







# Subsidiary Recursion in Coq

Aaron Stump   

Computer Science Dept., The University of Iowa, USA

Alex Hubers 

Computer Science, The University of Iowa, USA

Christopher Jenkins  

Computer Science, The University of Iowa, USA

Benjamin Delaware  

Computer Science, Purdue University, USA

---

## Abstract

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

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

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

**Digital Object Identifier** 10.4230/LIPIcs.ITP.2022.

## 1 Introduction: subsidiary recursion

Central to interactive theorem provers like Coq, Agda, Isabelle/HOL, Lean and others are terminating recursive functions over user-declared inductive datatypes [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 will be 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 applies to the standard datatypes in the Coq library, and does not require switching  
49 libraries or datatype definitions.

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

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

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

## 69 2 Interface for subsidiary recursion

70 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 (`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]). On this approach, one describes transformations to be performed on data as *algebras*, which can then be *folded* over data. The simplest form of algebras, namely *F*-algebras for functor *F*, 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, which we will use below:

```

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

```

Algebras for our subsidiary recursion are more complex than *F*-algebras. 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, thus enabling recursions to use Coq’s pattern-matching on the subdata structure *d*. So the style of coding against this interface retains a similar feel to structural recursion. Unlike with structural termination, though, the interface here is type-based and hence compositional.

We have previously dubbed this interface a *recursion universe* [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`)

Our development is parametrized by a signature functor *F* of type  $\mathbf{Set} \rightarrow \mathbf{Set}$ . It implements `Subrec` :  $\mathbf{Set}$ , which is the type to use for subsidiary recursion. This type

## 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 comes with `inn : F Subrec -> Subrec`, which behaves computationally like a constructor.  
102 `Subrec.v` does not itself provide an induction principle based on `inn`, however. Induction  
103 is derived later (Section 5). `Subrec.v` makes critical use of retractive-positive recursive  
104 types, to take a fixed-point of a construction based on `F`. We present these recursive types in  
105 Section 4.1 below.

106 For our examples, we will consider the specialization to the case of lists, parametrized  
107 by the type `A` of elements. In general, to use our development to get subsidiary recursion  
108 over some datatype, one must define a signature functor for the datatype. For lists, this is  
109 `ListF` of Figure 2. `List` is defined to be `Subrec`, with `F` instantiated to `ListF A`. This type  
110 `List` is not to be confused with the type `list` of lists in Coq’s standard library. As noted  
111 previously, our development is meant to be used in extension of existing inductive datatypes,  
112 not replacing them. The figure also shows constructors `mkNil` and `mkCons` for `List`, and  
113 types for conversion functions between `List` and `list` (code elided). One direction uses  
114 Coq’s structural recursion, the other uses subsidiary recursion, which we will see next.

### 115 2.3 Algebras for subsidiary recursion

116 `Subrec.v` also implements the notion of algebra we introduced informally above. The central  
117 definitions are in Figure 3. `KAlg` is the kind for the type-constructor for algebras, as we see  
118 in the definition of `Alg`. This type-constructor `Alg` is a fixed-point of the type `AlgF`. The  
119 fixed-point is taken using `MuAlg`, which implements our retractive-positive recursive types  
120 (Section 4.1) at kind `KAlg`.

121 We need a fixed-point here due to the input `fold (Algf)` of type `FoldT Alg R`. This is  
122 the type for fold functions which apply algebras (`Alg`) to data of type `R`. The variable `Alg`  
123 occurs only positively (but not strictly positively) in `AlgF`, because it occurs negatively in  
124 `FoldT Alg R` which occurs negatively in `AlgF Alg X`. So we can indeed take a fixed-point of  
125 `AlgF` to define the constant `Alg`.

126 Let us look at `AlgF`. As noted already, each recursion is based on an abstract type `R`,

```

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 .

```

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 following operations. Next we have a local `fold` function, which will 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`.

