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
```

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

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. Delayed Computations

As we have seen, the Haskell implementation cannot automatically be transferred to Agda, because the termination cannot be guaranteed. For that reason, we introduce a mechanism, which allows delaying computations. Beside that, the termination must take these delays into account.

Let us start with thinking about delaying computations. Types do not have any dependence on time: all their inhabitants are always available. To make them depend on time, we look at types indexed by some type of times. We call them *timed types*.

3.1. Timed Types.

The main ingredient for timed types, is thus the indexing type. The most straightforward option would be the natural, but we shall refrain to use them. Instead, we use Size for the indices. Before explaining why, let us precisely define what timed types are.

```
Time : Set
Time = Size

TimedSet = Time → Set
```

In the remainder of this section, we explain how to use timed types. For that we introduce several combinators: some of them to construct such types, some of them for actual programming. The simplest lifts operations on types to timed types.

```
⇒ : TimedSet → TimedSet → TimedSet
(A ⇒ B) i = A i → B i

⊗ : TimedSet → TimedSet → TimedSet
(A ⊗ B) i = A i × B i
```

```
c : Set → TimedSet
c A i = A
```

Timed types can be turned into types. This corresponds to universal quantification in first order logic. If we think of a time type as a predicate on times,

```
□ : TimedSet → Set
□ A = {i : Time} → A i
```

To see what is going on, let us look at an example of a timed type.

The natural numbers are generated by zero and the successor. We can define these operations on the timed natural as well. They are defined pointwise.

At every time, we have both zero and the successor of each timed natural number. Other operations, for example the minimum, are defined in the same way. Note that the function ⊓ takes the minimum of two natural numbers.

The last combinator represents delayed computations. For this, we need to look more closely to Agda's mechanism of sized types. An important operation on sizes is the order and we use it to define an order on types.

```
Time< : Time → Set
Time< i = Size< i
```

The set Time< i represents the times smaller than i. With this order, we can now defined delayed computations.

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

The only inhabitants ▷ A i are those of A j for j smaller than i. This means that something in A i is only accessible in ▷ A k for k greater than i. Or, in words, this means that inhabitants of A are only available in ▷ A at later times.

Before discussing examples, let us first look at how to program with delayed computations. First, we give ▷ the structure of an applicative functor.

```
pure : {A : Set} → □ (c A ⇒ ▷ A)
force (pure x) = x

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

Secondly, we give a fixpoint combinator and here the order plays a crucial role. The reason of that, is because it allows broader priductivity checks. The semantics guarantee that there is no infinitely decreasing sequence of sizes. In particular, if a size decrease in every recursive call of some function, then this map is productive.

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

4. Eliminating Traversals

Now we have developed sufficient material to formalize the program. We start by defining the relevant data types: a data type of natural numbers and a data type of trees. Remember that we already defined TimedNat as a constant family. The data type of trees is defined the same way.

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

This data type has two constructors TLeaf and TNode. We also have delayed versions of them.

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

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

fb-h : {T N : Set} → □(▷ N ⇒ c(T × N)) → □(▷(T × N) ⇒ c(T × N))
fb-h f x = f (pure proj₂ ⊛ x)

feedback : {T N : Set} → □(▷ N ⇒ c(T × N)) → T
feedback {T} {N} f = proj₁ (fix {T × N} (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 → □(▷ ℕ ⇒ c(Tree × ℕ))
rmb (Leaf x) n = (Leaf (force n) , x)
rmb (Node l r) n =
  let (l' , ml) = rmb l n
      (r' , mr) = rmb r n
  in (Node l' r' , ml ⊓ mr)

replaceMin : Tree → Tree
replaceMin t = feedback (rmb t)
```

5. Proving with Sized Types

Our next goal is to prove correctness. Before doing that, we need to define timed predicates, timed relations, and combinators on them.

```
TimedPredicate : TimedSet → Set₁
TimedPredicate A = {i : Time} → A i → Set

TimedRelation : TimedSet → TimedSet → Set₁
TimedRelation A B = {i : Time} → 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.

```
all : (A : TimedSet) → TimedPredicate A → TimedSet
all A B i = (x : A i) → B x

syntax all A (λ x → B) = ∏[ x ∈ A ] B

eq : (A : Set) → TimedRelation (c A) (c A)
eq A x y = x ≡ y

syntax eq A x y = x ≡[ A ]≡ y
```

6. Correctness

6.1. Specification.

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

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

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

6.2. Proof.

```
rmb₁ : □(∏[ t ∈ c Tree ]
        □(∏[ n ∈ ▷ ℕ ]
          (proj₁ (rmb t n) ≡[ Tree ]≡ replace t (force n))))
rmb₁ (Leaf x) n = refl
rmb₁ (Node l r) n =
  begin
    Node (proj₁ (rmb l n)) (proj₁ (rmb r n))
  ≡⟨ cong (λ z → Node z ) (rmb₁ l n) ⟩
    Node (replace l (force n)) (proj₁ (rmb r n))
  ≡⟨ cong (Node ) (rmb₁ r n) ⟩
    Node (replace l (force n)) (replace r (force n))
  ■

rmb₂ : □(∏[ t ∈ c Tree ]
        □(∏[ n ∈ ▷ ℕ ]
          (proj₂ (rmb t n) ≡[ ℕ ]≡ min-tree 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-tree l ⊓ proj₂ (rmb r n)
  ≡⟨ cong (λ z →  ⊓ z) (rmb₂ r n) ⟩
    min-tree l ⊓ min-tree r
  ■

rm-correct : □(∏[ t ∈ c Tree ] (replaceMin t ≡[ Tree ]≡ replaceMin-spec t))
rm-correct t =
  begin
    feedback (rmb t)
  ≡⟨ refl ⟩
    proj₁ (rmb t (pure proj₂ ⊛ ▷fix (fb-h (rmb t))))
  ≡⟨ rmb₁ t (pure proj₂ ⊛ ▷fix (fb-h (rmb t))) ⟩
    replace t (proj₂ (rmb t (pure proj₂ ⊛ ▷fix (fb-h (rmb t)))))
  ≡⟨ cong (replace t) (rmb₂ t (pure proj₂ ⊛ ▷fix (fb-h (rmb t)))) ⟩
    replace t (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.