Catamorphism Generation and Fusion Using Coq

Simon Robillard*,
*Univ Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France,
simon.robillard@univ-orleans.fr

***Abstract*—Catamorphisms are a class of higher-order functions that recursively traverse an inductive data structure to produce a value. An important result related to catamorphisms is the fusion theorem, which gives sufficient conditions to rewrite compositions of catamorphisms. We use the Coq proof assistant to automatically define a catamorphism and a fusion theorem according to an arbitrary inductive type definition. Catamorphisms are then used to define functional specifications and the fusion theorem is applied to derive efficient programs that match those specifications.**

Keywords*—*Program derivation; Category theory; Catamorphism; Fusion theorem; Interactive theorem prover; Coq

I. Introduction

Writing correct computer programs is not an easy task, and the need for efficiency makes it even harder. Even in declarative languages, that try to turn their focus on *what* the program does instead of *how*, the search for performance often leads to complex code, prone to hidden errors and difficult to maintain.

Program derivation is an old idea that answers this challenge [1], [2]. Starting from an "obviously correct" expression of a program, verified transformations are iteratively applied in order to obtain a semantically equivalent expression with improved efficiency. While this concept can be applied to any paradigm, it is particularly well suited to functional programming, in which one can write programs and reason about them in the same language. This constrasts with imperative programs, which can only be reasoned about with predicate logic, and often require more complex mathematical reasoning tools.

The Bird-Meertens formalism [3] provides an advanced theoretical framework for deriving functional programs. Based on category theory, it formalizes algebraic data types as objects equipped with operations characterized by universal properties [4]. These operations are used to build functional programs, while the universal properties lead to theorems that we can use to optimize said programs. An advantage of this approach is the fact that theorems are applicable to any algebraic data type. Another strong point is the relatively small number of theorems and their wide range of application.

One of the key concepts of the Bird-Meertens formalism is the notion of catamorphism, a higher-order function that recursively consumes an inductive data structure to produce a value. A strong result related to catamorphisms is the fusion theorem [5], which gives conditions under which the composition of an arbitrary function and a catamorphism can be rewritten as a single catamorphism. The idea driving this work is to automatically generate a catamorphism for any given type in a functional language, and to use the fusion theorem to optimize programs written using this function.

While automation of the derivation process is highly desirable (for example in an optimizing compiler), any non-trivial property of a function is undecidable in the general case, and therefore automated processes will always reach limits. In an attempt to get the best of both worlds, we use the interactive theorem prover Coq to conduct program derivations using catamorphisms and the fusion theorem. In this setting, users can rely on Coq for correctness and partial proof automation while still being able to provide valuable guidance to the system.

This paper is organized as follows. Section II gives a quick presentation of the Coq proof assistant, with a special attention to the features that are most relevant to this work. Section III defines the concept of catamorphism for an arbitrary type T. Section IV introduces the fusion theorem, while in section V we show how to use this theorem to prove functional equalities or to construct new functions. Section VI discusses related work, and section VII brings ideas for future work before concluding the paper.

II. A Brief Introduction to Coq

Coq [6] is a proof assistant based on the Curry-Howard correspondence [7] relating terms of a typed λ-calculus with proof trees of a logical system in natural deduction form. A logical formula is expressed as a type, and is said to be true if it is inhabited, i.e. there exists a term of that type. Proving a formula is accomplished by building a term of the correct type. The calculus behind Coq is the Calculus of (co)-Inductive Constructions, an extension of the initial Calculus of Constructions [8], [9].

From a more practical side, Coq can be seen as a functional programming language, close to OCaml or Haskell but with a richer type system. As a matter of fact, Coq is often used as follows: programs are developed in Coq and their properties are also proved in Coq. This is for example the case of the CompertC compiler, a compiler for the C language, implemented and certified using Coq [10].

A. Types and functions

An interesting feature of the Calculus of Constructions is that it does not distinguish between types and terms, meaning that types have their own types, in an infinitely ascending type hierarchy. Two types are built into the system kernel: `Prop` is meant to represent logical assertions while `Set` is reserved for values. Apart from those predefined types, there is the possibility to define algebraic types using the usual constructs.

