Subsidiary Recursion in Coq

Aaron Stump
Computer Science Dept., The University of Iowa, USA

Alex Hubers
Computer Science, The University of Iowa, USA

Christopher Jenkins
Computer Science, The University of Iowa, USA

Benjamin Delaware
Computer Science, Purdue University, USA

Abstract

This paper describes a functor-generic derivation in Coq of subsidiary recursion. On this recursion scheme, inner recursions may be initiated within outer ones, in such a way that outer recursive calls may be made on results from inner ones. The derivation utilizes a novel (necessarily weakened) form of positive-recursive types in Coq, dubbed retractive-positive recursive types. A corresponding form of induction is also supported. The method is demonstrated through several examples.

2012 ACM Subject Classification Software and its engineering → Recursion; Software and its engineering → Polymorphism

Keywords and phrases strong functional programming, recursion schemes, positive-recursive types, impredicativity

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.

1 Introduction: subsidiary recursion

Central to interactive theorem provers like Coq, Agda, Isabelle/HOL, Lean and others are terminating recursive functions over user-declared inductive datatypes [5, 7, 9, 4]. Termination is usually enforced by a syntactic check for structural decrease. This structural termination is sufficient for many basic functions. For example, the well-known `span` function from Haskell's standard library (`Data.List`) takes a list and returns a pair of the maximal prefix satisfying a given predicate `p`, and the remaining suffix:

```
span :: (a -> Bool) -> [a] -> ([a],[a])
span _ []     =  ([], [])
span p (x:xs) = if p x
                then let (ys,zs) = span p xs in (x:ys,zs)
                else ([],x:xs)
```

The sole recursive call is `span p xs`, and it occurs in a clause where the input list is of the form `x:xs`. So the input to the recursive call is a subdatum of the input, and hence this definition is structurally decreasing. In the appropriate syntax, it can be accepted without additional effort by all the mentioned provers.

This paper is about a more expressive form of terminating recursion, called **subsidiary recursion**. While performing an outer recursion on some input `x`, one may initiate an inner recursion on `x` (or possibly some of its subdata), preserving the possibility of further invocations of the outer recursive function. Let us see a simple example. The function `wordsBy` (from `Data.List.Extra`) breaks a list into its maximal sublists whose elements do not satisfy a predicate `p`. For example, `wordsBy isSpace " good day "` returns `["good","day"]`; so `wordsBy isSpace` has the same behavior as `words` (from `Data.List`). Code is in Figure 1.

