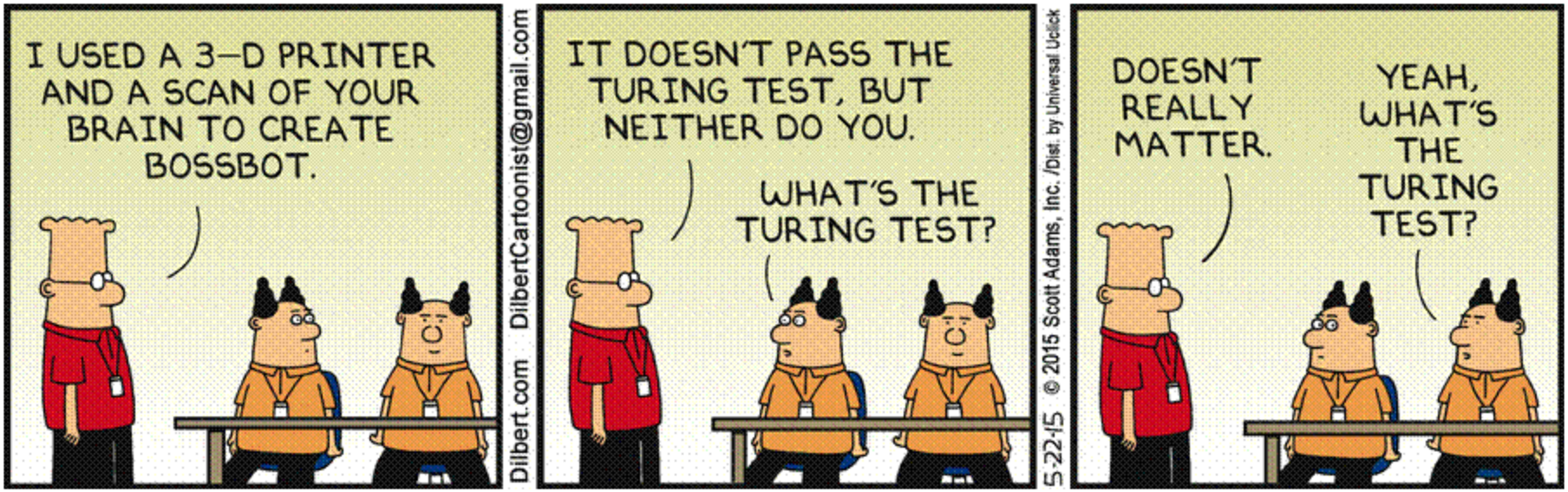
```
prop_gADTD2Tree_eq_gADTU2Tree :: Data a => a -> Bool
prop_gADTD2List_eq_gADTU2List :: Data a => a -> Bool
prop_gldD_eq_gldU :: Data a => a -> Bool
prop_guldD_eq_guldU :: Data a => a -> Bool
```

What - Requirements (or simplifications)

- As this is for metadata from a database, there is no need for recursive data structures – Trees are not supported in DDL and SQL.
- Values are often not available - only the type definition, so the term level is undefined.
- Only working in Haskell, so this is a context free transformation.
- Very much a “roll your own” tool.

Agenda

- Wh* - Who, Why and When
- What
- How

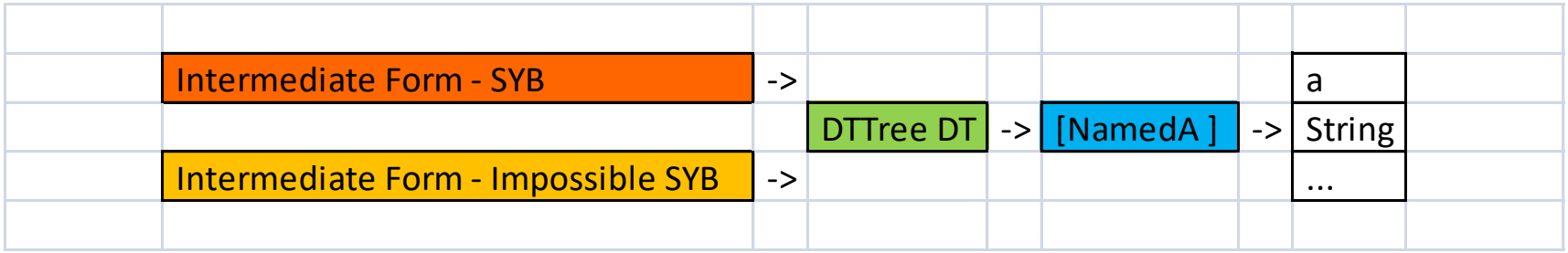


How - Comparing SYBs

Feature	SYB (Simon)	Impossible SYB (Oleg)
On Hackage	Yes - Data.Data	No - Oleg's web site
Ease of Use	Yes - OOB	No - transform to new intermediate form
Undefined	No	Yes

The advantage of having both is that the equivalent function property can be defined.

