Abstract

This paper describes a functor-generic derivation in Coq of subsidiary recursion. With 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

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 [8, 14, 17, 7].
Termination is usually enforced by a syntactic check for structural decrease, which is sufficient
for many basic functions. For example, the span function from Haskell’s prelude (Data.List)
takes a list and returns a pair of the maximal prefix whose elements satisfy a given predicate
p, and the remaining suffix:

```coq
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. Hence it is structurally decreasing. In the appropriate syntax, this definition 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
(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"]. Code
is in Figure 1. Recall that break p is equivalent to span (not . p). The first recursive call,
wordsBy p tl, is structural. But in the second, we invoke wordsBy p on a value obtained
Subsidiary Recursion in Coq

wordsBy :: (a -> Bool) -> [a] -> [[a]]

wordsBy p [] = []
wordsBy p (hd:tl) =
  if p hd
  then wordsBy p tl
  else let (w,z) = break p tl in
      (hd:w) : wordsBy p z

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

from another recursion, namely span. This is not allowed under structural termination, but
is permitted by subsidiary recursion.

### 1.1 Summary of results

This paper presents a functor-generic derivation of terminating subsidiary recursion and
induction in Coq. We emphasize that this is a derivation within the type theory of Coq,
and requires no axioms or other modifications to Coq, except the -impredicative-set flag.
Using this derivation, we present several example functions like wordsBy, and prove theorems
about them. A nice example is a definition of run-length encoding using span as a subsidiary
recursion, where we prove that encoding and then decoding returns the original list. Our
approach does not require switching libraries or datatype definitions.

An important technical novelty 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 input
types of constructors 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 of an even number (only)
of arrows. We present a way to derive a weakened form of positive-recursive type that is
sufficient for our examples (Section 4.1). The weakening is to require only that $F (\mu F)$ is a
retract of $\mu F$. Usually, these types are isomorphic. Hence, we dub these retractive-positive
recursive types. This weakening leads to noncanonical elements of $\mu$, but we will see how to
work around this. Our definition of retractive-positive recursive types makes essential use of
impredicative quantification, and hence is not legal in predicative theories 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. The code may be
found as release itp-2022 (dated prior to the ITP 2022 deadline) at https://github.com/
astump/coq-subsidiary. The paper references files in this codebase, as an aid to the reader
wishing to peruse the code.

## 2 Interface for subsidiary recursion

This section presents the interface our Coq development provides for subsidiary recursion.
Definition List := Subrec ListF.
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 parametrized by an element type A (List.v)

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 (cf. [22, 5, 11]). With 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 for functor F (called the signature functor of the datatype), are morphisms from FA to A, for carrier object A. From a programming perspective, an F-algebra is given input of type FA, and must compute a result of type A. An example of F is the signature functor for lists, parametrized by the type A of elements:

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

Algebras for our subsidiary recursion are more complex than F-algebras. Let us begin with an informal explanation. For reasons we will explain further below, the carrier of the algebra will be a functor X : Set -> Set. The algebra is presented with:

- a type R : Set, which will be this recursion’s view of the datatype.
- a function fold : FoldT Alg R, which allows one to initiate subsidiary recursions over data of type R. We will present the type FoldT Alg R below.
- a function rec : R -> X R, to use for making recursive calls, on any value of type R.
- and a 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 X R.

We will use Coq inductive types for the signature functors F of various datatypes. This allows algebras 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 recursion. Unlike with structural termination, though, the interface here is type-based and hence compositional.

We have previously dubbed interfaces for recursion recursion universes [20]. As in other domains using the term “universe”, we have a kind of space (here, R), which one cannot escape using certain operations. Other examples are the ordinal ϵ₀ and ω−, and the physical universe and traveling at the speed of light. Staying in the recursion universe is good, because we may recurse (via rec) on any value of type R. Some points must still be explained: why X has type Set -> Set, and the definition of FoldT. Let us see these details next.

2.2 Types for subsidiary recursion (Subrec.v, List.v)

The type over which one can recurse using our scheme of subsidiary recursion is called Subrec. It is parametrized by a signature functor F of type Set -> Set. Subrec comes with
Definition KAlg : Type := (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)(X : Set -> Set) : Set :=
  forall (R : Set)
    (fold : FoldT Alg R)
    (rec : R -> X R)
    (d : F R),
  X R.

Definition Alg : KAlg := MuAlg AlgF.

Definition fold : FoldT Alg Subrec.
Definition rollAlg : forall {X : Set -> Set}, AlgF Alg X -> Alg X.
Definition unrollAlg : forall {X : Set -> Set}, Alg X -> AlgF Alg X.