Interactive Theorem Proving 2022.
Editors: June Andronick and Leonardo da Moura; Article No. ; pp. :1–:8
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
wordsBy :: (a -> Bool) -> [a] -> [[a]]
wordsBy p [] = []
wordsBy p (hd:tl) =
  if p hd
  then wordsBy p tl
  else let (w,z) = span (not . p) tl in
        (hd:w) : wordsBy p z
```

Figure 1 Haskell code for `wordsBy`, demonstrating subsidiary recursion

The first recursive call, `wordsBy p tl`, is structural. But in the second, we invoke `wordsBy p` on a value obtained from another recursion, namely `span`. This is not allowed under structural termination, but will be permitted by subsidiary recursion as derived below.

1.1 Summary of results

This paper presents a functor-generic derivation of terminating subsidiary recursion and induction in Coq. We should emphasize that this is a derivation of this recursion scheme within the type theory of Coq. No axioms or other modifications to Coq of any kind are required. Based on this derivation, we present several example functions like `wordsBy`, and prove theorems about them. For example, we prove the expected property that the sublists returned by `wordsBy` consist of elements satisfying `not . p`. For another, we give a definition of run-length encoding as a subsidiary recursion using `span`, and prove that encoding and then decoding returns the original list. Our approach applies to the standard datatypes in the Coq library, and does not require switching libraries or datatype definitions.

An important technical novelty of our approach is a derivation of a weakened form of positive-recursive type in Coq. Coq (Agda, and Lean) restrict datatypes D to be strictly positive: in the type for any constructor of D, D cannot occur to the left of any arrows. Our derivation needs to use positive-recursive types, where D may occur to the left only of an even number of arrows. The restriction to strict positivity is enforced because full positive-recursive types are incompatible with Coq's type theory [3]. But we present a way to derive a weakened form that is sufficient for our examples (Section 4.1). The weakening is to require only that $F\ \mu$ is a retract of μ, where μ is the recursive type and $F\ \mu$ its one-step unfolding. Usually these types are isomorphic. Hence, we dub these **retractive-positive** recursive types. This weakening does have some negative consequences, but we will see how to handle them. Our definition of retractive-positive recursive types makes essential use of impredicate quantification, and hence cannot be soundly recapitulated in a predicative theory like Agda's.

We begin by summarizing the interface our derivation provides for subsidiary recursion (Section 2), and then see examples (Section 3). We next explain how the interface is actually implemented (Section 4), including our retractive-positive recursive types (Section 4.1). The interface for subsidiary induction is covered next (Section 5), and example proofs using it (Section 6). Related work is discussed in Section 7.

All presented derivations have been checked with Coq version 8.13.2, using command-line option `-impredicative-set`. The code may be found as release `itp-2022` (dated prior to the ITP 2022 deadline) at `https://github.com/astump/coq-subsidiary`.

2 Interface for subsidiary recursion

This section presents the interface our Coq development provides for subsidiary recursion.

2.1 The recursion universe

Our approach is within a long line of work using ideas from universal algebra and category theory to describe inductive datatypes and their recursion principles. On this approach, one describes transformations to be performed on data as *algebras*, which can then be *folded* over data. The simplest form of algebras, namely F-algebras, are morphisms from $F\ A$ to A, for carrier object A. From a programming perspective, an F-algebra is presented with $F\ A$, and must compute a value of type A.

Algebras for our subsidiary recursion are quite a bit more complex than this. First, for reasons we will explain further below, the carrier of the algebra will be an `X : Set -> Set`. Second, instead of being given just $F\ A$ as input, the algebra is presented with:

- a type `R : Set`
- a function `reveal : R -> C`, which reveals values of type `R` as really having *anchor type* `C`. This type `C` will either by the abstract type of some outer recursion, or else our development's version of the actual datatype one is recursing over.
- a function `fold : FoldT Alg R`, which allows one to initiate subsidiary recursions in which the anchor type is `R`. We will present its type `FoldT Alg R` below.
- a function `eval : R -> X R`, to use for making recursive calls, on any value of type `R`.
- and an *subdata structure* `d : F R`, where `F` is the signature functor for the datatype.

The algebra is then required to produce a value of type `F R`.

We use Coq inductive types for the signature functors `F` of various datatypes, thus enabling recursions to use Coq's pattern-matching on the subdata structure `d`. So the style of coding against this interface retains a similar feel to structural recursions. Unlike with structural termination, though, the interface here is type-based and hence compositional. As we will see, it supports nested and higher-order recursions.

As in previous work, we dub this interface a *recursion universe* [10]. As in other domains using the term "universe", we have an entity (here, `R`) from which one cannot escape by using the available operations (for other cases: ϵ_0 and ω^-, the physical universe and traveling at the speed of light). Staying in the recursion universe is good, because we may recurse (via `eval`) on any value of type `R`.

Some points must still be explained, particularly why `X` has type `Set -> Set`, and the definition of `FoldT`. Let us see these details next.

2.2 The interface in more detail

Let us consider two files from our development.

2.2.1 Subrec.v

This file is parametrized by a signature functor `F` of type `Set -> Set`. It provides the implementation of subsidiary recursion. Two crucial values are `Subrec : Set`, which is the type to use for subsidiary recursion; and `inn : F Subrec -> Subrec`, which is to be used as a constructor for that type. An important point, however, is that `Subrec.v` does not provide an induction principle based on `inn`. Induction is derived later (Section 5). `Subrec.v` makes

```
Inductive ListF(X : Set) : Set :=
| Nil : ListF X
| Cons : A -> X -> ListF X.

