ELIMINATING TRAVERSELS IN AGDA

1. Introduction

Let us consider the following problem:

Given a binary tree t with integer values in the leaves.
Replace every value in t by the minimum.

The most obvious way to solve this, would be by first traversing the tree to calculate the minimum and then traversing the tree to replace all values by that. Note that it is simple to prove the termination and we go twice through the tree.

However, there is a more efficient solution to this problem [6]. By using a *cyclic* program, he described a program for which only one traversal is needed. Normally, one defines functions on algebraic data types by using structural recursion and then proof assistants, such as Coq and Agda [5, 8], can automatically check the termination. Cyclic programs, on the other hand, are *not* structurally recursive and they do *not* necessarily terminate. One can show this particular one terminates using clocked type theory [3], but this has not been implemented in a proof assistant yet. Hence, this solution is more efficient, but the price we pay, is that proving termination becomes more difficult.

In addition, showing correctness requires different techniques. For structurally recursive functions, one can use structural induction. For cyclic programs, that technique is not available and thus broader techniques are needed.

This pearl describes an Agda implementation of this program together with a proof that it is terminating and a correctness proof [8]. Our solution is based on the work by Atkey and McBride [3] and the approach shows similarities to clocked type theory [4]. We start by giving an Haskell implementation to demonstrate the issues we have to tackle. After that we discuss the main tool for checking termination in Agda, namely *sized types*. Types are assigned sizes and if those decrease in recursive calls, then the program is productive. We then give the solution, which is terminating since Agda accepts it. Lastly, we demonstrate how to do proofs with sized types and we finish by proving correctness via equational reasoning.

2. The Haskell Implementation

Bird's original solution is the following Haskell program [6].

```
data Tree = Leaf Int | Node Tree Tree

replaceMin :: Tree → Tree
replaceMin t = let (r, m) = rmb t m in r
  where
    rmb :: Tree → Int → (Tree, Int)
    rmb (Leaf x) y = (Leaf y, x)
    rmb (Node l r) y =
      let (l′, ml) = rmb l y
```

$$
\begin{aligned}
&\quad (r', mr) = rmb\ r\ y \\
&\mathbf{in}\ (Node\ l'\ r', min\ ml\ mr)
\end{aligned}
$$

A peculiar feature of this program, is the call of rmb. Rather than defining m via structural recursion, it is defined via the fixed point of $rmb\ t$. As a consequence, systems such as Coq and Agda cannot automatically guarantee this function actually terminates [5, 8]. Beside that, showing correctness becomes more difficult since we cannot use just structural induction anymore.

Due to this, the termination of this program crucially depends on lazy evaluation. If $rmb\ t\ m$ would be calculated eagerly, then before unfolding rmb, the value m has to be known. However, this requires $rmb\ t\ m$ to be computed already and hence, it does not terminate.

All in all, to make this all work in a total programming language, we need a mechanism to allow general recursion, which produces productive functions. In addition, since the termination of general recursive functions requires lazy evaluation, we also need a way to annotate that an argument of a function is evaluated lazily. This is the exact opposite from Haskell where by default arguments are evaluated lazily and strictness is annotated.

3. Programming with Sized Types

Our goal is to define a function fix, which gives general recursion, and a data type $\triangleright$ representing delayed computations. Since lazy evaluation is about delaying computations and forcing them when needed, the type $\triangleright$ can be used to evaluate arguments lazily. To guarantee everything remains productive, we need broader termination checks and that is where *sized types* come into play [1].

3.1. Sized Types. Instead of just structural recursion, this allows general recursion and lazy evaluation. The main idea is that types are annotated with a size, which, intuitively, is an ordinal number. To check the termination of a not necessarily structural-recursive function, the sizes must decrease in every call.