The expressivity of the type system of Coq stems from its use of dependent products: this construct generalizes the arrow type $A \to B$ and allows one to define the type $\forall a : A,\ B$, where the value of B depends on a. This allows for rich type definitions, hence expressive formulae. Since types are terms, polymorphic types are just a particular case of dependent products.

Functions in Coq must be total, a fact that is somewhat balanced by the expressivity of types, as it is always possible to refine the domain of a function to make it total. Another approach is to use a sum type to provide a default return value. Functions must also be guaranteed to terminate to preserve the consistency of the system: since proofs are functions, a non-terminating function could be used as a proof for any formula. For functions where recursive calls are applied only to syntactic subterms of the argument, termination is automatically proven by the system. For more complicated cases, the user must provide a guaranty, such as a well-founded relation over input values.

B. Interactivity and automation

Coq also provides a proof mode; although it doesn't add to the expressivity of the system, this tool greatly facilitates the process of proving formulae. It allows the user to build proofs (terms) by using a sequential language of tactics, in a way similar to natural deduction. Although usually unnecessary, the proof mode can even be used to build terms of `Set`.

It is possible to combine tactics and even to write complex decision procedures, thanks to a dedicated tactic language [11]. For more advanced use cases, it is possible to use the Coq API to develop custom tactics in plugins written in OCaml. The same API offers many other functionalities to extend the capabilities of Coq. It should be noted that such OCaml plugins do not endanger the consistency of the system, which relies solely on type checking of terms and termination of functions.

Lastly, types and functions defined in Coq can be extracted to target languages such as OCaml and Haskell. In order to account for the simpler type systems of these languages, all elements of `Prop` are lost during the extraction process as well as some elements of dependent products. However this functionality allows verified programs conceived in Coq to be run efficiently on various platforms.

III. Datatypes and Catamorphisms

A. Generalized folds

Catamorphisms are arguably the most fundamental operation over an inductive type [12], as they can be used to define almost any function that operates by recursively traversing a data structure. They are a generalization of *fold*, the familiar higher-order function operating over lists. It is possible to build a similar higher-order function for any inductive data type, much in the same way that induction principles can be constructed for any type according to its constructors. Both catamorphisms and induction principles encapsulate the notion of recursion over an inductive type; indeed the definition of catamorphisms given below can be seen as a restricted version of the general induction principle that Coq automatically generates upon declaration of the type. While slightly less expressive than this induction principle, the function that we use is better suited to express program specifications and to derive efficient implementations.

In order to illustrate the usefulness of catamorphisms, let us give the definition of the catamorphism over a very simple inductive type, that of natural numbers $Nat ::= zero \mid succ\ Nat$. The catamorphism we get from this definition is a higher-order function with two arguments $f_1 : A$ and $f_2 : A \to A$. We shall use the notation $(\!|f_1, f_2|\!)_{Nat}$ to denote it, or simply $(\!|f_1, f_2|\!)$ if the type of the catamorphism is clear from the context. This function is defined by the recursive equations:

$$\begin{array}{lcl} (\!|f_1, f_2|\!)\ zero & = & f_1 \\ (\!|f_1, f_2|\!)\ succ\, n & = & f_2\ ((\!|f_1, f_2|\!)\ n) \end{array}$$

We can then define basic arithmetic operations as catamorphisms. The point-free notation we are using means that these are unary functions: one parameter is fixed while the actual argument is implicit.

- $+n = (\!|n, succ|\!)_{Nat}$
- $\times n = (\!|zero, (+n)|\!)_{Nat}$

We can give further examples of the usefulness of catamorphisms with the type of lists $List_A ::= nil \mid cons\ A\ List_A$. As the definition of this particular catamorphism is that of the familiar *fold* function, we do not recall it.

