Trees that Evolve Together

Shayan Najd
Laboratory for Foundations of Computer Science
The University of Edinburgh, Scotland, U.K.
sh.najd@gmail.com

August 8, 2018

Abstract

We study the notion of evolution in functional data types, as a new approach to the problem of decorating abstract syntax trees with additional information. We observed the need for such evolutions while redesigning the data types representing Haskell abstract syntax inside Glasgow Haskell Compiler (GHC).

Specifically, we generalise the Trees that Grow (TTG) idiom, to account for more refined forms of evolution in abstract syntax trees. The generalisation, while introduces a modest change, results in a remarkably more powerful approach. For example, the new approach can account for changes in the type of specific, but not all, subterms.

We show how such newly gained powers help with solving some practically challenging problems, such as handling source location decorations, introducing mutual recursion, and lightweight generic programming.

1 Introduction

A forest with one tree, is no forest; there are many trees in a forest, all growing together in harmony by passing of time.

A compiler with one abstract syntax data type, is no compiler; there are many abstract syntax data types in a compiler, all evolving together synchronously through passes of the compiler.

Trees that Grow (TTG) idiom (Najd and Peyton Jones, 2017) introduces a practical solution to the problem of extending abstract syntax data types in a functional setting. TTG supports a wide range of extensions to normal algebraic data types, and scales to support generalised algebraic data types. Extensions can account for the evolution of an abstract syntax data type through compiler passes, by assigning a different set of extensions for each pass-specific variant of an abstract syntax data type. For a specific pass, these sets of extensions can be synchronised across data types by sharing the same parameter used for describing the set of extensions. The same synchronisation process can be used to independently extend mutually-defined data types, straightforwardly.

Less straightforward is when two independently defined data types evolve to be mutually-defined.

For example, in Glasgow Haskell Compiler (GHC), expressions in the parsing, or type checking phase, are decorated with source locations, but the same form of expressions are entirely undecorated when used in meta programming, i.e, Template Haskell. Consider the following simplified data types describing expression in application form, in the parsing pass (denoted by $-^P$), type checking pass (denoted by $-^T$), and meta programming (denoted by $-^M$).

```
type LExp^P id = Loc (Exp^P id)
data Exp^P   id
 = App^P (LExp^P id) (LExp^P id)
 ⋮
```

```
type LExp^T id = Loc (Exp^T id)
data Exp^T   id
 = App^T Typ (LExp^T id) (LExp^T id)
 ⋮
```

```
data Exp^M id
 = App^M (Exp^M id) (Exp^M id)
 ⋮
```

where the wrapper for source location spans is defined as follows.

```
type Spn = (Integer , Integer)
```

```
data Loc trm = Loc Spn trm
```

We really would like to have, instead of three data types, one single base data type describing expressions, and define these three trees as instances of the base data type. This is exactly the goal of the TTG idiom. Following the TTG idiom, the base data type can be defined as the following.

```
type LExp^X x = Loc (Exp^X x)
data Exp^X   x
 = App^X !(X_App x) (LExp^X x) (LExp^X x)
 | New^X !(X_New x)
 ⋮
```

```
class X_Exp   x where
 type X_App x
 type X_New x
 ⋮
```

The base data type Exp^X clearly subsumes Exp^P and Exp^T by instantiating the extension type families such as X_{App}:

```
data P id
type LExp^P id = LExp^X (P id)
type Exp^P  id = Exp^X  (P id)
instance X_Exp (P id) where
 type      X_App (P id) = No_Fld
 type      X_New (P id) = No_Con
⋮
```

```
data T id
type LExp^T id = LExp^X (T id)
type Exp^T  id = Exp^X  (T id)
instance X_Exp (T id) where
 type      X_App (T id) = Typ
 type      X_New (T id) = No_Con
⋮
```

where the empty field extension type and the empty constructor extension type are defined as the following:

```
data No_Fld = No_Fld
```

```
data No_Con
```

It is straightforward to provide pattern synonyms, as the following, to hide the plumbing, which in turn makes the one-to-one correspondence evident.

```
pattern App^P :: LExp^P id → LExp^P id → Exp^P id
pattern App^P l m = App^X No_Fld l m
⋮
pattern App^T :: Typ → LExp^T id → LExp^T id → Exp^T id
pattern App^T a l m = App^X a l m
⋮
```

Nevertheless, the process is not as straightforward when it comes to subsuming Exp^M by instantiating Exp^X. The subexpression in Exp^X are of type $LExp^X$; the source location decorations are hard coded inside the subexpressions of Exp^X, deeming Exp^X an invalid choice for a base data type for subsuming Exp^M. Can we find a suitable base data type that subsumes all the three data types at once? Such a base data type should also be able to subsume data types reasonably similar to these three. What are the general guidelines?

