







# 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. 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

**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 [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:

```
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



© Aaron Stump, Alex Hubers, Christopher Jenkins, and Benjamin Delaware;  
licensed under Creative Commons License CC-BY 4.0

Interactive Theorem Proving 2022.

Editors: June Andronick and Leonardo da Moura; Article No. ; pp. 1–18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## XX:2 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

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

### 41 1.1 Summary of results

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

49 An important technical novelty is a derivation of a weakened form of positive-recursive  
50 type in Coq. Coq (Agda, and Lean) restrict datatypes  $D$  to be strictly positive: in the input  
51 types of constructors of  $D$ ,  $D$  cannot occur to the left of any arrows. Our derivation needs  
52 to use positive-recursive types, where  $D$  may occur to the left of an even number (only)  
53 of arrows. We present a way to derive a weakened form of positive-recursive type that is  
54 sufficient for our examples (Section 4.1). The weakening is to require only that  $F(\mu F)$  is a  
55 retract of  $\mu F$ . Usually, these types are isomorphic. Hence, we dub these **retractive-positive**  
56 recursive types. This weakening leads to noncanonical elements of  $\mu$ , but we will see how to  
57 work around this. Our definition of retractive-positive recursive types makes essential use of  
58 impredicative quantification, and hence is not legal in predicative theories like Agda's.

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

64 All presented derivations have been checked with Coq version 8.13.2. The code may be  
65 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  
66 wishing to peruse the code.

## 68 2 Interface for subsidiary recursion

69 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  $F A$  to  $A$ , for carrier object  $A$ . From a programming perspective, an  $F$ -algebra is given input of type  $F A$ , 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  $X : \mathbf{Set} \rightarrow \mathbf{Set}$ . The algebra is presented with:

- a type  $R : \mathbf{Set}$ , 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` :  $R \rightarrow X R$ , 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  $\epsilon_0$  and  $\omega^-$ , 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  $\mathbf{Set} \rightarrow \mathbf{Set}$ , 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  $\mathbf{Set} \rightarrow \mathbf{Set}$ . `Subrec` comes with

## XX:4 Subsidiary Recursion in Coq

```
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  $F : \text{Set} \rightarrow \text{Set}$  (Subrec.v)

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

105 In this paper, our examples use `ListF A` (shown above) to instantiate the parameter `F`.  
106 `List` is then defined to be `Subrec`. In general, to use our development to get subsidiary  
107 recursion over some datatype, one must define a signature functor for the datatype. Note that  
108 `List` is different from the type `list` of lists in Coq’s standard library. Our development is  
109 meant to be used in extension of existing inductive datatypes, not replacing them. The figure  
110 also shows constructors `mkNil` and `mkCons` for `List`, and typings for conversion functions  
111 between `List` and `list` (definitions elided).

### 112 2.3 Algebras for subsidiary recursion (Subrec.v)

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

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

123 Let us look further at `AlgF`. As noted already, each recursion is based on an abstract  
124 type `R`, representing the data upon which we will recurse. This is the first argument to a  
125 value of type `AlgF Alg X`. Reasoning parametrically, an algebra can assume nothing about `R`  
126 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 folding an algebra over constructed data `inn d` is equal to invoking the algebra on `fold` for the fold 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  $p : A \rightarrow \text{bool}$ , 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  $R : \text{Set}$  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`:

## XX:6 Subsidiary Recursion in Coq

```
Definition SpanAlg(p : A -> bool) : Alg (ListF A) SpanF :=
  rollAlg (fun R fo span xs =>
    match xs with
    | Nil => SpanNoMatch
    | Cons hd tl =>
      if p hd then
        match (span tl) with
        | SpanNoMatch => SpanSomeMatch [hd] tl
        | SpanSomeMatch l r => SpanSomeMatch (hd::l) r
        end
      else
        SpanNoMatch
    end).
end).
```

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

```
Definition span(p : A -> bool)(xs : List A) : list A * List A
:= spanr (fold (ListF A)) p xs.
```

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

```
Inductive SpanF(X : Set) : Set :=
  SpanNoMatch : SpanF X
| SpanSomeMatch : list A -> X -> SpanF X.
```

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

### 170 3.1.1 The algebra for `span`

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

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

### 184 3.1.2 Defining span from SpanAlg

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

## 192 3.2 The wordsBy function (WordsBy.v)

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

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



## XX:8 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

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

### 215 3.3 The mapThrough function (MapThrough.v)

216 This example shows how to write a combinator that factors out a subsidiary recursion.  
217 The Haskell library `Data.List.Extra` defines a function **repeatedly** in essentially the same  
218 way as **mapThrough** in Figure 8 (we propose a more informative name). This function behaves  
219 like the standard **map** function on lists, except that the function *f* that we are mapping (or  
220 “mapping through”) takes in not just the current element *a*, but also the tail *as*. It then  
221 returns the value *b* to include in the output list, and whatever sublist it wishes, upon which  
222 **mapThrough** will then recurse.

