DrIFT User Guide

version 2.2.3
10 February 2008

Noel Winstanley
nww@dcs.gla.ac.uk
John Meacham
john@foo.net

1 Introduction

This is a guide to using DrIFT, a type sensitive preprocessor for Haskell 98.

DrIFT is a tool which parses a Haskell module for structures (data & newtype declarations) and commands. These commands cause rules to be fired on the parsed data, generating new code which is then appended to the bottom of the input file, or redirected to another. These rules are expressed as Haskell code, and it is intended that the user can add new rules as required.

DrIFT is written in pure Haskell 98, however code it generates is free to make use of extensions when appropriate. DrIFT is currently tested against hugs and ghc.

1.1 So, What Does DrIFT do?

DrIFT allows derivation of instances for classes that aren't supported by the standard compilers. In addition, instances can be produced in separate modules to that containing the type declaration. This allows instances to be derived for a type after the original module has been compiled. As a bonus, simple utility functions can also be produced for types.

1.2 Features

- DrIFT comes with a set of rules to produce instances for all derivable classes given in the Prelude. There's a rule to produce instances of NFData (the original motivation of all this), and rules for utility functions on types also. The DrIFT implementation is also regularly updated with rules submitted by users.
- Code is generated using pretty-printing combinators. This means that the output is (fairly) well formatted, and easy on the human eye.
- Effort has been made to make the rule interface as easy to use as possible. This is to allow users to add rules to generate code specific to their own projects. As the rules are written in Haskell themselves, the user doesn't have to learn a new language syntax, and can use all Haskell's features.

Currently supported derivations are the following. This list is obtainable by running `DrIFT -l`.

```
Binary:
   Binary            efficient binary encoding of terms
   GhcBinary         byte sized binary encoding of terms
Debugging:
   Observable        HOOD observable
General:
   NFData            provides 'rnf' to reduce to normal form (deepSeq)
   Typeable          derive Typeable for Dynamic
Generics:
   FunctorM          derive reasonable fmapM implementation
   HFoldable         Strafunski hfoldr
   Monoid            derive reasonable Data.Monoid implementation
   RMapM             derive reasonable rmapM implementation
```

```
   Term               Strafunski representation via Dynamic
Prelude:
   Bounded
   Enum
   Eq
   Ord
   Read
   Show
Representation:
   ATermConvertible   encode terms in the ATerm format
   Haskell2Xml        encode terms as XML (HaXml<=1.13)
   XmlContent         encode terms as XML (HaXml>=1.14)
Utility:
   Parse              parse values back from standard 'Show'
   Query              provide a QueryFoo class with 'is', 'has',
                        'from', and 'get' routines
   from               provides fromFoo for each constructor
   get                for label 'foo' provide foo_g to get it
   has                hasfoo for record types
   is                 provides isFoo for each constructor
   test               output raw data for testing
   un                 provides unFoo for unary constructors
   update             for label 'foo' provides 'foo_u' to update it
                        and foo_s to set it
```

1.3 Why Do We Need DrIFT?

The original motivation for DrIFT came from reading one of the Glasgow Parallel Haskell papers on Strategies. Strategies require producing instances of a class which reduces to normal form (called NFData). It was commented that it was a shame that instances of NFData couldn't be automatically derived; the rules to generate the instances are simple, and adding instances by hand is tiresome. Many classes' instances follow simple patterns. This is what makes coding up instances so tedious: there's no thought involved!

The idea to extend DrIFT to work on imported types came from a discussion of the Haskell mailing list, arising from a point made by Olaf Chitil :

> Why is the automatic derivation of instances for some standard classes linked to data and newtype declarations? It happened already several times to me that I needed a standard instance of a data type that I imported from a module that did not provide that instance and which I did not want to change (a library; GHC, which I mainly want to extend by further modules, not spread changes over 250 modules). When declaring a new data type one normally avoids deriving (currently) unneeded instances, because it costs program code (and maybe one even wants to enable the user of the module to define his own instances).

The third feature of DrIFT, providing utility functions to manipulate new types, especially records was caused by finding oneself writing the same sort of code over and over again. These functions couldn't be captured in a class, but have a similar form for each type they are defined on. A thread on the Haskell mailing list made a related point: untagging and manipulating newtypes was more cumbersome than it should be.

1.4 An Example

Here's an example of what how DrIFT is used. This Haskell module contains commands to the DrIFT preprocessor. These are annotated with `{-! ... !-}`. After processing with DrIFT the generated code is glued on the bottom of the file, beneath a marker indicating where the new code starts. The machine generated code is quite long, and would really have been a drudge to type in by hand.