- $append\ l = (\!|l, cons|\!)_{List_A}$
- $length = (\!|zero, succ \circ outr|\!)_{List_A}$
- $sum = (\!|zero, +|\!)_{List_{Nat}}$
- $concat = (\!|nil, append|\!)_{List_{List_A}}$

The first function returns the result of appending a list l to the end of its argument list, while the second returns the length of its argument, thus giving an example of a catamorphism ranging over a different type. Note the use of $outr$, that returns the second element of a pair. It is one of the fundamental operations over product types; in this case its purpose is to ignore the actual elements of the list. The function sum returns the sum of a $List_{Nat}$ and $concat$ concatenates a list of lists into a single list.

Many more functions can be concisely specified as catamorphisms, and by combining such catamorphisms it is possible to specify non-trivial problems.

B. Definition for an arbitrary type

Having given a general idea of the concept, we will now define catamorphisms for any inductive type T with n constructors α_i. This definition is given in the context of the Calculus of Inductive Constructions, but can easily be translated to any language with algebraic types.

The catamorphism, ranging over polymorphic type A, takes n functions f_i as arguments. We shall refer to these arguments as the catamorphism parameters. Their types depend on those of the constructors:

$$\frac{\alpha_i : X_1 \to \cdots \to X_m \to T}{f_i : Y_1 \to \cdots \to Y_m \to A}$$

where $Y_j = A$ if $X_j = T$ and $Y_j = X_j$ otherwise. In addition to these parameters, the catamorphism takes an argument of type T, which will be refered to as the main argument. The body of the catamorphism is built by destructing the main argument as follows:

```
match $x$ with
| ···
| $\alpha_i\ x_1 \cdots x_p \implies f_i\ y_1 \cdots y_p$
| ···
end
```

where y_j is the result of a recursive call applied to x_j if $x_j : T$, or simply x_j otherwise. Termination of this function trivially follows from the fact that recursive calls are applied only to strict syntactic subterms of the main argument; it is automatically proven by the checker in Coq.

C. *Catamorphism over binary trees*

In order to illustrate this definition, we will show what function we obtain from the following type definition, which describes binary trees of natural numbers.

```
Inductive tree :=
  | null : tree
  | node : nat → tree → tree → tree.
```

The associated catamorphism has two parameters `fnull` and `fnode`. Its definition is given below; the keyword `fix` is similar to `fun` but indicates a recursive function, while the curly braces mean that argument `A` will be automatically infered whenever possible, lightening the notation.

```
Definition cata_tree
                {A : Type}
                (fnull : A)
                (fnode : nat → A → A → A) :=
  fix f (t : tree) : A :=
    match t with
    | null ⇒ fnull
    | node n l r ⇒ fnode n (f l) (f r)
    end.
```

And here are a few examples of instanciated catamorphisms:

```
Definition count : tree → nat :=
  cata_tree 0 (fun _ l r ⇒ 1 + l + r).

Definition height : tree → nat :=
  cata_tree 0 (fun _ l r ⇒ 1 + max l r).

Definition sum : tree → nat :=
  cata_tree 0 (fun n l r ⇒ n + l + r).

Definition flatten : tree → list nat :=
  cata_tree [] (fun n l r ⇒ l ++ n :: r).
```

D. *Parametric types*

While this example showcases catamorphisms for sum, product and recursive types, the Calculus of Inductive Constructions also includes *inductive parameters* and *real arguments* in type definitions.

The value of an inductive parameter is shared among all constructors. This is for example the case of the parameter `A` in this definition of polymorphic lists:

```
Inductive list (A : Type) : Set :=
| nil : list A
| cons : A → list A → list A.
```

On the contrary, the value of a real argument can vary across constructors. The type of length-indexed lists presented here features one real argument of type `nat`.

```
Inductive nlist : nat → Set :=
| nnil : list 0
| ncons : ∀n, A → list n → list (n + 1).
```

Inductive parameters can be dealt with by simply adding an argument to the catamorphism definition for each parameter of the type. For real arguments we use the same solution but the additional arguments must be a part of the fixpoint, since their values can change with every recursive call. Accordingly, recursive calls inside the body must have the correct arguments.

With this last feature, we are able to automatically generate a catamorphism for any inductive type definable in the context of the Calculus of Inductive Constructions. Catamorphism generation is however disabled for types in `Prop`. Elimination of arguments in `Prop` is only allowed for functions ranging over `Prop` (proofs), and there is little interest in using catamorphisms to build proofs.

