



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 [5, 7, 10, 4]. Termination is usually enforced by a syntactic check for structural decrease. This structural termination is sufficient for many basic functions. For example, the well-known `span` function from Haskell's standard library (`Data.List`) takes a list and returns a pair of the maximal prefix satisfying 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`. So the input to the recursive call is a subdatum of the input, and hence this definition is structurally decreasing. In the appropriate syntax, it 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` (from `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"]`; so `wordsBy isSpace` has the same behavior as `words` (from `Data.List`). Code is in Figure 1.



© 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) = span (not . p) tl in
       (hd:w) : wordsBy p z
```

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

44 The first recursive call, `wordsBy p tl`, is structural. But in the second, we invoke `wordsBy p`
45 on a value obtained from another recursion, namely `span`. This is not allowed under structural
46 termination, but will be permitted by subsidiary recursion as derived below.

47 1.1 Summary of results

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

56 An important technical novelty of our approach is a derivation of a weakened form of
57 positive-recursive type in Coq. Coq (Agda, and Lean) restrict datatypes D to be strictly
58 positive: in the type for any constructor of D , D cannot occur to the left of any arrows.
59 Our derivation needs to use positive-recursive types, where D may occur to the left of an
60 even number (only) of arrows. Coq requires strict positivity because in the presence of other
61 features of Coq's theory, full positive-recursive types lead to a paradox [3]. We present a
62 way to derive a weakened form of positive-recursive type that is sufficient for our examples
63 (Section 4.1). The weakening is to require only that $F \mu$ is a retract of μ , where μ is the
64 recursive type and $F \mu$ its one-step unfolding. Usually these types are isomorphic. Hence, we
65 dub these **retractive-positive** recursive types. This weakening has the negative consequence
66 of leading to a form of noncanonicity, but we will see how to work around this. Our definition
67 of retractive-positive recursive types makes essential use of impredicate quantification, and
68 hence cannot be soundly recapitulated in a predicative theory like Agda's.

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

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

2 Interface for subsidiary recursion

This section presents the interface our Coq development provides for subsidiary recursion.

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` : `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 `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 recursions. Unlike with structural termination, though, the interface here is type-based and hence compositional. As we will see, it supports nested and higher-order recursions.

As in previous work, we dub this interface a *recursion universe* [12]. 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 ϵ_0 and ω^- , 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, particularly why X has type $\mathbf{Set} \rightarrow \mathbf{Set}$, and the definition of `FoldT`. Let us see these and other 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

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`)

119 a constructor for that type. An important point, however, is that `Subrec.v` does not provide
120 an induction principle based on `inn`. Induction is derived later (Section 5). `Subrec.v` makes
121 critical use of retractive-positive recursive types, to take a fixed-point of a construction based
122 on `F`. We present these recursive types in Section 4.1 below.

123 2.2.2 List.v

124 This file specializes the development in `Subrec.v` to the case of lists (parametrized by the
125 type `A` of elements). In general, to use our development to get subsidiary recursion over some
126 datatype, one will have a similar “shim” file. For space reasons, we just give the example of
127 lists. The file defines the signature functor `ListF`, shown in Figure 2. We then define `List`
128 to be `Subrec`, with the instantiation of `F` to `ListF A`. This type `List` is not to be confused
129 with the type `list` of lists in Coq’s standard library. As noted previously, our development
130 is meant to be used in extension of existing inductive datatypes, not replacing them. The
131 figure also shows constructors `mkNil` and `mkCons` for `List`, and types for conversion functions
132 between `List` and `list` (see Section 4 for the code).

133 2.3 Algebras for subsidiary recursion

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

140 The type `FoldT Alg C` is the type for fold functions which apply algebras of type `Alg`
141 to data of type `C`, which we have already dubbed the *anchor type* of the recursion. At the
142 top level of code, the anchor type would just be `List` (for example). When one initiates a
143 subsidiary recursion, though, the anchor type will instead be the abstract type `R` for the
144 outer recursion.

145 The variable `Alg` occurs only positively (but not strictly positively) in `AlgF`, because
146 it occurs negatively in `FoldT Alg R` which occurs negatively in `AlgF Alg C X`. So we can
147 indeed take a fixed-point of `AlgF` to define the constant `Alg`.

148 Let us look at `AlgF`. As noted already, each recursion is based on an abstract type `R`,
149 representing the data upon which we will recurse. This is the first argument to a value of

```

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`)

150 type `AlgF Alg C X`. An algebra can assume nothing about `R` except that it supports the
 151 following operations. First there is `reveal`, which turns an `R` into a `C`. This reveals that the
 152 data of type `R` are really values of the anchor type of this recursion. Next we have `fold`,
 153 which will allow us to fold another algebra over data of type `R`. We will use `fold` to initiate
 154 subsidiary recursions. Then there is `eval`, for recursive calls on data of type `R`.