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 as having the type `C` outside the recursion. The function `fold` in the figure initiates top-level folds. We also can have functions between `Alg` and its `AlgF`-unfolding. We will return to the code for `Subrec.v` 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 stated as a theorem in Figure ???. 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.

### 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`)

This first example does not invoke subsidiary recursions, but will itself be used as a subsidiary recursion in other examples to follow. 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 would like to 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 are legal. But the second component will be a subdatum of the input list, and so can still have type `R`, enabling outer recursive calls. So we want:

```

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 4** 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.
```

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

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

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

### 173 3.1.1 The algebra for `span`

174 Figure 4 show the algebra `SpanAlg`, whose type is `Alg (ListF A) SpanF`. So we are  
175 defining an algebra (`Alg`) for the `ListF A` functor, with carrier `SpanF` of the required  
176 type `Set -> Set`. We use `rollAlg` to create an algebra from something whose type is an  
177 application of `AlgF`. This takes in all the components of the recursion universe: the abstract  
178 type `R`, the fold function (`fo`) for any subsidiary recursions (not needed here), a function we  
179 choose to name `span` for making recursive calls, and finally `xs : ListF A R`. The algebra  
180 pattern-matches on this `xs`. In the cases where it is empty or where its head (`hd`) does not  
181 satisfy `p`, we return `SpanNoMatch`. This signals to the caller that we really wished to return  
182 `([], xs)`, but could not because we do not have `xs` at type `R`. If the head does satisfy `p`, then  
183 we recurse on the tail (`tl : R`) by calling the provided `span : R -> SpanF R`. If `span tl`  
184 returns `SpanNoMatch`, that means that we should make `tl` the suffix in the pair we return

```

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 5** Functions derived from SpanAlg (Span.v)

185 (via SpanSomeMatch). Happily, we have `tl : R` here, so we can do this. In either case (for  
 186 return value of `span tl`), we add the head to the front of the prefix.

### 187 3.1.2 Defining span from SpanAlg

188 SpanAlg is used in the definition of `spanhr`, in Figure 5. This function invokes the fold  
 189 function it is given, on SpanAlg. The final twist is now in the definition of `spanr`. We call  
 190 `spanhr` on the input `xs : R`. If `spanhr` returns `SpanNoMatch`, then we are supposed to return  
 191 `([],xs)`, which we can do here, because we have `xs : R`. It was only inside the algebra that  
 192 we lost the information that the subdata structure of type `F R` is derived from a value of  
 193 type `R`. If `spanhr` returns `SpanSomeMatch l r`, then we return the nonempty prefix `(l)` and  
 194 the suffix `(r)`. We also define a version of `break` for subsidiary recursion (e.g., in `wordsBy`,  
 195 below).

## 196 3.2 The wordsBy function (WordsBy.v)

197 Let us now see how to write `wordsBy`, our example function from Section 1, using `breakr`  
 198 subsidiarily. The code is in Figure 6, assuming a type `A : Set`. The setup is similar to  
 199 that for `span`. We first define an algebra `WordsBy`, parametrized by the predicate `p`, of type  
 200 `Alg (ListF A) (Const (list (list A)))`. This says that `WordsBy p` is an algebra (Alg)  
 201 for the `ListF A` functor, with carrier `Const (list (list A))`. `Const` is a combinator for  
 202 creating the object part of constant functors; `FunConst` creates the morphism part (i.e., the  
 203 `fmap` function). We use `Const` where the return type of the algebra will not depend on its  
 204 abstract type `R`. Here, we are constructing from scratch a list of lists, so it will not be legal  
 205 to recurse on the list itself, or its (list) elements. So we just use the `list` type of Coq's  
 206 standard library.

207 The code for `WordsBy` is essentially the same as what we saw in Section 1. We pattern  
 208 match on `xs : ListF A R`. Recall that for this function, we are trying to drop elements  
 209 which satisfy `p`, and return a list of the sublists between maximal sequences of such elements.  
 210 In the `Cons` case, if the head (`hd`) satisfies the predicate, then we are supposed to drop it and  
 211 recurse. This is legal, because `tl : R` and `wordsBy : R -> list (list A)`. In the `else`



## XX:8 Subsidiary Recursion in Coq