A more difficult problem arises when one needs a catamorphism ranging over a dependent type, such as a function converting a list into a length-indexed list. For this we need to change the definition so that the output type $A : Type$ is replaced by $F_A : T \to Type$. The type of the catamorphism parameters must be modified accordingly. The resulting function is similar to the general induction principle that Coq automatically generates upon declaration of the inductive type. However this function is too cumbersome for most use cases and its type does not allow us to give a useful statement of the fusion theorem. We therefore chose to limit ourselves to catamorphisms ranging over non-dependent types.

E. *Other categorical morphisms*

Despite their important role, not every recursive program can be expressed as a catamorphism. Other categorical concepts can be used to formalize patterns of recursion: paramorphisms, anamorphisms and hylomorphisms. Each comes with its own universal properties and a dedicated fusion theorem [4].

One example of a function that does not fit the pattern of recursion that we have so far is the factorial function. It is possible, but awkward, to express it as catamorphism:

```
Definition factorial : nat → nat :=
  compose snd
    (cata_nat (0, 1)
      (fun x ⇒ let (n, m) := x
               in (n + 1, (n + 1) * m))).
```

Since catamorphisms do not use the recursive argument of the constructor (only the result of recursively applying some f to it), we have to artificially keep track of it by building a pair of natural numbers. Paramorphisms are a generalization of catamorphisms that can be used to express functions that recursively consume their argument, but also use its value [13]. Very little needs to be changed in our machinery to generate their definition: the body of the function is defined exactly as before, except that for each constructor argument $x_j : T$, the corresponding f is applied to both x_j itself and the result of a recursive call applied to x_j (therefore a paramorphism parameter has one more argument for each recursive constructor argument than the corresponding catamorphism parameter). We can now express the factorial function in a much more elegant manner.

```
Definition factorial' : nat → nat :=
 para_nat 1 (fun n m ⇒ (n + 1) * m).
```

The definition of the fusion theorem given in section IV can be adapted in the same way so that its conclusion applies to paramorphisms.

Anamorphisms, also called *unfolds*, are the converse of catamorphisms: from a value, they generate a recursive data structure. While not quite as natural a programming pattern, they can be used to provide elegant solutions to some problems [14]. Their implementation in a functional language requires the use of a boolean predicate as a stopping condition (or more generally, a n-valued function used to choose from the n constructors at each iteration). In a lazy language, anamorphisms can even be used to define functions that generate infinite objects. In the Calculus of Constructions however, inductive and co-inductive objects live in two separate worlds. Implementing anamorphisms in Coq would require that the user provide a termination guaranty for each anamorphism instance, thus making their use impractical. While the termination policy of Coq frees us from verifying some strictness conditions otherwise needed to apply fusion theorems, it also comes with some limitations to the expressivity of the functions we are able to define. Those same limitations prevent us from implementing hylomorphisms, functions that are constructed by composing a catamorphism and an anamorphism.

IV. Fusion Theorem

The fusion theorem (also called *promotion* by some authors) gives the conditions under which the composition of an arbitrary function h and a catamorphism can be rewritten as a single catamorphism. In real-world applications, such a rewriting leads to the program using one less intermediate data structure. At a minimum, this saves valuable allocation time, and in cases where the size of the intermediate data structure is larger than both the input and the output, this can reduce the asymptotic complexity of the function. This result has been called "the most useful tool in the arsenal of techniques for program derivation" [3]. While it can be seen as a single law in a categorical setting, we need to instantiate it for each inductive type for use in the context of our calculus. Here is how we define the statement of the theorem for an arbitrary type T:

$$H_1 \wedge \cdots \wedge H_n \implies h \circ (\![f_1, \cdots, f_n]\!)_T = (\![g_1, \cdots, g_n]\!)_T$$

The types of the n hypotheses H_i are again shaped according to the types of the constructors α_i.

$$\frac{\alpha_i : X_1 \to \cdots \to X_m \to T}{H_i : \forall x_1 \cdots x_m,\ g_i\ (y_1 \cdots y_m) = h\ (f_i\ x_1 \cdots x_m)}$$

where $y_j = h\ (x_j)$ if $X_j = T$ and $y_j = x_j$ otherwise.

