Deriving a Relationship from a Single Example

Neil Mitchell

ndmitchell@gmail.com

Abstract

Given an appropriate domain specific language (DSL), it is possible to describe the relationship between Haskell data types and many generic functions, typically type-class instances. While 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 have the relationship derived.

When deriving a relationship from only one example, it is important that the derived relationship is the intended one. 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 in the DERIVE tool, where over 60% of type classes are derived from a single example.

1. Introduction

In Haskell (Peyton Jones 2003), *type classes* (Wadler and Blott 1989) 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. Many libraries define new type classes, for example Trinder et al. (1998) define the NFData type class, which reduces a value to normal form. As an example, we can define a data type to describe some computer programming languages, and provide an NFData instance:

```
data Language = Haskell [Extension] Compiler
              | Whitespace
              | Java Version

instance NFData Language where
  rnf (Haskell x1 x2) = rnf x1 `seq` rnf x2 `seq` ()
  rnf (Whitespace  ) = ()
  rnf (Java x1     ) = rnf x1 `seq` ()
```

We also need to define NFData instances for the data types Extension, Compiler and Version. Any instance of NFData follows naturally from the structure of the data type: for each constructor, all fields have seq applied, before returning ().

Writing an NFData instance for a single simple data type is easy – but for multiple complex data types the effort can be substantial. The standard solution is to express the *relationship* between a data type and it's instance. In standard tools, such as DrIFT (Winstanley 1997), the person describing a relationship must be familiar with both the representation of a data type, and various code-generation functions. The result is that specifying a relationship is not as straightforward as one might hope.

Using the techniques described in this paper, these relationships can often be automatically inferred from a single example. To define the generation of *all* NFData instances, we require an example to be given for the Sample data type defined in Figure 1:

```
data Sample α = First
              | Second α α
              | Third  α
```

Figure 1. The Sample data type.

```
instance NFData α ⇒ NFData (Sample α) where
  rnf (First      ) = ()
  rnf (Second x1 x2) = rnf x1 `seq` rnf x2 `seq` ()
  rnf (Third x1   ) = rnf x1 `seq` ()
```

The NFData instance for Sample follows the same pattern as for Language. From this example, we can infer the general relationship. However, there are many possible relationships between the Sample data type and this result – for example the function might always generate the instance for Sample, regardless of the input type. We overcome this problem by requiring the relationship to be written in a domain specific language (DSL), and that the example has certain properties (see §2). With our restrictions, we can regain predictability.

1.1 Contributions

This paper makes the following contributions:

- We describe a scheme which allows us to infer 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 practical concerns (§5).
- We outline a method for deriving a relationship in our DSL, without resorting to unguided search (§4).
- We give results (§6), including reasons why our inference fails (§6.1). In our experience, over 60% of Haskell type classes can be derived using our method.

2. Our Derivation Scheme

In this section we define a general scheme for deriving functions, which we later use to derive type-class instance generators. In general terms, a function takes an input to an output. In our case, we restrict ourselves to functions that can be described by a DSL (domain specific language). We need an apply function to serve as an interpreter for our DSL, which takes a DSL and an input and produces an output. Our scheme can be implemented in Haskell as follows:

```
data Input
data Output
data DSL

apply :: DSL → Input → Output
```

Now we turn to the derivation scheme. Given a single result of the Output type, for a particular sample Input, we wish to derive a suitable DSL. It may not be possible to derive a suitable DSL, so our derivation function must allow for the possibility of failure. Instead of producing at most one DSL, we instead produce a list of DSLs, following the lead of Wadler (1985). Once again, we can implement this in Haskell as:

```
sample :: Input
derive  :: Output → [DSL]
```

We require our scheme to have two properties – correctness (it works) and predictability (it is what the user intended). We now define both of these properties more formally, along with restrictions necessary to achieve them.

2.1 Correctness

The derivation of a particular output is correct if all derived DSLs, when applied to the sample input, produce the original output:

$$\forall\ o \in \mathsf{Output} \bullet \forall\ d \in \mathsf{derive}\ o \bullet \mathsf{apply}\ d\ \mathsf{sample} \equiv o$$

Given an existing derive' function, which does not necessarily ensure correctness, we can create a correct version by filtering out the incorrect DSLs. By applying this modification we can remove some constraints from the derive' function – either simplifying the implementation, or gaining a higher assurance of correctness.

$$\mathsf{derive}\ o = [d \mid d \leftarrow \mathsf{derive}'\ o, \mathsf{apply}\ d\ \mathsf{sample} \equiv o]$$

2.2 Predictability

A derived relationship is predictable if the user can be confident that it matches their expectations. In particular, we don't want the user to have to understand the complex derive function to gain predictability. In this section we attempt to simplify the task of defining predictable derivation schemes.

Before defining predictability, it is useful to define congruence of DSLs. We define two DSLs to be congruent ($\cong$), if for every input they produce identical results – i.e. $\mathsf{apply}\ d_1 \equiv \mathsf{apply}\ d_2$.

$$d_1 \cong d_2 \iff \forall\ i \in \mathsf{Input} \bullet \mathsf{apply}\ d_1\ i \equiv \mathsf{apply}\ d_2\ i$$

Our derive function returns a list of suitable DSLs. To ensure consistency, it is important that the DSLs are all congruent – allowing us to choose any DSL as the answer.

$$\forall\ o \in \mathsf{Output} \bullet \forall\ d_1, d_2 \in \mathsf{derive}\ o \bullet d_1 \cong d_2$$

This property is dependent on the implementation of the derive function, so is insufficient for ensuring predictability. To ensure predictability we require that all results satisfying the correctness property are congruent:

$$\forall\ d_1, d_2 \in \mathsf{DSL} \bullet \\ \mathsf{apply}\ d_1\ \mathsf{sample} \equiv \mathsf{apply}\ d_2\ \mathsf{sample} \Rightarrow d_1 \cong d_2$$

The combination of this predictability property and the correctness property implies the consistency property. It is important to note that predictability does not impose conditions on the derive function, only on the DSL and sample input. The sample input is chosen by the author of the derivation scheme, so the user is not required to understand the reasons for it's form. To ensure predictability the user may have to know some details about the DSL, but hopefully these will not be too onerous.

2.3 Summary

If the predictability property holds for the DSL and sample value, and we use the modified derive in terms of derive', then any result produced by derive will be a valid relationship. These properties allow us to write the derive function focusing on other attributes (which we discuss in §4).

To use this general scheme, we need to instantiate it to our particular problem (§3), check the predictability property (§3.4), and implement a derive function (§4).

3. Deriving Instances

In this section we apply the scheme from §2 to the problem of deriving type class instances. We let the output type be Haskell source code and the input type be a representation of algebraic data types. The DSL contains features such as list manipulation, constant values, folds and maps. We first describe each type in detail, then discuss the restrictions necessary to satisfy the predictability property.

3.1 Output

We wish to generate any sequence of Haskell declarations, where a declaration is typically a function definition or type class instance. There are several ways to represent a sequence of declarations:

String A sequence of Haskell declarations can be represented as the string of the program text. However, the lack of structure in a string poses several problems. When constructing strings it is easy to generate invalid programs, particularly given the indentation and layout requirements of Haskell. It is also hard to recover structure from the program that is likely to be useful for deriving relationships.

