



# 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 [6, 11, 14, 5]. 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

44 from another recursion, namely `span`. This is not allowed under structural termination, but  
45 will be permitted by subsidiary recursion.

### 46 1.1 Summary of results

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

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

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

70 All presented derivations have been checked with Coq version 8.13.2, using command-line  
71 option `-impredicative-set`. The code may be found as release `itp-2022` (dated prior  
72 to the ITP 2022 deadline) at <https://github.com/astump/coq-subsidiary>. The paper  
73 references files in this codebase, as an aid to the reader wishing to peruse the code.

## 74 2 Interface for subsidiary recursion

75 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. 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, 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  $A$ .

Algebras for our subsidiary recursion are more complex. First, for reasons we will explain further below, the carrier of the algebra will be a functor  $X : \mathbf{Set} \rightarrow \mathbf{Set}$ . 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`). The algebra is presented with:

- a type  $R : \mathbf{Set}$ , 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` :  $\text{FoldT Alg } R$ , 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  $\text{FoldT Alg } R$  below.
- a function `eval` :  $R \rightarrow X\ R$ , 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* [17]. As in other domains using the term “universe”, we have an entity (here,  $R$ ) from which one cannot escape by using the available operations (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 `eval`) 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  $\text{FoldT}$ . Let us see these details next.

## 2.2 The interface in more detail

Let us consider two central files from our development.

### 2.2.1 Subrec.v

This file is parametrized by a signature functor  $F$  of type  $\mathbf{Set} \rightarrow \mathbf{Set}$ . It provides the implementation of subsidiary recursion. Two crucial values are `Subrec` :  $\mathbf{Set}$ , which is the type to use for subsidiary recursion; and `inn` :  $F\ \text{Subrec} \rightarrow \text{Subrec}$ , which is to be used as a constructor for that type. An important point, however, is that `Subrec.v` does not provide an induction principle based on `inn`. 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

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

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`)

### 117 2.2.2 List.v

118 This file specializes the development in `Subrec.v` to the case of lists (parametrized by the  
119 type `A` of elements). In general, to use our development to get subsidiary recursion over some  
120 datatype, one will have a similar “shim” file. For space reasons, we just give the example of  
121 lists. The file defines the signature functor `ListF`, shown in Figure 2. We then define `List`  
122 to be `Subrec`, with the instantiation of `F` to `ListF A`. This type `List` is not to be confused  
123 with the type `list` of lists in Coq’s standard library. As noted previously, our development  
124 is meant to be used in extension of existing inductive datatypes, not replacing them. The  
125 figure also shows constructors `mkNil` and `mkCons` for `List`, and types for conversion functions  
126 between `List` and `list`; one direction uses Coq’s structural recursion, the other uses our  
127 subsidiary recursion (code elided).

### 128 2.3 Algebras for subsidiary recursion

129 `Subrec.v` also defines the notion of algebra that is used for writing recursions. The central  
130 definitions are in Figure 3. `KAlg` is the kind for the type-constructor for algebras, as we see  
131 in the definition of `Alg`. This type-constructor `Alg` is a fixed-point of the type `AlgF`. The  
132 fixed-point is taken using `MuAlg`, which implements our retractive-positive recursive types  
133 (Section 4.1) at kind `KAlg`. Using `Alg` will require that `AlgF` only uses its parameter `Alg`  
134 positively. We will confirm this shortly.

135 `FoldT Alg C` is the type for fold functions which apply algebras of type `Alg` to data of  
136 type `C`, which we have already dubbed the *anchor type* of the recursion. At the top level of  
137 code, the anchor type would just be `List` (for example). When one initiates a subsidiary  
138 recursion, though, the anchor type will instead be the abstract type `R` for the outer recursion.  
139 The variable `Alg` occurs only positively (but not strictly positively) in `AlgF`, because it occurs  
140 negatively in `FoldT Alg R` which occurs negatively in `AlgF Alg C X`. So we can indeed take  
141 a fixed-point of `AlgF` to define the constant `Alg`.