More concretely, there is a type Size and *sized type* is a type indexed by Size.

$$\mathsf{SizedSet} = \mathsf{Size} \to \mathsf{Set}$$

On these sets, we need several operations. Some of them, like exponentials, sums, products, and constant families, are defined pointwise.

$$
\begin{aligned}
&\Rightarrow\ :\ \mathsf{SizedSet} \to \mathsf{SizedSet} \to \mathsf{SizedSet} \\
&A \Rightarrow B = \lambda\ i \to A\ i \to B\ i
\end{aligned}
$$

$$
\begin{aligned}
&\oplus\ :\ \mathsf{SizedSet} \to \mathsf{SizedSet} \to \mathsf{SizedSet} \\
&A \oplus B = \lambda\ i \to A\ i \uplus B\ i
\end{aligned}
$$

$$
\begin{aligned}
&\otimes\ :\ \mathsf{SizedSet} \to \mathsf{SizedSet} \to \mathsf{SizedSet} \\
&A \otimes B = \lambda\ i \to A\ i \times B\ i
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{c}\ :\ \mathsf{Set} \to \mathsf{SizedSet} \\
&\mathsf{c}\ A = \lambda\ \_ \to A
\end{aligned}
$$

A less intuitive operation, realizes sized types as actual types. Suppose, we have some sized type A and let us call its realization $\&\ A$. Then inhabitants of A are the same as functions assining to each size i an element of $A\ i$. Hence, & A is just a type of dependent functions.

```
& : SizedSet → Set
& A = {i : Size} → A i
```

3.2. **Delayed Computations.** The main ingredient in the machinery is a data type representing delayed computations. Lazy evaluation requires delaying calculations as some must only be done when they are forced. For example, elements in a lazy list are only computed when they are needed by some other function.

Since Agda is a total language, everything should remain productive and this is where sizes come into play. Productivity is guaranteed if the sizes decrease in each recursive call. Concretely, this means that whenever we have some delayed computation of size i, we can only force it with a size smaller than i.

The type Size< i represents those sizes smaller than i. Now we can define a sized type ▷ A as follows.

```
record ▷ (A : SizedSet) (i : Size) : Set where
  coinductive
  field force : {j : Size< i} → A j
open ▷ public
```

Note that this is a coinductive record meaning that we can use *copatterns* to define values [2]. Instead of saying how elements are constructed, it says how elements are destructed or, intuitively, how to make observations on elements. Taking this point of view, we can say that the function force makes an observation on a delayed computation.

To get a feeling how this all works, we look at the lazy natural numbers. This also explains why this mechanism allows lazy evaluation. We define the lazy natural numbers similar to the usual natural numbers, but the argument of the successor is delayed.

```
data LNat (i : Size) : Set where
  LZ : LNat i
  LS : ▷ LNat i → LNat i
```

A simple function we can define with these, computes the number infinity. On ℕ, the natural numbers defined inductively, we cannot define that, because it is not structurally recursive. The calculation does not terminate. However, if we evaluate the argument of the successor lazily, then this is no problem.

We define infinity with mutual recursion. The first function computes infinity as a lazy natural number and the second gives a delayed version of it.

```
infinity : & LNat
▷infinity : &(▷ LNat)
```

For infinity, we repeatedly need to apply LS. However, its argument is delayed, so we use ▷infinity for it. We define ▷infinity with copatterns meaning that we only need to give the value of force ▷infinity.

```
infinity {i} = LS (▷infinity {i})
force (▷infinity {i}) {j} = infinity {j}
```

Note that the sizes in each call decrease since j has type Size< i. It is actually unnecessary to write down all the sizes since Agda can determine them itself.

This type actually corresponds with the natural numbers in Haskell, because Haskell evaluates lazily. Due to lazy evaluation,

In the remainder, we shall need that $\triangleright$ is an applicative functor [7]. This means we need to define functions pure and _⊛_. Both are defined using copatterns.

$$\mathsf{pure} : \{A : \mathsf{SizedSet}\} \to \&\ A \to \&(\triangleright\ A)$$
$$\mathsf{force}\ (\mathsf{pure}\ x) = x$$

$$\circledast : \{A : \mathsf{SizedSet}\}\ \{B : \mathsf{SizedSet}\} \to \&(\triangleright(A \Rightarrow B) \Rightarrow \triangleright\ A \Rightarrow \triangleright\ B)$$
$$\mathsf{force}\ (f \circledast x) = \mathsf{force}\ f\ (\mathsf{force}\ x)$$