Note that while the conclusion of the theorem above is written in point-free notation, we cannot use the same notation in Coq. The definition of equality in Coq is not extensional, meaning that it differentiates between two implementations of a same function. We could axiomatize the extensionality lemma $\forall x,\ f\ x = g\ x \implies f = g$, but it is preferable to simply use a pointwise notation and have the conclusion read

$$\forall t : T,\ h\ ((\![f_1, \cdots, f_n]\!)\ t) = (\![g_1, \cdots, g_n]\!)\ t$$

Having formulated the statement of the theorem (its type), we need to provide a proof term. It would be feasible to build the proof term directly, again using the type constructor to guide the formation of the term. However this would require reimplementing some functionalities already offered by tactics. Instead, the proof term is built internally by the plugin very much like a user would, using the proof mode and a combination of tactics. The proof of each instance of the theorem is straightforward and follows the same pattern: it is carried out inductively on t, each inductive case being concluded with the appropriate hypothesis H_i.

Our plugin for Coq provides the command `Catamorphism`. This command takes as only parameter the name of an inductive type `t`; it adds to the environment two new definitions, `cata_t` and `cata_t_fusion`, that correspond respectively to the catamorphism of this type and its fusion theorem. A similar command `Paramorphism` is also available.

V. Applications

A. Composition of maps

A simple example of a program optimization lemma is the following, stating that the composition of two maps can be rewritten as a single map:

$$\forall f\ g,\ map\ g \circ map\ f = map\ (g \circ f)$$

This lemma can be proven by inductive reasoning on the argument. However if we observe that both $map\ f$ and $map\ (g \circ f)$ are catamorphisms, it becomes clear that the fusion theorem is applicable here. To show how it is used, we go back to our earlier definition of the type `tree` and modify it slightly in order to make the type polymorphic, by adding an inductive parameter. The catamorphism generated from this new definition is itself polymorphic w.r.t. the type of `tree`. We can now define *map* over trees:

```
Definition map {A B} (f : A → B) :=
  cata_tree null (fun x l r ⇒ node (f x) l r).
```

We now want to solve the goal:

```
map g (map f t) = map (compose g f) t
```

Application of the fusion theorem yields two goals.

```
null = map g null
```

and

```
node ((compose g f) y0) (map g y1) (map g y2)
= map g (node (f y0) y1 y2)
```

Simplification of the terms reveals that these goals are trivial equalities, both solved in a single step by the `auto` tactic, thus completing the proof. One notable advantage of this proof is that it doesn't explicitely make use of inductive reasoning: the inductive part of the proof is in fact already contained in the fusion proof. This is worth mentioning since inductive reasoning is often too complex for automated provers.

B. *Maximum segment sum optimization*

A larger example of application of the fusion theorem is given in [12], in which a function is derived to solve the maximum segment sum problem: given a list of integers l, compute the maximum of the sums of all sublists of l. We were able to formalize this derivation in Coq using our tool[1], in a way that very closely follows the proof published in the paper. First we generate the catamorphism function for the type of polymorphic lists, and use it to specify the problem, using no less than eight different catamorphism instances in the definition. Multiple transformation lemmas are then proven, all using the fusion theorem. By application of these lemmas, the initial specification (a function requiring cubic time and multiple intermediate data structures) is turned into a linear time function with only one intermediate list.

This provides a compelling example of program derivation using catamorphism fusion. On a less positive note, it also shows the need for user guidance in this endeavour. While in some cases fusion conditions can be verified automatically, for others we need to use a few key facts about the catamorphism parameters. For example, at one point in the calculation, we use the fact that the data type used is a monoid. Thankfully the facts used in this case are relatively simple. Another obstacle to automation arises when applying the derivation steps. A cost model for functions would be useful for verifying that fusion is indeed beneficial, or for allowing the machine to choose from two possible expressions of a program. To further complicate matters, some steps that we apply do not improve efficiency, but are required anyway to conduct further rewriting. In some instances, a calculation might require applying a fusion lemma from right to left. It is not obvious why and when these steps should be applied, especially in the context of automated reasoning.

