Free Theorems for Nested Types

ANONYMOUS AUTHOR(S)

1 LIMITATIONS OF FOLDS AND ISSUES DEFINING BUSHREVERSE

We would like to define a function `bushreverse` which reverses a bush. Bushes are not linear, so it is not immediately clear what reversing a bush should even mean. First consider the flattening of bushes as a natural transformation from Bush to List (see bush2list). The flattening of a reversed bush should be the reversal (on lists) of the flattening of the original bush, i.e., `forall b. bush2list (bushreverse b) = reverse (bush2list b)` or `bush2list ∘ bushreverse = reverse ∘ bush2list`. We would also like `bushreverse` to be an involution on bushes.

Clearly this function will have a recursive definition, so to define it in our calculus we will use a fold. Before defining `bushreverse` as a fold, let's consider how to define list reversal as a fold. There are several ways to do this. One method is to replace nil with nil and replace cons with a function which inserts an element at the back of a list. This cons replacement function can be defined in several ways, e.g.,

$\lambda\, x\, xs \rightarrow$ `xs ++ [ x ]`, or

$\lambda\, x \rightarrow$ `foldr (::) [ x ]` .

The first version makes use of the append function on lists. The second version simply replaces the nil constructor with a singleton list. The first cannot be defined in our system because we cannot define an append function for bushes (see next section). The second cannot be defined because the fold for bushes requires a fully polymorphic algebra, so the only possible replacement for bnil is bnil in this context. Writing the insertion function with foldr is much simpler than the case for bushes because there is only one nil constructor in a list, but a bush can have many occurences of bnil. We can also give a recursive definition without using append or foldr:

```
insert : ∀ {A : Set} → A → List A → List A
insert x [] = [ x ]
insert x (y :: ys) = y :: (insert x xs)
```

and we can define an analogous function for bushes:

```
binsert : ∀ {A : Set} → A → Bush (Bush A) → Bush A
binsert x bnil = bcons x bnil
binsert x (bcons bnil bbbx) = bcons x (binsert bnil bbbx)
binsert x (bcons (bcons y bx) bbbx) = bcons y (binsert (binsert x bx) bbbx)
```

and feed this as an argument to the fold for bushes to get a reversal function:

```
bushreverse : ∀ {A : Set} → Bush A → Bush A
bushreverse {A} = bfold {A} {Bush} bnil binsert
```

Note that the first explicit argument to bfold is completely determined by the types. The argument must have type `(∀ {A : Set} → Bush A)` and the only inhabitant of this type is bnil. Folds for (non first-order) nested types will require arguments of the form `(∀ {A : Set} → F A)` whenever the type has constructors with no arguments, and due to the universal quantification, the only inhabitants of this type will be these constructors themselves (e.g. pnil, bnil) with some

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

other possible combinations if there are constructors of type `(∀ {A : Set} → F A → F A → F A)`, for example.

This restriction means that the behavior of our reversal function must be specified entirely by binsert. The question is, can we define binsert without arbitrary term level recursion? If we try to define it as a fold over the `Bush (Bush A)` parameter, then we have no way to include x in that (fully polymorphic) fold. We could even try to define it as a fold over a bush built up from x such as `(bcons x bnil)`, but the same issue arises.

The fundamental issue here is that the binsert function takes two arguments, and there is no way to combine them together with a single fold. We can fold over each argument separately, but then we still have results to combine.

1.1 What can we define with folds over nested types?

These issues are not specific to the Bush datatype; they are inherent limitations of folds over nested types. In fact we are quite restricted in the types of functions we can define recursively. To define a function using a fold, its domain must be a fixed point, therefore types such as `(∀ {A : Set} → (Bush A × Bush A) → Bush A)` cannot be defined as a fold over the pair. Of course, we can fold over each bush in the pair individually, but we cannot interleave the computations in any way because the polymorphism required by the folds prohibits this.

Additionally, the return type of the result of the fold must be a functor. So even if we curry the previous type to get `(∀ {A : Set} → Bush A → (Bush A → Bush A))`, we can not define it as a fold over the first bush because `(Bush A → Bush A)` is not a functor. These two restrictions indicate that we cannot define many interesting functions which have multiple nested types as arguments.

The restriction stretches even further though, because sometimes the algebras supplied to a fold must have a recursive definition involving nested types. This is the case with `bushreverse` because even though it has a suitable type, `(∀ {A : Set} → Bush A → Bush A)`, the binsert function has a type which makes it incompatible with fold.

1.2 Generalized Folds

We can make our calculus slightly more expressive using generalized folds [**?**]. However, these generalized folds are mostly helpful for defining *reductions*, functions which take data of a monomorphic type as input and have a monomorphic return type. Specifically, generalized folds relax the constraint on the domain of the function being defined as a fold. Instead of requiring the domain to be a fixed point, the domain can be a fixed point composed with another functor. However, this does not help us in defining functions which have multiple nested types as arguments.