In this paper, we try to answer these questions, by generalising the TTG idiom, to support changes to the type of specific, not all, subexpressions. This general case, handles subsumption of Exp^P, Exp^T and Exp^M by one single base data type, where the non-trivial change (beyond the simpler forms already supported by TTG) is the evolution of the type of some subexpressions from Exp^X to $LExp^X$. For example, in the App^X constructor of the ideal base type both arguments change type from Exp^X to $LExp^X$ when instantiated for Exp^P and Exp^T, but remain untouched for Exp^M. Such ability to change the type of specific subterms is beyond the original TTG approach, but as we show is easy achievable by a modest generalisation.

To see the necessity of such generalisation, let us attempt to solve the problem of subsuming at once all the three data types Exp^P,Exp^T, and Exp^M by an approach dual to the one taken earlier: we start with a base data type where

source location decorations are not hard coded into subexpressions.

Following the original TTG idiom, we have the following fully undecorated base data type:

```
data Exp^X x
=    App^X !(X_App x) (Exp^X x) (Exp^X x)
|    New^X !(X_New x)
⋮
```

```
class X_Exp x where
 type X_App x
 type X_New x
 ⋮
```

Now to store the source location decorations for the subexpressions in Exp^P and Exp^T, we have two options: either we store them in the extension fields, or in the extension constructor. The former leads to a new programming interface that barely corresponds to the original one. In the former design, the distinction between undecorated and decorated subexpressions is destroyed, leading to unnecessary couplings of code handling decorations and undecorated expressions. For example, with the former design, source location decorations should be handled in every clause of matchings, whenever we pattern match on an expression; it unnecessarily couples the task of handling source location decorations with any other task inspecting expressions. The latter design, is more promising in that handling source location decorations remains decoupled from other task in that when matching on expressions only one clause needs to handle source location decorations, i.e., the clause handling the constructor extension. However, as we discuss shortly, the latter design has its own drawbacks.

The following is how we instantiate the base data type Exp^X, for the three data type, following the latter design of storing source location decorations in the constructor extension.

```
data P id
type LExp^P id = Exp^X (P id)
type Exp^P  id = Exp^X (P id)
instance X_Exp (P id) where
 type X_App  (P id) = No_Fld
 type X_New  (P id) = Loc (Exp^P id)
⋮
```

```
data T id
type LExp^T id = Exp^X (T id)
type Exp^T  id = Exp^X (T id)
instance X_Exp (T id) where
 type X_App  (T id) = Typ
 type X_New  (T id) = Loc (Exp^T id)
⋮
```

```
data M id
type Exp^M id = Exp^X (M id)
instance X_Exp (M id) where
 type X_App (M id) = No_Fld
 type X_New (M id) = No_Con
⋮
```

We can provide pattern synonyms for constructors such App as before. Moreover, we can also provide synonyms for the wrapper at each type:

```
pattern Loc^P :: Spn → Exp^P id → LExp^P id
pattern Loc^P sp m = New^X (Loc sp m)
pattern Loc^T :: Spn → Exp^T id → LExp^T id
pattern Loc^T sp m = New^X (Loc sp m)
```

Unfortunately, above does not suffice! There are two key problems with it.

The above TTG design allows for wrapping of an already-wrapped expression, an unintended behavior that the original design (i.e., in the original data types Exp^P and Exp^T) avoids via the mutual recursion. Above design, leads to redundant, and potentially incoherent, source location decorations. For example, the following is wrongly accepted:

```
badWrap :: LExp^P id → LExp^P id
badWrap m = Loc^P (0 , 0) m
```

Furthermore, the above TTG design, suffers from a dual problem: it allows for expressions without source location decorations, an unintended behavior that the original design also avoids via the mutual recursion. For example, the following is wrongly accepted:

```
badSelfApp :: Exp^P id→ Exp^P id
badSelfApp m = App^P m m
```

What we really need is the ability to change the type of subexpressions of App^X from $Exp^X\ x$ to $Loc\ (Exp^X\ x')$. We can do so with a modest generalisation of the original TTG: we allow the extension parameter of subterms to be set locally by the extension type family of the constructor containing the subterms. In the original TTG idiom, extension parameters were globally set by the data type.