C. *Constructive approach*

So far we have only used the fusion theorem to prove the equality of two expressions. A perhaps more interesting use of the theorem is to build new expressions. For this we use the conditions of the theorem as equations, the solutions of which are the parameters of a catamorphism that constitutes a more efficient implementation of the original composed function. If we go back to composing maps, consider the following lemma:

$$\forall f\ g, \exists x_1\ x_2,\ map\ g \circ map\ f = (\![x_1, x_2]\!)$$

The logic of Coq being constructive, solving this lemma can only be accomplished by providing concrete instances of x_1 and x_2 that we can reuse to create an optimizing rule. Existential quantification is not a built-in element of the Calculus of Constructions, but it is represented by the following type:

```
Inductive ex
        (A : Type)
        (P : A → Prop) : Prop :=
  ex_intro : ∀x : A, P x → ex A P.
```

The notation `exists x, P x` stands in place of the term `ex A (fun (x : A) => P x)`. However, using this definition presents a problem : due to the fact that `ex` is defined in `Prop`, it is not possible to destruct the proposition to extract the witness after the proof of existence has been completed. The standard library include a similar definition in `Set`, named `sig`. This type is commonly used to define the subset of `A` that verifies `P`, but we can also use it to represent an existentially quantified proposition in a way that allows us to remember the witnesses provided during a proof. Our tool provides some functions, notations and tactics specifically designed to facilitate the construction and destruction of terms of this type to carry out constructive proofs.

We can now represent the existentially quantified goal above. To solve it, we will introduce *existential variables* in the proof mode and apply the fusion lemma, which leaves us once more with two goals to solve (presented here in a slightly simplified form):

```
?1 = null
```

and

```
?2 y1 y2 y3 = node (g (f y1)) y2 y3
```

The first is trivially solved, at which point the value `null` is associated to the existential variable `?1`. For the second goal we can see that `?2` must be a ternary function. Solving this goal without manually giving an instance of `?2` requires performing β-expansion. No standard tactic allows this, but the API provides all that is needed to write custom tactics. We provide for this purpose the tactic `beta`, which applies this simple inference rule to any goal of the form $f\ x = e$:

$$\frac{f = \lambda y.e[x := y]}{f\ x = e}$$

Three successive applications of this tactic to the goal presented above yield the following:

```
?2 = (fun x1 x2 x3 ⇒ (node (g (f x1)) x2 x3)
```

This goal is trivially solved, giving us one instance of `?2` and allowing us to build a new catamorphism, which happens to be precisely `map (compose g f)`. Even in this constructive version, we are able to completely automate the proof using a very simple script.

[1]Source code available at http://traclifo.univ-orleans.fr/SDPP/wiki/SyDRec

VI. Related Work

There have been other attempts to use the results of the Bird-Meertens formalism beyond pen-and-paper derivations, including for parallel software [15]. The use of a theorem prover to provide a concrete implementation of these concepts can also be seen in [16], where the author uses PVS to apply hylomorphism fusion. As this particular fusion does not necessitate verifying preconditions, it has been used in entirely automatic tools targeting the language Haskell [17], [18]. In [19], the author describes the implementation of this type of fusion in a compiler for pH, an implicitely parallel dialect of Haskell.

Long before it was formalized as a categorical concept, hylomorphism fusion could be seen in optimizing algorithms such as supercompilation [20], fold-unfold [1] and deforestation [21], [22].

VII. Conclusion and Future Work

Writing functional programs with categorical morphisms instead of using explicitly recursive functions provides several benefits. It improves software design (with short, modular code that emphasizes meaning over implementation details) while exposing strong algebraic properties of the algorithms. Thanks to fusion theorems, this approach does not hinder performance. On the contrary, it may be used to discover elegant and efficient algorithms.

We showed how to define a catamorphism function and a fusion theorem for any inductive data type, including those with dependent products in their definition; we also illustrated the use of catamorphisms to specify problems, the resulting programs being often very short and easy to understand. The fusion theorem can then be used to prove that another, more efficient catamorphism is extensionally equal to that specification; it can also be used to construct such a fast implementation by using its conditions as equations and solving them.

