







# 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. 1–19

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.

## 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. First, for reasons we will explain further below, the carrier of the algebra will be a functor  $X : \text{Set} \rightarrow \text{Set}$ . Second, algebras have a specified *anchor type*  $C$ , which we can think of as the datatype *as viewed by a containing recursion* or else, if this is a top-level recursion, our development's version of the actual datatype (e.g., `List A`, for some  $A$ ). The algebra is presented with:

- a type  $R : \text{Set}$ , which will be this recursion's view of the datatype.
- a function `reveal` :  $R \rightarrow C$ , which reveals values of type  $R$  as really having the anchor type.
- a function `fold` : `FoldT Alg R`, which allows one to initiate subsidiary recursions in which the anchor type is  $R$ . Note that the algebra's anchor type is  $C$ , but for subsidiary recursions the anchor type changes (to  $R$ ). 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.

As in previous work, we dub this interface a *recursion universe* [20]. As in other domains using the term “universe”, we have a kind of space (here,  $R$ ) with operations that keep one in that space (for other cases: the ordinal  $\epsilon_0$  and  $\omega^-$ , 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  $R$ . One can use `reveal` to leave, but then `rec` can no longer be used. Some points must still be explained: why  $X$  has type  $\text{Set} \rightarrow \text{Set}$ , 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  $\text{Set} \rightarrow \text{Set}$ . It implements `Subrec` :  $\text{Set}$ , which is the type to use for subsidiary recursion. This type comes with `inn` :  $F \text{ Subrec} \rightarrow \text{Subrec}$ , which behaves computationally like a constructor. `Subrec.v` does not itself provide an induction principle based on `inn`, however. Induction is derived later (Section 5). `Subrec.v` makes critical use of retractive-positive recursive types, to take a fixed-point of a construction based on  $F$ . We present these recursive types in Section 4.1 below.

## XX:4 Subsidiary Recursion in Coq

```
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`)

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

### 121 2.3 Algebras for subsidiary recursion

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

127 We need a fixed-point here due to the input `fold (Algf)` of type `FoldT Alg C`. This is  
128 the type for fold functions which apply algebras (`Alg`) to data of type `C`, which we have  
129 already dubbed the *anchor type* of the recursion. At the top level of code, the anchor type  
130 would just be `List A` (for example). When one initiates a subsidiary recursion, though, the  
131 anchor type will instead be the abstract type `R` for the outer recursion. The variable `Alg`  
132 occurs only positively (but not strictly positively) in `AlgF`, because it occurs negatively in  
133 `FoldT Alg R` which occurs negatively in `AlgF Alg C X`. So we can indeed take a fixed-point  
134 of `AlgF` to define the constant `Alg`.

135 Let us look at `AlgF`. As noted already, each recursion is based on an abstract type `R`,  
136 representing the data upon which we will recurse. This is the first argument to a value  
137 of type `AlgF Alg C X`. Reasoning parametrically, an algebra can assume nothing about `R`  
138 except that it supports the following operations. First there is `reveal`, which turns an `R` into  
139 a `C`. This reveals that the data of type `R` are really values of the anchor type of this recursion.  
140 Next we have `fold`, which will allow us to fold another algebra over data of type `R`. We will  
141 use `fold` to initiate subsidiary recursions. Then there is `rec`, for recursive calls on data of  
142 type `R`.

143 As noted already, for subsidiary recursion, algebras have a carrier `X` which depends  
144 (functorially) on a type. This is so that (i) inside an inner recursion we may compute a result  
145 of some type that may mention `R`, but (ii) outside that recursion, the result will mention  
146 the anchor type `C`. The `rec` function returns something of type `X R`, and so does the algebra  
147 itself; this demonstrates (i). For (ii): if we look at the definition of `FoldT` in the figure, we

```

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

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

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

Definition Alg : KAlg := MuAlg AlgF.