1.4.1 Source Code

```
-- example script for DrIFT

module Example where
import Foo
{-!for Foo derive :  Read,NFData !-} -- apply rules to imported type

{-! global : is !-} -- global to this module
{-!for Data derive : update,Show,Read!-} -- stand alone comand syntax

{-!for Maybe derive : NFData !-} -- apply rules to prelude type

data Data = D {name :: Name,
constraints :: [(Class,Var)],
vars :: [Var],
body :: [(Constructor,[(Name,Type)])],
derive :: [Class],
statement :: Statement}

data Statement = DataStmt | NewTypeStmt
        deriving Eq {-!derive : Ord,Show,Read !-} -- abbreviated syntax
```

1.4.2 After processing with DrIFT

```
module Example where
import Foo
{-!for Foo derive : Read,NFData !-} -- apply rules to imported type

{-! global : is !-} -- global to this module
{-!for Data derive : update,Show,Read!-} -- stand alone comand syntax

{-!for Maybe derive : NFData !-} -- apply rules to prelude type

data Data = D {name :: Name,
                        constraints :: [(Class,Var)],
                        vars :: [Var],
```

```
                        body :: [(Constructor,[(Name,Type)])],
                        derive :: [Class],
                        statement :: Statement}

data Statement = DataStmt | NewTypeStmt
        deriving Eq {-!derive : Ord,Show,Read !-}

{-* Generated by DrIFT-v1.0 : Look, but Don't Touch. *-}
isD (D aa ab ac ad ae af) = True
isD _ = False

instance Ord Statement where
    compare DataStmt (DataStmt) = EQ
    compare DataStmt (NewTypeStmt) = LT
    compare NewTypeStmt (DataStmt) = GT
    compare NewTypeStmt (NewTypeStmt) = EQ

instance Show Statement where
    showsPrec d (DataStmt) = showString "DataStmt"
    showsPrec d (NewTypeStmt) = showString "NewTypeStmt"

instance Read Statement where
    readsPrec d input =
              (\ inp -> [((DataStmt) , rest)
                          | ("DataStmt" , rest) <- lex inp])
              input
              ++
              (\ inp ->
               [((NewTypeStmt) , rest)
                  | ("NewTypeStmt" , rest) <- lex inp])
              input

isDataStmt (DataStmt) = True
isDataStmt _ = False
isNewTypeStmt (NewTypeStmt) = True
isNewTypeStmt _ = False

instance (NFData a) => NFData (Maybe a) where
    rnf (Just aa) = rnf aa
    rnf (Nothing) = ()

body_u f r@D{body} = r{body = f body}
constraints_u f r@D{constraints} = r{constraints = f constraints}
derive_u f r@D{derive} = r{derive = f derive}
name_u f r@D{name} = r{name = f name}
statement_u f r@D{statement} = r{statement = f statement}
vars_u f r@D{vars} = r{vars = f vars}
```

```
body_s v =  body_u  (const v)
constraints_s v =  constraints_u  (const v)
derive_s v =  derive_u  (const v)
name_s v =  name_u  (const v)
statement_s v =  statement_u  (const v)
vars_s v =  vars_u  (const v)

instance Show Data where
    showsPrec d (D aa ab ac ad ae af) = showParen (d >= 10)
              (showString "D" . showChar '{' .
               showString "name" . showChar '=' . showsPrec 10 aa
               . showChar ',' .
               showString "constraints" . showChar '=' . showsPrec 10 ab
               . showChar ',' .
               showString "vars" . showChar '=' . showsPrec 10 ac
               . showChar ',' .
               showString "body" . showChar '=' . showsPrec 10 ad
               . showChar ',' .
               showString "derive" . showChar '=' . showsPrec 10 ae
               . showChar ',' .
               showString "statement" . showChar '=' . showsPrec 10 af
               . showChar '}')

instance Read Data where
    readsPrec d input =
          readParen (d > 9)
           (\ inp ->
            [((D aa ab ac ad ae af) , rest) | ("D" , inp) <- lex inp ,
             ("{" , inp) <- lex inp , ("name" , inp) <- lex inp ,
             ("=" , inp) <- lex inp , (aa , inp) <- readsPrec 10 inp ,
             ("," , inp) <- lex inp , ("constraints" , inp) <- lex inp ,
             ("=" , inp) <- lex inp , (ab , inp) <- readsPrec 10 inp ,
             ("," , inp) <- lex inp , ("vars" , inp) <- lex inp ,
             ("=" , inp) <- lex inp , (ac , inp) <- readsPrec 10 inp ,
             ("," , inp) <- lex inp , ("body" , inp) <- lex inp ,
             ("=" , inp) <- lex inp , (ad , inp) <- readsPrec 10 inp ,
             ("," , inp) <- lex inp , ("derive" , inp) <- lex inp ,
             ("=" , inp) <- lex inp , (ae , inp) <- readsPrec 10 inp ,
             ("," , inp) <- lex inp , ("statement" , inp) <- lex inp ,
             ("=" , inp) <- lex inp , (af , inp) <- readsPrec 10 inp ,
             ("}" , rest) <- lex inp])
           input

--  Imported from other files :-

instance Read Foo where
    readsPrec d input =
```

```
                  (\ inp -> [((Foo) , rest)
                              | ("Foo" , rest) <- lex inp]) input
                  ++
                  (\ inp -> [((Bar) , rest)
                              | ("Bar" , rest) <- lex inp]) input
                  ++
                  (\ inp -> [((Bub) , rest)
                              | ("Bub" , rest) <- lex inp]) input

instance NFData Foo where
    rnf (Foo) = ()
    rnf (Bar) = ()
    rnf (Bub) = ()
```