```
Definition WordsBy(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))) (WordsBy p) xs.
```

■ **Figure 6** Functions `wordsBy` and `wordsByr`, and the algebra they fold (`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 7** The `mapThrough` function in Haskell

212 case, we use `breakr` to obtain the maximal prefix `w` of `tl` that does not satisfy `p`, and the  
213 remaining suffix `z`.

214 Here we see the benefit of our approach. From Figure 5, the return type of `breakr`  
215 is `list A * R`, where `R` comes from the type `FoldT (ListF A) Alg R` of `fo`, from the  
216 definition of `AlgF` in Figure 3 (instantiating the functor with `ListF A`). This means that  
217 from the invocation of `breakr`, we get `w : list A` and `z : R`. And so we can indeed apply  
218 `wordsBy : R -> list (list A)` to `z` to recurse. The figure also shows the code for the  
219 subsidiary recursion `wordsByr`.

### 220 3.3 The `mapThrough` function (`MapThrough.v`)

221 In this example, we see how to write a combinator that factors out a subsidiary recursion.  
222 The Haskell library `Data.List.Extra` has a function `repeatedly`, defined essentially as in  
223 Figure 7, though we attempt a more informative name. This is like the standard `map` function  
224 on lists, except that the function `f` that we are mapping (or “mapping through”) takes in  
225 not just the current element `a`, but also the tail `as`. It then returns the value `b` to include in  
226 the output list, and whatever other list it wishes, upon which `mapThrough` will recurse.

227 To write this combinator using our infrastructure for subsidiary recursion, we need to  
228 supply the mapped function with the fold function for `mapThrough`’s recursion. This is so  
229 that the mapped function can initiate a subsidiary recursion, returning a value in the abstract  
230 type `R` of `mapThrough`’s recursion. So the type we will use for mapped functions is:

```
Definition mappedT(A B : Set) : Set :=
```



```

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 8** The algebra `MapThroughAlg` defining function `mapThrough` and `mapThroughr`; the code for those follows the pattern of `wordsBy` and `wordsByr` (Figure 6), so we omit it (`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 9** Run-length encoding in Haskell, using `mapThrough` and `span`

```

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

```

231 This type is more informative than the Haskell type, since it shows that the second component  
 232 of the returned value must have type `R`, and hence must be (hereditarily) a tail of the input.

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

### 239 3.4 Run-length encoding (`Rle.v`)

240 Finally, we have an example using our `mapThrough` combinator together with a subsidiary  
 241 recursion, to implement *run-length encoding*. This is a basic data-compression algorithm  
 242 where maximal sequences of  $n$  occurrences of element  $e$  are summarized by the pair  $(n, e)$  [19].  
 243 Haskell code is in Figure 9. Recall that `(== a)` tests its input for equality with `a`. The  
 244 `compressSpan` helper function gathers up all elements at the start of the tail `as` that are  
 245 equal to the head `a`. This prefix is returned as `p`, with the remaining suffix as `s`. The pair  
 246 `(1 + length p, a)` is returned to summarize `a` : `p`. The `mapThrough` combinator then  
 247 iterates `compressSpan` through the suffix `s`.