Now that we got lazy evaluation, we can move our attention to general recursion. For that we use an operation, called fix, which gives the fixpoint of maps between sized types. It is computed by repeatedly applying the given function. To guarantee that the sizes decrease in every call, only functions of type $\&(\triangleright\ A \Rightarrow A)$ are accepted. Concretely, this means that the function must evaluate its argument lazily. We define fix in a similar fashion to repeat.

$$\mathsf{fix} : \{A : \mathsf{SizedSet}\} \to \&(\triangleright\ A \Rightarrow A) \to \&\ A$$
$$\triangleright\mathsf{fix} : \{A : \mathsf{SizedSet}\} \to \&(\triangleright\ A \Rightarrow A) \to \&\ (\triangleright\ A)$$
$$\mathsf{fix}\ f\ \{i\} = f\ \{i\}\ (\triangleright\mathsf{fix}\ f\ \{i\})$$
$$\mathsf{force}\ (\triangleright\mathsf{fix}\ f\ \{i\})\ \{j\} = \mathsf{fix}\ f\ \{j\}$$

$$\mathsf{infinity\text{-}alt} : \&\ \mathsf{LNat}$$
$$\mathsf{infinity\text{-}alt} = \mathsf{fix}\ \mathsf{LS}$$

Again we write down the sizes explicitly, even though it is not necessary, to make clear this function is productive.

4. Eliminating Traversals

4.1. **The Setting.** With all the theory in place, we can give the required data types and basic functions. The values in the leafs all are natural numbers and we shall need a sized type of natural numbers. This is the same one as CNat, but now we define it in a more concise way. Beside that, we need the minimum of numbers.

$$\mathsf{SizedNat} : \mathsf{SizedSet}$$
$$\mathsf{SizedNat} = \mathsf{c}\ \mathbb{N}$$

$$\mathsf{min} : \&(\mathsf{SizedNat} \Rightarrow \mathsf{SizedNat} \Rightarrow \mathsf{SizedNat})$$
$$\mathsf{min}\ n\ m = n \sqcap m$$

Next we define a sized type of trees where the leafs are labelled with sized natural numbers. As usual, we have two constructors: Leaf and Node. We let both constructors preserve the size.

```
data Tree (i : Size) : Set where
  Leaf : SizedNat i → Tree i
  Node : Tree i → Tree i → Tree i
```

In the examples repeat and fix we discussed before, the definition required two steps. We needed an actual version, repeat and fix, and a delayed version, ▷repeat and ▷fix. For Leaf and Node we also need a delayed version. We define them using copatterns and the applicative structure of ▷.

```
▷Leaf : &(▷ SizedNat ⇒ ▷ Tree)
force (▷Leaf n) = Leaf (force n)

▷Node : &(▷ Tree ⇒ ▷ Tree ⇒ ▷ Tree)
▷Node t₁ t₂ = pure Node ⊛ t₁ ⊛ t₂
```

4.2. **The Algorithm.** Now we translate the Haskell program given in the introduction to Agda. Note that in the third line of the original program, we compute a fixpoint of the function *rmb t*. So, to describe the algorithm, we first need to say how to compute that fixpoint. For that, we use the function feedback.

```
fb-h : {B U : SizedSet} → &(▷ U ⇒ B ⊗ U) → &(▷(B ⊗ U) ⇒ B ⊗ U)
fb-h f x = f (pure proj₂ ⊛ x)

feedback : {B U : SizedSet} → &(▷ U ⇒ B ⊗ U) → & B
feedback f = proj₁ (fix (fb-h f))
```

Now we want to define the help function rmb, which will be the argument for feedback. We take ▷ Tree for B and SizedNat for U. Note that we must use ▷ Tree, because otherwise we would not be able to apply Node or ▷Node.

```
rmb : &(Tree ⇒ ▷ SizedNat ⇒ ▷ Tree ⊗ SizedNat)
rmb (Leaf x) n = (▷Leaf n , x)
rmb (Node l r) n =
  let rmbl = rmb l n
      rmbr = rmb r n
  in (▷Node (proj₁ rmbl) (proj₁ rmbr) , min (proj₂ rmbl) (proj₂ rmbr))

replaceMin : &(Tree ⇒ Tree)
replaceMin t = force (feedback (rmb t))
```

Since feedback (rmb t) has the type ▷ Tree, we must apply force.