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

```

■ **Figure 3** The type for algebras, parametrized over  $F : \text{Set} \rightarrow \text{Set}$  (Subrec.v)

```

Theorem FoldChar :
  forall (X : Set -> Set) (FunX : Functor X) (IdF : FmapId X FunX)
    (algf : AlgF Alg Subrec X) (d : F Subrec),
  fold X FunX (rollAlg algf) (inn d) =
  algf _ (fun x => x) fold (fold X FunX (rollAlg algf)) d .

```

148 see that folding an algebra of type `alg C X` over a value of type `C` produces a result of type  
 149 `X C`. Having a functor for the carrier of the algebra gives us the flexibility to type results  
 150 inside a recursion with the abstract type `R`, but view those results as having the anchor type  
 151 `C` outside the recursion.

152 The last definitions in the figure are for `fold`, which allows us to fold an `Alg` over a value  
 153 of type `Subrec`; and for mapping between `Alg` and its unfolding in terms of `AlgF`. We will  
 154 return to the code for `Subrec.v` in Section 4.

155 Finally, for a recursion scheme, one would like to see not just the typed interface, but  
 156 also the computation law. This is stated as a theorem in Figure ?? . Intuitively, it states  
 157 that folding an algebra over constructed data `inn d` is equal to invoking the algebra on  
 158 the identity function for `reveal` from the interface for algebras (Figure 3); `fold` for the fold  
 159 function; an invocation of `fold` with the algebra for the `rec` function; and `d` for the subdata  
 160 structure.

### 161 3 Examples of subsidiary recursion

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

### 163 3.1 The span function (Span.v)

164 Given a predicate  $p : A \rightarrow \text{bool}$ , and a value of type  $\text{List } A$ , we would like to compute  
 165 a pair of type  $\text{list } A * \text{List } A$ , where the first component is the maximal prefix whose  
 166 elements satisfy  $p$ , and the second is the remaining suffix. This is the typing for a top-level  
 167 recursion. More generally, though, given an anchor type  $R : \text{Set}$  along with a fold function  
 168 for that anchor type (i.e., of type  $\text{FoldT } (\text{Alg } (\text{ListF } A)) R$ ), we would like to map an  
 169 input list of type  $R$  to a pair of type  $\text{list } A * R$ . The first component of this pair is going to  
 170 be built up from scratch, and so cannot have type  $R$ ; we cannot statically ensure that outer  
 171 recursions on it are legal. But the second component will be a subdatum of the input list,  
 172 and so can still have type  $R$ , enabling outer recursive calls. So we want:

**Definition**  $\text{spanr}\{R : \text{Set}\}(\text{fo}:\text{FoldT } (\text{Alg } (\text{ListF } A)) R)$   
 $(p : A \rightarrow \text{bool})(xs : R) : \text{list } A * R.$

173 From this we can also define the top-level recursion, by supplying  $\text{fold } (\text{ListF } A)$ , which is  
 174 the function for folding an algebra over a list (Figure 3), for the argument  $\text{fo}$  of  $\text{spanr}$ :

**Definition**  $\text{span}(p : A \rightarrow \text{bool})(xs : \text{List } A) : \text{list } A * \text{List } A$   
 $:= \text{spanr } (\text{fold } (\text{ListF } A)) p xs.$

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

**Inductive**  $\text{SpanF}(X : \text{Set}) : \text{Set} :=$   
 $\text{SpanNoMatch} : \text{SpanF } X$   
 $| \text{SpanSomeMatch} : \text{list } A \rightarrow X \rightarrow \text{SpanF } X.$

181 The idea is that the recursion will signal if it is in the one tricky case where  $p$  does not  
 182 match the first element, by returning  $\text{SpanNoMatch}$ . Otherwise, it will be able to return, via  
 183  $\text{SpanSomeMatch}$ , a prefix and the suffix at type  $R$ . The prefix will be nonempty, and hence  
 184 the suffix will be at most the tail of  $xs$ . This suffix is available to the algebra in the subdata  
 185 structure of type  $\text{ListF } A R$ .

#### 186 3.1.1 The algebra for span

187 Figure 4 gives the algebra  $\text{SpanAlg}$  for computing  $\text{span}$ . The type of  $\text{SpanAlg } p C$  is

188  $\text{Alg } (\text{ListF } A) C \text{ SpanF}$

189 This states that we are defining an algebra ( $\text{Alg}$ ) for the  $\text{ListF } A$  functor, with anchor type  
 190  $C$  and carrier  $\text{SpanF}$ .  $\text{SpanF}$  has type  $\text{Set} \rightarrow \text{Set}$ , as required for the carriers of our algebras.  
 191 The definition of  $\text{SpanAlg}$  is actually parametrized by  $C$ , which is good, as it means we can  
 192 use  $\text{SpanAlg}$  for top-level or subsidiary recursions.

193 Let us continue through the code for  $\text{SpanAlg}$  (Figure 4). We use  $\text{rollAlg}$  to create an  
 194 algebra from something whose type is an application of  $\text{AlgF}$ . This takes in all the components  
 195 of the recursion universe: the abstract type  $R$ , the  $\text{reveal}$  function (not needed in this case),  
 196 the fold function ( $\text{fo}$ ) for any subsidiary recursions (also not needed here), a function we  
 197 choose to name  $\text{span}$  for making recursive calls, and finally  $xs : \text{ListF } A R$ . The algebra