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

```

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

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

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

Definition Alg : KAlg := MuAlg AlgF.

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 (`Subrec.v`)

149 As noted already, for subsidiary recursion, algebras have a carrier  $X$  which depends  
 150 (functorially) on a type. This is so that (i) inside an inner recursion we may compute a result  
 151 of some type that may mention  $R$ , but (ii) outside that recursion, the result will mention the  
 152 anchor type  $C$ . The `eval` function returns something of type  $X R$ , and so does the algebra  
 153 itself; this demonstrates (i). For (ii): if we look at the definition of `FoldT` in the figure, we  
 154 see that folding an algebra of type `alg C X` over a value of type  $C$  produces a result of type  
 155  $X C$ . Having a functor for the carrier of the algebra gives us the flexibility to type results  
 156 inside a recursion with the abstract type  $R$ , but view those results as having the anchor type  
 157  $C$  outside the recursion.

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

### 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 `List A`, we would like to compute  
 165 a pair of type `list A * 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 `FoldT (Alg (ListF A)) R`), we would like to map an  
 169 input list of type  $R$  to a pair of type `list A * R`. The first component of this pair is going to

## XX:6 Subsidiary Recursion in Coq

```

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 tl =>
      if p hd then
        match (span tl) with
        | SpanNoMatch => SpanSomeMatch [hd] tl
        | SpanSomeMatch l r => SpanSomeMatch (hd::l) r
        end
      else
        SpanNoMatch
    end).

```

■ **Figure 4** The algebra `SpanAlg` for the `span` function

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:

```

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

```

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

```

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

```

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

```

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

```

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

193 Figure 4 gives the algebra `SpanAlg` for computing `span`. The type of `SpanAlg p C` is

```

194 Alg (ListF A) C SpanF

```

195 This states that we are defining an algebra (`Alg`) for the `ListF A` functor, with anchor type  
 196 `C` and carrier `SpanF`. `SpanF` has type `Set -> Set`, as required for the carriers of our algebras.

```

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`

197 The definition of `SpanAlg` is actually parametrized by `C`, which is good, as it means we can  
 198 use `SpanAlg` for top-level or subsidiary recursions.

199 Let us continue through the code for `SpanAlg` (Figure 5). We use `rollAlg` to create an  
 200 algebra from something whose type is an application of `AlgF`. This takes in all the components  
 201 of the recursion universe: the abstract type `R`, the `reveal` function (not needed in this case),  
 202 the fold function (`fo`) for any subsidiary recursions (also not needed here), a function we  
 203 choose to name `span` for making recursive calls, and finally `xs : ListF A R`. The algebra  
 204 pattern-matches on this `xs`. In the cases where it is empty or where its head (`hd`) does not  
 205 satisfy `p`, we return `SpanNoMatch`. This signals to the caller that we really wished to return  
 206 `([],xs)`, but could not because we do not have `xs` at type `R`. If the head does satisfy `p`, then  
 207 we recurse on the tail (`tl : R`) by calling the provided `span : R -> SpanF R`. If `span tl`  
 208 returns `SpanNoMatch`, that means that we should make `tl` the suffix in the pair we return  
 209 (via `SpanSomeMatch`). Happily, we have `tl : R` here, so we can do this. In either case (for  
 210 return value of `span tl`), we add the head to the front of the prefix.

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

### 219 3.2 The wordsBy function (WordsBy.v)

220 Let us now see how to write `wordsBy`, our example function from Section 1, using `breakr`  
 221 subsidiarily. The code is in Figure 6, assuming a type `A : Set`. The setup is similar to that  
 222 for `span`. We first define an algebra `WordsBy`, parametrized by anchor type `C` (and also the  
 223 predicate `p`), of type

```

224 Alg (ListF A) C (Const (list (list A)))