5. Proving with Sized Types

Our next goal is to prove functional correctness of replaceMin. To formulate the specification, we need predicates and relations on sized types, universal quantification, and equality types

A predicate on A is a function giving a type for each $a : A$. A relation between A and B is a function giving a type for each $a : A$ and $b : B$. To make it sized, we let it depend on sizes.

```
SizedPredicate : SizedSet → Set₁
SizedPredicate A = {i : Size} → A i → Set

SizedRelation : SizedSet → SizedSet → Set₁
SizedRelation A B = {i : Size} → A i → B i → Set
```

Next we define universal quantification and for that we use dependent products. Given a sized type A and a predicate on A, we get another sized type.

$\prod$: (A : SizedSet) → SizedPredicate A → SizedSet
$\prod$ A B i = (x : A i) → B x

Furthermore, for each sized type A, we have a relation on A representing equality. For this we use propositional equality in Agda.

eq : (A : SizedSet) → SizedRelation A A
eq A x y = $x \equiv y$

Note that for $\prod$ and eq we use the Agda implementations of the dependent product and equality respectively. This means that we can use all previously defined functions for them and in paticular, we can use Agda's usual notation for equational reasoning.

The last relation we need, might seem a bit unexpected. It is another equality type, but for delayed computations. Rather than saying that elements are propositionally equal, it says that all observations on them are equal. More specifically, if applying force on them gives the same result, then they are equal.

When using this predicate, the elements might use force meaning that they depend on some size. Hence, we first define a sized set which for every size i gives the sizes smaller than i.

~ : {A : SizedSet} → &(▷ A) → &(▷ A) → Set
x ~ y = {i : Size} {j : Size< i} → force x {j} ≡ force y

pure-force : {A : SizedSet} → (x : &(▷ A)) → pure (force x) ~ x
pure-force x = refl

force-pure : {A : SizedSet} → (x : & A) → force (pure x) ≡ x
force-pure x = refl

identity : {A : SizedSet} (x : &(▷ A)) → pure (λ z → z) ⊛ x ~ x
identity x = refl

composition : {A B C : SizedSet}
    (g : &(▷(B ⇒ C))) (f : &(▷(A ⇒ B)))
    (x : &(▷ A))
    → pure (λ h_1 h_2 z → h_1(h_2 z)) ⊛ g ⊛ f ⊛ x ~ g ⊛ (f ⊛ x)
composition g f x = refl

homomorphism : {A B : SizedSet} (f : &(A ⇒ B)) (x : & A) → pure f ⊛ pure x ~ pure (f x)
homomorphism f x = refl

interchange : {A B : SizedSet} (f : &(▷(A ⇒ B))) (x : & A) → f ⊛ (pure x) ~ pure (λ z → z x) ⊛ f
interchange f x = refl

Size<Set : SizedSet

$\mathsf{Size{<}Set}\ i = \mathsf{Size{<}}\ i$

$\triangleright\mathsf{eq} : (A : \mathsf{SizedSet}) \to \mathsf{SizedRelation}\ (\mathsf{Size{<}Set} \Rightarrow \triangleright\ A)\ (\mathsf{Size{<}Set} \Rightarrow \triangleright\ A)$
$\triangleright\mathsf{eq}\ A\ \{i\}\ x\ y = \{j : \mathsf{Size{<}}\ i\} \to \mathsf{force}\ (x\ j)\ \{j\} \equiv \mathsf{force}\ (y\ j)$

6. Correctness

To write down the specification we first give the inefficient implementation of the algorithm. The specification says they are equal on every input. We need two help functions, which are the first and second projection of rmb.