```

Definition SpanAlg(p : A -> bool)(C : Set)
  : Alg (ListF A) C SpanF :=
  rollAlg (fun R reveal fo span xs =>
    match xs with
    | Nil => SpanNoMatch
    | Cons hd t1 =>
      if p hd then
        match (span t1) with
        | SpanNoMatch => SpanSomeMatch [hd] t1
        | SpanSomeMatch l r => SpanSomeMatch (hd::l) r
      end
    else
      SpanNoMatch
  end).

```

■ **Figure 4** The algebra `SpanAlg` for the `span` function (`Span.v`)

pattern-matches on this `xs`. In the cases where it is empty or where its head (`hd`) does not satisfy `p`, we return `SpanNoMatch`. This signals to the caller that we really wished to return `([], xs)`, but could not because we do not have `xs` at type `R`. If the head does satisfy `p`, then we recurse on the tail (`t1 : R`) by calling the provided `span : R -> SpanF R`. If `span t1` returns `SpanNoMatch`, that means that we should make `t1` the suffix in the pair we return (via `SpanSomeMatch`). Happily, we have `t1 : R` here, so we can do this. In either case (for return value of `span t1`), we add the head to the front of the prefix.

### 3.1.2 Defining `span` from `SpanAlg`

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

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

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



## XX:8 Subsidiary Recursion in Coq

```
Definition spanhr{R : Set}(fo:FoldT (Alg (ListF A)) R)
  (p : A -> bool)(xs : R) : SpanF R :=
  fo SpanF SpanFunctor (SpanAlg p R) 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)

225 The code for `WordsBy` is essentially the same as what we saw in Section 1. We pattern  
226 match on `xs : ListF A R`. Recall that for this function, we are trying to drop elements  
227 which satisfy `p`, and return a list of the sublists between maximal sequences of such elements.  
228 In the `Cons` case, if the head (`hd`) satisfies the predicate, then we are supposed to drop it and  
229 recurse. This is legal, because `tl : R` and `wordsBy : R -> list (list A)`. In the `else`  
230 case, we use `breakr` to obtain the maximal prefix `w` of `tl` that does not satisfy `p`, and the  
231 remaining suffix `z`.

232 Here we see the benefit of our approach. From Figure 5, the return type of `breakr` is  
233 `list A * R`, where `R` is the anchor type of the provided fold function `fo`. And `fo` has type  
234 `FoldT (ListF A) Alg R`, from the definition of `AlgF` in Figure 3 (instantiating the functor  
235 with `ListF A`). This means that from the invocation of `breakr`, we get `w : list A` and  
236 `z : R`. And so we can indeed apply `wordsBy : R -> list (list A)` to `z` to recurse. The  
237 figure also shows the code for the subsidiary recursion `wordsByr`.

### 238 3.3 The mapThrough function (MapThrough.v)

239 The Haskell library `Data.List.Extra` has a function `repeatedly`, defined essentially as  
240 in Figure 7, though we attempt a more informative name. This is like the standard `map`  
241 function on lists, except that the function `f` that we are mapping (or “mapping through”)  
242 takes in not just the current element `a`, but also the tail `as`. It then returns the value `b` to  
243 include in the output list, and whatever other list it wishes, upon which `mapThrough` will  
244 recurse.