```



## XX:8 Subsidiary Recursion in Coq

```
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.

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

■ **Figure 6** The `wordsBy` and `wordsByR` function, defined using an algebra

225 This says that `WordsBy p C` is an algebra (`Alg`) for the `ListF A` functor, with anchor type `C`,  
226 and carrier `Const (list (list A))`. `Const` is the combinator for creating the object part  
227 of constant functors; `FunConst` creates the morphism part (i.e., the `fmap` function). We use  
228 `Const` where the return type of the algebra will not depend on its abstract type `R`. Here, we  
229 are constructing from scratch a list of lists, so it will not be legal to recurse on the list itself,  
230 or its (list) elements. So we just use the `list` type of Coq’s standard library.

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

239 Here we see the benefit of our approach. From Figure 5, the return type of `breakr` is  
240 `list A * R`, where `R` is the anchor type of the provided fold function `fo`. And `fo` has type  
241 `FoldT (ListF A) Alg R`, from the definition of `AlgF` in Figure 3 (instantiating the functor  
242 with `ListF A`). This means that from the invocation of `breakr`, we get `w : list A` and  
243 `z : R`. And so we can indeed apply `wordsBy : R -> list (list A)` to `z` to recurse.

### 244 3.3 The `mapThrough` function (`MapThrough.v`)

245 The Haskell library `Data.List.Extra` has a function `repeatedly`, defined essentially as in  
246 Figure 7. We attempt a more informative name. This is like the standard `map` function on  
247 lists, except that the function `f` that we are mapping (or “mapping through”) takes in not  
248 just the current element `a`, but also the tail `as`. It then returns the value `b` to include in the



```

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

```

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

249 output list, and whatever other list it wishes, upon which `mapThrough` will recurse.  
 250 To write this combinator using our infrastructure for subsidiary recursion, we need to  
 251 supply the mapped function with the fold function for `mapThrough`'s recursion. This is so  
 252 that the mapped function can initiate a subsidiary recursion, returning a value in the abstract  
 253 type `R` of `mapThrough`'s recursion. So the type we will use for mapped functions is:

```

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

```

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

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

### 264 3.4 Run-length encoding (`Rle.v`)

265 Using `mapThrough`, we can quite concisely implement *run-length encoding*, a basic data-  
 266 compression algorithm where maximal sequences of  $n$  occurrences of element  $e$  are summarized  
 267 by the pair  $(n, e)$  [15]. Haskell code is in Figure 9. Recall that `(== a)` tests its input for  
 268 equality with `a`. The `compressSpan` helper function gathers up all elements at the start of  
 269 the tail `as` that are equal to the head `a`. This prefix is returned as `p`, with the remaining suffix

## XX:10 Subsidiary Recursion in Coq

```
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`

```
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` (Figure 8)

270 as `s`. The pair `(1 + length p, a)` is returned to summarize `a :: p`. The `mapThrough`  
271 combinator then iterates `compressSpan` through the suffix `s`.

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

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

277 It will be invoked by the code for `mapThrough` with a fold function `fo` with anchor type  
278 `R`, and then has the responsibility of mapping the tail at type `R` (second input) to a result  
279 upon which `mapThrough` should recurse (second component of the output pair). Then we  
280 define an algebra `RleAlg` by supplying `compressSpan` as the function to map through, to  
281 `MapThroughAlg` (Figure 8). Following the pattern seen above, we define function `rle` for  
282 top-level recursions using `fold` (we could also define a subsidiary version `rler`).

## 283 4 Derivation of subsidiary recursion

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

### 286 4.1 Retractive-positive recursive types (`Mu.v`)

287 As we have seen, our definitions require a form of positive-recursive types, to allow algebras  
288 to accept fold functions that themselves require algebras, and also for the definition of  
289 `Subrec` (which we will see in more detail in the next section). Full positive-recursive  
290 types are incompatible with Coq's type theory [4]. One can impose some restrictions on

```

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

large eliminations which then enable positive-recursive types [2], but this requires changing the underlying theory. Here we take a different approach, exploiting Coq’s impredicative polymorphism.

This is done in a file `Mu.v`, whose central definitions are in Figure 11. The development is parametrized by `F : Set -> Set` which is assumed to have an `fmap` function (morphism part of the functor) of type

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

which satisfies the identity-preservation law for functors:

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

Let us consider the code in Figure 11. The critical idea is embodied in the definition of `Mu`. Ideally, we would like to have a definition like

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

This is exactly what is used in approaches to modular datatypes in functional programming, like Swierstra’s [18]. But this definition is (rightly) rejected by Coq, as instantiations of `F` that are not strictly positive would be unsound.