Figure 3 The type for algebras, parametrized over F : Set -> Set (Subrec.v)

inn : F Subrec -> Subrec, which behaves computationally like a constructor. We will
later derive an induction principle for this type (Section 5). The definition of Subrec uses
retractive-positive recursive types, to take a fixed-point of a construction based on F. We
present these recursive types in Section 4.1 below.

In this paper, our examples use ListF A (shown above) to instantiate the parameter F.
List is then defined to be Subrec. In general, to use our development to get subsidiary
recursion over some datatype, one must define a signature functor for the datatype. Note that
List is different from the type list of lists in Coq's standard library. 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 typings for conversion functions
between List and list (definitions elided).

2.3 Algebras for subsidiary recursion (Subrec.v)

Let us now look more formally at the notion of algebra we introduced informally above. The
central definitions are in Figure 3. KAlg is the kind for the type-constructor for algebras, as
we see in the definition of Alg. This type-constructor Alg is a fixed-point of the type AlgF.
The fixed-point is taken using MuAlg, which implements our retractive-positive recursive
types (Section 4.1) at kind KAlg.

We need a fixed-point here since Alg occurs in the definition of AlgF. This is an essential
circularity, because we are trying to express that algebras take in fold functions, which
themselves may accept algebras. The variable Alg occurs negatively in FoldT Alg R which
occurs negatively in AlgF Alg X. Hence it occurs positively in AlgF, though not strictly
positively. So we can indeed take a fixed-point of AlgF to define the constant Alg.

Let us look further 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 X. Reasoning parametrically, an algebra can assume nothing about R
except that it supports the operations that follow. We have a local fold function, which will
Theorem FoldChar :
  forall (X : Set -> Set) (FunX : Functor X) (IdF : FmapId X FunX)
  (algf : AlgF Alg X) (d : F Subrec),
  fold X FunX (rollAlg algf) (inn d) =
  algf _ fold (fold X FunX (rollAlg algf)) d .

Figure 4 Computation law for subsidiary recursion, stated as a theorem

allow us to fold another algebra over data of type R. We will use fold to initiate subsidiary
recursions. Then there is rec, for recursive calls on data of type R. Finally, we have the
subdata structure d : F R.

As noted already, for subsidiary recursion, algebras have a carrier X which depends
(functorially) on a type. When we fold an algebra using a fold function (either global or local)
of type FoldT Alg C, (i) recursive calls may compute a result of type X R, mentioning the
abstract type R for that recursion; and (ii) outside that recursion, the result will have 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 with type C outside the recursion.
The function fold (Figure 3) initiates top-level folds. We also have functions rollAlg
and unrollAlg between Alg and its Algf-unfolding. We will fill in missing definitions (for
Subrec, inn, fold, etc.) in Section 4.

Finally, for a recursion scheme, one would like to see not just the typed interface, but
also the computation law. This is shown as a theorem in Figure 4. Intuitively, it states that
fold ing an algebra over constructed data inn d is equal to invoking the algebra on fold for
the rec function; an invocation of fold with the algebra for the rec function; and d for the
subdata structure. (FmapId expresses the identity law for functors, needed for the proof.)

3 Examples of subsidiary recursion

Having seen the interface for subsidiary recursion in Coq, let us consider now some examples.

3.1 The span function (Span.v)

Our first example is span discussed above. It does not invoke subsidiary recursions, but will
be used as a subsidiary recursion in later examples. Given a predicate p : A -> bool, and a
value of type List A, we would like to compute a pair of type list A * List A, where the
first component is the maximal prefix whose elements satisfy p, and the second is the remaining
suffix. This is the typing for a top-level recursion. More generally, though, given a type
R : Set along with a fold function for that type (i.e., of type FoldT (Alg (ListF A)) R),
we will map an input list of type R to a pair of type list A * R. The first component of this
pair is going to be built up from scratch, and so cannot have type R; we cannot statically
ensure that outer recursions on it would be legal. But the second component will always be
a subdatum of the input list, and so can still have type R. The typing is:

Definition spanr(R : Set)(fo:FoldT (Alg (ListF A)) R)
  (p : A -> bool)(xs : R) : list A * R.

From this we can also define the top-level recursion, by supplying fold (ListF A), which is
the function for folding an algebra over a list (Figure 3), for the argument fo of spanr:
**Subsidiary Recursion in Coq**

**Definition** SpanAlg(p : A -> bool) : Alg (ListF A) SpanF :=

\[
\text{rollAlg (fun R fo span xs =>}
\]

\[
\begin{align*}
\text{match xs with} \\
\text{Nil => SpanNoMatch} \\
| \text{Cons hd tl =>} \\
\text{if p hd then} \\
\text{match (span tl) with} \\
\text{SpanNoMatch => SpanSomeMatch [hd] tl} \\
| \text{SpanSomeMatch l r => SpanSomeMatch (hd::l) r} \\
\text{end} \\
\text{else} \\
\text{SpanNoMatch} \\
\end{align*}
\]

end).

