Printing Trees That Grow

Rodrigo Mesquita

June 2022

1 Introduction

Trees That Grow?? is a programming idiom to define extensible data types, which particularly addresses the need for decorating abstract syntax trees with different additional information accross compiler stages. With this newfound extensibility, we are able to share one AST data type accross compiler stages and other AST clients — both of which need to define their own extensions to the datatype. This extensibility comes from using type-level functions in defining the data types, and having the user instance them with the needed extension.

As an example, here is the extensible definition of an abstract syntax tree (AST).

```
type Var = String
data Typ = Int
         | Fun Typ Typ

data Expr p = Lit (XLit p) Integer
            | Var (XVar p) Var
            | Ann (XAnn p) (Expr p) Typ
            | Abs (XAbs p) Var (Expr p)
            | App (XApp p) (Expr p) (Expr p)
            | XExpr !(XXExpr p) -- Constructor extension point

type family XLit p
type family XVar p
type family XAnn p
type family XAbs p
type family XApp p
type family XXExpr p
```

And an AST with no additional decorations could be extended from the above definition as

```
data UD
```

```
type instance XLit    UD = ()
type instance XVar    UD = ()
type instance XAnn    UD = ()
type instance XAbs    UD = ()
type instance XApp    UD = ()
type instance XXExpr UD = Void
```

A drawback of this extensible definition of a datatype is that few can be done without knowing the particular instance of the datatype's extension. This means the defined AST is, by itself, unusable.

One of the promises of extensible data types is the reduction of duplicated code, therefore, we might be tempted to define generic functions or type-class instances for it. In the original paper some solutions are provided

- ignore the extension points, although we no longer give the user the flexibility of an function or instance that takes into consideration the extension points they defined.
- or make use of higher order functions in the implementation, allowing for some custom usage of the extension points, but still restricted within the context of the generic implementation.

The second option, while more flexible, still isn't sufficient when faced with the need to define a radically different implementation for a particular constructor of the datatype, in which we might want to additionally make use of the defined extensions. We might also note that to define functions generic over the field extension points, a lot of higher order functions or dictionaries must be passed to the functions, and the type-class instance of an extension point is the same regardless of the constructor its found in.

We are then faced with the unattractive choice of either reducing duplicated code at the cost of flexibility, or of requiring a complete implementation of the function from any user needing that extra bit of flexibility.

This paper describes an idiom to define generic functions over the extensible abstract syntax tree which allow drop-in definitions from the user that take their extension instance into account.

2 Watering Trees That Grow

We would like to construct a clever way of having generic definitions of functions over an extensible data type, definitions which allow the extensible data type user to override particular parts of the implementation and delegate to the generic implementation of the function the non-overriden cases — allowing for a possible complete reimplementation of the instance if desired.

At first sight, a function that can default to some other implementation can simply be a function that takes as parameter a higher-order function which is the default implementation itself.

With a small tweak, the default implementation itself always calls the so called *override* function and pass it the actual default implementation as an argument.

For example, if we were to write a pretty printer for the above defined AST, which by default works regardless of the extension points, but that can be overriden on some or all constructors, we could have

```
override :: (Expr p -> String) -> Expr p -> String

pprDefault :: Expr p -> String
pprDefault = override (\case
    Lit _ i -> show i
    Var _ s -> s
    Ann _ e t -> "(" <> printE e <> ") :: (" <> printT t <> ")"
    Abs _ v e -> "" <> v <> "." <> printE e
    App _ f v -> "(" <> printE f <> ") (" <> printE v <> ")"
    XExpr _ -> "")
```

This naive approach doesn't quite solve it for us. First, it's just a sketch. The function *override* should be bound by *pprDefault* ensuring each client can pass a different override function. However, most importantly, we must consider type-class instances whose function signatures we cannot change – instances that we want to be defined near the datatypes so as not to create orphan instances; and be able to write functions taking constraints rather than higher-order functions, so that we have a common language for this pattern and forget about which function is the right one to pass where.

Let's continue our example, saying we now want to create a generic instance for the *Show* type class instance. Show is defined as

```
class Show a where
    show :: a -> String
```

We now want to instance *Show* for our extensible AST right next to the datatype (we don't want orphan instances!), but making sure the instance can make use of the *override* method, such that the user can override which parts they desire of the default implementation.

So the following logical step is to create a class to abstract over the *override* pattern. Note, however, that this class musn't be client-specific, as it should work for all clients the same. It should have the following skeleton

```
instance TheOverrideClass ??? => Show (Expr p) where
    show = override defaultShowExpr
```

The challenge here is what instances the class? *Expr* is defined in the client independent side, so we really wouldn't like to have anything to do with it, since instancing it from the client side would mean an orphan instance. That leaves us with p, the pass parameter, which is defined by the client.

Calling the *override* type-class *TTG*, and for now saying the type of override to be that needed by *show*, we get

```
class TTG p where
    override :: (Expr p -> String) -> Expr p -> String
```

Meaning we could now have, (saying we cautiously turn on UndecidableInstances)

```
instance TTG p => Show (Expr p) where
    show = override defaultShowExpr
```

To address the next challenge of *override* not yet working for any function, we realize *override* really is a function that takes some function of type *f* and returns a function of the same type *f*, which can make use of the default one for the cases it doesn't want to override.

So we want to change the type signature of *override* to be polymorphic over *f*. Unfortunately, we must extend the type-class with a parameter specifying the type *f* of the function we're overriding. By no longer using the TTG parameter *p* on the *override* signature we need to turn on *AllowAmbiguousTypes*, which will require that *override* must always be called with an explicit type application specifying the *p*. This is fine, given that the override usage should be confined to the generic functions over the extensible data type.

```
class TTG p f where
    override :: f -> f

instance TTG p (Expr p -> String) => Show (Expr p) where
    show = override @p defaultShowExpr
```

Finally, we'll note that we might want to override, for the same pass *p*, two functions of the same type. To disambiguate between this, and make our instance more readable in the way, we'll use the *DataKinds* extension to annotate the *TTG* override instance with the function it should override.

The following example showcases the use of the final *TTG* override class to define generic functions for both the pretty printer and show classes (which have the same type).

Client-independent code

```
class TTG p (s :: Symbol) f where
    override :: f -> f

instance TTG p "ppr" (Expr p -> String) => Pretty (Expr p) where
    ppr = override @p @"ppr" defaultPprExpr

instance TTG p "show" (Expr p -> String) => Show (Expr p) where
    show = override @p @"show" defaultShowExpr
```

Client-specific code

```
data Decorated

type instance XLit    Decorated = String
```

```
type instance XVar    Decorated = ()
type instance XAnn    Decorated = Bool
type instance XAbs    Decorated = ()
type instance XApp    Decorated = ()
type instance XXExpr Decorated = Void

instance TTG p "ppr" (Expr p -> String) where
    override def = \case
        Lit s i -> "Ann:" <> s <> " -- " <> show i
        e@(Ann b e t) -> if b then def e else ""
        x -> def x

instance TTG p "show" (Expr p -> String) where
    override def = \case
        Lit s i -> "Ann:" <> s <> " -- " <> show i
        Ann b e t -> "Bool:" <> show b <> " -- " <> "(" <> printE e <> ") ::
        x -> def x
```

This approach still has its drawbacks. For one, this makes it impossible to reuse parts of the default implementation, leading to code duplication in branches similar to the default, and secondly, we depend on much type-level machinery which complicates the program code, requiring both *UndecidableInstances* and *AllowAmbigousTypes*, meaning the override function requires explicit type applications.