The first function, which replace all values in a tree by some other value, might seem not straightforward. One could define it without any delayed computation, but actually the value by which we replace everything, is delayed. This also means that the resulting tree is delayed as well. The reason for this, is that the number given to rmb is delayed.

$\mathsf{replace} : \&(\mathsf{Tree} \Rightarrow \triangleright\ \mathsf{SizedNat} \Rightarrow \triangleright\ \mathsf{Tree})$
$\mathsf{replace}\ (\mathsf{Leaf}\ x)\ n = \triangleright\mathsf{Leaf}\ n$
$\mathsf{replace}\ (\mathsf{Node}\ l\ r)\ n = \triangleright\mathsf{Node}\ (\mathsf{replace}\ l\ n)\ (\mathsf{replace}\ r\ n)$

The second function, which takes the minimum of a tree, is straightforward.

$\mathsf{min\text{-}tree} : \&(\mathsf{Tree} \Rightarrow \mathsf{SizedNat})$
$\mathsf{min\text{-}tree}\ (\mathsf{Leaf}\ x) = x$
$\mathsf{min\text{-}tree}\ (\mathsf{Node}\ l\ r) = \mathsf{min}\ (\mathsf{min\text{-}tree}\ l)\ (\mathsf{min\text{-}tree}\ r)$

All in all, we get the following inefficient implementation and specification.

$\mathsf{replaceMin\text{-}spec} : \&(\mathsf{Tree} \Rightarrow \mathsf{Tree})$
$\mathsf{replaceMin\text{-}spec}\ t = \mathsf{force}\ (\mathsf{replace}\ t\ (\mathsf{pure}\ (\mathsf{min\text{-}tree}\ t)))$

$\mathsf{valid} : \&(\mathsf{Tree} \Rightarrow \mathsf{Tree}) \to \mathsf{Set}$
$\mathsf{valid}\ f = \&(\prod\ \mathsf{Tree}\ (\lambda\ t \to \mathsf{eq}\ \mathsf{Tree}\ (f\ t)\ (\mathsf{replaceMin\text{-}spec}\ t)))$

In the remainder of this section, we prove that replaceMin satisfies the specification. We do this in three steps. First, we compute the first and second projection of rmb. These are given by replace and min-tree respectively. Second, we prove a lemma replace. Briefly, it says we can always make the second argument, of type $\triangleright$ SizedNat, of the form pure x. Third, we put it all together to prove the correctness theorem.

$\mathsf{rmb}_1 : \&(\prod\ \mathsf{Tree}\ (\lambda\ t \to$
$\quad\quad \&(\prod\ (\triangleright\ \mathsf{SizedNat})\ (\lambda\ n \to$
$\quad\quad\quad \mathsf{eq}\ (\triangleright\ \mathsf{Tree})\ (\mathsf{proj}_1\ (\mathsf{rmb}\ t\ n))\ (\mathsf{replace}\ t\ n)))))$
$\mathsf{rmb}_1\ (\mathsf{Leaf}\ x)\ n = \mathsf{refl}$
$\mathsf{rmb}_1\ (\mathsf{Node}\ l\ r)\ n =$
$\quad \mathsf{begin}$
$\quad\quad \mathsf{proj}_1\ (\mathsf{rmb}\ (\mathsf{Node}\ l\ r)\ n)$
$\quad \equiv\langle \text{ unfold } \mathsf{rmb} \rangle$
$\quad\quad \triangleright\mathsf{Node}\ (\mathsf{proj}_1\ (\mathsf{rmb}\ l\ n))\ (\mathsf{proj}_1\ (\mathsf{rmb}\ r\ n))$
$\quad \equiv\langle \text{ induction hypothesis, } \mathsf{rmb}_1\ l\ n \rangle$

```
    ▷Node (replace l n) (proj₁ (rmb r n))
  ≡⟨ induction hypothesis, rmb₁ r n ⟩
    ▷Node (replace l n) (replace r n)
  ≡⟨ fold replace ⟩
    replace (Node l r) n
  ■

rmb₂ : &(∏ Tree (λ t →
          &(∏ (▷ SizedNat) (λ n →
            eq SizedNat (proj₂ (rmb t n)) (min-tree t))))
          )
rmb₂ (Leaf x) n = refl
rmb₂ (Node l r) n =
  begin
    proj₂ (rmb (Node l r) n)
  ≡⟨ unfold rmb ⟩
    min (proj₂ (rmb l n)) (proj₂ (rmb r n))
  ≡⟨ induction hypothesis, rmb₂ r n ⟩
    min (proj₂ (rmb l n)) (min-tree r)
  ≡⟨ induction hypothesis, rmb₂ l n ⟩
    min (min-tree l) (min-tree r)
  ■
```