**Figure 5** The algebra SpanAlg for the span function (Span.v)


Before we define spanr, we must resolve a small problem. If the first element of the input list xs to span does not satisfy p, then span should return ([], xs). But when recursing on xs, we will see it only in the form of a subdata structure of type ListF A R. We will not be able to return it from our recursion at type R, and hence we would not be able to return ([], xs) as desired. To work around this, we will have our recursion return a value of the following type SpanF R (with X implicit for the constructors):

**Inductive** SpanF(X : Set) : Set :=

SpanNoMatch : SpanF X
| SpanSomeMatch : list A -> X -> SpanF X.

The idea is that the recursion will return SpanNoMatch to signal that it is in the one tricky case where p does not match the first element. Otherwise, it will be able to return, via SpanSomeMatch, a prefix and the suffix at type R. The prefix will be nonempty, and hence the suffix will be at most the tail of xs. This suffix is available to the algebra in the subdata structure of type ListF A R.

**3.1.1 The algebra for span**

Figure 5 shows the algebra SpanAlg, whose type is Alg (ListF A) SpanF. We are defining an algebra (Alg) for the ListF A functor, with carrier SpanF of the required type Set -> Set. We use rollAlg to create an algebra from something whose type is an application of AlgF. This takes in all the components of the recursion universe: the abstract type R, the fold function fo for any subsidiary recursions (not needed here), a function we choose to name span for making recursive calls, and finally xs : ListF A R. The algebra pattern-matches on this xs. In the cases where it is empty or where its head (hd) does not satisfy p, we return SpanNoMatch. This signals to the caller that we really wished to return ([], xs), but could not because we do not have xs at type R. If the head does satisfy p, then we recurse on the tail (tl : R) by calling the provided span : R -> SpanF R. If span tl returns SpanNoMatch, that means that we should make tl the suffix in the pair we return (via SpanSomeMatch).
Definition spanhr(R : Set)(fo:FoldT (Alg (ListF A)) R)
   (p : A -> bool)(xs : R) : SpanF R :=
   fo SpanFunctor (SpanAlg p) xs.

Definition spanr(R : Set)(fo:FoldT (Alg (ListF A)) R)
   (p : A -> bool)(xs : R) : list A * R :=
   match spanhr fo p xs with
   SpanNoMatch => ([],xs)
   | SpanSomeMatch l r => (l,r)
   end.

Definition breakr(R : Set)(fo:FoldT (Alg (ListF A)) R)
   (p : A -> bool)(xs : R) : list A * R :=
   spanr fo (fun x => negb (p x)) xs.

Figure 6 Functions derived from SpanAlg (Span.v)

Happily, we have tl : R here, so we can do this. For either possible return value of span tl,
we add the head to the front of the prefix.

3.1.2 Defining span from SpanAlg

SpanAlg is used in the definition of spanhr, in Figure 6. This function invokes the fold
function it is given, on SpanAlg. The final twist is now in the definition of spanr. We call
spanhr on the input xs : R. If spanhr returns SpanNoMatch, then we are supposed to return
([],xs), which we can do here, because we have xs : R. It was only inside the algebra that
we lost the information that the subdata structure of type F R is derived from a value of
type R. If spanhr returns SpanSomeMatch l r, then we return the nonempty prefix (l) and
the suffix (r). We also define a version breakr for subsidiary recursion.

3.2 The wordsBy function (WordsBy.v)

We now consider how to write the wordsBy function from Section 1, using breakr subsidiarily.
The code is in Figure 7, assuming a type A : Set. The setup is similar to that for span.
We first define an algebra WordsByAlg of type Alg (ListF A) (Const (list (list A))),
parametrized by a predicate p. This type expresses that WordsByAlg p is an algebra (Alg)
for the ListF A functor, with carrier Const (list (list A)). Const is a combinator for
creating the object part of constant functors; FunConst creates the morphism part (i.e., the
fmap function). We use Const where the return type of the algebra will not depend on its
abstract type R. Since we are constructing a list of lists from scratch, it will not be legal
to recurse on the list itself, or its (list) elements. So we just use the list type of Coq’s
standard library.

The code for WordsByAlg is essentially the same as wordsBy in Haskell, which we saw
in Section 1. Recall that we must drop elements that satisfy p, and return the list of
sublists between maximal sequences of such elements. The algebra pattern-matches on
xs : ListF A R. In the Cons case, if the head (hd) satisfies the predicate, then we drop
it and recurse. Legality of the recursive call follows by typing, since tl : R has the input
type required by wordsBy : R -> list (list A). Otherwise, we use breakr to obtain the
Subsidiary Recursion in Coq

Definition WordsByAlg(p : A -> bool) : Alg (ListF A) (Const (list (list A))) :=
rollAlg (fun R fo wordsBy xs =>
  match xs with
  Nil => []
  | Cons hd tl =>
    if p hd then
    wordsBy tl
    else
    let (w,z) := breakr fo p tl in
    (hd :: w) :: wordsBy z
  end).