Pretty printing combinators Some tools such as DrIFT (Winstanley 1997) generate Haskell code using pretty printing combinators. These combinators supply more structure than strings, but the structure is linked to the presentation, rather than the meaning of constructs.

Typed abstract syntax tree (AST) The standard way of working with Haskell source code is using a typed AST – an AST where different types of fragment (i.e. declarations, expressions and patterns) are restricted to different positions within the tree. The first version of DERIVE used a typed AST, specifically Template Haskell (Sheard and Peyton Jones 2002). This approach preserves all the structure, and makes it reasonably easy to ensure the generated program is syntactically correct. By combining a typed AST with a parser and pretty printer we can convert between strings as necessary.

Untyped abstract syntax tree (AST) An untyped AST is an AST where all fragments have the same type, and types do not restrict where a fragment may be placed. The removal of types increases the number of invalid programs that can be represented – for example a declaration could occur where an expression was expected. However, by removing types we increase the similarity of the tree, in turn simplifying function that operate on the tree in a uniform manner.

For our purposes, it is clear that both strings and pretty printing combinators are unsuitable – they lack sufficient structure to implement the derive operation. The choice between a typed and untyped AST is one of safety vs simplicity. The use of a typed AST in the first version of DERIVE caused many complexities – notably the DSL was hard to represent in a well-typed manner and some functions had to be duplicated for each type. The loss of safety from using an untyped AST is not too serious, as both DSLs and ASTs are automatically generated, rather than being written by hand. Therefore, we chose to use untyped ASTs for the current version of DERIVE. We discuss possible changes to regain type safety in §8.

While we work internally with an untyped AST, existing Haskell libraries for working with ASTs use types. To allow the use of existing libraries we start from a typed AST and collapse it

to a single type, using the Scrap Your Boilerplate generic programming library (Lämmel and Peyton Jones 2003, 2004).

The use of Template Haskell in the first version of DERIVE provided a number of advantages – it is built in to GHC and can represent a large range of Haskell programs. Unfortunately, there were also a number of problems:

- Being integrated in to GHC ensures Template Haskell is available everywhere GHC is, but also means that Template Haskell cannot be upgraded separately. Users of older versions of GHC cannot take advantage of improvements to Template Haskell, and every GHC upgrade requires modifications to DERIVE.
- Template Haskell does not support new GHC extensions – they are often implemented several years later. For example, Template Haskell does not yet support view patterns.
- Template Haskell allows generated instances to be used easily by GHC compiled programs, but it makes the construction of a standalone preprocessor harder.
- If Template Haskell is also used to read the input data type (as it was in the first version of DERIVE) then only data types contained in compilable modules can be used. In particular, all necessary libraries must be compiled before an instance could be generated.
- The API of Template Haskell is relatively complex, and has some inconsistencies. In particular the Q monad caused much frustration.

We have implemented the current version of DERIVE using the haskell-src-exts library (Broberg 2009). The haskell-src-exts library is well maintained, supports most Haskell extensions [1] and operates purely as a library. We convert the typed AST of haskell-src-exts to a universal data type:

```
data Output = OString String
            | OInt Int
            | OList [Output]
            | OApp String [Output]
```

OString and OInt represent strings and integers. The OList constructor generates a list from a sequence of Output values. The expression OApp c xs represents the constructor c with fields xs. For example Just $[1,2]$ would be represented by the expression OApp "Just" [OList [OInt 1, OInt 2]].

Our Output type can represent many impossible values, for example the expression OApp "Just" [] (wrong number of fields) or OApp "Maybe" [] (not a constructor). We consider any Output value that does not represent a haskell-src-exts value to be an error. The root Output value must represent a value of type [Decl]. We can translate between our Output type and the haskell-src-exts type [Decl]:

```
toOutput   :: [Decl] → Output
fromOutput :: Output → [Decl]
```

We have implemented these functions using the SYB generics library (Lämmel and Peyton Jones 2004), specifically we have implemented the more general:

```
toOut   :: Data α ⇒ α → Output
fromOut :: Data α ⇒ Output → α
```

These functions are partial – they only succeed if the Output value represents a well-typed haskell-src-exts value. When operating on the Output type, we are working without type safety. However, provided all DSL values are constructed by derive, and that derive only constructs well-formed DSL values, our fromOutput function will be safe.

[1] Haskell-src-exts supports even more extensions than GHC!

3.2 Input

While the output type is largely dictated by the need to generate Haskell, we have more freedom with the input type. The input type represents Haskell data types, but we can choose which details to include, and thus which relationships we can represent. For example, we can include the module name in which the data type is defined, or we can omit this detail. We choose not to include the module name, which eliminates some derivations, for example the Typeable type class (Lämmel and Peyton Jones 2003).

Our Input type represents algebraic data types. We include details such as the arity of each constructor (ctorArity), the 0-based index of each constructor (ctorIndex) and the number of type variables (dataVars), but omit details such as types and record field names. Our Input type is:

```
data Input = Input
  {dataName :: String, dataVars :: Int, dataCtors :: [Ctor]}
data Ctor = Ctor
  {ctorName :: String, ctorIndex :: Int, ctorArity :: Int}
```

Values of Input for the Sample data type (Figure 1) and the Language data type (§1) are:

```
sampleType :: Input
sampleType = Input "Sample" 1
  [Ctor "First"  0 0
  ,Ctor "Second" 1 2
  ,Ctor "Third"  2 1]

languageType :: Input
languageType = Input "Language" 0
  [Ctor "Haskell"    0 2
  ,Ctor "Whitespace" 1 0
  ,Ctor "Java"       2 1]
```

The Input constructor contains the name of the data type, and the number of type variables the data type takes. For each constructor we record the name, 0-based index, and arity. These choices allow derivations to depend on the arity or index of a constructor, but not the types of a constructors arguments. In §6 we consider possible extensions to the Input type.

3.3 DSL

Our DSL type is given in Figure 2, and our apply function is given in Figure 3. The operations in the DSL are split in to six groups – we first give a high-level overview of the DSL, then return to each group in detail. The apply function is written in terms of applyEnv, where an environment is passed including the input data type, and other optional fields. Some functions in the DSL add to the environment (i.e. MapCtor), while others read from the environment (i.e. CtorName). Any operation reading a value from the environment must be nested within an operation placing that value in the environment.