223 To write this combinator using our infrastructure for subsidiary recursion, we will use  
224 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 MapThroughAlg{B : Set}(f:mappedT A B)
  : Alg (ListF A) (Const (list B)) :=
  rollAlg (fun R fo mapThrough xs =>
    match xs with
    | Nil => []
    | Cons hd tl =>
      let (b,c) := f R fo hd tl in
      b :: mapThrough c
    end).
Definition mapThroughr{R : Set}(fo:FoldT (Alg (ListF A)) R)
  {B : Set}(f:mappedT A B) : R -> list B.
Definition mapThrough{B : Set}(f:mappedT A B) : List A -> list B.

```

■ **Figure 9** The algebra `MapThroughAlg` defining the functions `mapThrough` and `mapThroughr`; code for the latter is omitted, as it follows the pattern of `wordsBy` and `wordsByr` of Figure 7 (`MapThrough.v`)

```

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

```

■ **Figure 10** Run-length encoding in Haskell, using `mapThrough` and `span`

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

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

### 236 3.4 Run-length encoding (`Rle.v`)

237 Finally, we have an example using our `mapThrough` combinator together with a subsidiary  
 238 recursion, to implement *run-length encoding*. This is a basic data-compression algorithm  
 239 where maximal sequences of  $n$  occurrences of element  $e$  are summarized by the pair  $(n, e)$  [19].  
 240 A Haskell implementation of this algorithm is in Figure 10. Recall that `(== a)` tests its  
 241 input for equality with `a`. The `compressSpan` helper function gathers up all elements at  
 242 the start of the tail `as` that are equal to the head `a`. This prefix is returned as `p`, with the  
 243 remaining suffix as `s`. The pair `(1 + length p, a)` is returned to summarize `a :: p`. The  
 244 `mapThrough` combinator then iterates `compressSpan` through the suffix `s`.

245 Assuming `A : Set` and an equality test `eqb : A -> A -> bool` on it, we port this code  
 246 to our Coq infrastructure in Figure 11. The function `compressSpan` is written at the type

## XX:10 Subsidiary Recursion in Coq

```
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) : list (nat * 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`)

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

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

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

### 255 4 Derivation of subsidiary recursion

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

#### 258 4.1 Retractive-positive recursive types (Mu.v)

259 As we have seen, our definitions require a form of positive-recursive types, to allow algebras  
260 to accept fold functions that themselves require algebras, and also for the definition of  
261 `Subrec` (which we will see in more detail in the next section). Full positive-recursive types  
262 are incompatible with Coq's type theory [6]. One can impose some restrictions on large  
263 eliminations which then enable positive-recursive types [3], but this requires changing the  
264 underlying theory. Here we derive a different solution.

265 Our starting point is a type scheme `F : Set -> Set`, with an `fmap` function (morphism  
266 part of the functor) of type

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

267 which satisfies the identity-preservation law for functors:

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

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

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

```

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 ( $\text{Mu.v}$ )

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

273 Instead, we define  $\text{Mu}$  in Figure 12, to weaken this ideal  $\text{Mu}'$  to a strictly positive  
 274 approximation. Instead of taking in  $F \text{ Mu}$ , the constructor  $\text{mu}$  accepts an input of type  
 275  $F R$ , for some type  $R$  for which we have a function of type  $R \rightarrow \text{Mu}$ . The impredicative  
 276 quantification of  $R$  is essential here: we will instantiate it with  $\text{Mu}$  itself in the definition  
 277 of  $\text{inMu}$  (Figure 12). So this approach would not work in a predicative theory like Agda’s.  
 278 The quantification of  $R$  can be seen as applying a technique due to Mendler, of introducing  
 279 universally quantified variables for problematic type occurrences, to a datatype constructor.  
 280 We will review this in Section 7.

281 Returning to Figure 12, we have functions  $\text{inMu}$  and  $\text{outMu}$ , which make  $F \text{ Mu}$  a retraction  
 282 ( $\text{outIn}$ ) of  $\text{Mu}$ : the composition of  $\text{outMu}$  and  $\text{inMu}$  is (extensionally) the identity on  $F \text{ Mu}$ .  
 283 But the reverse composition cannot be proved to be the identity, because of the basic problem  
 284 of **noncanonicity** that arises with this definition.

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