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

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

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



```

Definition WordsBy(p : A -> bool)(C : Set)
  : Alg (ListF A) C (Const (list (list A))) :=
  rollAlg (fun R reveal 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 R) 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

## XX:10 Subsidiary Recursion in Coq

```

Definition MapThroughAlg{B : Set}(f:mappedT A B)
  (C : Set) : Alg (ListF A) C (Const (list B)) :=
  rollAlg (fun R reveal 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)

```

rule :: Eq a => [a] -> [(Int,a)]
rule = 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

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

### 257 3.4 Run-length encoding (Rle.v)

258 Using mapThrough, we can quite concisely implement *run-length encoding*, a basic data-  
 259 compression algorithm where maximal sequences of  $n$  occurrences of element  $e$  are summarized  
 260 by the pair  $(n, e)$  [19]. Haskell code is in Figure 9. Recall that (== a) tests its input for  
 261 equality with a. The compressSpan helper function gathers up all elements at the start of  
 262 the tail as that are equal to the head a. This prefix is returned as p, with the remaining suffix  
 263 as s. The pair (1 + length p, a) is returned to summarize a :: p. The mapThrough  
 264 combinator then iterates compressSpan through the suffix s.

265 Assuming A : Set and an equality test eqb : A -> A -> bool on it, we port this code  
 266 to our Coq infrastructure in Figure 10. The function compressSpan is written at the type  
 267 mappedT A (nat \* A) that will be required by mapThrough. Unfolding the definition of  
 268 mappedT, compressSpan has type:

```

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

```

269 It will be invoked by the code for mapThrough with a fold function fo with anchor type R,  
 270 and then has the responsibility of extracting from the tail at type R (second input) a result

```

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

271 upon which `mapThrough` should recurse (second component of the output pair). Then we  
 272 define an algebra `RleAlg` by supplying `compressSpan` as the function to map through, to  
 273 `MapThroughAlg` (Figure 8). Following the pattern seen above, we define function `rle` for  
 274 top-level recursions using `fold` (we could also define a subsidiary version `rler`).

## 275 4 Derivation of subsidiary recursion

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

### 278 4.1 Retractive-positive recursive types (`Mu.v`)

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

285 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
```

286 which satisfies the identity-preservation law for functors:

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

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

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

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

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

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

## XX:12 Subsidiary Recursion in Coq

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

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

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

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

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

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

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

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

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

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

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

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

316 Noncanonicity must be handled carefully when reasoning about functions defined with our  
317 interface. We will see an example in Section 6. First, though, let us complete the exposition  
318 of our implementation of subsidiary recursion.

```

Definition SubrecF(C : Set) :=
  forall (X : Set -> Set) (FunX : Functor X), Alg C 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`)

## 319 4.2 The implementation of `Subrec` (`Subrec.v`)

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

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

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

328 The rest of the interface for `Subrec` is shown in Figure 13. We have `fold`, which is a fold  
 329 function with anchor type `Subrec`. To fold an algebra `alg` with carrier `X` (with `fmap` function  
 330 given by `FunX`) over `d : Subrec`, we `unroll` the definition of `Subrec` and apply that to the  
 331 algebra (with its carrier).

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

- 339 ■ `Subrec` is passed as the value for the abstract type `R`; this is what enables all the rest of  
 340 the components to have the desired types, since we will pass values that have `Subrec`  
 341 where the interface mentions `R`.
- 342 ■ the identity function is passed as the value for `reveal : R -> Subrec`.
- 343 ■ The function `fold`, which expects an algebra with anchor type `Subrec`, is passed as the  
 344 fold function of type `FoldT Alg R`.
- 345 ■ For the `rec : R -> X R` function, we pass `(fold X xmap alg) : Subrec -> X Subrec`.
- 346 ■ For the subdata structure of type `F R`, we pass `d : F Subrec`.

347 Finally, Figure 13 defines `out` as a subsidiary recursion, given any fold function with its  
 348 anchor type `R`. Outside the recursion, `d` has type `F R`; inside the recursion it has type `F R'`  
 349 where `R'` is the abstract type of the subsidiary recursion. So `out` implements the idea that  
 350 unfolding an abstract type one step is just a trivial case of subsidiary recursion.

