

Deriving Generic Functions by Example (+10 years)

Neil Mitchell

<http://ndmitchell.com>

Guess the function

Input: aBc(

- (cBa
- bCd)
- ABC(
- aaBBcc((
- ac

Basic idea

$$f :: A \rightarrow B$$

- I pick: $a \in A$
- You pick f , give me b (where $b = f a$)
- I infer f
 - Correct for a ($b = f a$)
 - Correct for all A (predictable)

Concrete example

Let's derive 'is' functions for Haskell types

a: data MyType = Foo | Bar

b: isFoo Foo{} = True; isFoo _ = False

isBar Bar{} = True; isBar _ = False

You do not need to write down f.

Want to be sure f is what you wanted.

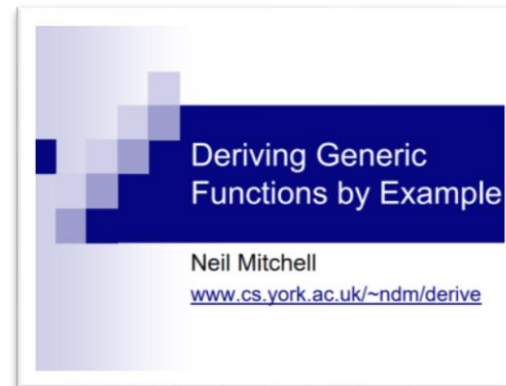
What happened to this work?

Where this work went

- 2007: York Doctoral Symposium (YDS) paper/talk
- 2008: York Programming Languages and Systems (PLASMA) talk
- 2009: Approaches and Applications of Inductive Programming (AAIP) keynote talk and reviewed post-publication
- 2007-2017: DERIVE open source project

2007: YDS

- My PhD involved learning to write English
 - With much thanks and credit to Colin Runciman
- YDS was a paper I wrote without Colin reading
 - Reading back, it's not too bad (6 small pages)
- All about an algorithm for inferring 'f' for one specific use case



2008: PLASMA

- More theory about how the algorithm worked, a bit more principled
 - f now quantified, can lift between quantifiers
- A sales pitch for the associated open-source DERIVE tool



2009: AAIP

- Invited to give a talk at a workshop
 - They'd seen my YDS work through my blog posts
- More formal and generic – less intuition
- Reviewed post-submission, 12 pages in 2-column style

Deriving a Relationship from a Single Example

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



Deriving a Relationship from a Single Example

Neil Mitchell
ndm@cs.ox.ac.uk

Abstract

Given an appropriate domain-specific language (DSL), it is possible to describe the relationship between Haskell data types and more general, abstract, datatype operations. However, describing the relationship is possible, it is not always an easy task. There is an alternative – simply give one example output for a carefully chosen input, and from the relationship derived.

When deriving a relationship from only one example, it is less precise than the derived relationship in the Haskell case. We identify general restrictions on the DSL, and on the provided example, to ensure a level of predictability. We then apply these restrictions in practice to derive the relationship between Haskell data types and generic functions. We have used our scheme to derive the Haskell code, where one Haskell type class was derived from a single example.

1. Introduction

In Haskell (Grossman 2001), type classes (Walker and Blum 1998) are used to provide similar operations for many data types. For each data type of interest, a user must define an associated instance. The instance definitions usually follow a highly regular pattern. Some libraries define new type classes, for example Haskell 98 (Haskell 2002) defines the `Monad` type class, which includes a number of standard laws. As an example, we can define a data type to describe user response programming language, and provide an `Monad` instance:

```
data Language = Haskell | Javascript | Cplusplus
              | Java Version
              | Java Version

instance Monad Language where
  ref (Haskell a, x) = ref a, ref a, ref a, ref a
  ref (Javascript a) = ref a, ref a, ref a, ref a
  ref (Cplusplus a) = ref a, ref a, ref a, ref a
```

We also need to define `Monad` instances for the data types themselves. Complete and Veridic, any instance of `Monad` has been naturally from the structure of the data type. For each instance of `Monad` has been applied, before returning `()`.

Writing an `Monad` instance for a single example data type is easy – one has to provide a single example data type for the relationship. The related relation is to express the relationship between a data type and its instance. In Haskell terms, such as `Monad` (Haskell 2002), the given description of the relationship can be used to derive the implementation of a data type, and a user can use the relationship to derive the implementation of a data type, and a user can use the relationship to derive the implementation of a data type.

Using the technique described in this paper, these relationships can often be automatically derived from a single example. To derive the generation of `Monad` instances, we require an example to be given for the `Monad` data type defined in Figure 1.

```
data Example a = First
              | Second a
              | Third a

Figure 1. The Example datatype.
```

Figure 1. The Example datatype.

```
instance Monad Example where
  ref (First a) = ()
  ref (Second a) = ref a, ref a, ref a, ref a
  ref (Third a) = ref a, ref a, ref a, ref a
```

The `Monad` instance for `Example` follows the same pattern as the `Language`. From this example, we can derive the general relationship. However, there are some possible relationships between the `Example` data type and the `Monad` – for example the function might always generate the `Second` for `Example` regardless of the input type. We overcome this problem by requiring the relationship to be defined in domain-specific language (DSL), and that the example has certain properties (see §3). With our restrictions, we can derive predictability.

1.1 Contributions

- We describe a scheme which allows us to take predictable and correct relationships (§2).
- We describe how this scheme is applicable to instance generation both in a high-level manner (§3), and more detailed procedural concrete (§5).
- We outline a method for deriving a relationship in an DSL, without resorting to unguided search (§6).
- We give concrete DSLs, including reasons why our solution fails (§6.1), to use experience, user skills of Haskell type classes can be derived using our method.