Definition inList : ListF List -> List := inn ListF.
Definition mkNil : List := inList Nil.
Definition mkCons (hd : A) (tl : List) : List := inList (Cons hd tl).
Definition toList : list A -> List.
Definition fromList : List -> list A.
```

Figure 2 Some basics from `List.v`, specializing the functor-generic derivation of subsidiary recursion to lists (`List.v`)

critical use of retractive-positive recursive types, to take a fixed-point based on `F`. We will present the definition of those in Section 4.1.

2.2.2 List.v

This file specializes the development in `Subrec.v` to the case of lists (parametrized by the type `A` of elements). In general, to use our development to get subsidiary recursion over some datatype, one will have a similar "shim" file. The file defines the signature functor `ListF`, shown in Figure 2. Using `Subrec`, we then get a type `List`. This is not to be confused with the type `list` of lists in Coq's standard library. As noted previously, our development is meant to be used in extension of existing inductive datatypes, not replacing them. The figure also shows constructors `mkNil` and `mkCons` for `List`, and types for conversion functions between `List` and `list` (see Section 4 for code).

2.3 Algebras for subsidiary recursion

`Subrec.v` also defines the notion of algebra that is used for writing recursions. The central definitions are in Figure 3. `KAlg` is the kind for the type-constructor for algebras, as we see in the very last definition of the figure, for `Alg`. This type-constructor `Alg` is a fixed-point of the type `AlgF`. The fixed-point is taken using `MuAlg` (Section 4.1 below). Doing so requires that `AlgF` only use its parameter `Alg` positively. We will confirm this shortly.

The type `FoldT Alg C` is the type for fold functions which apply algebras of type `Alg` to data of type `C`, which we have already dubbed the *anchor type* of the recursion. At the top level of code, the anchor type would just be `List` (for example). When one initiates a subsidiary recursion, though, the anchor type will instead by the abstract type `R` for the outer recursion.

The variable `Alg` occurs only positively (but not strictly positively) in `AlgF`, because it occurs negatively in `FoldT Alg R` which occurs negatively in `AlgF Alg C X`. So we can indeed take a fixed-point of `AlgF` to define the constant `Alg`.

Let us look at `AlgF`. As noted already, each recursion is based on an abstract type `R`, representing the data upon which we will recurse. This is the first argument to a value of type `AlgF Alg C X`. An algebra can assume nothing about `R` except that it supports the following operations. First there is `reveal`, which turns an `R` into a `C`. This reveals that the data of type `R` are really values of the datatype (`List`, for example), if this is a top-level recursion; or else really belong to some outer recursion universe, if this is an inner recursion. Next we have `fold`, which will allow us to fold another algebra over data of type `R`. We

```
Definition KAlg  : Type := Set -> (Set -> Set) -> Set.

Definition FoldT(alg : KAlg)(C : Set) : Set :=
  forall (X : Set -> Set) (FunX : Functor X), alg C X -> C -> X C.

Definition AlgF(Alg: KAlg)(C : Set)(X : Set -> Set) : Set :=
  forall (R : Set)
         (reveal : R -> C)
         (fold : FoldT Alg R)
         (eval : R -> X R)
         (d : F R),
         X R.

