Specifying and Verifying a Transformation of Recursive Functions into Tail-Recursive Functions

Axel Suárez Polo[1], José de Jesús Lavalle Martínez[1], Iván Molina Rebolledo[1]

[1] Benemérita Universidad Autónoma de Puebla, Puebla, Mexico

author1@xxx.xx, author2@xxx.xx

Abstract.

It is well known that some recursive functions admit a tail recursive counterpart which have a more efficient time-complexity behavior. This paper presents a formal specification and verification of such process. A monoid is used to generate a recursive function and its tail-recursive counterpart. Also, the monoid properties are used to prove extensional equality of both functions. In order to achieve this goal, the Agda programming language and proof assistant is used to generate a parametrized module with a monoid, via dependent types. This technique is exemplified with the length, reverse, and indices functions over lists.

Keywords. Dependent Types, Formal Specification and Verification, Tail Recursion, Accumulation, Program Transformation

1 Introduction

Dependently typed programming languages provide an expressive system that allows both programming and theorem proving. Agda is an implementation of such a kind of language [6]. Using these programming languages, it can be proved that two functions return the same output when they receive the same input, which is a property known as *extensional equality* [5].

Programs can be developed using a transformational approach, where an initial program whose correctness is easy to verify is written, and after that, it is transformed into a more efficient program that preserves the same properties and semantics [12]. Proving that the transformed program works the same way as the original program is usually done by using *algebraic reasoning* [3], but this can also be done using dependently typed programming [11], with the advantage of the proof being verified by the compiler.

The *accumulation* strategy is a well-known program transformation technique to improve the efficiency of recursive functions [4]. This technique is the focus of this paper, in which dependently typed programming is used to develop a strategy to prove extensional equality between the original recursive programs and their tail-recursive counterparts.

The source code of this paper is available at https://github.com/ggzor/specifying-verifying-tail-recursion.

2 A simple example: list length

Let us start with a simple example: a function to compute the length of a list. This function can be defined recursively as follows:

```
  ℕ::
length : List A →ℕ
length [] = 0
length (x :: xs) = suc (length xs)
```

Nonetheless, this function requires space proportional to the length of the list due to the recursive calls. This program can be transformed into a tail-recursive function, which can be optimized automatically by the compiler to use constant space [2]. The transformed function is shown below:

```
  ℕ::→ℕ::
length-tail : List A →ℕ→ℕ
length-tail [] n = n
length-tail (x ::
xs) n = length-tail xs (suc n)
```

In this example, it is clear to see that both functions return the same result for every possible list we provide as input. This fact can be represented in Agda using dependent function types:

ℕ::→ℕ::→ℕ→ℕ::≡∀ℕ→≡::∀→≡≡

```
lengthlength-tail : ∀ (xs : List A)→
                      length xs ≡
length-tail xs 0
```

The notion of "sameness" used here is the one of *intensional equality*, which is an inductively defined family of types [7, 11] with the following definition:

```
data ≡__ {a} {A : Set a} (x : A) : A →
Set a where
  instance refl : x ≡ x
```

This means that two terms are equal if they are exactly the same term. Additionally, in Agda, if both terms reduce to the same term, we can state that they are intensionally equal. For example, $\text{refl} : 2 + 3 \equiv 5$.

This notion of equality together with the addition of the universal quantifier, allows us to state a kind of equality for functions, known as *point-wise equality* or *extensional equality* [5].

To prove extensional equality for the length functions, we can proceed inductively over the list, which has the [] and ::xxs cases[1]:

```
≡
lengthlength-tail : ∀ (xs : List A)→
                      length xs ≡
length-tail xs 0≡
lengthlength-tail [] = ?≡
lengthlength-tail (x :: xs) = ?
```

The base case is trivial, because both sides of the equality in the result type reduce to the same term:

$$\begin{aligned} \text{length } [] &= 0 \qquad \text{(by definition)} \\ \text{length-tail } [] \; 0 &= 0 \end{aligned}$$

Therefore, we can fill the first hole in our proof with refl:

```
≡
lengthlength-tail [] = refl
```

[1] The ? symbols are holes, which must be filled later to complete the proof, but are useful to write the proof incrementally.