Some operations require particular types – for example Reverse requires it's argument to evaluate to OList. Where possible we have annotated these restrictions in the DSL definition using comments. We have used view patterns, as implemented in GHC 6.10 (The GHC Team 2009), to perform matches on the evaluated argument DSLs. Our use of view patterns can be understood with the simple translation[2]:

[2] View-patterns and pattern-guards in GHC have different scoping behaviour, but this difference does not effect our apply function.

```
data DSL
        -- Constants
    = String String
    | Int Int
    | List [DSL]
    | App String DSL  {-[α] -}
        -- Operations
    | Concat DSL  {-[[α]] -}
    | Reverse DSL  {-[α] -}
    | ShowInt DSL  {-Int -}
        -- Fold
    | Fold DSL DSL
    | Head
    | Tail
        -- Constructors
    | MapCtor DSL
    | CtorIndex
    | CtorArity
    | CtorName
        -- Fields
    | MapField DSL
    | FieldIndex
        -- Custom
    | DataName
    | Application DSL  {-[Exp] -}
    | Instance [String] String DSL  {-[InstDecl] -}
```

Figure 2. DSL data type

```
f (Reverse (f → OList xs)) = ...
   ≡
f (Reverse v1) | OList xs ← f v1 = ...
   ≡
f (Reverse v1) | case v2 of OList {} → True; _ → False = ...
   where v2 = f v1; OList xs = v2
```

Some operations have restrictions on what their arguments must evaluate to, and what environment values must be available. It would be possible to capture many of these invariants using either phantom types (Fluet and Pucella 2002) or GADTs (Peyton Jones et al. 2006). However, for simplicity, we choose not to.

3.3.1 Constants

We include constants in our DSL, so we can lift values of Output to values of DSL. The String, Int, List operations are directly equivalent to the corresponding Output values. The App constructor is similar to OApp, but instead of taking a *list* of arguments, App takes a single argument, which must to evaluate to an OList. Requiring an OList rather than an explicit list allows the arguments to App to be constructed by operations such as Reverse or Concat.

3.3.2 Operations

The operations group consists of useful functions for manipulating lists, strings and integers. The operations have been added as required, based on functions in the Haskell Prelude. The Concat operation corresponds to concat, and concatenates either a list of lists, or a list of strings. The Reverse operation performs reverse on a list. The ShowInt operation performs show, converting an integer to a string. We have only included functions for which we have found a specific need, for example Reverse cannot be applied to a string, even though there is a sensible interpretation. We do not

```
apply :: DSL → Input → Output
apply dsl input = applyEnv dsl Env {envInput = input}

data Env = Env {envInput :: Input
               , envCtor :: Ctor
               , envField :: Int
               , envFold :: (Output, Output)}

applyEnv :: DSL → Env → Output
applyEnv dsl env@(Env input ctor field fold) = f dsl
  where
  vars = take (dataVars input) $ map (:[]) ['a' ..]

  f (Instance ctx hd body) =
    OApp "InstDecl"
        [toOut
          [ClassA (UnQual $ Ident c) [TyVar $ Ident v]
           | v ← vars, c ← ctx]
        , toOut $ UnQual $ Ident hd
        , toOut [foldl TyApp
          (TyCon $ UnQual $ Ident $ dataName input)
          [TyVar $ Ident v | v ← vars]]
        , f body]

  f (Application (f → OList xs)) =
    foldl1 (λa b → OApp "App" [a, b]) xs

  f (MapCtor dsl) = OList [applyEnv dsl env {envCtor = c}
    | c ← dataCtors input]
  f (MapField dsl) = OList [applyEnv dsl env {envField = i}
    | i ← [1 .. ctorArity ctor]]

  f DataName = OString $ dataName input
  f CtorName = OString $ ctorName ctor
  f CtorArity = OInt $ ctorArity ctor
  f CtorIndex = OInt $ ctorIndex ctor
  f FieldIndex = OInt $ field

  f Head = fst fold
  f Tail = snd fold
  f (Fold cons (f → OList xs)) =
    foldr1 (λa b → applyEnv cons env {envFold = (a, b)}) xs

  f (List xs) = OList $ map f xs
  f (Reverse (f → OList xs)) = OList $ reverse xs
  f (Concat (f → OList [])) = OList []
  f (Concat (f → OList xs)) = foldr1 g xs
      where g (OList x) (OList y) = OList (x ++ y)
            g (OString x) (OString y) = OString (x ++ y)
  f (String x) = OString x
  f (Int x) = OInt x
  f (ShowInt (f → OInt x)) = OString $ show x
  f (App x (f → OList ys)) = OApp x ys
```

Figure 3. The apply function.

provide an append or $(+\!\!+)$ operation, but one can be created from a combination of List and Concat. These operations are all simple, and would be appropriate for many DSLs.

Some examples of these operations in use are:

```
Concat (List [String "hello ", String "world"])
  ≡ OString "hello world"
Reverse (List [Int 1, Int 2, Int 3])
  ≡ OList [OInt 3, OInt 2, OInt 1]
ShowInt (Int 42) ≡ OString "42"
```

3.3.3 Fold

The Fold operation corresponds to foldr1, but can be combined with Reverse to simulate foldl1. The first argument of Fold is a function – a DSL containing Head and Tail operations. The second argument must evaluate to a list containing at least one element. If the list has exactly one element, that is the result. If there is more than one element, then Head is replaced by the first element, and Tail is replaced by a fold over the remaining elements. This can be described by:

```
Fold fn [x] = x
Fold fn (x : xs) = fn [x / Head, Fold fn xs / Tail]
```

For example, to implement concat in terms of an Append operation would be Fold (Append Head Tail) (ignoring the case of the empty list). The fold operation is more complicated than the previous operations, but may still be useful to other DSLs.

3.3.4 Constructors

To insert information from the constructors we provide MapCtor. This operation generates a list, with the argument DSL evaluated once with each different constructor in the environment. The argument to MapCtor may contain CtorName, CtorIndex and CtorArity operations, which retrieve the information associated with the constructor. CtorName produces a string, while the others produce integers. An example of MapCtor on the Sample data type is:

```
MapCtor CtorName ≡ OList
  [OString "First", OString "Second", OString "Third"]
```

3.3.5 Fields

The MapField operation is similar to MapCtor, but maps over each field within a constructor. MapField is only valid within MapCtor. Within MapField, the FieldIndex operation returns the 1-based index of the current field. While most indexing in Haskell is 0-based, fields usually correspond to variable indices (i.e. x_1), which tend to be 1-based. As an example of MapField, using Second as the constructor in the environment:

```
Concat (List [List [CtorName],
  MapField (Concat (List [String "v", ShowInt FieldIndex]))])
  ≡ ["Second", "v1", "v2"]
```

3.3.6 Custom

The final set of operations are all specific to our particular problem. The simplest operation in this group is DataName, which returns the string corresponding to the name of the data type.

The second operation is Application. The haskell-src-exts library uses binary application, where multiple applications are often nested – we provide Application to represent vector application. Vector application is often used to call constructors with arguments resulting from MapField.

The final operation is Instance, and is used to represent a common pattern of instance declaration. For example, given the type Either α β, a typical instance declaration might be:

```
instance (Show α, Ord α, Show β, Ord β) ⇒
  ShowOrd (Either α β) where ...
```

This pattern requires each type variable to be a member of a set of type classes. The resulting instance construction is:

```
Instance ["Show", "Ord"] "ShowOrd" ...
```

The Instance fields describe which classes are required for each type variable (i.e. Show and Ord in this example), what the main class is (i.e. ShowOrd), and a DSL to generate the body. To specify this pattern without a specific Instance operation would require operations over type variables – something we do not support.