248 Assuming `A` : `Set` and an equality test `eqb` : `A -> A -> bool` on it, we port this code  
 249 to our Coq infrastructure in Figure 10. The function `compressSpan` is written at the type  
 250 `mappedT A (nat * A)` that will be required by `mapThrough`. Unfolding the definition of  
 251 `mappedT`, `compressSpan` has 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 10** The function `rle` for run-length encoding, and the algebra `RleAlg` defining it in terms of `MapThroughAlg` of Figure 8 (`Rle.v`)

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

252 It will be invoked by the code for `mapThrough` with a fold function `fo` of type `FoldT (Alg (ListF A)) R`.  
253 It then has the responsibility of extracting from the tail at type `R` (second input) a result  
254 upon which `mapThrough` should recurse (second component of the output pair). Then we  
255 define an algebra `RleAlg` by supplying `compressSpan` as the function to map through, to  
256 `MapThroughAlg` (Figure 8). Following the pattern seen above, we define function `rle` for  
257 top-level recursions using `fold` (we could also define a subsidiary version `rler`).

### 258 4 Derivation of subsidiary recursion

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

#### 261 4.1 Retractive-positive recursive types (`Mu.v`)

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

268 Assume `F : Set -> Set`, with an `fmap` function (morphism part of the functor) of type

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

269 which satisfies the identity-preservation law for functors:

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

270 Then we make the definitions of Figure 11. The critical idea is embodied in the definition of  
271 `Mu`. Ideally, we would like to have a definition like

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

272 This is exactly what is used in approaches to modular datatypes in functional programming,  
273 like Swierstra's [21]. But this definition is (rightly) rejected by Coq, as instantiations of `F`  
274 that are not strictly positive would be unsound.

275 The definition of `Mu` weakens this to a strictly positive approximation:

```

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 11** Derivation of retractive-positive recursive types  $(\text{Mu.v})$

```

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

```

276 Instead of taking in  $F \text{ Mu}$ , constructor `mu` accepts an input of type  $F R$ , for some type  $R$  for  
 277 which we have a function of type  $R \rightarrow \text{Mu}$ . The impredicative quantification of  $R$  is essential  
 278 here: we instantiate it with  $\text{Mu}$  itself in the definition of `inMu` (Figure 11). So this approach  
 279 would not work in a predicative theory like Agda's. The quantification of  $R$  can be seen  
 280 as applying a technique due to Mendler, of introducing universally quantified variables for  
 281 problematic type occurrences, to a datatype constructor. We will review this in Section 7.

282 Returning to Figure 11, we have functions `inMu` and `outMu`, which make  $F \text{ Mu}$  a retraction  
 283 (`outIn`) of  $\text{Mu}$ : the composition of `outMu` and `inMu` is (extensionally) the identity on  $F \text{ Mu}$ .  
 284 But the reverse composition cannot be proved to be the identity, because of the basic problem  
 285 of **noncanonicity** that arises with this definition.

286 For a simple example: suppose we instantiate  $F$  with `ListF` (of Figure 2). Our derivation  
 287 uses a different type that wraps  $F$ , but this will show the issue in a simple form. Let us  
 288 temporarily define `List A` as  $\text{Mu} (\text{ListF } A)$  (again, for subsidiary recursion do not use just  
 289 `ListF` directly). The canonical way to define the empty list would be, implicitly instantiating  
 290  $F$  to `ListF A`,

```

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

```

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