Definition Alg : KAlg := MuAlg AlgF.
```

Figure 3 The type for algebras (`Subrec.v`)

will use `fold` to initiate subsidiary recursions. Then there is `eval`, which is used to make recursive calls on data of type `R`.

As noted already, for subsidiary recursion, algebras have a carrier `X` which depends (functorially) on a type. This is so that (i) inside an inner recursion we may compute a result of some type that may mention `R`, but (ii) outside that recursion, the result will mention the anchor type `C`. The `eval` function returns something of type `X R`, and so does the algebra itself; this demonstrates (i). For (ii): if we look at the definition of `FoldT` in the figure, we see that folding an algebra of type `alg C X` over a value of type `C` produces a result of type `X C`. Having a functor for the carrier of the algebra gives us the flexibility to type results inside a recursion with the abstract type `R`, but view those results as having the anchor type `C` outside the recursion.

3 Examples of subsidiary recursion

4 Derivation of subsidiary recursion

4.1 Retractive-positive recursive types

As we have seen, our definitions require a form of positive-recursive types, to allow algebras to accept fold functions that themselves require algebras, and also for the definition of `Subrec`. But as recalled already, full positive-recursive types are incompatible with Coq's type theory [3]. It is worth noting that one can impose some restrictions on large eliminations which then allow positive-recursive types [2]. This approach would require changing the underlying theory. To avoid this, we here take a different approach, exploiting Coq's impredicative polymorphism.

This is done in a file `Mu.v`, whose central definitions are in Figure 4. The development is parametrized by `F : Set -> Set` which is assumed to have an `fmap` function (morphism part of the functor) of type

```
forall A B : Set, (A -> B) -> F A -> F B
```

which satisfies the identity-preservation law for functors:

```
Inductive Mu : Set :=
  mu : forall (R : Set), (R -> Mu) -> F R -> Mu.

Definition inMu(d : F Mu) : Mu :=
  mu Mu (fun x => x) d.

Definition outMu(m : Mu) : F Mu :=
  match m with
  | mu A r d => fmap r d
  end.

Lemma outIn(d : F Mu) : outMu (inMu d) = d.
```

Figure 4 Derivation of retractive-positive recursive types

```
fmapId : forall (A : Set)(d : F A), fmap (fun x => x) d = d
```

Let us consider the code in Figure 4. The critical idea is embodied in the definition of `Mu`. We would like to have a definition like

```
Inductive Mu' : Set := mu' : F Mu' -> Mu'.
```

This is exactly what is used in approaches to modular datatypes in functional programming, like Swierstra's [11]. But this definition is (rightly) rejected by Coq, as it would unsoundly enable instantiations of `F` that are not strictly positive.

Instead, the definition of `Mu` in Figure 4 weakens this ideal definition to a strictly positive approximation:

```
Inductive Mu : Set :=
  mu : forall (R : Set), (R -> Mu) -> F R -> Mu.
```

Instead of taking in `F Mu`, constructor `mu` accepts an input of type `F R`, for some type `R` for which we have a function of type `R -> Mu`. The impredicative quantification of `R` is essential here: we instantiate it with `Mu` itself in the definition of `inMu` (Figure 4). So this approach would not work in a predicative theory like Agda's. The quantification of `R` can be seen as applying a technique due to Mendler, of introducing universally quantified variables for problematic type occurrences, to a datatype constructor. We will review this in Section 7.

Returning to Figure 4, we have functions `inMu` and `outMu`, which make `F Mu` a retraction (`outIn`) of `Mu`: the composition of `outMu` and `inMu` is (extensionally) the identity on `F Mu`. But the reverse composition cannot be proved to be the identity, because of the basic problem of **noncanonicity** that arises with this definition.

For a simple example of noncanonicity, suppose we instantiate `F` with `ListF` (of Figure 2). Please note that as `Mu` is used in our derivation of subsidiary recursion, we will not instantiate this `F` with the signature functor of a datatype directly; but this will show the issue in a simple form. Let us temporarily define `List A` as `Mu (ListF A)` (again, for subsidiary recursion we use a different functor than just `ListF` directly). The canonical way to define the empty list would be, implicitly instantiating `F` to `ListF A`,

```
Definition mkNil := mu (List A) (fun x => x) (NilF A)
```

But given this, there are infinitely many other equivalent definitions. For any `Q : Set`, we could take