This process (either proving an equality or constructing a new function) can be entirely automated for simple cases. Complex optimizations do on the other hand necessitate human guidance, both to construct equality rules and to apply them.

Our tool could benefit from being extended with other results from the Bird-Meertens formalism for function calculations, e.g. regarding mutually recursive functions [23] or the representation of conditionals through distributive categories [24]. Another possible extension of this tool would be to allow programs to be specified as relations. Much theoretical work has been done in this direction [25] and Coq makes it possible to formalize relations. Many problems that are not easily specified as functions (non-deterministic problems, optimization problems, converses…) could then be studied. Translating the ideas of this calculs of relations into the world of Coq presents challenges, but it would enable the derivation of many interesting programs.

Acknowledgements

This work is partly supported by the PaPDAS project funded by ANR (France, ANR-2010-INTB-0205-02) and JST (Japan, 10102704).

References

[1] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 44–67, 1977.

[2] J. Backus, "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1977.

[3] R. Bird and O. De Moor, *The algebra of programming*. Prentice Hall, 1997.

[4] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," in *Functional Programming Languages and Computer Architecture*. Springer, 1991, pp. 124–144.

[5] G. Malcolm, "Data structures and program transformation," *Science of computer programming*, vol. 14, no. 2, pp. 255–279, 1990.

[6] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer, 2004.

[7] W. A. Howard, "The formulae-as-types notion of construction," in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin and J. R. Hindley, Eds. Academic Press, 1980, pp. 479–490.

[8] T. Coquand and G. Huet, "The calculus of constructions," *Information and computation*, vol. 76, no. 2, pp. 95–120, 1988.

[9] C. Paulin-Mohring, *Inductive definitions in the system Coq rules and properties*. Springer, 1993.

[10] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[11] D. Delahaye, "A tactic language for the system Coq," in *Logic for Programming and Automated Reasoning (LPAR)*, vol. 1955. Springer, 2000, pp. 85–95.

[12] J. Gibbons, "Calculating functional programs," in *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Springer, 2002, pp. 151–203.

[13] L. Meertens, "Paramorphisms," *Formal Aspects of Computing*, vol. 4, no. 5, pp. 413–424, 1992.

[14] J. Gibbons and G. Jones, "The under-appreciated unfold," in *International Conference on Functional Programming*. ACM, 1998, pp. 273–279.

[15] B. J. Alexander and A. L. Wendelborn, "Automated transformation of BMF programs," in *Workshop on Object Systems and Software Architectures (2004: Victor Harbor, South Australia)*, 2004.

[16] N. Shankar, "Steps towards mechanizing program transformations using PVS," *Science of Computer Programming*, vol. 26, no. 1, pp. 33–57, 1996.

[17] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi, "A calculational fusion system HYLO," 1997.

[18] F. Domínguez and A. Pardo, "Exploiting algebra/coalgebra duality for program fusion extensions," in *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*. ACM, 2011, p. 6.

[19] J. B. Schwartz *et al.*, "Eliminating intermediate lists in pH," Master's thesis, Massachusetts Institute of Technology, 1999.

[20] V. F. Turchin, "The concept of a supercompiler," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 3, pp. 292–325, 1986.

[21] P. Wadler, "Deforestation: Transforming programs to eliminate trees," *Theoretical computer science*, vol. 73, no. 2, pp. 231–248, 1990.

[22] S. D. Marlow, "Deforestation for higher-order functional programs," Ph.D. dissertation, University of Glasgow, 1995.

[23] M. M. Fokkinga, "Law and order in algorithmics," Ph.D. dissertation, University of Twente, 1992.

[24] J. Robin and B. Cockett, "Conditional control is not quite categorical control," in *IV Higher Order Workshop, Banff 1990*. Springer, 1991, pp. 190–217.

[25] R. C. Backhouse, P. Hoogendijk, E. Voermans, and J. van der Woude, "A relational theory of datatypes," Eindhoven University of Technology, Dept. of Mathematics and Computer Science, Tech. Rep., 1992.