Thesis Proposal: Verified tail recursive folds through dissection

Carlos Tomé Cortiñas

April 4, 2018

Supervised by Wouter Swierstra
Second supervisor Alejandro Serrano Mena

Contents

1 Verified tail-recursive fold for binary trees

In this chapter, we present the transformation of a catamorphism for binary trees with natural numbers in the leaves onto a tail-recursive function. Moreover, we show how to prove that the resulting function terminates and that it is correct (for every possible input it computes the same result as the original fold).
In order to do so, in section 1.1 we introduce the type of binary trees along with several examples of evaluation functions and how their common structure can be abstracted into a catamorphism. Then, in section 1.2, we explain the concept of a Zipper with its many type indexed variants and in section 1.3.

1.1 Binary trees

The type of binary trees with natural numbers on the leaves is represented in *Agda*using the following inductive type.

```
data Tree : Set where
  Tip  : ℕ    → Tree
  Node : Tree → Tree → Tree
```

As examples of binary trees for example we can have:

photo example

We can use binary trees to represent the syntax of arithmetic expressions. A value of type Tree is an expression where the Node constructor stands for addition and the natural numbers stand for themselves. Under this view, we can interpret (or evaluate) an expression into a natural number using a function eval.

```
eval : Tree → ℕ
eval (Tip n)      = n
eval (Node t₁ t₂) = eval t₁ + eval t₂
```

The function eval is defined compositionally (in the style of denotational semantics) where the value of a Node is computed by composing the values for the left and right

subtree. Analogously, we can have another interpetation function that pretty prints a binary tree. This is a function of type Tree → String.

```
pretty : Tree → String
pretty (Tip n)      = show n
pretty (Node t1 t2) = pretty t1 ++ " + " ++ pretty t2
```

Both interpretations functions share some underlying structure. In the case of eval, when evaluating the constructor Node we combine the results of both subtrees using the addition operator + while when pretty printing we combine them by concatenating the resulting Strings.
Abstracting the concrete functions used in each case into an algebra.

```
record TreeAlg : Set1 where
  field
    A      : Set
    TipA   : ℕ → A
    NodeA  : A → A → A
```

A has to be in TreeAlg or it should be made parametric over it?

We can define a general catamorphism that folds a tree into a value of the type specified by the algebra.

```
treeCata : (tAlg : TreeAlg) → Tree → A tAlg
treeCata tAlg (Tip n)      = TipA tAlg n
treeCata tAlg (Node t1 t2) = NodeA tAlg (treeCata tAlg t1) (treeCata tAlg t2)
```

Using the catamorphism, the eval function can be re-implemented as follows.

```
eval : Tree → ℕ
eval = treeCata tAlg
  where
    tAlg : TreeAlg
    tAlg = record {A = ℕ; TipA = id; NodeA = \_ + \_}
```

1.2 Zipper

Following Huet's idea of a *Zipper*[1], any leaf of a binary tree can be represented as a pair of the natural number it holds and the path from the root of the tree to the position it occupies.

```
UZipper : Set
UZipper = ℕ × Stack
```

At each Node constructor, the path has to record whether the leaf occurs in the left or the right subtree. Moreover, from a value of *Zipper* we should be able to reconstruct the original tree thus each time the path choses between left or right it has to keep track of the remaining subtree.

```
Stack : Set
Stack = List (Tree ⊎ Tree)
```

In fig. 1, there are two examples of *Zipper* where the leaf in focus is marked with a box □. The *Zipper* shown on the bottom is a tuple with the path to the leaf and the natural number. In the path, a left arrow ← (correspondingly a right arrow →) represents that the rest of the path points to a leaf in the left (right) subtree of the Node.
If we were able to freeze a traversal from left to right over a binary tree, a value of type *Zipper* would represent a concrete state of this procedure. At each leaf, everything