For the inductive case, we can reduce both sides of the equality instantiated with the argument, and check what is necessary to prove. Note that this can be done automatically by querying Agda, and it is particularly useful when using the Agda mode in Emacs [15]. The reductions are shown below and follow from the definition:

$$\begin{aligned} \text{length } (x :: xs) &= \text{suc } (\text{length } xs) \\ \text{length-tail } (x :: xs) \; 0 &= \text{length-tail } xs \; (\text{suc } 0) \\ &= \text{length-tail } xs \; 1 \end{aligned}$$

We need to prove that $\text{suc } (\text{length } xs \equiv) \text{length-tail } xs \; 1$. This time, we cannot simply use refl, because both sides do not reduce to the same term. For this reason, we can proceed to call this function recursively with the tail of the list. This is justified because of the Curry-Howard correspondence, and the fact that we are making a proof by induction. The result of the recursive call gives us the induction hypothesis:

```
≡
lengthlength-tail (x :: xs) =
  let ind-h = ≡lengthlength-tail xs
   in ?
```

The type of ind-h is $\text{length } \equiv xs \text{length-tail } xs \; 0$. The left sides of the induction hypothesis and what we are proving are almost the same. To make them match, we can apply the *congruence* property of equality, which has the following type:

```
cong : ∀ (f : A → B) {x y} → x ≡
y → f x ≡ f y
```

Applying this function to the induction hypothesis, we get the function below:

```
≡
lengthlength-tail (x :: xs) =
  let ind-h = ≡lengthlength-tail xs
      suc-cong = cong suc ind-h
   in ?
```

The suc-cong term has the type:

```
suc (length xs) ≡
suc (length-tail xs 0)
```

As we can see the left sides match, so we can change our goal to prove that the right side of suc-cong is equal to the right side of the goal; by making use of the transitive property of equality, which has the following type in Agda:

```
trans : ∀ {x y z} → x ≡ y → y ≡
z → x ≡ z
```

Therefore, now our proof is:

```
≡
lengthlength-tail (x :: xs) =
  let ind-h = ≡lengthlength-tail xs
      suc-cong = cong suc ind-h
   in trans suc-cong ?
```

The type of the term required to fill the hole is:

```
suc (length-tail xs 0) ≡
length-tail xs 1
```

We need to "pull" the 1 from the accumulator somehow, and convert it to a suc call. We can extract this new goal into a helper function:

```
ℕ::→ℕ::→ℕ→ℕ::≡∀ℕ→≡::
length-pull : ∀ (xs : List A)→
                suc (length-tail xs 0) ≡
length-tail xs 1
```

We can try to prove this goal by straightforward induction over the list, but we reach a dead end:

```
length-pull [] = refl
length-pull (x :: xs) = ?
```

The base case is trivial, following the definitions of the function, both terms reduce to 1. The problem is the inductive case, which reduces as follows:

$$\begin{aligned}\mathrm{suc}\ (\mathrm{length\text{-}tail}\ \ (x::xs)\,0) &= \mathrm{suc}\ (\mathrm{length\text{-}tail}\ \ xs\ \ (\mathrm{suc}\ \ 0)) \\ &= \mathrm{suc}\ (\mathrm{length\text{-}tail}\ \ xs\ \ 1) \\ \mathrm{length\text{-}tail}\ \ (x::xs)\,1 &= \mathrm{length\text{-}tail}\ \ xs\ \ (\mathrm{suc}\ \ 1) \\ &= \mathrm{length\text{-}tail}\ \ xs\ \ 2\end{aligned}$$

So, we are left with the following goal, which is very similar to the one we started with:

```
suc (length-tail xs 1) ≡
length-tail xs 2
```

We could try to prove this proposition by straightforward induction too, but that would require to prove a similar proposition for the next values 2 and 3, and so on.

To solve this issue, we can use a *generalization* strategy to prove this inductive property [1]. The generalized property will allow us to vary the value of the accumulator in the different cases of the inductive proof, but we will need to introduce another variable for it. It is important to note that after processing the first n items of the list, we will get $n + \mathrm{length\text{-}tail}\ \ xs\ 0$ on the left side and $\mathrm{length\text{-}tail}\ \ xs\ n$ on the right one. Combining the generalization strategy and this fact, we can see that the property we have to prove is:

```
ℕ::→ℕ::→ℕ→ℕ::≡
length-pull-generalized :∀
      (xs : List A) (n p : ℕ)→
      n + length-tail xs p ≡
length-tail xs (n + p)
```

This function can be proved by induction over the list:

```
length-pull-generalized [] n p = refl
length-pull-generalized (x ::
xs) n p = ?
```

The base case is trivial, because replacing the xs argument with [], and following a single reduction step on both sides, the common term $n + p$ is reached.