Definition wordsByr{R : Set}(fo:FoldT (Alg (ListF A)) R)
(p : A -> bool)(xs : R) :
list (list A) :=
fo (Const (list (list A))) (FunConst (list (list A))) (WordsByAlg p) xs.

Definition wordsBy(p : A -> bool)(xs : List A) :
list (list A) :=
wordsByr (fold (ListF A)) p xs.

Figure 7 The algebra WordsByAlg, and functions wordsBy and wordsByr folding it (WordsBy.v)

| mapThrough : (a -> [a] -> (b, [a])) -> [a] -> [b] |
| mapThrough f [] = [] |
| mapThrough f (a:as) = b : mapThrough f as' |
| where (b, as') = f a as |

Figure 8 The mapThrough function in Haskell

maximal prefix w of tl that does not satisfy p, and the remaining suffix z.
Here we see the benefit of our approach. From Figure 6, the return type of breakr is
list A * R, where R comes from the type FoldT (ListF A) Alg R of fo. This means that
from the invocation of breakr, we get w : list A and z : R. Thus, it is legal to apply
wordsBy : R -> list (list A) to z to recurse. The figure also shows the code for the
subsidiary recursion wordsByr and top-level recursion wordsBy.

3.3 The mapThrough function (MapThrough.v)

This example shows how to write a combinator that factors out a subsidiary recursion.
The Haskell library Data.List.Extra defines a function repeatedly in essentially the same
way as mapThrough in Figure 8 (we propose a more informative name). This function behaves
like the standard map function on lists, except that the function f that we are mapping (or
“mapping through”) takes in not just the current element a, but also the tail as. It then
returns the value b to include in the output list, and whatever sublist it wishes, upon which
mapThrough will then recurse.
To write this combinator using our infrastructure for subsidiary recursion, we will use
this type for mapped functions:

Definition mappedT(A B : Set) : Set :=
forall(R : Set)(fo:FoldT (Alg (ListF A)) R), A -> R -> B * R.
Definition \texttt{MapThroughAlg}\{B : \text{Set}\}(f:\text{mappedT A B})
: \text{Alg (ListF A} \ (\text{Const (list B)})\) :=
\text{rollAlg (fun R fo mapThrough xs =>}
\text{match xs with}
\text{Nil => []}
| \text{Cons hd tl} =>
\text{let (b,c) := f R fo hd tl in}
\text{b :: mapThrough c}
\text{end).}

Definition \texttt{mapThroughr}\{R : \text{Set}\}(fo:FoldT (\text{Alg (ListF A)}) R)
{B : \text{Set}\}(f:\text{mappedT A B}) : R \rightarrow \text{list B}.

Definition \texttt{mapThrough}\{B : \text{Set}\}(f:\text{mappedT A B}) : \text{List A} \rightarrow \text{list B}.

\textbf{Figure 9} The algebra \texttt{MapThroughAlg} defining the functions \texttt{mapThrough} and \texttt{mapThroughr}; code for the latter is omitted, as it follows the pattern of \texttt{wordsBy} and \texttt{wordsByr} of Figure 7 (\texttt{MapThrough.v})

\texttt{rle :: Eq a => [a] -> [(Int,a)]}
\texttt{rle = mapThrough compressSpan}
\texttt{where compressSpan a as =}
\texttt{let (p,s) = span (== a) as in}
\texttt{((1 + length p, a),s)}

\textbf{Figure 10} Run-length encoding in Haskell, using \texttt{mapThrough} and \texttt{span}

This \texttt{mappedT} type is more informative than the Haskell type, since it shows that the second component of the returned value must have type \(R\), and hence must be (hereditarily) a tail of the input. We need to supply mapped functions with the fold function to use, which will come from \texttt{mapThrough}'s recursion. Mapped functions need this to initiate subsidiary recursions, returning a value in the abstract type \(R\) of \texttt{mapThrough}'s recursion.

Given this definition, Coq code for \texttt{mapThrough} is shown in Figure 9. \texttt{MapThroughAlg} is similar to the Haskell code in Figure 8, though when we call \(f\), we must supply the abstract type \(R\) and fold function \(fo\). From the definition of \texttt{mappedT}, we have that \(b : B\) and \(c : R\), so we may indeed invoke \texttt{mapThrough} : \(R \rightarrow \text{list B}\) on \(c\). Note that as we are building up a new list from scratch (rather than just extracting some tail of the input list), we just return \texttt{list B}; we cannot perform further subsidiary recursion on the output.

