JFP **13** (1): 139–144, January 2003.
DOI: 10.1017/S0956796803001217 Printed in the United Kingdom

Chapter 10

Specification of Derived Instances

A *derived instance* is an instance declaration that is generated automatically in conjunction with a `data` or `newtype` declaration. The body of a derived instance declaration is derived syntactically from the definition of the associated type. Derived instances are possible only for classes known to the compiler: those defined in either the Prelude or a standard library. In this appendix, we describe the derivation of classes defined by the Prelude.

If T is an algebraic datatype declared by:

$$\texttt{data}\ cx\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \ \texttt{=}\ \ K_1\ t_{11}\ \ldots\ t_{1k_1}\ |\ \cdots\ |\ K_n\ t_{n1}\ \ldots\ t_{nk_n}$$
$$\texttt{deriving}\ \texttt{(}C_1\texttt{,}\ \ldots\texttt{,}\ C_m\texttt{)}$$

(where $m \geq 0$ and the parentheses may be omitted if $m = 1$) then a derived instance declaration is possible for a class C if these conditions hold:

1. C is one of `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, or `Read`.
2. There is a context cx' such that $cx' \Rightarrow C\ t_{ij}$ holds for each of the constituent types t_{ij}.
3. If C is `Bounded`, the type must be either an enumeration (all constructors must be nullary) or have only one constructor.
4. If C is `Enum`, the type must be an enumeration.
5. There must be no explicit instance declaration elsewhere in the program that makes $T\ u_1\ \ldots\ u_k$ an instance of C.

For the purposes of derived instances, a `newtype` declaration is treated as a `data` declaration with a single constructor.

If the `deriving` form is present, an instance declaration is automatically generated for $T\ u_1\ \ldots\ u_k$ over each class C_i. If the derived instance declaration is impossible for any of the C_i then a static error results. If no derived instances are required, the `deriving` form may be omitted or the form `deriving ()` may be used.

Each derived instance declaration will have the form:

$$\texttt{instance (}cx\texttt{, }cx'\texttt{) =>}\ C_i\ (T\ u_1\ \ldots\ u_k)\ \texttt{where \{}\ d\ \texttt{\}}$$

where d is derived automatically depending on C_i and the data type declaration for T (as will be described in the remainder of this chapter).

The context cx' is the smallest context satisfying point (2) above. For mutually recusive data types, the compiler may need to perform a fixpoint calculation to compute it.

The remaining details of the derived instances for each of the derivable Prelude classes are now given. Free variables and constructors used in these translations always refer to entities defined by the `Prelude`.

10.1 Derived Instances of `Eq` and `Ord`

The class methods automatically introduced by derived instances of `Eq` and `Ord` are `(==)`, `(/=)`, `compare`, `(<)`, `(<=)`, `(>)`, `(>=)`, `max`, and `min`. The latter seven operators are defined so as to compare their arguments lexicographically with respect to the constructor set given, with earlier constructors in the datatype declaration counting as smaller than later ones. For example, for the `Bool` datatype, we have that `(True > False) == True`.

Derived comparisons always traverse constructors from left to right. These examples illustrate this property:

$$\begin{array}{lcl} \texttt{(1,undefined) == (2,undefined)} & \Rightarrow & \texttt{False} \\ \texttt{(undefined,1) == (undefined,2)} & \Rightarrow & \perp \end{array}$$

All derived operations of class `Eq` and `Ord` are strict in both arguments. For example, `False <=` $\perp$ is $\perp$, even though `False` is the first constructor of the `Bool` type.

10.2 Derived Instances of `Enum`

Derived instance declarations for the class `Enum` are only possible for enumerations (data types with only nullary constructors).

https://doi.org/10.1017/S0956796803001217 Published online by Cambridge University Press

The nullary constructors are assumed to be numbered left-to-right with the indices 0 through $n-1$. The `succ` and `pred` operators give the successor and predecessor respectively of a value, under this numbering scheme. It is an error to apply `succ` to the maximum element, or `pred` to the minimum element.