2. Other Derivation Scheme

In this section we define a general scheme for deriving functions, which we later use to derive operations around generation. In general terms, a function takes an input as an output. In our case, we require functions to function that can be described by a DSL, domain-specific languages. We need an input function to serve as an example for the DSL, which takes a DSL, and an input and produces an output. The scheme can be implemented in Haskell as follows:

```
data Input
data Output
data DSL
apply :: DSL --> Input --> Output
```

Formal setup

We pick all of:

- Input *the input type*
- Output *the output type*
- DSL *type of things describing functions*
- sample :: Input *chosen input*
- apply :: DSL \rightarrow Input \rightarrow Output *apply f*
- derive :: Output \rightarrow Maybe DSL *guess f*

Correctness

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

If derive succeeds,
it must work for the example

Predictability

$\forall i \in \text{Input},$

$d_1, d_2 \in \text{DSL},$

$\text{apply } d_1 \text{ sample} \equiv \text{apply } d_2 \text{ sample} \Rightarrow$

$\text{apply } d_1 i \equiv \text{apply } d_2 i$

If any input can distinguish two DSLs
it must be sample

Predictability not influenced
by derive!

Guess the function solved

Input: aBc(

Output: ac

Function: Pick odd indices, filter isLower

Option 1: Change sample to aBcd(

Option 2: Only permit one of those in DSL

2007-2017: DERIVE tool

- Generates instances
 - 60% of instances defined by example
 - Some instances have been moved into GHC
- Moderately successful Haskell tool
 - <https://github.com/ndmitchell/derive>
 - 843 commits
 - 10 forks, 15 stars, 3 watchers
 - 14 contributors (most a couple of patches)

DERIVE: End of the line

- There are lots of newer instances it can't do
 - Projects now ship an instance deriver with the instance, rather than centrally
- New way to define generic instances with GHC
- Examples define simple instances, which are the easiest ones anyway
- I don't personally use it anymore

What happened to me?

Personal life

- -10Y Move to Cambridge
- -8Y Got married (Emily)
- -5Y Had child (Henry)



Hobby/Mission

The Haskell logo consists of a stylized 'H' symbol on the left, formed by two overlapping chevron shapes pointing right, with a purple-to-blue gradient. To the right of the symbol is the word 'Haskell' in a bold, dark grey, sans-serif typeface.

Haskell

Jobs

- 3 month Google Summer of Code
- 3 month internship at Credit Suisse
- 8 years at Standard Chartered
- 1 year at Barclays

Expected to have to abandon Haskell, instead been programming it for a decade, and also learnt finance

Academic

- Supercompilation
 - Extension of my PhD, paper in ICFP 2010
 - Had a few PhD students follow my work
 - Mostly fizzled out (apart from Russia)
- Build systems (Shake)
 - Required by Standard Chartered
 - Papers at ICFP 2012, Haskell Symposium 2016
 - Going strong: GHC switching, companies use it
 - <http://shakebuild.com/>

Open Source

- Lots and lots of projects (too many)
 - Biggest: Hoogle, Shake, HLint, Ghcid
 - Recent: Hexml, Weeder, Profiterole
 - Contribute: Foundation, Alga
- All on GitHub <https://github.com/ndmitchell/>

Talks/Blog

- Still talk at user groups/conferences
 - 46 talks since 2004, recently 2-4 a year
 - All on <http://ndmitchell.com>
 - Where I got all the material for this talk from
- Blog with 307 posts
 - I write 4-8 posts a year (should do more)
 - 976,729 views (not including aggregator sites)
 - Initially just writing practice

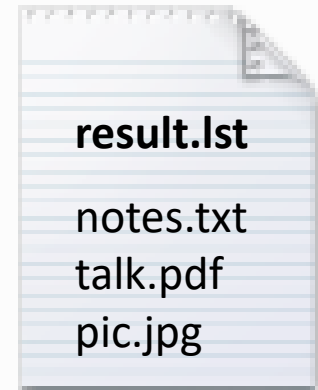
Shake overview

- Haskell EDSL for writing build systems – alternative to Make
 - Monadic dependencies
 - Unchanging dependencies
 - Non-file dependencies
 - Lots of engineering
- Vastly better for *generated* files

Shake example

```
import Development.Shake
import System.FilePath

main = shakeArgs shakeOptions $ do
  want ["result.tar"]
  "*.tar" %> \out -> do
    need [out -<.> "lst"]
    contents <- readFileLines $ out -<.> "lst"
    need contents
    cmd "tar -cf" [out] contents
```



Shake users

- **Standard Chartered** have been using Shake since 2009, 1000's of compiles per day.
- **factis research GmbH** use Shake to compile their Checkpad MED application.
- **Samplecount** using Shake since 2012, producing several open-source projects for working with Shake.
- **CovenantEyes** use Shake to build their Windows client.
- **Keystone Tower Systems** has a robotic welder with a Shake build system.
- **FP Complete** use Shake to build Docker images.
- **Genomics Plc** use Shake for the build system, their first major use of Haskell in the company.

Conclusions

- YDS was fun, resulted in my first invited talk
- Suggestions:
 - Do lots of things that interest you
 - Make some of those things good
 - Tell people what you are doing (blogs, talk etc)
 - Be open about your work
 - Start your website/blog now