How - 2 SYB intermediate forms are used



- 2 Algebraic Data Type Abstractions from SYB and immediate SYB are inspected.
- Both are converted into a common DTTree DT.
- This is flattened into a NamedA list
- The NamedA is used for final generic functions

How – SYB or Data.Data

- By: Simon Peyton Jones and Ralf Lammel
- Hackage: Data.Data
- Main Function: gmapQ
- Easier to use, but does not handle undefined types.
- Read the paper: ‘Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming’
- https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Data.html
- Next to look at GHC.Generics

How – Impossible SYB

- By: Oleg Kiselyov
- Hackage: Not available
- Main Function: gunfold
- Hard to work with, but handles undefined types.
- Read the web page : ‘Seemingly impossible generic map in SYB’
- http://okmij.org/ftp/Haskell/generics.html#gmap
- The functions process the immediate sub terms of an algebraic data type and enable transformations.
- I only used a subset of his functions.

How – gshowQ using SYB

```
--   show using gmapQ to serialise a type
gshowQ :: Data a => a -> String
gshowQ a = intercalate sepCharS ( gshow2 a )
--  uses OOB SYB constructors
--  create a gshowX for all allowed ADT types
gshow2 :: Data a => a -> [String]
gshow2  = gshowRecurse2 `ext1Q` gshowList2 `ext1Q` gshowMaybe2
    `extQ` gshowString2 `extQ` gshowChar2 `extQ` gshowInt2 `extQ`
    gshowBool2XLS `extQ` gshowFloat2 `extQ` gshowDay2 `extQ`
    gshowInteger2 `extQ` gshowDouble2 `extQ`  gshowUTCTime2
--    convert True to 1 and False to 0 for easy use in Excel
gshowBool2XLS :: Bool -> [String]
gshowBool2XLS b = [if b then "1" else "0"]
--  that's it
```

How - generic default for undefined types using Impossible SYB

```
--    generic default for undefined types  (a really unsafe head here! ☹ )
gdefaultU :: Data a => a -> a
gdefaultU a= ( fst.head.gread ) ((gd2S.gdU) a) )

--    common fn to convert namedA to final string
gd2S :: [NamedA String String] -> String
gd2S ns = trim ( ( foldr ( . ) id ( intercalate [showChar ' '] [( map namedAFunction ns )] ) ) "" )

--    from an undefined type
gdU :: Data a => a -> [NamedA String String]
gdU a  = gd gADTU2ListU a

--    initial call to traverse ADT AST to build up a default record
gd :: Data a => ( a -> [DT] ) -> a -> [NamedA String String]

--   new entry point to flatten undefined tree
gADTU2ListU ::  Data a => a -> [DT]
gADTU2ListU a = updatedTFieldIndex2 ( flattenDFS (gADTU2Tree a) )
--    This follows the hylomorphism pattern – an anamorphism (fold) followed by a catamorphism (map)
```

How – DTTree – Intermediate Generic Data Type

```
data DTType = EnumDT      --          EnumDT is any Sum type
      | BaseDT            --          BaseDT can be Int, Float, Char, Bool, Double, String
      | ProductDT         --          ProductDT is any product type
      | MonadDT           --          MonadDT "Just" "(:)" constructors are used to identfy [] and Maybe
      | CloseBracketDT
      | CloseListBracketDT deriving (  Eq, Ord, Typeable, Show, Read  )

data DT = DT {
    dTName :: String
  , dTType :: DTType
  , dTConfields :: [String]
  , dTFieldIndex :: Int        --   +1 if a field ( enum or base, 0 if Product or Bracket
  , dTShowIndex :: Int         --   +1 if shown ( enum, base or product ), 0 if Bracket
  , dTAccessor :: Maybe String --    name of accessor function
  , dTIsInMonad :: Bool        --   is this field inside a Monad such as [] or Maybe?
  , dTMonadLevel :: Int        --   what level in this field inside a Monad?
  } deriving (  Typeable, Show, Ord, Read, Eq )

data DTTree a = DTNode {
     dTRootLabel :: Maybe a,
     dTSubForest :: [DTTree a] } deriving (  Typeable, Show, Ord, Read, Eq, Functor, Foldable )
```