Instead, the definition of `Mu` in Figure 11 weakens this ideal definition to a strictly positive approximation:

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

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

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

## XX:12 Subsidiary Recursion in Coq

```
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`

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

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

326 But given this, there are infinitely many other equivalent definitions. For any `Q : Set`, we  
327 could take

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

329 Since `fmap f (NilF A)` equals just `NilF B` for `f : A -> B`, if we apply `outMu` (of Figure 11)  
330 to `mkNil'` or `mkNil`, we will get `NilF (List A)`. But critically, `mkNil` and `mkNil'` are not  
331 equal, neither definitionally nor provably. One can define a function that puts `Mu` values in  
332 normal form by folding `inMu` over them. Then `mkNil` and `mkNil'` will have the same normal  
333 form, and be equivalent in that sense. But the fact that they are not provably equal is what  
334 we term noncanonicity.

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

### 338 4.2 The implementation of `Subrec` (`Subrec.v`)

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

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

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

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

352 More interesting is the definition of `inn`, which is the critical point where the recursion  
353 universe is implemented. To create a value of type `Subrec` from data of type `F Subrec`, the  
354 definition of `inn` rolls a value of type `SubrecF Subrec` (we saw this type unfolded at the

```

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`

start of this section). This value takes in a carrier `X`, its `fmap` function `xmap`, and an algebra `alg` with that carrier. Note that the anchor type of this algebra is `Subrec`. It will then call `alg` (after unrolling it) with implementations for the components of the recursion universe (cf. Section 2.1, also Figure 3):

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

Finally, Figure 13 defines `out` as a subsidiary recursion, given any fold function with its anchor type `R`. The code for `out` just folds an algebra over the input of type `R`, where that algebra simply returns the subdata structure it is given. Outside the recursion, this has type `F R`; inside the recursion it has type `F R'` where `R'` is the abstract type of the subsidiary recursion (named in the figure just for discussion here). So `out` implements the idea that unfolding an abstract type one step is just a trivial case of subsidiary recursion.

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

We have seen how to write subsidiary recursions in Coq. But can one reason about these? To wrap up this paper, we will see an interface for subsidiary induction in Coq, and example proofs written using this interface. Subsidiary induction is written just as the natural extension of subsidiary recursion, which worked over `Sets`, to `Subrec`-predicates. The development is parametrized by a functor `F` and a functor `Fi : (Subrec -> Prop) -> (Subrec -> Prop)` over `Subrec`-indexed propositions (i.e., predicates). Just as functors need an `fmap` function, we here need an indexed version, of type `fmapiT Subrec Fi` (definition elided.)

The central definitions for the type `Subreci : Subrec -> Prop` are given in Figure 14. Where having a value `x` of `Subrec` entitles us to define subsidiary recursions to inhabit types `X Subrec`, a value of type `Subreci x` lets us prove properties of `x` by subsidiary induction. Briefly: `kMo` is the kind for *motives*, namely predicates on `Subrec` [12]. `KAlgi` is the kind for indexed algebras. `FoldTi` is the indexed version of `FoldT`: it expresses provability of `X C` for `d`, based on an indexed algebra and a value of type `C d`, where `C` is the anchor type, now indexed. `AlgFi` and `Algi` are indexed versions of the algebras we saw for recursion.

## XX:14 Subsidiary Recursion in Coq

```
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

388 The `eval` function (Figure 3) has now become an induction hypothesis: given any `d` where  
389 `R d` holds, `ih` proves `X R d`. A value of type `R d` is thus a license to induct on `d`. Finally,  
390 the algebra is given a subdata structure indexed by `d : Subrec`, and must produce a proof  
391 of `X R d`. `Subreci` is defined as the suitably indexed fixed-point of `SubrecFi`, which is the  
392 natural indexed version of `SubrecF`.

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

## 398 6 Examples of subsidiary induction

399 For proving the main theorem about run-length encoding, we need several lemmas about  
400 `span`, shown in Figure 16. For lack of space, we just state the properties. The first says that  
401 appending the results of a call to `span` returns the original list (module some conversions to  
402 `list` from `List`). The second uses the inductive type `Forall` from Coq’s standard library  
403 to state that all the elements of the prefix returned by `span` satisfy `p`. These lemmas are  
404 proved using an indexed algebra where the indexed anchor type is not used (so the carriers  
405 are constant indexed-functor returning the types shown). But `GuardPresF` uses the indexed  
406 anchor type (its argument `S`), to express that whenever `spanh` returns a suffix `r`, that suffix  
407 satisfies the indexed anchor type. This enables us to invoke an outer induction hypothesis on  
408 this suffix, when using `span` subsidiarily. Using these lemmas, we can write a short proof

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 (Alg ListF ListFi) R d.
```