3.4 \textbf{Run-length encoding (Rle.v)}

Finally, we have an example using our \texttt{mapThrough} combinator together with a subsidiary recursion, to implement \textit{run-length encoding}. This is a basic data-compression algorithm where maximal sequences of \(n\) occurrences of element \(e\) are summarized by the pair \((n, e)\) [19].

A Haskell implementation of this algorithm is in Figure 10. Recall that \((== a)\) tests its input for equality with \(a\). The \texttt{compressSpan} helper function gathers up all elements at the start of the tail as that are equal to the head \(a\). This prefix is returned as \(p\), with the remaining suffix as \(s\). The pair \((1 + \text{length } p, a)\) is returned to summarize \(a : : p\). The \texttt{mapThrough} combinator then iterates \texttt{compressSpan} through the suffix \(s\).

Assuming \(A : \text{Set}\) and an equality test \(eqb : A \rightarrow A \rightarrow \text{bool}\) on it, we port this code to our Coq infrastructure in Figure 11. The function \texttt{compressSpan} is written at the type
Definition compressSpan : mappedT A (nat * A) :=
  fun R fo hd tl =>
    let (p,s) := spanr fo (eqb hd) tl in
    ((succ (length p),hd), s).

Definition RleCarr := Const (list (nat * A)).
Definition RleAlg : Alg (ListF A) RleCarr :=
  MapThroughAlg compressSpan.
Definition rle(xs : List A) :=
  fold (ListF A) RleCarr (FunConst (list (nat * A))) RleAlg xs.

Figure 11 The function rle for run-length encoding, and the algebra RleAlg defining it in terms of MapThroughAlg of Figure 9 (Rle.v)

mappedT A (nat * A) that will be required by mapThrough. Unfolding the definition of mappedT, we see that compressSpan has this type:

forall(R : Set)(fo:FoldT (Alg (ListF A)) R), A -> R -> (nat * A) * R.

It is invoked by the code for mapThrough with fo : FoldT (Alg (ListF A)) R. Then compressSpan will extract from the tail at type R (second input) the suffix upon which mapThrough should recurse (second component of the output pair). Then we define an algebra RleAlg by supplying compressSpan as the function to map through, to MapThroughAlg (Figure 9). Following the pattern seen above, we define function rle for top-level recursions using fold (we could also define a subsidiary version rler).

4 Derivation of subsidiary recursion

Let us now consider the implementation of the interface we have used for the preceding examples. The first step is our weakened form of positive-recursive types.

4.1 Retractive-positive recursive types (Mu.v)

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 (which we will see in more detail in the next section). Full positive-recursive types are incompatible with Coq’s type theory [6]. One can impose some restrictions on large eliminations which then enable positive-recursive types [3], but this requires changing the underlying theory. Here we derive a different solution.

Our starting point is a type scheme F : Set -> Set, with 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:

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

Then we make the definitions of Figure 12. The critical idea is embodied in the definition of Mu. Ideally, we would like to use this definition:

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

XX:10 Subsidiary Recursion in Coq
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 12 Derivation of retractive-positive recursive types (\(\text{Mu}.v\))

This is exactly what is used in many approaches to modular datatypes in functional program-
ning, like Swierstra’s [21]. But this definition is (rightly) rejected by Coq, as instantiations
of \(F\) that are not strictly positive would be unsound.

Instead, we define \(\text{Mu}\) in Figure 12, to weaken this ideal \(\text{Mu}'\) to a strictly positive
approximation. Instead of taking in \(F\ \text{Mu}\), the constructor \(\text{mu}\) accepts an input of type
\(F\ R\), for some type \(R\) for which we have a function of type \(R -> \text{Mu}\). The impredicative
quantification of \(R\) is essential here: we will instantiate it with \(\text{Mu}\) itself in the definition
of \(\text{inMu}\) (Figure 12). 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 12, we have functions \(\text{inMu}\) and \(\text{outMu}\), which make \(F\ \text{Mu}\) a retraction
\((\text{outIn})\) of \(\text{Mu}\): the composition of \(\text{outMu}\) and \(\text{inMu}\) is (extensionally) the identity on \(F\ \text{Mu}\).

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 \(\text{ListF} A\) (from
Section 2.1). Our derivation actually uses a different type that wraps \(F\), but using \(\text{ListF} A\)
demonstrates the issue in a simple form. Let us temporarily define \(\text{List} A\) as \(\text{Mu}\) (\(\text{ListF} A\))
(again, for subsidiary recursion do not use just \(\text{ListF}\) directly). The canonical way to define
the empty list would be:

\[
\text{Definition mkNil} := \text{mu (List A) (fun x => x) (NilF A)}
\]

But given this, there are infinitely many other equivalent definitions. For any \(Q : \text{Set}\), we
could take

\[
\text{Definition mkNil'} := \text{mu Q (fun x => mkNil) (NilF A)}
\]

Since \(\text{fmap f (NilF A)}\) equals \(\text{NilF B}\) for \(f : A -> B\), if we apply \(\text{outMu}\) (of Figure 12) to
\(\text{mkNil'}\) or \(\text{mkNil}\), we will get \(\text{NilF (List A)}\). But critically, \(\text{mkNil}\) and \(\text{mkNil'}\) are not equal,
neither definitionally nor provably. Of course, one could define a function that puts \(\text{Mu}\) values
in canonical form by folding \(\text{inMu}\) over them. Then \(\text{mkNil}\) and \(\text{mkNil'}\) would be equivalent.