The `toEnum` and `fromEnum` operators map enumerated values to and from the `Int` type; `toEnum` raises a runtime error if the `Int` argument is not the index of one of the constructors.

The definitions of the remaining methods are

```
enumFrom x           = enumFromTo x lastCon
enumFromThen x y     = enumFromThenTo x y bound
                      where
                        bound | fromEnum y >= fromEnum x = lastCon
                              | otherwise                = firstCon
enumFromTo x y       = map toEnum [fromEnum x .. fromEnum y]
enumFromThenTo x y z = map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

where `firstCon` and `lastCon` are, respectively, the first and last constructors listed in the `data` declaration. For example, given the datatype:

```
data  Color = Red | Orange | Yellow | Green  deriving (Enum)
```

we would have:

```
[Orange ..]         ==  [Orange, Yellow, Green]
fromEnum Yellow     ==  2
```

10.3 Derived Instances of `Bounded`

The `Bounded` class introduces the class methods `minBound` and `maxBound`, which define the minimal and maximal elements of the type. For an enumeration, the first and last constructors listed in the `data` declaration are the bounds. For a type with a single constructor, the constructor is applied to the bounds for the constituent types. For example, the following datatype:

```
data  Pair a b = Pair a b deriving Bounded
```

would generate the following `Bounded` instance:

```
instance (Bounded a,Bounded b) => Bounded (Pair a b) where
  minBound = Pair minBound minBound
  maxBound = Pair maxBound maxBound
```

https://doi.org/10.1017/S0956796803001217 Published online by Cambridge University Press

10.4 Derived Instances of `Read` and `Show`

The class methods automatically introduced by derived instances of `Read` and `Show` are `showsPrec`, `readsPrec`, `showList`, and `readList`. They are used to coerce values into strings and parse strings into values.

The function `showsPrec d x r` accepts a precedence level `d` (a number from `0` to `11`), a value `x`, and a string `r`. It returns a string representing `x` concatenated to `r`. `showsPrec` satisfies the law:

```
showsPrec d x r ++ s  ==  showsPrec d x (r ++ s)
```

The representation will be enclosed in parentheses if the precedence of the top-level constructor in `x` is less than `d`. Thus, if `d` is `0` then the result is never surrounded in parentheses; if `d` is `11` it is always surrounded in parentheses, unless it is an atomic expression. (Recall that function application has precedence `10`.) The extra parameter `r` is essential if tree-like structures are to be printed in linear time rather than time quadratic in the size of the tree.

The function `readsPrec d s` accepts a precedence level `d` (a number from `0` to `10`) and a string `s`, and attempts to parse a value from the front of the string, returning a list of (parsed value, remaining string) pairs. If there is no successful parse, the returned list is empty. Parsing of an un-parenthesised infix operator application succeeds only if the precedence of the operator is greater than or equal to `d`.

It should be the case that

`(x, "")` is an element of `(readsPrec d (showsPrec d x ""))`

That is, `readsPrec` should be able to parse the string produced by `showsPrec`, and should deliver the value that `showsPrec` started with.

`showList` and `readList` allow lists of objects to be represented using non-standard denotations. This is especially useful for strings (lists of `Char`).

`readsPrec` will parse any valid representation of the standard types apart from strings, for which only quoted strings are accepted, and other lists, for which only the bracketed form `[`...`]` is accepted. See Chapter 8 for full details.

The result of `show` is a syntactically correct Haskell expression containing only constants, given the fixity declarations in force at the point where the type is declared. It contains only the constructor names defined in the data type, parentheses, and spaces. When labelled constructor fields are used, braces, commas, field names, and equal signs are also used. Parentheses are only added where needed, *ignoring associativity*. No line breaks are added. The result of `show` is readable by `read` if all component types are readable. (This is true for all instances defined in the Prelude but may not be true for user-defined instances.)

Derived instances of `Read` make the following assumptions, which derived instances of `Show` obey:

https://doi.org/10.1017/S0956796803001217 Published online by Cambridge University Press

- If the constructor is defined to be an infix operator, then the derived `Read` instance will parse only infix applications of the constructor (not the prefix form).

- Associativity is not used to reduce the occurrence of parentheses, although precedence may be. For example, given