## 351 5 Interface for subsidiary induction (`Subreci.v`)

352 We have seen how to write subsidiary recursions in Coq. But can one reason about these? To  
 353 wrap up this paper, we will see an interface for subsidiary induction in Coq, and example proofs

## XX:14 Subsidiary Recursion in Coq

```

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 (fun x => x) 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)

354 written using this interface. Subsidiary induction is written just as the natural extension  
 355 of subsidiary recursion, which worked over **Sets**, to **Subrec**-predicates. The development is  
 356 parametrized by a functor **F** and a functor **Fi** : (**Subrec** -> **Prop**) -> (**Subrec** -> **Prop**)  
 357 over **Subrec**-indexed propositions (i.e., predicates). Just as functors need an **fmap** function,  
 358 we here need an indexed version, of type **fmapiT Subrec Fi** (definition elided.)

359 The central definitions for the type **Subreci** : **Subrec** -> **Prop** are given in Figure 14.  
 360 Where having a value **x** of **Subrec** entitles us to define subsidiary recursions to inhabit types  
 361 **X Subrec**, a value of type **Subreci x** lets us prove properties of **x** by subsidiary induction.  
 362 Briefly: **kMo** is the kind for *motives*, namely predicates on **Subrec** [15]. **KAlg** is the kind for  
 363 indexed algebras. **FoldTi** is the indexed version of **FoldT**: it expresses provability of **X C** for  
 364 **d**, based on an indexed algebra and a value of type **C d**, where **C** is the (indexed) anchor type.  
 365 **AlgFi** and **Algi** are indexed versions of the algebras we saw for recursion. The **rec** function  
 366 (Figure 3) has now become an induction hypothesis: given any **d** where **R d** holds, **ih** proves  
 367 **X R d**. A value of type **R d** is thus a license to induct on **d**. Finally, the algebra is given a  
 368 subdata structure indexed by **d** : **Subrec**, and must produce a proof of **X R d**. **Subreci** is  
 369 defined as the suitably indexed fixed-point of **SubrecFi**, which is the natural indexed version  
 370 of **SubrecF**.

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

## 6 Examples of subsidiary induction

377 For proving the main theorem about run-length encoding, we need several lemmas about  
 378 **span**, shown in Figure 16. For lack of space, we just state the properties. The first says that  
 379 appending the results of a call to **span** returns the original list (module some conversions to  
 380 **list** from **List**). The second uses the inductive type **Forall** from Coq’s standard library  
 381 to state that all the elements of the prefix returned by **span** satisfy **p**. These lemmas are  
 382 proved using an indexed algebra where the indexed anchor type is not used (so the carriers  
 383 are constant indexed functors returning the types shown). But **GuardPresF** uses the indexed  
 384 anchor type (its argument **S**), to express that whenever **spanh** returns a suffix **r**, that suffix  
 385 satisfies the indexed anchor type. This enables us to invoke an outer induction hypothesis on  
 386 this suffix, when reasoning subsidiarily about **span**. Using these lemmas, we can write a short  
 387 proof by subsidiary induction of the following, where **rld** : **list (nat \* A)** -> **list A** is

```

Definition kMo := Subrec -> Prop.
Definition KAlgi := kMo -> (kMo -> kMo) -> Set.
Definition FoldTi(alg : KAlgi)(C : kMo) : kMo :=
  fun d => forall (X : kMo -> kMo) (xmap : fmapiT Subrec X),
    alg C X -> C d -> X C d.

Definition AlgFi(A : KAlgi)(C : kMo)(X : kMo -> kMo) : Set :=
  forall (R : kMo)
    (reveal : (forall (d : Subrec), R d -> C d))
    (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 C 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)



## XX:16 Subsidiary Recursion in Coq

```

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`

388 the obvious decoding function:

```

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

```

389 We invoke the lemmas about `span` subsidiarily, so that we may apply our induction hypothesis  
 390 to the suffix that `span` returns (on which `mapThrough` then recurses). For example, the  
 391 lemma for `GuardPresF` takes in the indexed fold function `foi` from the outer induction (for  
 392 `RldRle`), to show that the abstract predicate `R` applies to the suffix `r` returned by `span`. This  
 393 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.

