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].

$\textbf{data}\ \mathit{Tree} = \mathit{Leaf}\ \mathit{Int} \mid \mathit{Node}\ \mathit{Tree}\ \mathit{Tree}$

$\mathit{replaceMin} :: \mathit{Tree} \to \mathit{Tree}$
$\mathit{replaceMin}\ t = \textbf{let}\ (r, m) = \mathit{rmb}\ (t, m)\ \textbf{in}\ r$
$\quad \textbf{where}$
$\qquad \mathit{rmb} :: (\mathit{Tree}, \mathit{Int}) \to (\mathit{Tree}, \mathit{Int})$
$\qquad \mathit{rmb}\ (\mathit{Leaf}\ x, y) = (\mathit{Leaf}\ y, x)$
$\qquad \mathit{rmb}\ (\mathit{Node}\ l\ r, y) =$
$\qquad\quad \textbf{let}\ (l', \mathit{ml}) = \mathit{rmb}\ (l, y)$

$$(r', mr) = rmb\ (r, y)$$
$$\textbf{in}\ (Node\ l'\ r', min\ ml\ mr)$$

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. Sized Types

A sized type is a family indexed by sizes. Formally, we define it as follows.

```
SizedSet = Size → Set
```

To work with sized types, we define several combinators. These come in two flavors. Firstly, we have combinators to construct sized types. The first two of these are analogs of the function type and product type.

```
_⇒_ : SizedSet → SizedSet → SizedSet
(A ⇒ B) i = A i → B i

_⊗_ : SizedSet → SizedSet → SizedSet
(A ⊗ B) i = A i × B i
```

As usual, $\otimes$ binds stronger than $\Rightarrow$. Beside those, two combinators, which relate types and sized types. Types can be transformed into sized types by taking the constant family.

```
c : Set → SizedSet
c A i = A
```

Conversly, we can turn sized types into types by taking the product. This operation is called the *box modality*, and we denote it by $\square$.

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

The last construction we need, represents delayed computations.

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

Secondly, we define combinators to define terms of sized types. We start by giving $\triangleright$ the structure of an applicative functor.

```
pure : {A : SizedSet} → □ A → □(▷ A)
force (pure x) i = x

_⊛_ : {A B : SizedSet} → □(▷(A ⇒ B) ⇒ ▷ A ⇒ ▷ B)
force (f ⊛ x) i = force f i (force x i)
```

Lastly, we have a fixpoint combinator, which takes the fixpoint of productive function.

```
fix : {A : SizedSet} → □(▷ A ⇒ A) → □ A
▷fix : {A : SizedSet} → □(▷ A ⇒ A) → □ (▷ A)
fix f {i} = f (▷fix f {i})
force (▷fix f {i}) j = fix f {j}
```

Now let us see all of this in action via a simple example. Our goal is to compute the fixpoint of $f(x, y) = (1, x)$.

```
const₁ : {N L P : SizedSet}
  → □(▷ N ⇒ P)
  → □(▷(L ⊗ N) ⇒ P)
const₁ f x = f (pure proj₂ ⊛ x)
```

Note that $f(x, y) = (1, x)$ is constant in the second coordinate. We define it as follows.

```
  solution : ℕ × ℕ
  solution =

    let f : □(▷(▷(c ℕ) ⊗ c ℕ)
              ⇒ ▷(c ℕ) ⊗ c ℕ)
        f = const₁ (λ x → x , 1)

        fixpoint : □(▷(c ℕ) ⊗ c ℕ)
        fixpoint = fix f

        (n , m) = fixpoint
    in  force n ∞ , m
```

4. Eliminating Traversals

We start by with the usual definition of binary trees.

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

Next we define lazy versions of the constructors Leaf and Node. To do so, we use that ▷ is an applicative functor.

```
▷Leaf : □(▷(c ℕ) ⇒ ▷ (c Tree))
▷Leaf n = pure Leaf ⊛ n
```

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

The function rmb takes a tree t and lazily evaluated natural number n and it returns a pair. The first coordinate of that pair is a lazily evaluated tree, which is t with each value replaced by n. The second coordinate is the minimum of t.

```
rmb : □(c Tree ⊗ ▷(c ℕ) ⇒ (▷(c Tree)) ⊗ c ℕ)
rmb (Leaf x , n) = (▷Leaf n , x)
rmb (Node l r , n) =
  let (l' , ml) = rmb (l , n)
      (r' , mr) = rmb (r , n)
  in (▷Node l' r' , ml ⊓ mr)
```