```

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

```

293 Since `fmap f (NilF A)` equals just `NilF B` for  $f : A \rightarrow B$ , if we apply `outMu` (of Figure 11)  
 294 to `mkNil'` or `mkNil`, we will get `NilF (List A)`. But critically, `mkNil` and `mkNil'` are not  
 295 equal, neither definitionally nor provably. One can define a function that puts  $\text{Mu}$  values in  
 296 normal form by folding `inMu` over them. Then `mkNil` and `mkNil'` will have the same normal  
 297 form, and be equivalent in that sense. But the fact that they are not provably equal is what  
 298 we term noncanonicity.

299 Noncanonicity must be handled carefully when reasoning about functions defined with our  
 300 interface. We will see an example in Section 6. First, though, let us complete the exposition  
 301 of 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 12** Definition of Subrec as a fixed-point of SubrecF (Subrec.v)

### 302 4.2 The implementation of Subrec (Subrec.v)

303 The type Subrec is defined in Figure 12, as a fixed-point of SubrecF : Set -> Set. We  
304 take this fixed-point with Mu, discussed in the previous section, and obtain roll and unroll  
305 functions between SubrecF Subrec and Subrec. Unrolling Subrec gives us the type

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

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

311 The rest of the interface for Subrec is shown in Figure 13. To fold an algebra alg with  
312 carrier X (with fmap function given by FunX) over d : Subrec, we unroll the definition of  
313 Subrec and apply that to the algebra (with its carrier).

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

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

326 Finally, Figure 13 defines out as a subsidiary recursion, given a fold function. Outside  
327 the recursion, d has type F R; inside the recursion it has type F R' where R' is the abstract  
328 type of the subsidiary recursion. So out implements the idea that unfolding an abstract type  
329 one step is just a trivial case of subsidiary recursion.

### 330 5 Interface for subsidiary induction (Subreci.v)

331 We have seen how to write subsidiary recursions in Coq. But can one reason about  
332 these? To wrap up this paper, we will briefly see the interface to our development  
333 of subsidiary induction in Coq, and example proofs written using this interface. Sub-  
334 subsidiary induction is the natural extension of subsidiary recursion, which worked over  
335 Sets, to Subrec-predicates. The development is parametrized by a functor F and a func-  
336 tor Fi : (Subrec -> Prop) -> (Subrec -> Prop) over Subrec-indexed propositions (i.e.,

```

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 13** The rest of the interface for Subrec (Subrec.v)

337 predicates). Just as functors need an `fmap` function, we here need an indexed version, of  
 338 type `fmapiT Subrec Fi` (definition elided.)

339 The central definitions for the type `Subreci : Subrec -> Prop` are given in Figure 14.  
 340 Where having a value `x` of `Subrec` entitles us to define subsidiary recursions to inhabit types  
 341 `X Subrec`, a value of type `Subreci x` lets us prove properties of `x` by subsidiary induction.  
 342 Briefly: `kMo` is the kind for *motives*, namely predicates on `Subrec` [15]. `KAlgi` is the kind for  
 343 indexed algebras. `FoldTi` is the indexed version of `FoldT`: it expresses provability of `X C` for  
 344 `d`, based on an indexed algebra and a value of type `C d`, where `C` is the (indexed) anchor type.  
 345 `AlgFi` and `Algi` are indexed versions of the algebras we saw for recursion. The `rec` function  
 346 (Figure 3) has now become an induction hypothesis: given any `d` where `R d` holds, `ih` proves  
 347 `X R d`. A value of type `R d` is thus a license to induct on `d`. Finally, the algebra is given a  
 348 subdata structure indexed by `d : Subrec`, and must produce a proof of `X R d`. `Subreci` is  
 349 defined as the suitably indexed fixed-point of `SubrecFi`, which is the natural indexed version  
 350 of `SubrecF`.

351 For lists, we instantiate `Fi` with `ListFi`, shown in Figure 15. This is just the indexed  
 352 version of `ListF`. Given a list `A`, `toListi` returns a value of type `Listi` (`toList xs`).  
 353 This can be understood as saying that for any list (from Coq’s standard library), we can  
 354 reason by subsidiary induction to prove properties of `toList xs`. We also introduce an  
 355 abbreviation `ListFoldTi` for the type of indexed fold functions over lists.