3.4 Restrictions for Predictability

To ensure predictability there must be no non-congruent DSL values which give equal results when applied to the sample input. Currently this invariant is violated – consider the counterexample DataName vs String "Sample". When applied to the sample input, both will generate OString "Sample", but when applied to other data types they generate different values. To regain predictability we impose three additional restrictions on the DSL:

1. The strings Sample, First, Second and Third cannot be contained in any String construction. Therefore, in the above example, String "Sample" is invalid.
2. All instances must be constructed with Instance.
3. Within MapCtor we require that the argument DSL *must* include CtorName.

We have already seen an example of the first restriction in practice, and the second restriction has similar motivation – to avoid making something constant when it should not be. Now let us examine the third restriction, with a practical example:

```
instance Arities (Sample α) where
  arities _ = [0, 2, 1]
```

Given this instance, we could either infer the arities function always returns $[0, 2, 1]$, or it returns the arity of each constructor. While a human can spot the intention, there is a potential ambiguity. Using the second restriction, we conclude that this must represent the constant operation. To derive a version returning the arities we can write:

```
instance Arities (Sample α) where
  arities _ = [const 0 First{}
              , const 2 Second{}
              , const 1 Third{}]
```

While this code code is more verbose, any good optimiser (i.e. GHC) will generate identical code. We return to the issue of possible simplifications in §5.2.

While our DSL has forms of iteration (i.e. MapCtor), it does not have any conditional constructs such as **if** or **case**. The lack of conditionals is important for predictability – for every possible choice it would be necessary for the Sample type to choose all branches, thus increasing the size of Sample.

The restrictions in this section ensure that no fragment of output can be represented by both a constant and be parameterised by the data type. The Sample type ensures no fragment can be parameterised in multiple ways, by having different artiy/index values for some constructors – explaining why the Second constructor has arity 2, while the Third has arity 1. The restrictions in this section, along with the Sample data type, ensure predictability. We have

checked the predictability property using QuickCheck (Claessen and Hughes 2000).

4. Implementing derive

This section covers the implementation of a derive function, as described in §2. There are many ways to write a derive function, our approach is merely one option – we hope that the scheme we have described provides ample opportunity for experimentation.

Before implementing derive it is useful to think about which properties are desirable. It is not necessary to guarantee correctness (see §2.1), but our method chooses to only generate correct results. We have shown that our DSL and sample input guarantee predictability without regard to the derive function, provided we meet the restrictions in §3.4, which we obey. We want our derive function to terminate, and ideally terminate within a reasonable time bound. Finally, we would like the derive function to find an answer if one exists, i.e.:

$$\forall\ o \in \mathsf{Output}, d \in \mathsf{DSL} \bullet \mathsf{null}\ (\mathsf{derive}\ o) \Rightarrow \mathsf{apply}\ d\ \mathsf{sample} \not\equiv o$$

We were unable to implement a derive function meeting this property for our problem which performed acceptably. Our method is a trade off between runtime and success rate, with a particular desire to succeed for real-world examples.

Our derive implementation is based around a parameterised guess. Each fragment of output is related to a guess – a DSL parameterised by some aspect of the environment. For example, OString `"First"` results in the guess CtorName parameterised by the first constructor. Concretely, our central Guess type is:

```
data Guess = Guess DSL
           | GuessCtr Int DSL   -- 0-based index
           | GuessFld Int DSL   -- 1-based index

derive :: Output → [DSL]
derive o = [d | Guess d ← guess o]

guess :: Output → [Guess]
```

Applying guess (OString `"First"`) produces a guess of GuessCtr 0 CtorName. The GuessCtr and GuessFld guesses are paramterised by either constructors or fields, and can only occur within MapCtor or MapField respectively. The Guess guess is either parameterised by the entire data type, or is a constant which does not refer to the environment at all.

To generate a guess for the entire output, we start by generating guesses for each leaf node of the Output value, then work upwards combining them. If at any point we see an opportunity to apply one of our custom rules (i.e. Instance), we do so. The important considerations are how we create guesses for the leaves, how we combine guesses together, and where we apply our custom rules. We require that all generated guesses are correct, defined by:

$$\forall\ o \in \mathsf{Output} \bullet \forall\ g \in \mathsf{guess}\ o \bullet \mathsf{applyGuess}\ g \equiv o$$

```
applyGuess :: Guess → Output
applyGuess (Guess      d) = applyEnv d
  Env {envInput = sample}
applyGuess (GuessCtr i d) = applyEnv d
  Env {envInput = sample, envCtor = dataCtors sample !! i}
applyGuess (GuessFld i d) = applyEnv d
  Env {envInput = sample, envField = i}
```

4.1 Guessing Constant Leafs

4.1.1 String

To guess an OString value is simple – if it has a banned substring (i.e. Sample or one of the constructors) we generate an appropriately parameterised guess, otherwise we use the constant string. Some examples:

```
OString "hello" ≡ Guess (String "hello")
OString "Sample" ≡ Guess DataName
OString "First" ≡ GuessCtr 0 CtorName
OString "isThird" ≡ GuessCtr 2
  (Concat (List [String "is", CtorName]))
```

4.1.2 Application

The guess for an OApp is composed of two parts – the name of the constructor to apply and the list of arguments. The name of the constructor in App always exactly matches that in OApp. The arguments to App are created by applying guess to the list, and wrapping the generated DSL in App op. The guess for OApp can be written as:

```
guess (OApp op xs) = map (lift (App op)) (guess (OList xs))

lift :: (DSL → DSL) → Guess → Guess
lift f (Guess      d) = Guess      (f d)
lift f (GuessCtr i d) = GuessCtr i (f d)
lift f (GuessFld i d) = GuessFld i (f d)
```

4.1.3 Integer

Given an integer there may be several suitable guesses. An integer could be a constant, a constructor index or arity, or a field index. We can guess an OInt as follows:

```
guess (OInt i) =
  [GuessFld i  FieldIndex | i ∈ [1,2]] ++
  [GuessCtr 1 CtorIndex  | i ≡ 1] ++
  [GuessCtr 1 CtorArity  | i ≡ 2] ++
  [Guess (Int i)]
```

And some examples:

```
OInt 0 ≡ [Guess (Int 0)]
OInt 1 ≡ [GuessFld 1 FieldIndex, GuessCtr 1 CtorIndex
         , Guess (Int 1)]
OInt 2 ≡ [GuessFld 2 FieldIndex, GuessCtr 1 CtorArity
         , Guess (Int 2)]
OInt 3 ≡ [Guess (Int 3)]
```

When guessing an OInt, we never generate guesses for any constructors other than Second (represented by GuessCtr 1) – the reason is explained in §4.2.3.