But they would still not be provably equal, which is the problem of noncanonicity. We will
see how to work around this in Section 6. First, though, let us complete the exposition of
our implementation of subsidiary recursion.
Subsidiary Recursion in Coq

Definition SubrecF(C : Set) :=
  forall (X : Set -> Set) (FunX : Functor X), Alg X -> X C.
Definition Subrec := Mu SubrecF.
Definition roll : SubrecF Subrec -> Subrec.
Definition unroll : Subrec -> SubrecF Subrec.

Figure 13 Definition of Subrec as a fixed-point of SubrecF (Subrec.v)

4.2 The implementation of Subrec (Subrec.v)

The type Subrec is defined in Figure 13, as a fixed-point of SubrecF : Set -> Set. We build this fixed-point using Mu from the previous section, and obtain roll and unroll functions between SubrecF Subrec and Subrec. The type SubrecF Subrec is definitionally equal to

forall (X : Set -> Set) (FunX : Functor X), Alg X -> X Subrec

So Subrec is the type of functions which, for all algebras with functorial carrier X, compute a value of type X Subrec. This is a generalization of the functor-generic type ∀ X. Alg X -> X for the Church encoding, where Alg X is F X -> X. We elide the implementation of the roll and unroll functions, but we note that unroll makes use of functoriality of carriers X.

The rest of the interface for Subrec is shown in Figure 14. To fold an algebra alg with carrier X (with fmap function given by FunX) over d : Subrec, we unroll d and apply that to the algebra (with its carrier).

More interesting is the definition of inn, which is the critical point where the recursion universe is implemented. To create a value of type Subrec from data of type F Subrec, the definition of inn rolls a value of type SubrecF Subrec (we saw this type unfolded at the start of this section). This value takes in a carrier X, its fmap function xmap, and an algebra alg with that carrier. It will then call alg (after unrolling it) with implementations for the components of the recursion universe (cf. Section 2.1, also Figure 3):

= Subrec is passed as the value for the abstract type R; this is what enables all the rest of the components to have the desired types, since we will pass values that have Subrec where the interface mentions R.

= The function fold : FoldT Alg Subrec is passed as the fold function of type FoldT Alg R.

= For the rec : R -> X R function, we pass (fold X xmap alg) : Subrec -> X Subrec.

= For the subdata structure of type F R, we pass d : F Subrec.

Finally, Figure 14 defines out as a subsidiary recursion, given a fold function. Outside the recursion, d has type F R; inside the recursion it has type F R' where R' is the abstract type of the subsidiary recursion. Intuitively, out implements the idea that unfolding an abstract type one step is just a trivial case of subsidiary recursion.

5 Interface for subsidiary induction (Subreci.v)

We have seen how to write subsidiary recursions in Coq. But can one reason about these? We turn now briefly to our interface for subsidiary induction in Coq, and some example proofs written using this interface. Subsidiary induction is the natural extension of subsidiary recursion, which worked over Sets, to Subrec-predicates. The development is parametrized
Definition fold : FoldT Alg Subrec :=
  fun X FunX alg d => unroll d X FunX alg.

Definition inn : F Subrec -> Subrec :=
  fun d => roll (fun X xmap alg =>
                 unrollAlg alg Subrec fold (fold X xmap alg) d).

Definition out{R: Set}(fo:FoldT Alg R) : R -> F R :=
  fo F FunF (rollAlg (fun R' _ d => d)).

Figure 14 The rest of the interface for Subrec (Subrec.v)

by a functor F and a functor Fi : (Subrec -> Prop) -> (Subrec -> Prop) over Subrec-indexed propositions (i.e., predicates). Just as functors need an fmap function, here we need an indexed version, of type fmapIT Subrec Fi (definition elided.)