```

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

```

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

```

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

```

292 Since  $\text{fmap } f \text{ (NilF } A)$  equals  $\text{NilF } B$  for  $f : A \rightarrow B$ , if we apply  $\text{outMu}$  (of Figure 12) to  
 293  $\text{mkNil}'$  or  $\text{mkNil}$ , we will get  $\text{NilF } (\text{List } A)$ . But critically,  $\text{mkNil}$  and  $\text{mkNil}'$  are not equal,  
 294 neither definitionally nor provably. Of course, one could define a function that puts  $\text{Mu}$  values  
 295 in canonical form by folding  $\text{inMu}$  over them. Then  $\text{mkNil}$  and  $\text{mkNil}'$  would be equivalent.  
 296 But they would still not be provably equal, which is the problem of noncanonicity. We will  
 297 see how to work around this in Section 6. First, though, let us complete the exposition of  
 298 our implementation of subsidiary recursion.

## XX:12 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`)

### 299 4.2 The implementation of `Subrec` (`Subrec.v`)

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

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

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

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

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

- 317 ■ `Subrec` is passed as the value for the abstract type `R`; this is what enables all the rest of  
318 the components to have the desired types, since we will pass values that have `Subrec`  
319 where the interface mentions `R`.
- 320 ■ The function `fold : FoldT Alg Subrec` is passed as the fold function of type `FoldT Alg R`.
- 321 ■ For the `rec : R -> X R` function, we pass `(fold X xmap alg) : Subrec -> X Subrec`.
- 322 ■ For the subdata structure of type `F R`, we pass `d : F Subrec`.

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

### 327 5 Interface for subsidiary induction (`Subreci.v`)

328 We have seen how to write subsidiary recursions in Coq. But can one reason about these?  
329 We turn now briefly to our interface for subsidiary induction in Coq, and some example  
330 proofs written using this interface. Subsidiary induction is the natural extension of subsidiary  
331 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)

332 by a functor  $F$  and a functor  $Fi : (\text{Subrec} \rightarrow \text{Prop}) \rightarrow (\text{Subrec} \rightarrow \text{Prop})$  over Subrec-  
 333 indexed propositions (i.e., predicates). Just as functors need an `fmap` function, here we need  
 334 an indexed version, of type `fmapiT Subrec Fi` (definition elided.)

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

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

## 351 6 Examples of subsidiary induction

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

```

Theorem RldRle (xs : list A): rld (rle (toList xs)) = xs.

```

## XX:14 Subsidiary Recursion in Coq