4.2 Lists

Lists are the most complex values to guess. To guess a list requires a list of suitable guesses for each element, which can be collapsed into a single guess. Given a suitable collapse function we can write:

```
guess (OList xs) = mapMaybe
  (liftM fromLists ∘ collapse ∘ toLists) (mapM guess xs)

fromLists = lift Concat
toLists   = map (lift (λx → List [x]))

collapse :: [Guess] → Maybe Guess
```

The mapM function uses the list monad to generate all possible sequences of lists. The toLists function lifts each guess to a

singleton list, and the fromLists function concatenates the results – allowing adjacent guesses to be collapsed without changing the result type. The function collapse applies the following three rules, returning a Just result if any possible sequence of rule applications reduces the list to a singleton element.

4.2.1 Promotion

The promotion rule adds a parameter to a guess. We can promote Guess to either GuessFld or GuessCtr, with any parameter value. The value Guess d, can be promoted to either of GuessCtr i d or GuessFld i d, for any index i. The promotion rule does not reduce the number of elements in the list, but allows other rules to apply, in particular the conjunction rule.

4.2.2 Conjunction

If two adjacent guesses have the same parameter value, they can be combined in to one guess. For example, given GuessCtr 2 d_1 and GuessCtr 2 d_2 we produce GuessCtr 2 (Concat (List $[d_1, d_2]$)). This rule shows the importance of each guess evaluating to a list.

4.2.3 Sequence

The sequence rule introduces either MapField or MapCtor from a list of guesses. Given two adjacent guesses we can apply the rule:

```
(GuessFld 1 d1) (GuessFld 2 d2)
  | applyGuess (GuessFld 2 d1) ≡ applyGuess (GuessFld 2 d2)
  = GuessCtr 1 (MapField d1)
```

It is important that the fields are in the correct order, one of the DSL values (in this case d_1) is applicable to both problems, and the resultant guess is paramterised by the Second constructor (which has two fields). We also permit sequences in reverse order, which we generate by reversing the list before, and inserting a Reverse afterwards.

The sequence construction for fields can be extended to constructors by demanding three guesses parameterised by consecutive constructors. For constructors we only check using the DSL relating to the Second constructor, as this DSL is the only one that could have a MapField construct within it. Because we only test against the Second DSL, we can avoid generating CtorArity and CtorIndex guesses for the other constructors. We also require that when creating a MapCtor the guess contains a CtorName, to ensure the restrictions from §3.4 are met.

4.3 Folds

The addition of fold to our DSL is practically motivated – a number of real derivations require it. Currently we only attempt to find folds in a few special cases. We require folds to start with one of the following patterns:

```
OApp m [OApp m [x, op, y], op, z]
OApp m [x, op, OApp m [y, op, z]]
```

Given such a pattern, we continue down the tree finding all matching patterns of op and m. After constructing a fold we then apply guess to the residual list.

4.4 Application

As with fold, the introduction of Application is practically motivated. We replace any sequence of left-nested OApp `"App"` expressions with Application.

4.5 Instance

As per the restrictions given in §3.4, the only way of creating an Instance value as output is to use the Instance DSL operator – it is forbidden to use App `"Instance"`. Given this restriction, we translate values to Instance where they follow the pattern set out in §3.3.6.

5. Using Derived DSLs

This section discusses possible uses of a DSL after it has been derived. We start by showing how to simplify a DSL, then how to simplify the output produced by applying a DSL. Finally we give some alternative uses for a DSL, other than applying it to an input. We use the Arities type class from §3.4 as a recurring example.

5.1 DSL Simplification

We can replace a DSL with a simplified version provided the simplified version is congruent to the original. Using the apply function from Figure 3, we can determine a number of identities:

```
Concat (List (a ++ [List xs, List ys] ++ b)) ≡
  Concat (List (a ++ [List (xs ++ ys)] ++ b))
Concat (List (a ++ [List []] ++ b)) ≡ Concat (List (a ++ b))
Concat (List [x]) ≡ x
Concat (List []) ≡ List []
```

To simplify a DSL we apply these identities from left to right wherever they occur, using the Uniplate generics library (Mitchell and Runciman 2007).

Unfortunately, even after simplifying the DSL, small examples still produce complex DSLs[3]. As an example, we give an abbreviated form of the Arities DSL – the full DSL is given in Appendix A. To simplify the presentation we have omitted some haskell-src-ext nodes (i.e. Ident, UnQual, SrcLoc), and added some syntactic sugar. We have written all DSL constructors in lower-case, and used upper-case for App constructors. After these translations, the Arities DSL is:

```
⟦instance [] "Arities" ⟦InsDecl (FunBind ⟦Match
  "arities"
  ⟦PWildCard⟧
  Nothing
  (List (mapCtor (application
    ⟦Var "const", Int ctorArity, RecConstr ctorName [][]⟧
  )))
  (BDecls [][])⟧
)⟧⟧
```

5.2 Output Simplification

To obey the restrictions from §3.4 we require the addition of const applications in the Arities instance. While these const applications will be optimised away by a compiler, their removal simplifies the output for human readers. After apply, we translate the Output type to a haskell-src-exts type using the function fromOutput (§3.1). We then perform a number of simplifications, mainly simple constant folding, often using inbuilt knowledge of particular functions. Some of these simplifications can be applied by GHC (i.e. const), while others can't as they involve recursive functions (i.e. length).

We present the output simplification directly as we have implemented it, using the Uniplate generics library (Mitchell and Runciman 2007). All of the rules operate at the expression level, and each rule is correct individually. To easily express matches we introduce ($\simeq$), which converts complex expressions to strings before checking for equality.

```
simplify :: Biplate α Exp ⇒ α → α
simplify = transformBi f
```

[3] Hence the advantage of having these relationships derived, rather than writing them by hand.

```
  where
    x ≃ y = prettyPrint x ≡ y

    f (App op (List xs))
      | op ≃ "length" = Lit $ Int $ fromIntegral $ length xs
      | op ≃ "head" = head xs
    f (InfixApp (Lit (Int i)) op (Lit (Int j)))
      | op ≃ "-" = Lit $ Int $ i − j
      | op ≃ ">" = Con $ UnQual $ Ident $ show $ i > j
    f (InfixApp x op y) | op ≃ "`const`" = x
    f (App (App con x) y) | con ≃ "const" = x
    f (Paren (Var x)) = Var x
    f (Paren (Lit x)) = Lit x
    f x = x
```

Some of these simplifications could be applied directly to the DSL, for example the removal of const. However, other simplifications can't be performed until after apply has been called, such as the reduction of $(\mathsf{CtorArity} - 1)$. Currently we do not perform any of these simplifications directly to the DSL, only to the output.

5.3 DSL Usage

The obvious way to use a value of our DSL is to apply it to an input to generate an output, a haskell-src-exts AST. From an AST we can pretty-print it, and compile the resulting code. Alternatively we can use the haskell-src-meta library (Morrow 2009) to translate the output into Template Haskell, which can be integrated into GHC compiled programs.