2 User Guide

This chapter assumes that DrIFT has already been installed and the environment variables set up. The installation is handled in Chapter 5 [Installation], page 14.

Briefly, the way DrIFT works is

1. parse the input file, looking for commands and data & newtype statements.
2. generate code by executing the commands, which apply rules to types.
3. if any commands remain unexecuted, this means the types aren't declared in this module, so DrIFT searches for them in imported modules.
4. append the generated code to the bottom of the file (overwriting any previously generated code)

Rules can be applied to any types defined using a `data` or `newtype` statement. Rules can't be applied to types defined using `type`, as this only produces a synonym for a type. **Don't try to use rules on type synonyms.**

2.1 The Command Line

DrIFT processes standard Haskell scripts (suffix '`.hs`') and literate scripts (suffix '`.lhs`'). Currently, only literate code using `>` is accepted: DrIFT doesn't understand the TEX style of literate programming using `\begin{code}`.

If you've compiled up an executable from the source code (or are using Runhugs) to run DrIFT over a file type :-

`DrIFT` *`filename`*

Alternatively, for Hugs, use :-

`runhugs DrIFT` *`filename`* (run DrIFT over filename)

2.2 Command Syntax

Commands to DrIFT are entered into Haskell code in the form of *annotations*. DrIFT's annotations start with `{-!` and finish with `!-}`. (This is so they don't clash with the compiler annotations given to GHC or HBC). There are three forms of command.

- **Stand–Alone Command** (syntax : `{-! for` *`type`* `derive :` *`rule1`*`,`*`rule2`*`,... !-}`) This is the basic form of DrIFT command. It asks DrIFT to apply the listed rules to the specified type. If the type is parameterised, e.g. `Maybe a`, just enter the type name into the command, omitting any type variables. DrIFT assumes that types given are currently in scope, and will first search the current module. If it fails to find a matching type definition, the prelude and any imported modules are also searched. This is the only command which allows code to be generated for a type defined in another module.
- **Abbreviated Command** (syntax : `{-! derive :`*`rule1`*`,`*`rule2`*`,... !-}`) This command is appended to the end of a `data` or `newtype` definition, after the deriving clause, if present. It applies the listed rules to the type it is attached to.
- **Global Command** (syntax : `{-! global :`*`rule1`*`,`*`rule2`*`,... !-}` This command applies the listed rules to all types defined within the module. Note that this command doesn't cause code to be generated for types imported from other modules.

For an example of these commands in use, See Section 1.4 [An Example], page 3.

2.2.1 Notes on Using Commands

- The stand-alone and global commands should be entered on a line by themselves, starting in the first column, (as with other top-level declarations, such as `infix`, `import`,`newtype`). It doesn't matter what position they occur within the module.
- In a literate file, all commands should be entered on a 'code' line (one starting with `>`).
- Commands may be commented out by using `--` and `{- .. -}` in the usual way.
- If two commands apply the same rule to a type, then two sets of identical code will be produced. This will cause a 'multiple definition' error when the processed module is compiled/interpreted. **Don't do it!**

2.3 Emacs DrIFT mode

For Emacs fans, Hans W Loidl `hwloidl@dcs.gla.ac.uk` has written a script which allows DrIFT to be run within a buffer.