The central definitions for the type Subreci : Subrec -> Prop are given in Figure 15. Where having a value x of Subrec entitles us to define subsidiary recursions to inhabit types X Subrec, a value of type Subreci x lets us prove properties of x by subsidiary induction. Briefly: \$kMo\$ is the kind for motives, namely predicates on Subrec [15]. \$\text{KAlg}_i\$ is the kind for indexed algebras. \$\text{FoldTi}\$ is the indexed version of \$\text{FoldT}\$: it expresses provability of \$X C\$ for \$d\$, based on an indexed algebra and a value of type \$C d\$. \$\text{AlgFi}\$ and \$\text{Algi}\$ are indexed versions of the algebras we saw for recursion. The \$\text{rec}\$ function from Figure 3 is now an induction hypothesis: given any \$d\$ where \$R d\$ holds, \$\text{ih}\$ proves \$X R d\$. A value of type \$R d\$ is thus a license to induct on \$d\$. Finally, the algebra is given a subdata structure indexed by \$d : \text{Subrec}\$, and must produce a proof of \$X R d\$. \$\text{Subreci}\$ is defined as the suitably indexed fixed-point of \$\text{SubrecFi}\$, which is the natural indexed version of \$\text{SubrecF}\$.

For lists, we instantiate \$\text{Fi}\$ with \$\text{ListFi}\$, shown in Figure 16. This is just the indexed version of \$\text{ListF}\$. Given a list \$A\$, \$\text{toListi}\$ returns a value of type \$\text{Listi} (\text{toList} \text{xs})\$. This can be understood as saying that for any list (from Coq’s standard library), we can reason by subsidiary induction to prove properties of \$\text{toList} \text{xs}\$. We also introduce an abbreviation \$\text{ListFoldTi}\$ for the type of indexed fold functions over lists.

6 Examples of subsidiary induction

To prove the main theorem about run-length encoding, we need the three lemmas about \$\text{span}\$ shown in Figure 17. For lack of space, we just state the properties. The first says that appending the results of a call to \$\text{span}\$ returns the original list (module some conversions to \$\text{list}\$ from \$\text{List}\$). The second uses the inductive proposition \$\text{Forall}\$ from Coq’s standard library to state that all the elements of the prefix returned by \$\text{span}\$ satisfy \$p\$. These lemmas are proved using indexed algebras with constant (indexed) carriers. In contrast, \$\text{GuardPresF}\$ uses its argument \$S\$ to express that whenever \$\text{spanh}\$ returns a suffix \$r\$, that suffix satisfies \$S\$. This enables us to invoke an outer induction hypothesis on this suffix, when reasoning subsidiarily about \$\text{span}\$. Using these lemmas, we can write a short proof by subsidiary induction of the following theorem, where \$\text{rld} : \text{list} (\text{nat} \times A) -> \text{list} A\$ is the obvious decoding function:

Theorem RldRle (xs : list A) : rld (rle (toList xs)) = xs.
Definition kMo := Subrec -> Prop.
Definition KAlgi := (kMo -> kMo) -> Set.
Definition FoldTi(alg : KAlgi)(C : kMo) : kMo :=
  fun d => forall (X : kMo -> kMo) (xmap : fmapiT Subrec X),
  alg X -> C d -> X C d.

Definition AlgFi(A: KAlgi)(X : kMo -> kMo) : Set :=
  forall (R : kMo)
  (fo : (forall (d : Subrec), FoldTi A R d))
  (ih : (forall (d : Subrec), R d -> X R d))
  (d : Subrec),
  Fi R d -> X R d.

Definition Algi := MuAlgi Subrec AlgFi.

Definition SubrecFi(C : kMo) : kMo :=
  fun d => forall (X : kMo -> kMo) (xmap : fmapiT Subrec X), Algi X -> X C d.
Definition Subreci := Mui Subrec SubrecFi.

Definition foldi(i : Subrec) : FoldTi Algi Subreci i.
Definition inni(i : Subrec)(fd : Fi Subreci i) : Subreci i.

Figure 15 Interface for subsidiary induction (Subreci.v)

Definition lkMo := List -> Prop.
Inductive ListFi(R : lkMo) : lkMo :=
  nilFi : ListFi R mkNil
  | consFi : forall (h : A)(t : List), R t -> ListFi R (mkCons h t).

Definition Listi := Subreci ListF ListFi.
Definition toListi(xs : list A) : Listi (toList xs) := listFoldi xs Listi inni.

Definition ListFoldTi(R : List -> Prop)(d : List) : Prop :=
  FoldTi ListF (Algi ListF ListFi) R d.

Figure 16 The indexed version ListFi of ListF (List.v)
Definition SpanAppendF(p : A -> bool)(xs : List A) : Prop :=
  forall (l : list A)(r : List A),
  span p xs = (l,r) ->
  fromList xs = l ++ (fromList r).

Definition spanForallF(p : A -> bool)(xs : List A) : Prop :=
  forall (l : list A)(r : List A),
  span p xs = (l,r) ->
  Forall (fun a => p a = true) l.

Definition GuardPresF(p : A -> bool)(S : List A -> Prop)(xs : List A) : Prop :=
  forall (l : list A)(r : List A),
  spanh p xs = SpanSomeMatch l r ->
  S r.