5.3.1 Specialised Instance Generators

From a DSL we can generate a specialised instance generator, that takes an input and produces an output directly, without the interpretative step of the apply function. This construction corresponds to the first Futamura projection (Futamura 1999). For example with Arities, we could produce:

```
generateArities :: Input → [Decl]
generateArities input = [InstDecl srcLoc []
  (UnQual $ Ident "Arities")
  [foldl TyApp
    (TyCon $ UnQual $ Ident $ dataName input)
    (map (TyVar ∘ Ident) vars)]
  [InsDecl (FunBind [Match srcLoc
    (Ident "arities") [PWildCard] Nothing
    ...
    (BDecls [])])]]
  where vars = take (dataVars input) $ map (:[]) ['a'..]
```

Since our apply and fromOutput functions are both terminating, the generateArities function can be constructed as:

```
generateArities = fromOutput ∘ apply aritiesDSL
```

The fromOutput and apply functions can then be unfolded and reduced until aritiesDSL has disappeared.

The first version of DERIVE generated a string corresponding to the source code of a specialised instance generator – primarily because it lacked a complete representation of the DSL. For the new version of DERIVE we do not create specialised instance generators – the only benefit would be the removal of interpretive overhead, which we believe to be negligible.

5.3.2 Dynamic Instance Generators

In Haskell each instance is defined by some fragment of source code, and new instances cannot be constructed at runtime. However, using Haskell's reflection capabilities (Lämmel and Peyton Jones 2004), one instance can define implementations for many data types. For example, all algebraic data types can be given an Arities instance with:

```
instance Data d_type ⇒ Arities d_type where
  arities _ =
    [ const (d_ctorArity d_ctor) (d_ctorNull d_ctor :: d_type)
    | d_ctor ← d_dataCtors (undefined :: d_type)]
```

This instance declaration was generated automatically from the Arities DSL. The instance requires that the type support the Data class, allowing type information to be queried at runtime. The expression makes use of a number of library functions defined by DERIVE, namely d_dataCtors, d_ctorArity and d_ctorNull – all defined in terms of operations within the Data class.

To use a dynamic instance generator it is necessary to enable some Haskell extensions. The first is ScopedTypeVariables, which allows the d_type variable to be bound in the instance declaration head and used within instance member functions. The second extension is to allow unrestricted overlapping instances, so that custom Arities declarations can be provided for basic types. Finally, it is necessary to have Data and Typeable instances for each type of interest – these can be derived automatically using either the DERIVE tool[4] or the extension DeriveDataTypeable.

Currently the creation of dynamic instances is limited to a small number of examples, but we believe many more instances could be dealt with. However, there are some instances which cannot be produced dynamically. We have identified two cases so far:

1. DSLs which generate name bindings using information from the data type, such as the name of the constructor (i.e. isFirst), cannot be constructed.
2. If an instance makes use of a particular type class on fields, but that class does not have an instance for all types implementing Data, then the instance will not type check.

The use of dynamic instances removes the inconvenience of a separate preprocessor, but only works on a restricted set of instances. Dynamic instances increase runtime, due to the overhead of reflection and the reduction of optimisation opportunities. Previously, only a handful of classes have provided dynamic instances – the only one we are aware of is the Binary class. One reason for not providing dynamic instances is that they are complex to write – use of the SYB libraries requires an intricate combination of type-level and value-level programming. Using DERIVE many type classes could have dynamic instances created with ease, by first deriving an instance from one example and then translating the DSL.

6. Results

This section discusses the results of using our automatic derivation scheme on real examples. We first categorise the instances we are unable to derive, then share some of the tricks we have developed to succeed with more examples. For each limitation we discuss possible modifications to our system to overcome it. Finally we give timing measurements for our implementation.

6.1 Limitations of Automatic Derivation

The instance generation scheme given is not complete – there exist instances whose generator cannot be determined. The DERIVE tool (Mitchell and O'Rear 2007) generates instances for user defined data types. Of the 24 instances supported by DERIVE, 15 are derived from one example, while 9 require hand-written instance

[4] While the Data instance can be derived from a single example, alas the Typeable instance cannot (see §6.1.1), but it is still available within DERIVE.

generators. All the examples which can't be derived are due to the choices of abstraction in our Input type. We now discuss each of the pieces of information lacking from Input that result in some instances being inexpressible.

6.1.1 Module Names

Some type classes require information about the module containing a type, for example Typeable instances (Lämmel and Peyton Jones 2003) follow the pattern:

```
typename_Language = mkTyCon "ModuleName.Language"

instance Typeable Language where
  typeOf _ = mkTyConApp typename_Language []
```

The Typeable class performs runtime type comparison, so each distinct type needs a distinct string to compare, and the module name is used to disambiguate. Our Input type does not include the module name, so cannot be used to derive Typeable. It would be possible to define the string `"Module.Name"` as the module name of the sample, and treat it in a similar manner to the string `"Sample"`. However, the only instance we are aware of that requires the module name is Typeable, so we do not provide module information.

6.1.2 Infix Constructors

Some instances treat infix constructors differently, for example the Show instance on a prefix constructor is:

```
instance Show PrefixConstructor where
  show (Prefix x y) = "Foo " ++ show x ++ " " ++ show y
```

But using an infix constructor:

```
instance Show InfixConstructor where
  show (x :+: y) = show x ++ " :+: " ++ show y
```

Our Input type does not express whether a constructor is infix or prefix, so cannot choose the appropriate behaviour. The loss of infix information mainly effects instances which display information to the user, i.e. Show and pretty printing (Hughes 1995). For most type classes, the infix information is not used, and infix constructors can be bracketed and treated as prefix $(:+:)$. To deal with infix constructors would require an infix constructor added to the Sample data type, and modifications to the DSL to allow different results to be generated depending on infix information. These changes would pose difficulties to predictability and require all example instances to have at least one additional case defined – we do not consider this a worthwhile trade off for a small number of additional instances.

6.1.3 Record-based definitions

Haskell provides records, which allow some fields to be labelled. Some operations make use of the record fields within a data type, for example using the data type:

```
data Computer = Desktop {memory :: Int}
              | Laptop  {memory :: Int, weight :: Int}
```

It is easy to write the definition:

```
hasWeight Desktop{} = False
hasWeight Laptop{}  = True
```

Where hasWeight returns True if the weight selector is valid for that constructor, and False if weight $x \equiv \bot$. Unfortunately our Input type does not contain information about records, so cannot express this definition. There are only a few type classes which exhibit label specific behaviour, such as Show which outputs the field name if present.

Record fields are not present in our Sample type, but could be added. The difficulty is that Haskell allows for one field name to be shared by multiple constructors, and allows some constructors to have field names while others do not. This flexibility results in a massive number of possible combinations, and so a Sample type with sufficient generality would require many constructors. Allowing records would be more feasible for a language such as F#, where records contain only one constructor and all fields must be named.

6.1.4 Type-based definitions

Our Sample data type has a simple type structure, and our DSL does not allow decisions to be made on the basis of type – these restrictions means some type classes can't be defined. For example, a Monoid instance processes items of the same type using mappend, but items of a different type using mempty. Several other type classes require type specific behaviour, including Functor, Traversable and Uniplate.