The commands available are

- `M-x hwl-derive`, `C-c d d` runs DrIFT over the current buffer, and then updates the buffer.
- `M-x hwl-derive-insert-standalone`, `C-c d s` inserts a template for a standalone command into the current buffer at the cursor position.
- `M-x hwl-derive-insert-local`, `C-c d l` inserts a template for an abbreviated command.
- `M-x hwl-derive-insert-global`, `C-c d g` inserts a template for a global command

In 'hugs-mode' these functions are also available vie a menu item in the hugs menu.

3 Standard Rules

Heres a listing of the rules that come pre-defined with DrIFT. If you want a more detailed idea of how they work, their definitions are in the file '`StandardRules.hs`', and are (fairly) well documented. In the following list the **highlighted** text is the name of the rule, as used in commands. The naming convention for rules is names starting with a capital generate an instance for the class of the same name. Sets of functions are generated by a name beginning with a lower case letter.

3.1 Prelude Classes

The classes **Eq**, **Ord**, **Enum**, **Show**, **Read** & **Bounded** are described in the Haskell report as being derivable; DrIFT provides rules for all these.

3.2 Other Classes

Originally, **NFData** (for Normal Form evaluation strategies) was the only other class to have a rule. But now, there are rules for many more classes from 3rd-party libraries, e.g. **XmlContent** from HaXml, **Binary** from nhc98, **Term** from Strafunski, **FunctorM** for Generics, **Observable** for HOOD debugging, **Typeable** for dynamics, and so on. For a full list, use the `--list` command-line option.

3.3 Utilities

- **un** attempts to make newtypes a little nicer to use by providing an untagging function. This rule can only be used on types defined using `newtype`.

  For a type `newtype Foo a = F a`,

  un produces the function `unFoo :: Foo a -> a`.
- **is** produces predicates that indicate the presence of a constructor. This is only useful for multi-constructor datatypes (obviously).

  For a type `data Foo = Bar | Bub`, **is** generates

  `isBar :: Foo -> Bool` and `isBub :: Foo -> Bool`.
- **has** produces predicates that indicate the presence of a label. This can only be used with types where at least one of the constructors is a labelled record. Note that labels can be shared between constructors of the same type.

  For a type `data Foo a = F{bar :: a,bub :: Int}` **has** generates

  `hasbar :: Foo a-> Bool` and `hasbub :: Foo a -> Bool`.
- **update** produces functions that update fields within a record type. This rule can only be used with a type where at least on of the constructors is a labelled record.

  For a type `data Foo a = F{bar :: a, bub ::Int}` **update** generates

  `bar_u :: (a -> a) -> Foo a -> Foo a` and

  `bub_u :: (Int -> Int) -> Foo a -> Foo a` which apply a function to a field of a record, and then return the updated record. If the value does not have the given field then the value is returned unchanged.

  `bar_s :: a -> Foo a -> Foo a` and `bub_s ::Int -> Foo a -> Foo a` are also generated, and are used to set the value of a field in a record.

- **test** dumps the parsed representation of a datatype to the output. This is be useful for debugging new rules, as the user can see what information is stored about a particular type.

4 Rolling Your Own

Programmers who only wish to use the pre-defined rules in DrIFT don't need to read or understand the following section. However, as well as using the supplied rules, users are encouraged to add their own. There is a stub module '`UserRules.hs`' in the source, to which rules can be added.

If a compiled version of DrIFT is being used, the program will then have to be recompiled before the new rules can be used. However, if the Runhugs standalone interpreter is used, this is not necessary. Due to the way Runhugs searches for modules to load, a user may have many copies of the UserRules module. The UserRules module in the current directory will be loaded first. If that is not present, then the `HUGSPATH` environment variable is searched for the module. So it is possible to have a default UserRules module, and specialised ones for particular projects.

4.1 The Basic Idea

A rule is a tuple containing a string and a function. The string is the name of the rule, and is used in commands in an input file. The function maps between the abstract representation of a datatype and text to be output (A sort of un-parser, if you like). The best way to understand this is to have a look at the existing rules in '`StandardRules.hs`'. This module is quite well documented.

4.2 How is a Type Represented?

A type is represented within DrIFT using the following data definition.

```
>data Statement = DataStmt | NewTypeStmt deriving (Eq,Show)

>data Data = D {       name :: Name,           -- type name
>                      constraints :: [(Class,Var)],
>                      vars :: [Var],          -- Parameters
>                      body :: [Body],
>                      derives :: [Class],     -- derived classes
>                      statement :: Statement}
>           | Directive
>           | TypeName Name deriving (Eq,Show)

>type Name = String
>type Var = String
>type Class = String
```

A `Data` type represents one parsed `data` or `newtype` statement. These are held in a `D` constructor record (the `Directive` and `TypeName` constructors are just used internally by DrIFT). We'll now examine each of the fields in turn.