Figure 17 Statements of three lemmas about span (directory SpanPfs)

Definition spanForall2F(p : A -> bool)(xs : List A) : Prop :=
  Forall (fun a => p a = true) (fromList xs) ->
  span p xs = (fromList xs, getNil xs).

Figure 18 A statement of the property that span returns the empty suffix, computed using getNil to avoid noncanonicity problems, if all elements satisfy p

We invoke the lemmas about span subsidiarily, so that we may apply our induction hypothesis to the suffix that span returns (on which mapThrough then recurses). For example, the lemma for GuardPresF takes in the indexed fold function foi from the outer induction (for RldRle), to show that the abstract predicate R applies to the suffix r returned by span. This enables the outer induction hypothesis (for RldRle) to be applied.

Lemma guardPres(R : List A -> Prop)(foi:forall d : List A, ListFoldTi R d)
  (p : A -> bool)(xs : List A)(rxs : R xs)
  (l: list A)(r : List A)(e: span p xs = (l,r)) : R r.

Finally, as promised, a note on noncanonicity. When proving properties about subsidiary recursions on xs : List A, one should be aware that nothing prevents the property from being applied to noncanonical Lists. For example, suppose we wish to prove that if all elements of a list satisfy p, then the suffix returned by span is empty. It is dangerous to phrase this as “the suffix equals mkNil”, because for a noncanonical input xs, span will return that same noncanonical xs as the suffix (and so it may be a noncanonical empty list, not equal to mkNil). The solution in this case is to use a function getNil (List.v) that computes an empty list from xs. The statement that one can prove is shown in Figure 18.

7 Related Work

Termination. In some tools, like Coq, Agda, and Lean, termination is checked statically, based on structural decrease. Others, like Isabelle/HOL, allow one to write recursions first, and prove (possibly with automated help) their termination afterwards [12]. These tools all support well-founded recursion, but in constructive type theory, evidence of well-foundedness
then propagates through code. Our approach, while less general, does not clutter code with proofs. Subsidiary recursion can be seen as a generalization of nested recursion, which allows recursive calls of the form \( f (f x) \) [13]. In subsidiary recursion, these are generalized to the form \( f (g x) \), where \( g \) could be \( f \) or another recursively defined function. See the survey by Bove et al. for more on partiality and recursion in theorem provers [4].

Our work contributes to the program proposed by Owens and Slind, of broadening the scope of functional programs that can be accommodated in ITPs [18]. The goal of terminating recursion has been advocated in the literature on programming languages under the name strong functional programming [23]. Our method is similar to the technique of sized types, in providing a type-based method for termination [2]. With sized types, datatypes are indexed with abstract sizes, which must then be propagated through code, using dependent types. In contrast, our approach relies just on polymorphism, and does not require dependent types for writing subsidiary recursions. (Subreci, for reasoning about such recursions, of course does use dependent types).

Uustalu and Vene developed a categorical view of a recursion scheme allowing one level of subsidiary recursion, and illustrated it in Haskell with an artificial example [25]. In contrast, our scheme allows arbitrary finite nestings of recursion, and we illustrate it in Coq with realistic examples. It seems that generalizing the carriers of algebras to functors is the critical step enabling such examples.

Mendler-style recursion. Mendler introduced the idea of using universal abstraction to support compositional termination checking [16]. He proposed a functor-generic recursor of type \( \forall X. (\forall R. (R \to X) \to F R \to X) \to \mu F \to X \). We have applied this idea to the constructor of the type \( Mu \) (Section 4.1). Previous work explored the categorical perspective on Mendler-style recursion, and showed how to reduce it to basic catamorphisms (i.e., structural recursion) [24]. Another considered its use with negative type schemes [1]. Previous work from our group showed how to derive inductive datatypes in Cedille using extensions of the Mendler encoding [9, 10]. Here, we do not derive inductive types, but rather a terminating recursion scheme for existing datatypes.

8 Conclusion

We have seen a derivation in Coq of a scheme for terminating subsidiary recursion, where recursions may be nested and outer recursive calls may be made on the results of inner recursions. We saw examples invoking the \texttt{span} function as a subsidiary recursion, for functions \texttt{wordsBy} and run-length encoding. We also looked briefly at the extension of this interface to support subsidiary induction, with example lemmas about \texttt{span}, and the decoding correctness theorem for run-length encoding. There are many other interesting examples we can develop in Coq with this interface, including natural-number division, which may invoke subtraction as a subsidiary recursion. Another example is Harper’s regular-expression matcher, which previous work showed can be implemented in Cedille using a form of nested recursion that is subsumed by subsidiary recursion [20]. We may also attempt to extend the recursion universe further, to allow other forms of recursion like divide-and-conquer, where some (necessarily limited) ability to recurse on values built using constructors is required.