```
Definition mkNil' := mu Q (fun x => mkNil) (NilF A)
```

Since `fmap f (NilF A)` equals just `NilF B` for `f : A -> B`, if we apply `outMu` (of Figure 4) to `mkNil'` or `mkNil`, we will get `NilF (List A)`. But critically, `mkNil` and `mkNil'` are not equal, neither definitionally nor provably. One can define a function that puts `Mu` values in normal form by folding `inMu` over them. Then `mkNil` and `mkNil'` will have the same normal form, and be equivalent in that sense. But the fact that they are not provably equal is what we term noncanonicity.

Noncanonicity leads to some issues, as we turn next to the problem of inductive reasoning about subsidiary recursions. With some care, however, we can avoid pitfalls, leaving us with a form of positive-recursive type that enables our definitions to go through.

5 Interface for subsidiary induction

6 Examples of subsidiary induction

7 Related Work

7.1 Termination

In some tools, like Coq, Agda, and Lean, termination is checked statically, based on structural decrease at recursive calls. Others, like Isabelle/HOL, allow one to write recursions first, and prove (possibly with automated help) their termination afterwards [6].

It has not escaped the notice of designers of ITPs that structural recursion is not the only form of terminating recursion. All the mentioned tools provide support for well-founded recursion, where for recursive calls, one must show that the parameter of recursion has decreased in some well-founded order.

Subsidiary recursion can be seen as a generalization of *nested recursion*, which allows recursive calls of the form `f (f x)`. In subsidiary recursion, these are generalized to the form `f (g x)`, where `g` could be `f` or another recursively defined function.

7.2 Mendler encoding

Mendler introduced the basic idea of using universal abstraction to support compositional termination checking; an accessible source is [8]. This recursor has type

$$\forall X. (\forall R. (R \to X) \to F\ R \to X) \to \mu\ F\ \to X$$

We have adopted this idea to the constructor of the type `Mu` (Section 4.1). Previous work explored the categorical perspective on Mendler-style recursion [12]. Others have explored the possibility of using it with negative type schemes [1].

References

1. Ki Yung Ahn and Tim Sheard. A hierarchy of mendler style recursion combinators: Taming inductive datatypes with negative occurrences. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 234–246, New York, NY, USA, 2011. ACM.
2. Frédéric Blanqui. Inductive types in the calculus of algebraic constructions. *Fundam. Informaticae*, 65(1-2):61–86, 2005. URL: `http://content.iospress.com/articles/fundamenta-informaticae/fi65-1-2-04`.

3 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988. URL: `https://doi.org/10.1007/3-540-52335-9_47`, `doi:10.1007/3-540-52335-9\_47`.

4 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. URL: `https://doi.org/10.1007/978-3-030-79876-5_37`, `doi:10.1007/978-3-030-79876-5\_37`.

5 The Agda development team. *Agda*, 2021. Version 2.6.2.1. URL: `https://agda.readthedocs.io/en/v2.6.2.1/`.

6 Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL*. URL: `https://isabelle.in.tum.de/doc/functions.pdf`.

7 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2021. Version 8.13.2. URL: `http://coq.inria.fr`.

8 N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1):159 – 172, 1991.

9 Wolfgang Naraschewski and Tobias Nipkow. Isabelle/hol, 2020. URL: `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`.

10 Aaron Stump, Christopher Jenkins, Stephan Spahn, and Colin McDonald. Strong functional pearl: Harper's regular-expression matcher in cedille. *Proc. ACM Program. Lang.*, 4(ICFP):122:1–122:25, 2020. URL: `https://doi.org/10.1145/3409004`, `doi:10.1145/3409004`.

11 Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008. URL: `https://doi.org/10.1017/S0956796808006758`, `doi:10.1017/S0956796808006758`.

12 Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic J. of Computing*, 6(3):343–361, September 1999.