- `name` holds the name of the new datatype as a string.
- `constraints` list the type constraints for the type variables of the new type. e.g. for `data (Eq a) => Foo a = F a`, the value of `constraints` would be `[("Eq","a")]`.
- `vars` contains a list of the type variables in the type. For the previous example, this would simply be `["a"]` .

- `body` is a list of the constructors of the type, and the information associated with them. We'll come back to this in a moment.
- `derives` lists the classes that the type an instance of though using the `deriving` clause.
- `statement` indicates whether the type was declared using a `newtype` or `data` statement

4.2.1 The Body

```
>data Body = Body { constructor :: Constructor,
>                   labels :: [Name],
>                   types :: [Type]} deriving (Eq,Show)

>type Constructor = String
```

The body type holds information about one of the constructors of a type. `constructor` is self-explanatory. `labels` holds the names of labels of a record. This will be blank if the constructor isn't a record. `types` contains a representation of the type of each value within the constructor. The definition of `Type` is as follows.

```
>data Type      = Arrow Type Type -- fn
>               | Apply Type Type -- application
>               | Var String      -- variable
>               | Con String      -- constructor
>               | Tuple [Type]    -- tuple
>               | List Type   -- list
> deriving (Eq,Show)
```

Few of the deriving rules supplied have actually needed to use this type information, which I found quite surprising. If you do find you need to use it, one example is the Haskell2Xml rule.

4.3 Pretty Printing

Instead of producing a string as output, rules produce a value of type `Doc`. This type is defined in the Pretty Printing Library implemented by Simon Peyton-Jones. The pretty printer ensures that the code is formatted for readability, and also handles problems such as indentation. Constructing output using pretty printing combinators is easier and more structured than manipulating strings too. For those unfamiliar with these combinators, have a look at the module '`Pretty.lhs`' and the web page `http://www.cse.ogi.edu/~simonpj/` or for more detail the paper *The Design of a Pretty Printing Library, J. Hughes*

4.4 Utilities

Upon the pretty printing library, DrIFT defines some more formatting functions which make regularly occurring structures of code easier to write. These structures include simple instances, blocks of code, lists, etc. The utilities are in the module '`RuleUtils.hs`' and should be self explanatory.

4.5 Adding a new rule

A rule has type `type Rule = (String,Data -> Doc)`. Once you have written your mapping function and chosen an appropriate name for the rule, add this tuple to the list `userRules`

`:: [Rule]` in module '`UserRules.hs`'. Recompile if necessary. DrIFT will then call this rule when its name occurs in a command in an input file.

5 Installation

DrIFT isn't a large or complicated application, so it shouldn't be too hard for anyone to get it up and running. For the platform you want to install for, read the corresponding section below, then see Section 5.4 [Environment Variables], page 14

5.1 GHC

the automake script should automatically detect any ghc or nhc installation and use that to build and install DrIFT. First run `./configure` . To compile, type `make all`. The executable produced '`DrIFT`' can then be installed with `make install`.

5.2 Hugs

The DrIFT code comes as a set of Haskell modules. You want to copy all these to somewhere in your `HUGSPATH`, then you can load and run DrIFT in any directory.

5.3 Runhugs

Edit the first line of the the file '`DrIFT`' to point to your copy of `runhugs`. Copy '`DrIFT`' to somewhere on your `PATH`, and the remainder of the source ('`*.hs`','`*.lhs`') to a directory in your `HUGSPATH`

5.4 Environment Variables

In you environment set `DERIVEPATH` to the list of directories you wish derive to search for modules / interfaces.

`DERIVEPATH` is quite fussy about the format the list should take :-

- each path should be separated by ':'
- no space inserted anywhere
- no final '/' on the end of a path

For instance

good - `/users/nww/share/hugs/lib:/users/nww/share/hugs/lib/hugs`

bad - `/users/nww/share/hugs/lib/: /users/nww/share/hugs/lib/hugs/`

5.5 Installing the Emacs DrIFT Mode

Edit '`derive.el`' so that the variable `hwl-derive-cmd` contains your copy of the DrIFT executable. Place '`derive.el`' into a directory on your `load-path`, byte-compile it and put the following command into your '`.emacs`' file:

```
(load "derive")
```

6 Bugs and Shortcomings

- DrIFT doesn't check for commands applying the same rule to a type.
- No support for TeX-style literate code.

Table of Contents