```
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`

363 We invoke the lemmas about `span` subsidiarily, so that we may apply our induction hypothesis  
 364 to the suffix that `span` returns (on which `mapThrough` then recurses). For example, the  
 365 lemma for `GuardPresF` takes in the indexed fold function `foi` from the outer induction (for  
 366 `RldRle`), to show that the abstract predicate `R` applies to the suffix `r` returned by `span`. This  
 367 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.

```

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

## 376 7 Related Work

377 **Termination.** In some tools, like Coq, Agda, and Lean, termination is checked statically,  
 378 based on structural decrease. Others, like Isabelle/HOL, allow one to write recursions first,  
 379 and prove (possibly with automated help) their termination afterwards [12]. These tools all  
 380 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  $\forall X. (\forall R. (R \rightarrow X) \rightarrow F\ R \rightarrow X) \rightarrow \mu\ F \rightarrow X$ . 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 `span` function as a subsidiary recursion, for functions `wordsBy` and run-length encoding. We also looked briefly at the extension of this interface to support subsidiary induction, with example lemmas about `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.

## 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 Gilles Barthe, Maria João Frade, Eduardo Giménez, Luís Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Math. Struct. Comput. Sci.*, 14(1):97–141, 2004. URL: <https://doi.org/10.1017/S0960129503004122>, doi:10.1017/S0960129503004122.
- 3 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>.
- 4 Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive theorem provers - an overview. *Mathematical Structures in Computer Science*, 26(1):38–88, 2016. URL: <https://doi.org/10.1017/S0960129514000115>, doi:10.1017/S0960129514000115.
- 5 Robin Cockett and Dwight Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. AMS, 1992.
- 6 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](https://doi.org/10.1007/3-540-52335-9_47), doi:10.1007/3-540-52335-9\_47.
- 7 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](https://doi.org/10.1007/978-3-030-79876-5_37), doi:10.1007/978-3-030-79876-5\_37.
- 8 The Agda development team. *Agda*, 2021. Version 2.6.2.1. URL: <https://agda.readthedocs.io/en/v2.6.2.1/>.
- 9 Denis Firsov, Richard Blair, and Aaron Stump. Efficient mendler-style lambda-encodings in cedille. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 235–252. Springer, 2018.
- 10 Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in cedille. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 215–227. ACM, 2018.
- 11 Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.
- 12 Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL*. URL: <https://isabelle.in.tum.de/doc/functions.pdf>.
- 13 Alexander Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Autom. Reasoning*, 44(4):303–336, 2010. URL: <https://doi.org/10.1007/s10817-009-9157-2>, doi:10.1007/s10817-009-9157-2.
- 14 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2021. Version 8.13.2. URL: <http://coq.inria.fr>.
- 15 Conor McBride. Elimination with a motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes*

- 473        *in Computer Science*, pages 197–216. Springer, 2000. URL: [https://doi.org/10.1007/](https://doi.org/10.1007/3-540-45842-5_13)  
474        [3-540-45842-5\\_13](https://doi.org/10.1007/3-540-45842-5_13), doi:10.1007/3-540-45842-5\_13.
- 475    16    N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus.  
476        *Annals of Pure and Applied Logic*, 51(1):159 – 172, 1991.
- 477    17    Wolfgang Naraschewski and Tobias Nipkow. Isabelle/hol, 2020. URL: [http://www.cl.cam.](http://www.cl.cam.ac.uk/research/hvg/Isabelle/)  
478        [ac.uk/research/hvg/Isabelle/](http://www.cl.cam.ac.uk/research/hvg/Isabelle/).
- 479    18    Scott Owens and Konrad Slind. Adapting functional programs to higher order logic. *Higher-*  
480        *Order and Symbolic Computation*, 21(4):377–409, 2008. URL: [https://doi.org/10.1007/](https://doi.org/10.1007/s10990-008-9038-0)  
481        [s10990-008-9038-0](https://doi.org/10.1007/s10990-008-9038-0), doi:10.1007/s10990-008-9038-0.
- 482    19    David Salomon and Giovanni Motta. *Handbook of Data Compression*. Springer, 2009.
- 483    20    Aaron Stump, Christopher Jenkins, Stephan Spahn, and Colin McDonald. Strong func-  
484        tional pearl: Harper’s regular-expression matcher in cedille. *Proc. ACM Program. Lang.*,  
485        4(ICFP):122:1–122:25, 2020. URL: [https://doi.org/10.1145/](https://doi.org/10.1145/3409004)  
486        [3409004](https://doi.org/10.1145/3409004).
- 487    21    Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008. URL:  
488        <https://doi.org/10.1017/S0956796808006758>, doi:10.1017/S0956796808006758.
- 489    22    Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. Foundational, composi-  
490        tional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In  
491        *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012,*  
492        *Dubrovnik, Croatia, June 25-28, 2012*, pages 596–605. IEEE Computer Society, 2012. URL:  
493        <https://doi.org/10.1109/LICS.2012.75>, doi:10.1109/LICS.2012.75.
- 494    23    D. A. Turner. Elementary Strong Functional Programming. In *Proceedings of the First*  
495        *International Symposium on Functional Programming Languages in Education*, FPLE ’95,  
496        page 1–13, Berlin, Heidelberg, 1995. Springer-Verlag.
- 497    24    Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic J. of*  
498        *Computing*, 6(3):343–361, September 1999.
- 499    25    Tarmo Uustalu and Varmo Vene. The Recursion Scheme from the Cofree Recursive Comonad.  
500        *Electron. Notes Theor. Comput. Sci.*, 229(5):135–157, 2011. URL: [https://doi.org/10.1016/](https://doi.org/10.1016/j.entcs.2011.02.020)  
501        [j.entcs.2011.02.020](https://doi.org/10.1016/j.entcs.2011.02.020), doi:10.1016/j.entcs.2011.02.020.