For the correctness theorem, we need one lemma which gives the value of force (replace t n).

```
replace< : &(Tree ⇒ ▷ SizedNat ⇒ Size<Set ⇒ ▷ Tree)
replace< t n  = replace t n

replace-pure : &(Tree ⇒ ▷ SizedNat ⇒ Size<Set ⇒ ▷ Tree)
replace-pure t n j = replace t (pure (force n))

▷replace : &(∏ Tree (λ t →
              &(∏ (▷ SizedNat) (λ n →
                ▷eq Tree (replace< t n) (replace-pure t n))))
              )
▷replace (Leaf x) n = refl
▷replace (Node l r) n =
  begin
    force (replace (Node l r) n)
  ≡⟨ refl ⟩
    force (▷Node (replace l n) (replace r n))
  ≡⟨ refl ⟩
```

```
    force (pure Node ⊛ (replace l n) ⊛ (replace r n))
  ≡⟨ refl ⟩
    force (pure Node) (force (replace l n)) (force (replace r n))
  ≡⟨ cong (λ z → force (pure Node) z (force (replace r n))) (▷replace l n) ⟩
    force (pure Node) (force (replace l (pure (force n)))) (force (replace r n))
  ≡⟨ cong (λ z → force (pure Node) (force (replace l (pure (force n)))) z) (▷replace r n) ⟩
    force (replace (Node l r) (pure (force n)))
  ∎

rm-correct : valid replaceMin
rm-correct t =
  begin
    replaceMin t
  ≡⟨ unfold replaceMin, feedback ⟩
    force (proj₁ (fix (λ x → rmb t (pure proj₂ ⊛ x))))
  ≡⟨ unfold fix ⟩
    force (proj₁ (rmb t (pure proj₂ ⊛ ▷fix (fb-h (rmb t)))))
  ≡⟨ rewrite rmb₁ ⟩
    force (replace t (pure proj₂ ⊛ ▷fix (fb-h (rmb t))))
  ≡⟨ rewrite ▷replace ⟩
    force (replace t (pure (force (pure proj₂ ⊛ ▷fix (fb-h (rmb t))))))
  ≡⟨ unfold pure ⟩
    force (replace t (pure (proj₂ (fix (fb-h (rmb t))))))
  ≡⟨ unfold fix ⟩
    force (replace t (pure (proj₂ (rmb t ((pure proj₂ ⊛ ▷fix (fb-h (rmb t))))))))
  ≡⟨ rewrite rmb₂ ⟩
    force (replace t (pure (min-tree t)))
  ∎
```

References

[1] Andreas Abel. Termination checking with types. *RAIRO-Theoretical Informatics and Applications*, 38(4):277–319, 2004.

[2] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

[3] Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. In *ACM SIGPLAN Notices*, volume 48, pages 197–208. ACM, 2013.

[4] Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. The clocks are ticking: No more delays! In *Logic in Computer Science (LICS), 2017 32nd Annual ACM/IEEE Symposium on*, pages 1–12. IEEE, 2017.

[5] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[6] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta informatica*, 21(3):239–250, 1984.

[7] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.

[8] Ulf Norell. Dependently typed programming in agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.