The inductive case is more interesting. Reducing both sides of the equation proceeds as follows:

$$n + \mathrm{length\text{-}tail}\ \ (x::xs)\ p = n + \mathrm{length\text{-}tail}\ \ xs\ \ (\mathrm{suc}\ \ p)$$
$$\mathrm{length\text{-}tail}\ \ (x::xs)\,(n+p) = \mathrm{length\text{-}tail}\ \ xs\ \ (\mathrm{suc}\ \ (n+p))$$

We can see that we have pretty much the induction hypothesis, with the only difference being the accumulating parameter p. Nevertheless, as we have generalized the proposition, we can pick a value for p when using the induction hypothesis:

```
length-pull-generalized (x ::
xs) n p =
  length-pull-generalized xs n (suc p)
```

This takes us closer to the goal we want to prove. Unfortunately, we are left with the following goal after performing the substitution of p with suc p:

```
n + length-tail xs (suc p) ≡
length-tail xs (n + suc p)
```

This is almost what we want, except for suc $(n + p)$ not being equal to $n + \text{suc}\ p$. However, these two terms are indeed equal, but not definitionally, because the plus function is defined by induction on the first argument, and not on the second one:

```
_+_ : Nat → Nat → Nat
zero  + m = m
suc n + m = suc (n + m)
```

Therefore, applying reduction steps does not allow Agda to deduce the equality of these two terms. Fortunately, the fact that these terms are equal can be easily proved inductively as follows:

```
+-suc : ∀ m n → m + suc n ≡
suc (m + n)
+-suc zero    n = refl
+-suc (suc m) n = cong suc (+-suc m n)
```

The remaining step is to "replace" the suc $(n + p)$ term with $n + \text{suc}\ p$. Agda provides the rewrite construct to perform this transformation:

```
  ℕ::→ℕ::→ℕ→ℕ::≡∀ℕ→≡
length-pull-generalized (x :: xs) n p
  rewrite (sym (+-suc n p))
        = length-pull-generalized xs n (suc p)
```

We make use of the *symmetric* property of equality in the rewriting step, which allows us to flip the sides of the equality:

```
sym : ∀ {x y} → x ≡ y → y ≡ x
```

With all this in place, we can finally prove the remaining goals, giving as a result the complete proof:

```
  ℕ::→ℕ::→ℕ→ℕ::≡
length-pull-generalized :∀
     (xs : List A) (n p : ℕ)→
     n + length-tail xs p ≡
length-tail xs (n + p)
length-pull-generalized [] n p = refl
length-pull-generalized (x :: xs) n p
  rewrite (sym (+-suc n p))
        = length-pull-generalized xs n (suc p)

length-pull : ∀ (xs : List A)→
              suc (length-tail xs 0) ≡
length-tail xs 1
length-pull xs = length-pull-generalized xs 1 0≡

lengthlength-tail : ∀ (xs : List A)→
                    length xs ≡
length-tail xs 0≡
lengthlength-tail [] = refl≡
lengthlength-tail (x :: xs) =
  let ind-h = ≡lengthlength-tail xs
      suc-cong = cong suc ind-h
      suc-pull = length-pull xs
   in trans suc-cong suc-pull
```

3 Another example: list reverse

The list reversal function follows a similar pattern to the one we have seen before:

```
  ::
reverse : List A -> List A
reverse [] = []
reverse (x :: xs) = reverse xs ++ (x ::
[])

reverse-tail : List A -> List A -> List A
reverse-tail [] ys = ys
reverse-tail (x ::
xs) ys = reverse-tail xs (x :: ys)
```

It should not come as a surprise that the equality proof is very similar too:

```
  ::::::::≡
reverse-pull-generalized :∀
   (xs ys zs : List A)→
   reverse-tail xs ys ++ zs≡
        reverse-tail xs (ys ++ zs)
```

```
reverse-pull-generalized [] ys zs = refl
reverse-pull-generalized (x ::
xs) ys zs =
  reverse-pull-generalized xs (x ::
ys) zs

reverse-pull :∀
   (x : A) (xs : List A)→
   reverse-tail xs [] ++ (x :: [])≡
       reverse-tail xs (x :: [])
reverse-pull x xs =
  reverse-pull-generalized xs [] (x ::
[])≡

reversereverse-tail : ∀
(xs : List A)→
                      reverse xs ≡
reverse-tail xs []≡
reversereverse-tail [] = refl≡
reversereverse-tail (x :: xs) =
  let ind-h = ≡reversereverse-tail xs
      append-cong = cong (_++ (x ::
[])) ind-h
      append-pull = reverse-pull x xs
   in trans append-cong append-pull
```