Now we define the actual program.

```
gconst₁ : {N T TN : SizedSet}
  → (f : □(T ⊗ ▷ N ⇒ TN))
  → (t : □ T)
  → □(▷(▷ T ⊗ N) ⇒ TN)
gconst₁ f t = const₁ (curry f t)

replaceMin : Tree → Tree
replaceMin t =
```

We first give the equation of which we take the fixpoint.

```
let f : □(▷(▷ (c Tree) ⊗ c ℕ) ⇒ ▷ (c Tree) ⊗ c ℕ)
    f = gconst₁ rmb t

    fixpoint : □(▷ (c Tree) ⊗ c ℕ)
    fixpoint = fix f

in force (proj₁ fixpoint) ∞
```

5. Proving with Sized Types

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

For sized predicates, we only need one combinator, which represents universal quantification. We define it pointwise using the dependent product of types.

```
all : (A : SizedSet) → SizedPredicate A → SizedSet
all A B i = (x : A i) → B x

syntax all A (λ x → B) = ∏[ x ∈ A ] B
```

If we want to prove an equation involving force, we need to give it all required arguments. One of those arguments, is a size smaller than i. For this reason, we define the following sized type.

```
Size<Set : SizedSet
Size<Set i = Size< i
```

6. Functional Correctness

```
replace : Tree → ℕ → Tree
replace (Leaf x) n = Leaf n
replace (Node l r) n = Node (replace l n) (replace r n)

min : Tree → ℕ
min (Leaf x) = x
min (Node l r) = min l ⊓ min r

replaceMin-spec : Tree → Tree
replaceMin-spec t = replace t (min t)
```

The proof of functional correctness goes in three step. We start by computing rmb and for that, we compute its first and second coordinate. Since the first projection of rmb is computed lazily, we need to force it.

```
rmb₁ : □(∏[ p ∈ Size<Set ⊗ c Tree ⊗ ▷(c ℕ) ]
             let (j , t , n) = p
             in force (proj₁ (rmb (t , n))) j
                ≡
                replace t (force n j))
rmb₁ (j , Leaf x , n) = refl
rmb₁ (j , Node l r , n) =
  begin
    force (▷Node
             (proj₁ (rmb (l , n)))
             (proj₁ (rmb (r , n))))
          j
  ≡⟨⟩
    Node
      (force (proj₁ (rmb (l , n))) j)
      (force (proj₁ (rmb (r , n))) j)
  ≡⟨ cong (λ z → Node z _) (rmb₁ (j , l , n)) ⟩
    Node
      (replace l (force n j))
      (force (proj₁ (rmb (r , n))) j)
  ≡⟨ cong (λ z → Node _ z) (rmb₁ (j , r , n)) ⟩
    Node
      (replace l (force n j))
      (replace r (force n j))
  ∎
```

The second projection is easier.

```
rmb₂ : □(∏[ p ∈ c Tree ⊗ ▷(c ℕ) ]
              let (t , n) = p
              in proj₂ (rmb (t , n))
                 ≡
                 min t)
rmb₂ (Leaf x , n) = refl
rmb₂ (Node l r , n) =
  begin
    proj₂ (rmb (l , n)) ⊓ proj₂ (rmb (r , n))
  ≡⟨ cong (λ z → z ⊓ _) (rmb₂ (l , n)) ⟩
    min l ⊓ proj₂ (rmb (r , n))
  ≡⟨ cong (λ z → _ ⊓ z) (rmb₂ (r , n)) ⟩
    min l ⊓ min r
  ∎
```

Now we use them both to compute replaceMin.

```
rm-correct : (t : Tree)
  → replaceMin t ≡ replaceMin-spec t
rm-correct t =
  begin
    replaceMin t
  ≡⟨⟩
    force (proj₁ (rmb (t , pure proj₂ ⊛ ▷fix (gconst₁ rmb t)))) ∞
  ≡⟨ rmb₁ (∞ , t , pure proj₂ ⊛ ▷fix (gconst₁ rmb t)) ⟩
    replace t (proj₂ (fix (gconst₁ rmb t)))
  ≡⟨⟩
    replace t (proj₂ (rmb (t , _)))
  ≡⟨ cong (replace t) (rmb₂ (t , _)) ⟩
    replace t (min 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.