■ **Figure 15** The indexed version ListFi of ListF

```
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

409 by subsidiary induction of the following, where rld : list (nat \* A) -> list A is the  
410 obvious decoding function:

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

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

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

420 Finally, as promised, a note on noncanonicity. When proving properties about subsidiary  
421 recursions on **xs : List A**, one should be aware that nothing prevents the property from  
422 being applied to noncanonical Lists. For example, suppose we wish to prove that if all  
423 elements of a list satisfy **p**, then the suffix returned by **span** is empty. It is dangerous to



## XX:16 Subsidiary Recursion in Coq

phrase this as “the suffix equals `mkNil`”, because for a noncanonical input `xs`, `span` will return that same noncanonical `xs` as the suffix (and so it may be a noncanonical empty list, not equal to `mkNil`). The solution in this case is to use a function `getNil (List.v)` that computes an empty list from `xs`. So the statement that one can prove is:

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

## 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 [9]. Well-founded recursion replaces structural decrease with decrease in a well-founded ordering. At least 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. For more on partiality and recursion in theorem provers, see [3].

Subsidiary recursion can be seen as a generalization of *nested recursion*, which allows recursive calls of the form `f (f x)` [10]. In subsidiary recursion, these are generalized to the form `f (g x)`, where `g` could be `f` or another recursively defined function.

**Mendler encoding.** Mendler introduced the basic idea of using universal abstraction to support compositional termination checking; an accessible source is [13]. He introduces 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 works explored the categorical perspective on Mendler-style recursion [19], and its use with negative type schemes [1]. Previous work from our group showed how to derive inductive datatypes in Cedille using encodings extending the Mendler encoding [7, 8, 16].

## 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 [17]. We may also attempt to extend the recursion universe further, to allow other forms of recursion like divide-and-conquer, where some (necessarily limited) ability to recurse on values built using constructors is required.

## References

- 1 Ki Yung Ahn and Tim Sheard. A hierarchy of mendler style recursion combinators: Taming inductive datatypes with negative occurrences. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP '11*, pages 234–246, New York, NY, USA, 2011. ACM.
- 2 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>.
- 3 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.
- 4 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.
- 5 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.
- 6 The Agda development team. *Agda*, 2021. Version 2.6.2.1. URL: <https://agda.readthedocs.io/en/v2.6.2.1/>.
- 7 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.
- 8 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.
- 9 Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL*. URL: <https://isabelle.in.tum.de/doc/functions.pdf>.
- 10 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.
- 11 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2021. Version 8.13.2. URL: <http://coq.inria.fr>.
- 12 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 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.
- 13 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.
- 14 Wolfgang Naraschewski and Tobias Nipkow. Isabelle/hol, 2020. URL: <http://www.cl.cam.ac.uk/research/hvg/Isabelle/>.
- 15 David Salomon and Giovanni Motta. *Handbook of Data Compression*. Springer, 2009.

- 512    **16**    Aaron Stump. From realizability to induction via dependent intersection. *Ann. Pure Appl.*  
513           *Log.*, 169(7):637–655, 2018. URL: <https://doi.org/10.1016/j.apal.2018.03.002>, doi:10.  
514           1016/j.apal.2018.03.002.
- 515    **17**    Aaron Stump, Christopher Jenkins, Stephan Spahn, and Colin McDonald. Strong func-  
516           tional pearl: Harper’s regular-expression matcher in cedille. *Proc. ACM Program. Lang.*,  
517           4(ICFP):122:1–122:25, 2020. URL: <https://doi.org/10.1145/3409004>, doi:10.1145/  
518           3409004.
- 519    **18**    Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008. URL:  
520           <https://doi.org/10.1017/S0956796808006758>, doi:10.1017/S0956796808006758.
- 521    **19**    Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic J. of*  
522           *Computing*, 6(3):343–361, September 1999.