There are minor variations in the function signatures and the order of the parameters, but the structure is identical:

- Start proving by induction on the list.
- Fill the base case with refl .
- Take the inductive hypothesis by using a recursive call.
- Apply *an operator* to both sides of the equality, using cong.
- Create a function to pull the accumulator, and prove it using a generalized version of this function that allows varying the accumulator.
- Compose the two equalities using the trans function.

4 Generalization

Starting from the function definitions, we can see that they follow the same recursive pattern, we can write this pattern in Agda, which is just a specialization of a fold function [9, 10]:

```
::≡→→→≡llrr
reduce : List A → R
reduce [] = empty
reduce (x :: xs) = f x <> reduce xs
```

where

- R is the result type of the function.
- empty is the term to return when the list is empty.
- f is a function to transform each element of the list into the result type.
- <> is the function to combine the current item and the recursive result.

In the case of the length function, the result type is $\mathbb{N}$, the natural numbers; empty is 0; the function to transform each element is a constant function that ignores its argument and returns 1; and the function to combine the current item and the result of the recursive call is the addition function.

For the reverse function, the result type is the same type as the original list, List A; empty is the empty list; the function to transform each element creates just a singleton list from its parameter; and the function to combine the current transformed item and the result of the recursive call, is the flipped concatenation function. The flipping is necessary to make the function concatenate its first argument to the right:

$$
\begin{aligned}
\mathrm{reduce}\ (x :: xs) &= (\lambda a \to a :: [])\ x <> \mathrm{reduce}\ xs \\
&= (x :: []) <> \mathrm{reduce}\ xs \\
&= (\lambda xs\ ys \to ys +\!+ xs)\ (x :: [])\ (\mathrm{reduce}\ xs) \\
&= \mathrm{reduce}\ xs\ +\!+ (x :: [])
\end{aligned}
$$

The functions that follow this pattern, can be defined in a tail-recursive way as follows:

```
::≡→→→≡llrr→::
reduce-tail : List A → R → R
reduce-tail [] r = r
reduce-tail (x ::
xs) r = reduce-tail xs (r <> f x)
```

We can check manually that this function matches the tail-recursive definition in the case of the reverse function:

$$
\begin{aligned}
\text{reduce-tail}\ (x :: xs)\ r &= \text{reduce-tail}\ \ xs\ \ (r\ <> (\lambda a \to a :: [])\ x)\\
&= \text{reduce-tail}\ \ xs\ \ (r\ <> (x :: [])\\
&= \ \ldots\ \ xs\ \ ((\lambda xs\ \ ys \to ys ++ xs)\ r\ (x :: []))\\
&= \text{reduce-tail}\ \ xs\ \ ((x :: []) ++ r)\\
&= \text{reduce-tail}\ \ xs\ \ (x :: r)
\end{aligned}
$$

Now we can proceed to prove that these two functions are extensionally equal in the general case. The proof follows the same pattern as the one for the length function:

```
::≡→→→≡llrr→::→→::∀→≡::∀→≡lr::ll≡
reducereduce-tail : ∀ (xs : List A)→
                          reduce xs ≡
reduce-tail xs empty≡
reducereduce-tail [] = refl≡
reducereduce-tail (x :: xs) =
  let ind-h = ≡reducereduce-tail xs
      op-cong = cong (f x <>_) ind-h
      op-pull = reduce-pull (f x) xs
   in trans op-cong op-pull
```

We make use of a piece of syntactic sugar called *sections*, which allows us to write the function λ(→rf x <> r) as (f x <>_). Apart from that, the proof is identical to the ones we have seen before.