## 356 6 Examples of subsidiary induction

357 For proving the main theorem about run-length encoding, we need several lemmas about  
 358 `span`, shown in Figure 16. For lack of space, we just state the properties. The first says that  
 359 appending the results of a call to `span` returns the original list (module some conversions to  
 360 `list` from `List`). The second uses the inductive type `Forall` from Coq’s standard library to  
 361 state that all the elements of the prefix returned by `span` satisfy `p`. These lemmas are proved  
 362 using indexed algebras with constant (indexed) carriers. In contrast, `GuardPresF` uses its  
 363 argument `S` to express that whenever `spanh` returns a suffix `r`, that suffix satisfies `S`. This  
 364 enables us to invoke an outer induction hypothesis on this suffix, when reasoning subsidiarily  
 365 about `span`. Using these lemmas, we can write a short proof by subsidiary induction of the  
 366 following, where `rld : list (nat * A) -> list A` is the obvious decoding function:

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

367 We invoke the lemmas about `span` subsidiarily, so that we may apply our induction hypothesis  
 368 to the suffix that `span` returns (on which `mapThrough` then recurses). For example, the

## 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 14** 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 15** 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 16** 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 17** A statement of the property that `span` returns the empty suffix, computed using `getNil` to avoid noncanonicity problems, if all elements satisfy `p`