How – NamedA
Intermediate Generic Flattened List

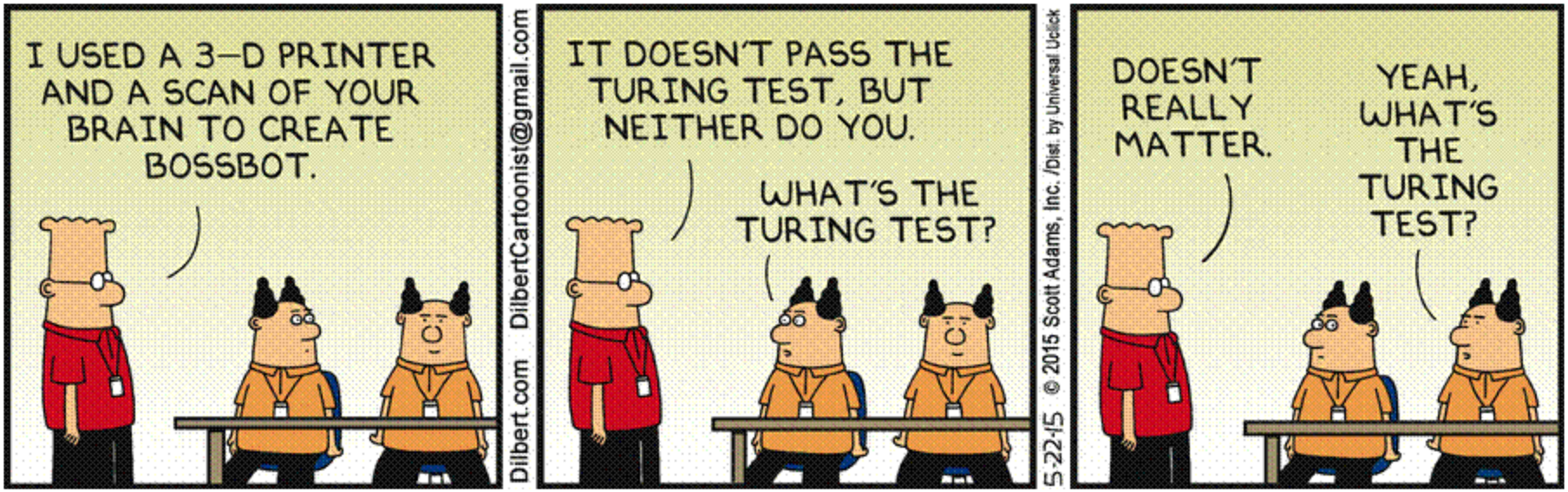
```
--    provides a universal intermediate form for all Algebraic Data Types
data NamedA b c = NamedA {
     namedAName :: String      --   type name
   , namedAValue :: b                    --   for functions, store the function name
   , namedAFunction :: b -> c  --   for simple type, use id, otherwise put function
   , namedADT :: DT                      --   the source DT
   , namedAFieldIndex :: Int   --   position in a flattened list including constructors
   , namedAShowIndex :: Int    --   position in a flattened list excluding constructors
   , namedAAccessorName :: String        --   field name ( to be used in header )
   , namedAIsInMonad :: Bool             --   is field in a Monad such as [] or Maybe?
   , namedAMonadLevel :: Int             --   what level is this field in the type tree
   , namedAConfields :: [String]         --   string list for enum type checking
   } deriving (  Typeable  )
```

How – finally generic LOAD undefined types using Impossible SYB

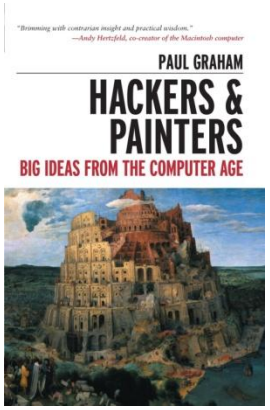
```
--    wraps gload in a greadF
gloadU :: Data a => a -> [String] -> a
gloadU  a xs     = greadF a ( gloadU2String a xs )

--     takes type and string list and builds a correctly formatted string for the type
gloadU2String :: Data a => a -> [String] -> String
gloadU2String  a xs  = gload2String ( gldU  a ) xs

--     extracts the NamedA list from this type
gldU :: Data a => a -> [NamedA String String]

--    simply fold in the xs using the ii index to each field (there is a length check for ii not shown)
--    creates a function f [String] -> String
gload2String :: [NamedA String String] -> [String] -> String
gload2String ns xs = concatMap ( name2Function xs ) ns

--    allocate a string argument to a function name
name2Function :: [String] -> NamedA String String -> String
```

Agenda

- Wh* - Who, Why and When
- What
- How

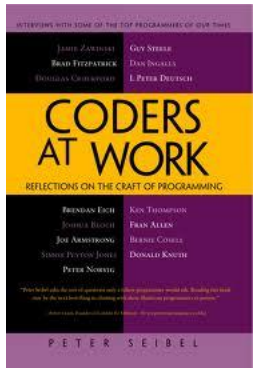


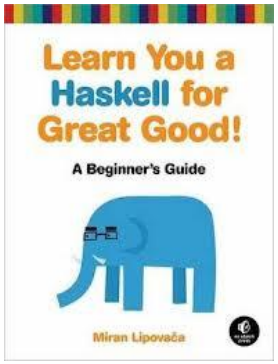
Continuing the Odyssey



- Hackers & Painters
- http://paulgraham.com/hackpaint.html

- Where the best thinkers are heading:
- http://www.codersatwork.com/
  - Donald Knuth
  - Joe Armstrong
  - Simon Peyton Jones
  - Peter Norvig
  - Guy Steele
- Finally, get your hands dirty with
- http://learnyouahaskell.com/





- Please reach out on linkedin: https://www.linkedin.com/in/matthewlawler/