However, to prove the accumulator pulling function, we need to use a different strategy. We are required to prove that:

```
::≡→→→≡llrr→::→→::∀→≡::
reduce-pull :∀
   (r : R) (xs : List A)→
   r <> reduce-tail xs empty≡
   reduce-tail xs (empty <> r)
```

To do this, we can prove this proposition by induction over the list, which requires us to prove the proposition when xs is []:

$$
\begin{aligned}
r\ <> \text{reduce-tail}\ \ []\ \ \text{empty} = r\ <> \text{empty}\\
\text{reduce-tail}\ \ []\ \ (\text{empty} <> r) = \text{empty} <> r
\end{aligned}
$$

So we are required to prove that r <> ≡emptyempty <> r. We could require the <> function to be commutative, but we can "ask for less" by just requiring empty to be a left and right identity for <>, this is expressed in Agda as:

```
<>-lidentity : ∀ (r : R) →
empty <> r ≡ r
<>-ridentity : ∀ (r : R) →
r <> empty ≡ r
```

This way, we can use those identities to rewrite our goals, and make them match over the term r, and then, complete the base case using the trivial equality proof refl:

```
::≡→→→≡llrr→::→→::∀→≡::∀→≡
reduce-pull r []
  rewrite <>-lidentity r
        | <>-ridentity r = refl
```

The inductive case goal is:

$$
\begin{aligned}
r\ <> \text{reduce-tail}\ \ (x :: xs)\ \text{empty}\\
= r\ <> \text{reduce-tail}\ \ xs\ \ (\text{empty} <> f\,x)\\
\text{reduce-tail}\ \ (x :: xs)\ (\text{empty} <> r)\\
= \text{reduce-tail}\ \ xs\ \ ((\text{empty} <> r) <> f\,x)
\end{aligned}
$$

Which cannot be proved directly by straightforward induction, as we have seen before, but at least we can simplify it by using the left identity property over empty <> f x and then over empty <> r:

```
::≡→→→≡llrr→::→→::∀→≡::∀→≡lr
reduce-pull r (x :: xs)
  rewrite <>-lidentity (f x)
        | <>-lidentity r
        = reduce-pull-generalized r (f x) xs
```

Finally, we just need to prove the generalized accumulation pulling function, which has the following type signature:

```
::≡→→→≡llrr→::→→::
reduce-pull-generalized :∀
  (r s : R) (xs : List A)→
  r <> reduce-tail xs s ≡
reduce-tail xs (r <> s)
```

Note that the base case is trivial, and it is quite similar to the ones we have already proved, so we are going to focus on the inductive case. Following the same kind of reductions we have been doing before, we can see that our goal is:

```
r <> reduce-tail (x :: xs) s
    = r <> reduce-tail xs (s <> f x)
reduce-tail (x :: xs) (s <> r)
    = reduce-tail xs ((r <> s) <> f x)
```

Following the generalization strategy, we have to call the function recursively, replacing the s by s <> f x, which almost gives what it is required, except that the right hand side accumulator is associated wrongly.

```
r <> reduce-tail xs (s <> f x)≡
    reduce-tail xs (r <> (s <> f x))
```

Associativity is indeed the last property that the <> function needs to satisfy. This can be expressed in Agda straightforwardly as:

```
<>-assoc : ∀ (r s t : R)→
           (r <> s) <> t ≡
r <> (s <> t)
```

Which helps us complete the proof:

```
::≡→→→≡llrr→::→→::∀→≡
reduce-pull-generalized r s [] = refl
reduce-pull-generalized r s (x :: xs)
  rewrite <>-assoc r s (f x)
      = reduce-pull-generalized r (s <> f x) xs
```

All of these properties match the definition of a monoid. We can complete the formalization and encapsulate it in a ready to use parametrized module, using the standard library definition of a monoid:

```
::≡
open import Algebra.Structures using (IsMonoid)

module GenericBasic
  {A : Set}
  {R : Set}
  (f : A → R)
  (_<>_ : R → R → R)
  (empty : R)
  (m : IsMonoid ≡__ _<>_ empty)
  where

open IsMonoid m using ()
  renaming ( ˡidentity to <>-ˡidentity
           ; ʳidentity to <>-ʳidentity
           ; assoc to <>-assoc
           )
```

5 Using the module with the examples

With the module in place, we can start using it to derive the recursive function, the tail-recursive counterpart, and the proof that both functions are extensionally equal.