```
infixr 4 :$
data T = Int :$ T  |  NT
```

  then:

  – `show (1 :$ 2 :$ NT)` produces the string `"1 :$ (2 :$ NT)"`.
  – `read "1 :$ (2 :$ NT)"` succeeds, with the obvious result.
  – `read "1 :$ 2 :$ NT"` fails.

- If the constructor is defined using record syntax, the derived `Read` will parse only the record-syntax form, and furthermore, the fields must be given in the same order as the original declaration.

- The derived `Read` instance allows arbitrary Haskell whitespace between tokens of the input string. Extra parentheses are also allowed.

The derived `Read` and `Show` instances may be unsuitable for some uses. Some problems include:

- Circular structures cannot be printed or read by these instances.

- The printer loses shared substructure; the printed representation of an object may be much larger than necessary.

- The parsing techniques used by the reader are very inefficient; reading a large structure may be quite slow.

- There is no user control over the printing of types defined in the Prelude. For example, there is no way to change the formatting of floating point numbers.

10.5 An Example

As a complete example, consider a tree datatype:

```
data Tree a = Leaf a | Tree a :^: Tree a
     deriving (Eq, Ord, Read, Show)
```

Automatic derivation of instance declarations for `Bounded` and `Enum` are not possible, as `Tree` is not an enumeration or single-constructor datatype. The complete instance declarations for `Tree` are shown in Figure 10.1, Note the implicit use of default class method definitions – for example, only `<=` is defined for `Ord`, with the other class methods (`<`, `>`, `>=`, `max`, and `min`) being defined by the defaults given in the class declaration shown in Figure 6.1 (page 85).

https://doi.org/10.1017/S0956796803001217 Published online by Cambridge University Press

```
infixr 5 :^:
data Tree a =  Leaf a  |  Tree a :^: Tree a

instance (Eq a) => Eq (Tree a) where
        Leaf m == Leaf n  =  m==n
        u:^:v  == x:^:y   =  u==x && v==y
             _ == _       =  False

instance (Ord a) => Ord (Tree a) where
        Leaf m <= Leaf n  =  m<=n
        Leaf m <= x:^:y   =  True
        u:^:v  <= Leaf n  =  False
        u:^:v  <= x:^:y   =  u<x || u==x && v<=y

instance (Show a) => Show (Tree a) where

        showsPrec d (Leaf m) = showParen (d > app_prec) showStr
          where
             showStr = showString "Leaf " . showsPrec (app_prec+1) m

        showsPrec d (u :^: v) = showParen (d > up_prec) showStr
          where
             showStr = showsPrec (up_prec+1) u .
                       showString " :^: "      .
                       showsPrec (up_prec+1) v
                -- Note: right-associativity of :^: ignored

instance (Read a) => Read (Tree a) where

        readsPrec d r =  readParen (d > up_prec)
                         (\r -> [(u:^:v,w) |
                                 (u,s) <- readsPrec (up_prec+1) r,
                                 (":^:",t) <- lex s,
                                 (v,w) <- readsPrec (up_prec+1) t]) r

                      ++ readParen (d > app_prec)
                         (\r -> [(Leaf m,t) |
                                 ("Leaf",s) <- lex r,
                                 (m,t) <- readsPrec (app_prec+1) s]) r

up_prec  = 5    -- Precedence of :^:
app_prec = 10   -- Application has precedence one more than
                -- the most tightly-binding operator
```

Figure 10.1: Example of Derived Instances

https://doi.org/10.1017/S0956796803001217 Published online by Cambridge University Press