The lack type information has other consequences. For example, we can write the definition:

```
fromFirst  (First    )    = const First{}    $ tuple0
fromSecond (Second x1 x2) = const Second{}   $ tuple2 x1 x2
fromThird  (Third  x1   ) = const Third{}    $ tuple1 x1
```

This function returns the elements contained within a constructor, generalising operations such fromJust, and has seen extensive use in the Yhc compiler (The Yhc Team 2007). When compiled with GHC this code generates a warning that no top-level type signatures have been given. These type signatures can be inferred, but the Haddock documentation tool (Marlow 2002) won't include functions lacking type signatures. Without type information in Input, we can't generate appropriate type signatures.

We see no easy way to include type information in our derivation scheme – types have too much variety, and different type classes make use of different type information. It may be possible to identify some restricted type information that could be used for a subset of type-based instances, but we have not done so.

6.2 Practical Experiences

This section describes our experiences of specifying instances in a form suitable for derivation. Ideally, we would write all instances in a natural way, but sometimes we need to make concessions to our derivation algorithm. Using the techniques given here, it seems possible to write most instances which are based on information included in the Input type.

6.2.1 Brackets Matter

The original DERIVE program used Template Haskell, which include brackets in the abstract syntax tree. For example, the expressions (First) and First are considered equal. However, using haskell-src-exts, brackets are explicit and care must be taken to ensure every constructor has the same level of bracketing. Examples of otherwise unnecessary brackets can be seen in §6.1.4, where the constructor First is bracketed. Currently some redundant brackets are removed by the transformations described in §5.2.

6.2.2 Variable Naming

When naming variables it is important that a sequence of variables follow a pattern. For example, in §6.1.4 we use Second x_1 x_2, rather than Second x y. By naming variables with consecutive numbers we are able to derive the fields correctly.

6.2.3 Explicit Fold Base-Case

When performing a fold, it is important to explicitly include the base-case. In the introductory example of NFData the Second

alternative is specified as rnf x_1 `seq` rnf x_2 `seq` (), however we can show that:

$\forall\, x \bullet \mathsf{rnf}\ x\ \texttt{`seq`}\ () \equiv \mathsf{rnf}\ x$

Therefore we could write the Second alternative more compactly as rnf x_1 `seq` rnf x_2. However, doing so would mean there was not one consistent pattern suitable for all constructors, and the derivation would fail. In general, when considering folds, the base-case should always be written explicitly.

6.2.4 Empty Record Construction

One useful feature of Haskell records is the empty record construction. The expression Second{} creates the value Second $\bot$ $\bot$. This expression is useful for generating constructors to pass as the second argument to const[5], for some generic programming operations, and for values that are lazily evaluated. The pattern Second{} matches all Second constructors, regardless of their fields.

6.2.5 Constructor Count

Some instances aren't inductive – for example Binary instances require a tag indicating which constructor has been stored, but only if there is more than one constructor. This pattern can be written as:

```
instance Binary α ⇒ Binary (Sample α) where
  put x = case x of
    First          → do putTag 0
    Second x1 x2   → do putTag 1; put x1; put x2
    Third  x1      → do putTag 2; put x1
    where
      useTag = length [First{}, Second{}, Third{}] > 1
      putTag = when useTag ∘ putWord8

  get = do
    i ← getTag
    case i of
      0 → do return (First)
      1 → do x1 ← get; x2 ← get; return (Second x1 x2)
      2 → do x1 ← get; return (Third x1)
      _ → error "Corrupted binary data for Sample"
    where
      useTag = length [First{}, Second{}, Third{}] > 1
      getTag = if useTag then getWord8 else return 0
```

The value length [First{}, Second{}, Third{}] is used to compute the number of constructors in the data type, which can be tested to get the correct behaviour. This pattern is used in other classes, for example Enum and Arbitrary. Using the simplifications from §5.2 we can remove the test and produce code specialised to the number of constructors.

The pattern for the number of constructors is useful, but seems a little verbose. In the first version of DERIVE the constructor count was guessed from the number 3. Unfortunately, the inclusion of this guess breaks the restrictions we have imposed for predictability. Another way of simplifying this pattern would be to introduce a meta function ctorCount, which expanded to the number of constructors. This solution would mean inputs were not real example instances, and would require users to learn part of the DSL – something we have tried to avoid. In the end, we simply accept that the constructor count is slightly verbose.

6.3 Timing Properties

We have implemented the methods described in this paper, and have used them to guess all 15 examples referred to in §6.1, along with 2 additional test cases. For each example we perform the following steps:

1. We derive the DSL from an example.
2. We apply the DSL (without output optimisation) to the Sample data type and check it matches the input example.
3. We apply the DSL to three other data types, namely lists, the eight element tuple and the expression type from the Yhc Core library (Golubovsky et al. 2007).

To perform all steps for 17 examples takes 0.3 seconds when compiled with GHC -O0 on a laptop with a 2GHz CPU and 1Gb of RAM. We consider these times to be more than adequate, so have not carried out further experiments or investigated additional optimisations.

7. Related Work

An earlier version of the DERIVE tool was presented in a previous paper (Mitchell 2007). The previous work described only the derivation algorithm. There was no intermediate DSL, and no predictability. Given a single example the tool could produce multiple different answers, and would always use the first generated – not always corresponding to the users intention. This paper presents a much more general scheme, along with many improvements to the previous work. Some of the areas of future work in the previous paper have been tackled, such as dynamic instances (see §5.3.2). Crucial improvements have been made to the derivation algorithm, particularly when dealing with lists.

We are unaware of any work (other than our own) that attempts to automatically derive Haskell type classes. Therefore we split the remaining related work in to two sections – that which explicitly defines instance relationships, and that which tries to derive relationships.

7.1 Specifying Type Classes

From an end-user perspective, the DrIFT tool (Winstanley 1997) is similar to DERIVE – both take data types and produce associated instances. To add a type-class to DrIFT the programmer manually writes a translation from input types to Haskell source code, using pretty-printing combinators. There is no automatic derivation of instance generators, and no underlying DSL. As a result, it is substantially easier to add generators which can be derived from one example to DERIVE.

Another mechanism for specifying type classes is to use generic type classes (Hinze and Peyton Jones 2000), a language extension supported by GHC. A programmer can write default instances for type classes in terms of the structure of a type, using unit, products and sums. There are many restrictions on such classes, including restrictions on the type of instance methods and the structure of the input type. Using the abstraction of products and sums, it is impossible to represent many instances such as those dealing with records or containing type specific behaviour.

7.2 Deriving Relationships

The purpose of our work is to find a pattern, which is generalised to other situations. Genetic algorithms (Goldberg 1989) are often used to automatically find patterns in a data set. Genetic algorithms work by evolving a hypothesis (a gene sequence) which is tested against a sample problem. While genetic algorithms are good for search, they usually use a heuristic to measure closeness – so lack the exactness of our approach.

There is much research on learning relationships from a collection of input/output pairs, often using only positive examples (Kitzelmann 2007). Some work tackles this problem using exhaus-

[5] `"Second"` would also work, but the use of a string feels too unpleasant.

tive search (Katayama 2008), a technique that could possibly replace our derive function. Instead of using specific examples, some work generalises a set of non-recursive equations into a recursive form (Kitzelmann and Schmid 2006; Kitzelmann 2008). All these pieces of work require a set of input/output examples, in contrast to our method that requires only one output for a specific input.