The length function uses the usual sum monoid over the natural numbers:

```
ℕ
open import GenericBasic
  {A = ℕ} λ( _ →
1) _+_ 0 +-0-isMonoid
  renaming ( reduce to length
           ; reduce-tail to length-tail
           ; ≡reducereduce-tail to ≡lengthlength-
           )
```

The reverse function requires us to create an instance of a flipped monoid for ++, which can be done with the already defined properties for list concatenation, but flipping them when necessary.

```
  ℕℕλ→≡≡≡2::rl→≡
++-flipped-isMonoid {A} = record
  { isSemigroup = record
    { isMagma = record
      { isEquivalence = isEquivalence
      ; •-cong = 2cong (flip _++_)
      }
    ; assoc = λ x y z →
sym (++-assoc z y x)
    }
  ; identity = ++-ridentity , ++-lidentity
  }
```

Finally, the indices function also requires us to create a custom monoid. The original indices function specialized for lists of natural number is the following:

```
  ℕ≟::
indices : ℕ→ List ℕ→ List ℕ
indices n [] = []
indices n (x :: xs) with n ≟ x
... | yes _ = 0 ::
map suc (indices n xs)
... | no _ = map suc (indices n xs)
```

The monoid for this function has the following operation and identity element:

```
  ℕℕλ→≡≡≡2::rl→≡•2λ→rlℕλ→::≡≡rr2≡≡↔≡∀→≡→≡→≡
IndicesData : Set
IndicesData = ℕ × List ℕ

empty : IndicesData
empty = 0 , []

_<>_ : IndicesData → IndicesData →
IndicesData
(ln , ll) <> (rn , rl) =
  ln + rn , ll ++ map (ln +_) rl
```

6 Conclusions

A technique to prove extensional equality between a recursive function and its tail-recursive counterpart has been presented, along with an Agda module to generate the functions and the proof automatically from an arbitrary monoid. The tail-recursive function generally improves the time complexity of the original recursive function and opens the possibility of performing tail-call optimization by the compiler, leading to a more space efficient function execution [2, 13].

There are some caveats with this technique which are exemplified by the indices function. Even though the generated function avoids mapping over the entire recursive call result, it introduces inefficiency by doing nested concatenations to the left, which leads to quadratic time complexity. This could be solved by using higher order functions as the accumulating monoid [8], but proving the corresponding monoid laws will require to be able to transform extensional equality to intensional equality, which is not possible in Agda without using *cubical type theory* [5, 14], but that is out of the scope of this paper.

Further work can be done in order to generalize this result to arbitrary *recursive data types* and *recursion schemes* [10].

References

1. **Abdali, S. K. & Vytopil, J. (1984).** Generalization heuristics for theorems related to recursively defined functions. *Proceedings of the Fourth AAAI Conference on Artificial Intelligence*, pp. 1–5.
2. **Bauer, A. (2003).** Compilation of functional programming languages using gcc—tail calls. *Master's thesis, Institut für Informatik, Technische Universität München, Germany.*
3. **Bird, R. & De Moor, O. (1996).** The algebra of programming. *NATO ASI DPD*, pp. 167–203.
4. **Bird, R. S. (1984).** The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 6, No. 4, pp. 487–504.
5. **Botta, N., Brede, N., Jansson, P., & Richter, T. (2021).** Extensional equality preservation and verified generic programming. *Journal of Functional Programming*, Vol. 31.
6. **Bove, A., Dybjer, P., & Norell, U. (2009).** A brief overview of agda–a functional language with dependent types. *International Conference on Theorem Proving in Higher Order Logics*, Springer, pp. 73–78.
7. **Dybjer, P. (1994).** Inductive families. *Formal aspects of computing*, Vol. 6, No. 4, pp. 440–465.

8. **Hughes, R. J. M. (1986).** A novel representation of lists and its application to the function “reverse”. *Information processing letters*, Vol. 22, No. 3, pp. 141–144.
9. **Hutton, G. (1999).** A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, Vol. 9, No. 4, pp. 355–372.
10. **Meijer, E., Fokkinga, M., & Paterson, R. (1991).** Functional programming with bananas, lenses, envelopes and barbed wire. *Conference on functional programming languages and computer architecture*, Springer, pp. 124–144.
11. **Mu, S.-C., Ko, H.-S., & Jansson, P. (2008).** Algebra of programming using dependent types. *International Conference on Mathematics of Program Construction*, Springer, pp. 268–283.
12. **Pettorossi, A. & Proietti, M. (1993).** Rules and strategies for program transformation. *Formal Program Development*, pp. 263–304.
13. **Rubio-Sánchez, M. (2017).** *Introduction to recursive programming*. CRC Press.
14. **Vezzosi, A., Mörtberg, A., & Abel, A. (2021).** Cubical agda: A dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, Vol. 31.
15. **Wadler, P. (2018).** Programming language foundations in agda. *Brazilian Symposium on Formal Methods*, Springer, pp. 56–73.