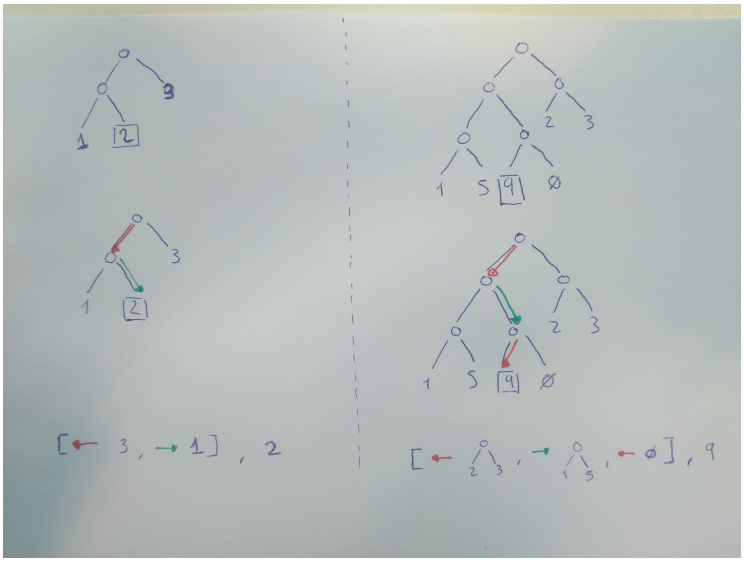

Figure 1: Example of *Zipper*

that has already been traversed appears to the left of the Stack while the parts of the tree that still have to be processed show up on the right.
In order to represent the state of folding a binary tree from left to right, we can enrich the type of the Stack save not only the left subtrees (in case the path points to the right →) but also the value it evaluates (using the catamorphism) to and a proof of this equality.
The rest of the constructions we are going to show assume there is an ambient tree algebra and that it is constant across all definitions and proofs. To achieve this in *Agda*, we create a new module parametrized by a TreeAlg.

```
module _ (tAlg : TreeAlg) where
  open TreeAlg tAlg
```

By **open**ing $tAlg$ we bring into scope the type A and the fields $TipA$ and $NodeA$ without need to make any further reference to $tAlg$. The type of Stack is now defined as follows.

```
Stack : Set
Stack = List (Tree ⊎ □ A □ a → □ Tree □ t
                             → treeCata tAlg t ≡ a)

pattern Left  t       = inj₁ t
pattern Right a t eq  = inj₂ (a , t , eq)
```

The only piece of information required to compute the catamorphism of a Tree are the a values that come from evaluating the subtrees that appear to the left of the position under focus. However, as we will show later in , it is neccesary for proving termination to keep around the Tree where the value came from. We say that t and the equality proof in Stack are computationally irrelevant. However, in *Agda* we cannot express such fact[1].

ref

Given a Stack and a Tree (we can always embed a ℕ into a Tree using Tip), the original Tree can be reconstructed by recursing over the Stack. At each step, it is known to which subtree the position belongs.

```
plug⇓ : Tree → Stack → Tree
plug⇓ t [] = t
```

[1] In other systems like Coq the Tree would live in Prop

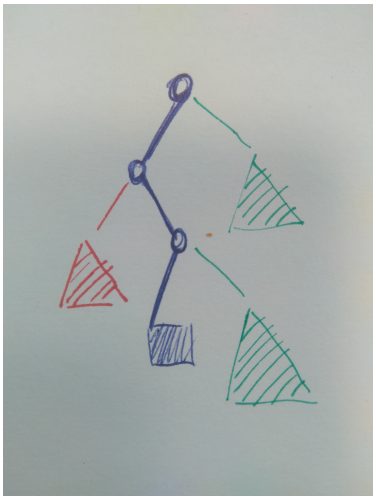

Figure 2: Example of freezing a traversal

```
plug⇓ t (Left t₁  ::  s)       =  Node (plug⇓ t s) t₁
plug⇓ t (Right _ t₁ _  ::  s)  =  Node t₁ (plug⇓ t s)
```

Until now, we have only considered that the Stack represents the path from the root of the tree way down to the focused leaf. We can instead reverse the Stack part of the *Zipper* so the path travels bottom-up from the leaf up to the root of the tree.

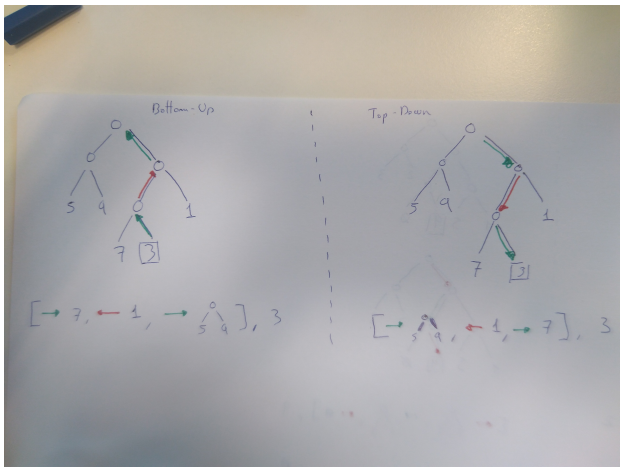

Figure 3: Bottom-Up versus Top-Down *Zipper*

Under this view, we can reconstruct the original Tree from the Stack.

```
plug⇑  :  Tree  →  Stack  →  Tree
plug⇑ t (Left   t₁  ::  s)      =  plug⇑ (Node t t₁) s
plug⇑ t (Right _ t₁ _  ::  s)  =  plug⇑ (Node t₁ t) s
plug⇑ t []  =  t
```