The closest work we are aware of is that of the theorem proving community. Induction is a very common tactic for writing proofs, and well supported in systems such as HOL Light (Harrison 1996). Typically the user must suggest the use of induction, which the system checks for validity. Automatic inference of an induction argument has been tried (Mintchev 1994), but is rarely successful. However, these systems all work from one positive example, attempting to determine a reasonably restricted pattern.

8. Conclusions and Future Work

We have presented a scheme for deriving a DSL from one example, which we have used to automatically derive instance generators for Haskell type classes. Our technique has been implemented in the DERIVE tool, where 60% of instance generators are specified by example. The ease of creating new instances has enabled several users to contribute instance generators. The DERIVE tool can be downloaded from Hackage[6], and we encourage interested users to try it out.

One of the key strengths of our derivation scheme is that concerns of correctness and predictability are separated from the main derivation function. Correctness is easy to test for, so incorrect derivations can simply be discarded. Predictability is a property of the DSL and sample input, and can be determined in isolation from the derivation function. The derivation function merely needs to take a best guess at what derivation might work, allowing greater freedom to experiment.

We see several lines of future work:

- By deriving an explicit DSL, we can reuse the DSL for other purposes. We have already shown the creation of dynamic instances in §5.3.2, but there are other possible uses. A DSL could be used to prove properties, for example that all Eq instances are reflexive, or that put/get in Binary are inverses. Another use might be to generate human readable documentation of an instance. We suspect there are many other uses.
- The Sample data type (Figure 1) allows many instances to be inferred – but more would be desirable. We have discussed possible extensions in §6.1, but none seems to offer compelling benefits. An alternative approach would be to introduce new sample data types with features specifically for certain types of definition. Care would have to be taken that these definitions still preserved predictability, and did not substantially increase the complexity of writing examples.
- While our scheme is implemented in a typed language, most of the actual DSL operations work upon a universal data type with runtime type checking – essentially a dynamically typed language. In order to preserve types throughout we could make use of GADTs (Peyton Jones et al. 2006).
- We have implemented our scheme specifically for instance generators in Haskell, but the same scheme could be applied to other computer languages and other situations. One possible target would be F#, where there are interfaces instead of type classes. Another target could be an object-orientated language, where design patterns (Gamma et al. 1995) are popular.

[6] `http://hackage.haskell.org/package/derive`

Computers are ideally suited to applying a relationship using new parameters, but specifying these relationships can be complex and error prone. By specifying a single example, instead of the relationship, a user can focus on what they care about, rather than the mechanism by which it is implemented.

Acknowledgements

Thanks to Stefan O'Rear for help writing the first version of the DERIVE tool. Thanks to Niklas Broberg for the excellent haskell-src-exts library. Thanks to Hongseok Yang for fruitful discussions on the original instance generation work. Thanks to Mike Dodds for constructive criticism on earlier drafts.

A. Arities DSL

This section presents the full Arities DSL, a simplified version of which is shown in §5.1.

```
List [Instance [] "Arities" (List [App "InsDecl" (
  List [App "FunBind" (List [List [
    App "Match" (List
      [App "Ident" (List [String "arities"])
      , List [App "PWildCard" (List [])]
      , App "Nothing" (List [])
      , App "UnGuardedRhs" (List [App "List" (List [
        MapCtor (Application (List
          [App "Var" (List [App "UnQual" (List [
            App "Ident" (List [String "const"])])])
          , App "Lit" (List [App "Int" (List [CtorArity])])
          , App "RecConstr" (List [App "UnQual" (List [
            App "Ident" (List [CtorName])]), List []])]
        ))
      ])])
      , App "BDecls" (List [List []])]
    )
  ])]
)])]
```

References

Niklas Broberg. haskell-src-exts. `http://www.cs.chalmers.se/~d00nibro/haskell-src-exts/`, 2009.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. ICFP '00*, pages 268–279. ACM Press, 2000.

Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *Proc. TCS '02*, pages 448–460, Deventer, The Netherlands, 2002.

Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, January 1989.

Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core – from Haskell to Core. *The Monad.Reader*, 1(7): 45–61, April 2007.

John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proc. Formal Methods in*

Computer-Aided Design, FMCAD'96, volume 1166 of *LNCS*, pages 265–269. Spinger-Verlag, 1996.

Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proc Haskell Workshop 2000*. Elsevier Science, September 2000.

John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925, 1995.

Susumu Katayama. Efficient exhaustive generation of functional programs using Monte-Carlo search with iterative deepening. In *PRICAI 2008: Trends in Artificial Intelligence*, volume 5351, pages 199–210. LNCS, 2008.

Emanuel Kitzelmann. Data-driven induction of recursive functions from input/output-examples. In *Proceedings of the Workshop on Approaches and Applications of Inductive Progamming (AAIP'07)*, pages 15–26, 2007.

Emanuel Kitzelmann. Data-driven induction of functional programs. In *Proc. ECAI 2008*. IOS Press, July 2008.

Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs – An explanation based generalization approach. *Journal of Machine Learning Research*, 7(Feb):429–454, 2006.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. TLDI '03*, pages 26–37. ACM Press, March 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proc. ICFP '04*, pages 244–255. ACM Press, 2004.

Simon Marlow. Haddock, a Haskell documentation tool. In *Proc. Haskell Workshop 2002*, Pittsburgh Pennsylvania, USA, October 2002. ACM Press.

Sava Mintchev. Mechanized reasoning about functional programs. In K. Hammond, D. N. Turner, and P. M. Sansom, editors, *Functional Programming*, pages 151–166. Springer, Berlin, Heidelberg, 1994.

Neil Mitchell. Deriving generic functions by example. In Jan Tobias Mühlberg and Juan Ignacio Perna, editors, *Proc. York Doctoral Symposium 2007*, pages 55–62. Tech. Report YCS-2007-421, University of York, October 2007.

Neil Mitchell and Stefan O'Rear. Derive - project home page. `http://community.haskell.org/~ndm/derive/`, March 2007.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proc. Haskell '07*, pages 49–60. ACM, 2007.

Matt Morrow. haskell-src-meta. `http://hackage.haskell.org/package/haskell-src-meta`, 2009.

Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proc. ICFP '06*, pages 50–61. ACM Press, 2006.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proc. Haskell Workshop '02*, pages 1–16. ACM Press, 2002.

The GHC Team. The GHC compiler, version 6.10.3. `http://www.haskell.org/ghc/`, May 2009.

The Yhc Team. The York Haskell Compiler – user manual. `http://www.haskell.org/haskellwiki/Yhc`, February 2007.

Philip Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon Peyton Jones. Algorithm + strategy = parallelism. *JFP*, 8(1):23–60, January 1998.

Philip Wadler. How to replace failure by a list of successes. In *Proc. FPCA '85*, pages 113–128. Springer-Verlag New York, Inc., 1985.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL '89*, pages 60–76. ACM Press, 1989.

Noel Winstanley. Reflections on instance derivation. In *1997 Glasgow Workshop on Functional Programming*. BCS Workshops in Computer Science, September 1997.