369 lemma for `GuardPresF` takes in the indexed fold function `foi` from the outer induction (for  
 370 `RldRle`), to show that the abstract predicate `R` applies to the suffix `r` returned by `span`. This  
 371 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.

```

372 Finally, as promised, a note on noncanonicity. When proving properties about subsidiary  
 373 recursions on `xs : List A`, one should be aware that nothing prevents the property from  
 374 being applied to noncanonical `Lists`. For example, suppose we wish to prove that if all  
 375 elements of a list satisfy `p`, then the suffix returned by `span` is empty. It is dangerous to  
 376 phrase this as “the suffix equals `mkNil`”, because for a noncanonical input `xs`, `span` will  
 377 return that same noncanonical `xs` as the suffix (and so it may be a noncanonical empty list,  
 378 not equal to `mkNil`). The solution in this case is to use a function `getNil` (`List.v`) that  
 379 computes an empty list from `xs`. The statement that one can prove is shown in Figure 17.

## 380 7 Related Work

381 **Termination.** In some tools, like Coq, Agda, and Lean, termination is checked statically,  
 382 based on structural decrease. Others, like Isabelle/HOL, allow one to write recursions first,  
 383 and prove (possibly with automated help) their termination afterwards [12]. These tools all  
 384 support well-founded recursion, but in constructive type theory, evidence of well-foundedness  
 385 then propagates through code. In contrast, our approach here, while less general, does not  
 386 clutter code with proofs. Subsidiary recursion can be seen as a generalization of *nested*



387 *recursion*, which allows recursive calls of the form  $\mathbf{f} \ (\mathbf{f} \ x)$  [13]. In subsidiary recursion,  
 388 these are generalized to the form  $\mathbf{f} \ (\mathbf{g} \ x)$ , where  $\mathbf{g}$  could be  $\mathbf{f}$  or another recursively defined  
 389 function. For more on partiality and recursion in theorem provers, see [4].

390 Our work contributes to the program proposed by Owens and Slind, of broadening the  
 391 scope of functional programs that can be accommodated in ITPs [18]. The goal of terminating  
 392 recursion has been advocated in the literature on programming languages under the name  
 393 *strong functional programming* [23]. Uustalu and Vene developed a categorical view of a  
 394 recursion scheme allowing one level of subsidiary recursion, and illustrated it in Haskell with  
 395 an artificial example [25]. In contrast, our scheme allows arbitrary finite nestings of recursion,  
 396 and we illustrate it in Coq with realistic examples. It seems that generalizing the carriers  
 397 of algebras to functors is the critical step enabling such examples. Like Uustalu and Vene,  
 398 we find that subsidiary recursion subsumes course-of-value recursion. Our method is also  
 399 similar to the technique of sized types, in providing a type-based method for termination [2].  
 400 With sized types, datatypes are indexed with abstract sizes, which must then be propagated  
 401 through code, using dependent types. Our **Subrec** does not require dependent types, resulting  
 402 in just polymorphically typed code for our examples (**Subreci**, for proofs, of course does use  
 403 dependent types).

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

## 414 8 Conclusion

415 We have seen a derivation in Coq of a scheme for terminating subsidiary recursion, where  
 416 recursions may be nested and outer recursive calls may be made on results of inner recursions.  
 417 We saw examples invoking the **span** function as a subsidiary recursion, for functions **wordsBy**  
 418 and run-length encoding. We also looked briefly at the extension of this interface to support  
 419 subsidiary induction, with example lemmas about **span**, and the decoding correctness theorem  
 420 for run-length encoding. There are many other interesting examples we can develop in Coq  
 421 with this interface, including natural-number division, which may invoke subtraction as  
 422 a subsidiary recursion. Another example is Harper’s regular-expression matcher, which  
 423 previous work showed can be implemented in Cedille using a form of nested recursion that is  
 424 subsumed by subsidiary recursion [20]. We may also attempt to extend the recursion universe  
 425 further, to allow other forms of recursion like divide-and-conquer, where some (necessarily  
 426 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*

- 478        *in Computer Science*, pages 197–216. Springer, 2000. URL: [https://doi.org/10.1007/](https://doi.org/10.1007/3-540-45842-5_13)  
479        [3-540-45842-5\\_13](https://doi.org/10.1007/3-540-45842-5_13), doi:10.1007/3-540-45842-5\_13.
- 480    16    N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus.  
481        *Annals of Pure and Applied Logic*, 51(1):159 – 172, 1991.
- 482    17    Wolfgang Naraschewski and Tobias Nipkow. Isabelle/hol, 2020. URL: [http://www.cl.cam.](http://www.cl.cam.ac.uk/research/hvg/Isabelle/)  
483        [ac.uk/research/hvg/Isabelle/](http://www.cl.cam.ac.uk/research/hvg/Isabelle/).
- 484    18    Scott Owens and Konrad Slind. Adapting functional programs to higher order logic. *Higher-*  
485        *Order and Symbolic Computation*, 21(4):377–409, 2008. URL: [https://doi.org/10.1007/](https://doi.org/10.1007/s10990-008-9038-0)  
486        [s10990-008-9038-0](https://doi.org/10.1007/s10990-008-9038-0), doi:10.1007/s10990-008-9038-0.
- 487    19    David Salomon and Giovanni Motta. *Handbook of Data Compression*. Springer, 2009.
- 488    20    Aaron Stump, Christopher Jenkins, Stephan Spahn, and Colin McDonald. Strong func-  
489        tional pearl: Harper’s regular-expression matcher in cedille. *Proc. ACM Program. Lang.*,  
490        4(ICFP):122:1–122:25, 2020. URL: [https://doi.org/10.1145/](https://doi.org/10.1145/3409004)  
491        [3409004](https://doi.org/10.1145/3409004).
- 492    21    Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008. URL:  
493        <https://doi.org/10.1017/S0956796808006758>, doi:10.1017/S0956796808006758.
- 494    22    Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. Foundational, composi-  
495        tional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In  
496        *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012,*  
497        *Dubrovnik, Croatia, June 25-28, 2012*, pages 596–605. IEEE Computer Society, 2012. URL:  
498        <https://doi.org/10.1109/LICS.2012.75>, doi:10.1109/LICS.2012.75.
- 499    23    D. A. Turner. Elementary Strong Functional Programming. In *Proceedings of the First*  
500        *International Symposium on Functional Programming Languages in Education*, FPLE ’95,  
501        page 1–13, Berlin, Heidelberg, 1995. Springer-Verlag.
- 502    24    Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic J. of*  
503        *Computing*, 6(3):343–361, September 1999.
- 504    25    Tarmo Uustalu and Varmo Vene. The Recursion Scheme from the Cofree Recursive Comonad.  
505        *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)  
506        [j.entcs.2011.02.020](https://doi.org/10.1016/j.entcs.2011.02.020), doi:10.1016/j.entcs.2011.02.020.