The only difference between both interpretations of the *Zipper* is that to translate from one to the other we just need to reverse the Stack. We can show that indeed the original Tree is preserved by the conversion.

```
plug⇑−to−plug⇓  :  ∀ t s  →  plug⇑ t s ≡ plug⇓ t (reverse s)
plug⇑−to−plug⇓ t s  =  ⋯
```

```
plug⇓−to−plug⇑ : ∀ t s → plug⇓ t s ≡ plug⇑ t (reverse s)
plug⇓−to−plug⇑ t s = ⋯
```

1.3 One-step of a fold

The *Zipper* type represents a snapshot of the internal state of a catamorphism over a binary tree. From one value of *Zipper* we can write a pair of functions, load and unload that combined together perform one step of the fold.

```
load : Tree → Stack → UZipper ⊎ A
load (Tip x) s        = inj₁ (x , s)
load (Node t₁ t₂) s   = load t₁ (Left t₂ :: s)

unload : (t : Tree) → (a : A) → (eq : treeCata tAlg t) ≡ a → Stack → UZipper ⊎ A
unload t a eq []                    = inj₂ a
unload t a eq (Left t′ :: s)        = load t′ (Right a t eq :: s)
unload t a eq (Right a′ t′ eq′ :: s) = unload (Node t′ t) (NodeA a′ a) (cong₂ NodeA eq′ eq) s
```

The function unload traverses the Stack combining all the results of evaluating the subtrees that were on the left of the position until either reaches a Node that has an unevaluated subtree on the right (Left constructor) or the Stack is empty and the Tree has been fully evaluated. When the former case occurs the load function is in charge of traversing the subtree on the right to find the leftmost leaf.

maybe talk about irrelevance of t and eq?

A folding step just consists of calling unload passing appropriate arguments.

```
step : UZipper → UZipper ⊎ A
step (n , s) = unload (Tip n) (TipA n) refl s
```

More graphically, the proccess of folding the tree one step is depicted in in fig. 4,

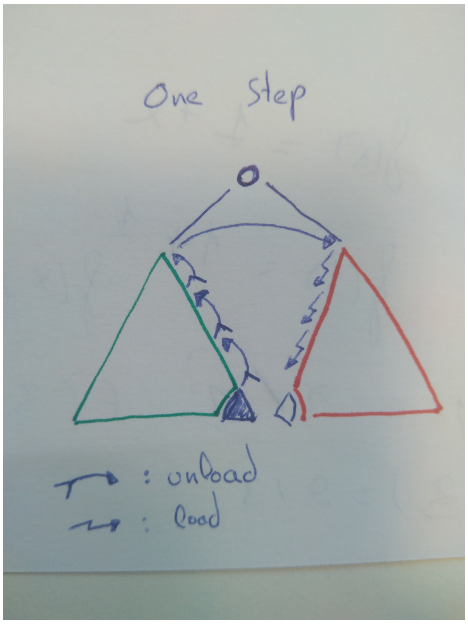

Figure 4: One step of fold

1.4 Folding a Tree

From the machinery we have developed so far one would be inclined to think that in order to fold a Tree it will suffice to find a fixed point of the function step.

```
foldTree : Tree → ℕ
foldTree t with load t []
... | inj₂ n = n
... | inj₁ z = rec z
  where
    rec : UZipper → ℕ
    rec z with step z
    ... | inj₁ z′ = rec z′
    ... | inj₂ n = n
```

However, it does not work. *Agda*'s termination checker is right when it warns that the z' passed as an argument to the recursive call on rec is not structurally smaller than z and therefore the fixed point of step might not exists[2].
From an outer perspective, it is fair to assert that for any possible input of type UZipper, the function rec has a fixed point. A value of UZipper focus on the position of a concrete leaf in a value of type Tree, and the function step refocuses the to the next leaf on the right. Because there are finitely many leaves on a Tree, the function rec must always terminate.
In order to solve this problem, in the next subsections we will develop the notion of indexed *Zipper* and show how it allows us to define an order relation between positions (*Zipper*s) of the same Tree. This will be the key to define rec by structural recursion.

1.5 Indexed *Zipper*

The current type of *Zipper*, UZipper does not encode too much information. Given a value, it is not possible to statically know the Tree from which it represents a position. If we are to compare a pair of UZipper we need them to represent positions on the same Tree otherwise it does not make sense.
We can address this shortcoming in the current representation by creating a new type wrapper over UZipper that is type indexed by the Tree where it is a position.

```
plugZ⇓ : UZipper → Tree
plugZ⇓ (t , s) = plug⇓ (Tip t) s
data Zipper⇓ (t : Tree) : Set where
  _,_ : (z : UZipper) → plugZ⇓ z ≡ t → Zipper⇓ t
```

From an indexed *Zipper* we can forget the extra information and recover the original UZipper.

```
forget⇓ : ∀ {t : Tree} → (z : Zipper⇓ t) → UZipper
forget⇓ (z , _) = z
```

References

[1] Gérard Huet and Inria Rocquencourt France. The zipper. *JFP*, 1997.

[2] rec loops forever