Here is an example of how Exp^X looks like in such a generalised setting:

```
data Exp^X x
 = ∀ x0 x1.
   App^X !(X_App x x0 x1) (Exp^X x0) (Exp^X x1)
 | New^X !(X_New x)
 ⋮
```

```
class X_Exp x where
 type X_App x x0 x1
 type X_New x
 ⋮
```

Notice the only difference from earlier is that the extension parameters of subexpressions are set locally via extension type families rather than globally. For example, notice that the parameter x_0 of the first subexpression in App^X is declared locally, apart from the global x.

This modest change, has a remarkable impact on increasing the range of changes that a base data type can support. For example, here is how the new Exp^X can subsume all the three data types.

```
data LP id
type LExp^P id = Exp^X (LP id)
instance X_Exp (LP id) where
 type X_App (LP id) x0 x1 = No_Con
 type X_New (LP id) = Loc (Exp^P id)
⋮
data P id
type Exp^P id = Exp^X (P id)
instance X_Exp (P id) where
 type X_App (P id) x0 x1
 = Prf ( x0 ~ LP id
       , x1 ~ LP id )
 type X_New (P id) = No_Con
⋮
```

```
data LT id
type LExp^T id = Exp^X (LT id)
instance X_Exp (LT id) where
 type X_App (LT id) x0 x1 = No_Con
 type X_New (LT id) = Loc (Exp^T id)
⋮
data T id
type Exp^T  id = Exp^X (T id)
instance X_Exp (T id) where
 type X_App (T id) x0 x1
 = (Typ , Prf ( x0 ~ LT id
              , x1 ~ LT id ))
 type X_New (T id) = No_Con
⋮
```

```
data M id
type Exp^M id = Exp^X (M id)
instance X_Exp (M id) where
 type X_App (M id) x0 x1 = Prf ( x0 ~ M id , x1 ~ M id )
 type X_New (M id)       = No_Con
⋮
```

We can provide pattern synonyms as before, in particular for App constructors, we have the following.

```
pattern App^P :: LExp^P id → LExp^P id → Exp^P id
pattern App^P  l m = App^X Prf l m
pattern App^T :: Typ → LExp^T id → LExp^T id → Exp^T id
pattern App^T a l m = App^X (a , Prf) l m
pattern App^M :: Exp^M id → Exp^M id → Exp^M id
pattern App^M  l m = App^X Prf l m
```

where we use the following to wrap explicit proofs of equality (and other constraints in general).

```
data Prf p where
 Prf :: p ⇒ Prf p
```

In above, we define two variants of Exp^X for each data type Exp^P and Exp^T: one decorated with source locations at the top-level, another undecorated at the top-level. Subexpressions in one variant refers to the other alternatively, e.g., the subexpressions of Exp^P are of type $LExp^P$ while the subexpression of $LExp^P$

is of type Exp^P. The switching between the two variants is guaranteed by the proof instance Prf in the field extension of App^X. Now, the following is rightly rejected

```
badWrap :: LExp^P id → LExp^P id
badWrap m = Loc^P (0 , 0) m
```

with the following error message

```
. Couldn't match type 'LP' with 'P'
  Expected type: Exp^P id
    Actual type: LExp^P id
```

and, the following is also rightly rejected

```
badSelfApp :: Exp^P id→ Exp^P id
badSelfApp m = App^P m m
```

with the following error message

```
. Couldn't match type 'P' with 'LP'
  Expected type: LExp^P id
    Actual type: Exp^P id
```

This pattern mimics the one-to-one correspondence between mutually recursive definition of data types and their equivalent GADT definition using indices.

For example, the following mutually-recursive definition

```
data A
 = C B
 | D A
```

```
data B
 = E A
```

is in a one-to-one correspondence to the following GADT flat definition.

```
data S = A | B
data El (s :: S)
 = (s ∼ A) ⇒ C (El B)
 | (s ∼ A) ⇒ D (El A)
 | (s ∼ B) ⇒ E (El A)
```

But, the scope of the introduced generalisation of TTG is not limited to mutually-recursive definitions. We will explain the idea further throughout the paper.

References

Najd, S. and Peyton Jones, S. (2017). Trees that grow. *J. UCS*, 23(1):42–62.