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 of a recursion scheme supporting the initiation of subsidiary recursions, in Coq. The approach supports making outer recursive calls on the results of subsidiary recursions. The derivation utilizes a novel (necessarily weakened) form of positive-recursive types in Coq. Subsidiary induction is also supported. Using this derivation, several examples are demonstrated.

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 [2, 4, 5, 1]. 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`. The input to the recursive call is a subdatum of the input for that clause, 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 recursion. Let us see a simple example. The function `wordsBy` (`Data.List.Extra`) breaks a list into the list of 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` (`Data.List`). Code is in Figure 1. The first recursive call,

```
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

`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.

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. (So 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` contain no elements satisfying the predicate `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. The 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. Full positive-recursive types are incompatible with Coq's type theory. But we present a way to derive a weakened form that is sufficient for our examples (Section 6.1). The weakening is to require just a retraction between recursive type μ and its one-step unfolding $F\ \mu$, instead of an isomorphism. Hence, we dub these **retractive-positive** recursive types.

We begin by summarizing the interface our derivation provides for subsidiary recursion (Section 2), and then see examples (Section 3). The interface for subsidiary induction is covered next (Section 5), and example proofs using it (Section **??**). We conclude with a walk-through of the derivation of that first interface (for subsidiary recursion), in Section 6. Related work is discussed in Section 7.

All presented derivations have been checked with Coq version 8.13.2. 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 purposes of presenting the interface for subsidiary recursion, let us consider three files in our codebase:

- `Subrec.v`: parametrized by a signature functor `F` of type `Set -> Set`, this provides

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

Definition inList : ListF List -> List := inn ListF.
Definition outList : List -> ListF List := out ListF (fold 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

```
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 interface for algebras

(among much else, to be discussed below) `Subrec : Set`, as the type to use for subsidiary recursion; and also a constructor `inn : F Subrec -> Subrec` for that type.

- `List.v`: parametrized by the type `A : Set` of list elements, this file specializes the development in `Subrec.v` to the case of lists. This is done using defining the signature functor `ListF`, shown in Figure 2, giving us the type `List`. This is not to be confused with the type `list` of lists in Coq's standard library. The figure also shows constructors `mkNil` and `mkCons` for `List`, and types for conversion functions between `List` and `list` (code omitted).
- `Span.v`: this implements the `span` function mentioned above, as an *algebra*.

2.1 Algebras for subsidiary recursion

Our approach is within a long line of work using ideas from universal algebra and category theory to implement inductive datatypes and recursion principles, in type theory. On this approach, one describes transformations to be performed on data as *algebras*, which can then be *folded* over data.

The interface for algebras is presented in Figure 3. Let us consider the definitions. `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`, to be discussed in Section 6.1 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`. At the top level of code, `C` would just be `List` (for example). When one initiates a subsidiary recursion, though, this type `C` will instead by what we like to call the *recursion universe* of that recursion. Each recursion is based on an abstract type `R`, representing the data upon which we will recurse. Indeed, `AlgF` says that algebras take in such an `R` as their first argument. This means that an algebra can assume nothing about `R` except that it has the operations given next in the listing of `AlgF`. Thus, `R` functions like a universe for recursion: one may perform the various given operations, without leaving the type `R` (which is good, because we may recurse on elements of type `R`).

And what are the operations on `R`? First there is `reveal`, which turns an `R` into a `C`. This reveals that the data of type `R` are really lists (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 will use `fold` to initiate subsidiary recursions. Then there is `eval`, which is used to make recursive calls on data of type `R`.

This is a good place to highlight that for subsidiary recursion, it is crucial that 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 type `C`. The `eval` function returns something of type `X R`, demonstrating (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`.

3 Examples of subsidiary recursion

4 Interface for subsidiary induction

5 Examples of subsidiary induction

6 Derivation of subsidiary recursion

6.1 Retractive-positive recursive types

7 Related Work

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 [3].

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 is a form of recursion allowing recursive calls of the form `f\ (f\ x)`. In subsidiary recursion, these are generalized to the form `f\ (g\ x)`, where `g` is another recursively defined function.

References

1 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`.

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

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

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

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