```

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

## 7 Related Work

**Termination.** In some tools, like Coq, Agda, and Lean, termination is checked statically, based on structural decrease. Others, like Isabelle/HOL, allow one to write recursions first, and prove (possibly with automated help) their termination afterwards [12]. These tools all support well-founded recursion, but in constructive type theory, evidence of well-foundedness then propagates through code. In contrast, our approach here, 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. For more on partiality and recursion in theorem provers, see [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]. 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. Like them, we find that subsidiary recursion subsumes course-of-value recursion. The technique of sized types is similar to our method in providing a type-based method for termination [2]. This technique enriches datatypes with abstract sizes, which must then be propagated through code, using dependent types. Our **Subrec** does not require dependent types, resulting in just polymorphically typed code for the examples we saw (**Subreci**, for proofs, of course does use dependent types).

**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 adopted 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 using these methods, but rather apply them to justify 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 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.

## 447 — References

- 448 1 Ki Yung Ahn and Tim Sheard. A hierarchy of mendler style recursion combinators: Taming  
449 inductive datatypes with negative occurrences. In *Proceedings of the 16th ACM SIGPLAN*  
450 *International Conference on Functional Programming, ICFP '11*, pages 234–246, New York,  
451 NY, USA, 2011. ACM.
- 452 2 Gilles Barthe, Maria João Frade, Eduardo Giménez, Luís Pinto, and Tarmo Uustalu. Type-  
453 based termination of recursive definitions. *Math. Struct. Comput. Sci.*, 14(1):97–141, 2004.  
454 URL: <https://doi.org/10.1017/S0960129503004122>, doi:10.1017/S0960129503004122.
- 455 3 Frédéric Blanqui. Inductive types in the calculus of algebraic constructions. *Fundam.*  
456 *Informaticae*, 65(1-2):61–86, 2005. URL: <http://content.iospress.com/articles/fundamenta-informaticae/fi65-1-2-04>.
- 457 4 Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive  
458 theorem provers - an overview. *Mathematical Structures in Computer Science*, 26(1):38–88, 2016.  
459 URL: <https://doi.org/10.1017/S0960129514000115>, doi:10.1017/S0960129514000115.
- 460 5 Robin Cockett and Dwight Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor,  
461 *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings.  
462 AMS, 1992.
- 463 6 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and  
464 Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn,*  
465 *USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages  
466 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/  
467 3-540-52335-9\_47.
- 468 7 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and program-  
469 ming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction -*  
470 *CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July*  
471 *12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages  
472 625–635. 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:  
473 10.1007/978-3-030-79876-5\_37.
- 474 8 The Agda development team. *Agda*, 2021. Version 2.6.2.1. URL: [https://agda.readthedocs.](https://agda.readthedocs.io/en/v2.6.2.1/)  
475 [io/en/v2.6.2.1/](https://agda.readthedocs.io/en/v2.6.2.1/).
- 476 9 Denis Firsov, Richard Blair, and Aaron Stump. Efficient mendler-style lambda-encodings in  
477 cedille. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th*  
478 *International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC*  
479 *2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer*  
480 *Science*, pages 235–252. Springer, 2018.
- 481 10 Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in  
482 cedille. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN*  
483 *International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA,*  
484 *USA, January 8-9, 2018*, pages 215–227. ACM, 2018.
- 485 11 Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh,  
486 1987.
- 487 12 Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL*. URL: [https://isabelle.](https://isabelle.in.tum.de/doc/functions.pdf)  
488 [in.tum.de/doc/functions.pdf](https://isabelle.in.tum.de/doc/functions.pdf).
- 489 13 Alexander Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Au-*  
490 *tom. Reasoning*, 44(4):303–336, 2010. URL: <https://doi.org/10.1007/s10817-009-9157-2>,  
491 doi:10.1007/s10817-009-9157-2.
- 492 14 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2021.  
493 Version 8.13.2. URL: <http://coq.inria.fr>.
- 494 15 Conor McBride. Elimination with a motive. In Paul Callaghan, Zhaohui Luo, James McKinna,  
495 and Robert Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES*  
496 *2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes*  
497

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