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

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

167 3 Examples of subsidiary recursion

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

169 3.1 The span function (Span.v)

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

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

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

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

185 Before we define spanr , we must resolve a small problem. If the first element of the input
 186 list xs to span does not satisfy p , then span should return $([], \text{xs})$. But when recursing
 187 on xs , we will see it only in the form of a subdata structure of type $F R$. We will not be able
 188 to return it from our recursion at type R , and hence we would not be able to return $([], \text{xs})$
 189 as desired. To work around this, we will have our recursion return a value of type $\text{SpanF } R$:

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

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

198 Figure 4 gives the algebra SpanAlg for computing span , and the code for spanr . We
 199 elide the proof SpanFunctor that SpanF is indeed a Functor , and make X implicit in the
 200 constructors of SpanF . The type of $\text{SpanAlg } p \ C$ is

```
201 Alg (ListF A) C SpanF
```

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

206 Let us continue through the code for SpanAlg (Figure 4). We use rollAlg to create an
 207 algebra from something whose type is an application of AlgF . This takes in all the components
 208 of the recursion universe: the abstract type R , the reveal function (not needed in this case),
 209 the fold function (fo) for any subsidiary recursions (also not needed here), a function we
 210 choose to name span for making recursive calls, and finally $\text{xs} : \text{ListF } A \ R$. The algebra
 211 pattern-matches on this xs . In the cases where it is empty or where its head (hd) does not

```

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).

```

```

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.

```

■ **Figure 4** The algebra `SpanAlg` for the `span` function, and some functions based on it

212 satisfy `p`, we return `SpanNoMatch`. This signals to the caller that we really wished to return
 213 `([],xs)`, but could not because we do not have `xs` at type `R`. If the head does satisfy `p`, then
 214 we recurse on the tail (`tl : R`) by calling the provided `span : R -> SpanF R`. If `span tl`
 215 returns `SpanNoMatch`, that means that we should make `tl` the suffix in the pair we return
 216 (via `SpanSomeMatch`). Happily, we have `tl : R` here, so we can do this. In either case (for
 217 return value of `span tl`), we add the head to the front of the prefix. We define `spanhr` to
 218 invoke the fold function it is given, on the algebra (`SpanAlg`).

219 The final twist is now in the definition of `spanr`. We call `spanhr` on the input `xs : R`.
 220 If `spanhr` returns `SpanNoMatch`, then we are supposed to return `([],xs)`, which we can do
 221 here, because we have `xs : R`. It was only inside the algebra that we lost the information
 222 that the subdata structure of type `F R` is derived from a value of type `R`. If `spanhr` returns
 223 `SpanSomeMatch`, then the return value gives us the nonempty prefix (`l`) and the suffix (`r`),
 224 which we then return.

225 We can easily define `break`, in Figure 5. The function `breakr` is a version of `break` that
 226 can be used for subsidiary recursion, similarly to `spanr` for `span`. Such a function always
 227 takes in a fold function (`fo`) with anchor type `R`, which then is used to fold the algebra in
 228 question.

XX:8 Subsidiary Recursion in Coq

```
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.

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

■ **Figure 5** The `break` function and its more flexible version, `breakr`, defined in terms of `spanr` (Figure 4)

229 3.2 The wordsBy function (WordsBy.v)

230 Let us now see how to write `wordsBy`, our example function from Section 1, using `spanr`
231 as a subsidiary recursion. The code is in Figure 6, assuming a type `A : Set`. The setup is
232 similar to that for `span`. We first define an algebra `WordsBy`, parametrized by anchor type `C`
233 (and also the predicate `p`), of type

```
234 Alg (ListF A) C (Const (list (list A)))
```

235 This says that `WordsBy p C` is an algebra (`Alg`) for the `ListF A` functor, with anchor type `C`,
236 and carrier `Const (list (list A))`. `Const` is the combinator for creating the object part
237 of constant functors; `FunConst` creates the morphism part (i.e., the `fmap` function). We use
238 it `Const` here and in other examples where the return type of the algebra will not depend on
239 its abstract type `R`. Here, we are constructing from scratch a list of lists, so it will not be
240 legal to recurse on the list itself, or its `(list)` elements. Instead, we just use the `list` type of
241 Coq's standard library.

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

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

255 3.3 The mapThrough function (MapThrough.v)

256 The Haskell library `Data.List.Extra` has a function `repeatedly`, which is defined essentially
257 as follows; I have attempted a more informative name:

```
258 mapThrough :: (a -> [a] -> (b, [a])) -> [a] -> [b]
259 mapThrough f [] = []
260 mapThrough f (a:as) = b : mapThrough f as'
261   where (b, as') = f a as
```



```

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

262 The idea is that the function is like the standard `map` function on lists, except that here,
 263 the function `f` that we are mapping (or “mapping through”) takes in not just the current
 264 element `a`, but also the tail `as`. It then returns the value `b` to include in the output list, and
 265 whatever other list it wishes, upon which `mapThrough` will recurse.

266 We can write this combinator using our infrastructure for subsidiary recursion. For this
 267 to work, we need to supply the mapped function with the fold function for `mapThrough`’s
 268 recursion. This is so that the mapped function can initiate a subsidiary recursion, returning
 269 a value in the abstract type `R` of `mapThrough`’s recursion. So the type we will use for mapped
 270 functions is:

```

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

```

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

276 Given this definition, the code for `mapThrough` and `mapThroughr` is in Figure 7. The
 277 code for `MapThroughAlg` is very similar (discounting syntax) to the Haskell code above. Here,
 278 though, when we call `f`, we must supply the abstract type `R` and fold function `fo`. Then,
 279 from the definition of `mappedT`, we have that `b` : `B` and `c` : `R`. So we may indeed invoke
 280 `mapThrough` : `R -> List B` on `c`. Note that as we are building up a new list from scratch
 281 (rather than just extracting some tail of the input list), we just return `list B`; we cannot
 282 perform further subsidiary recursion on the output.

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 :=
  fo (Const (list B)) (FunConst (list B)) (MapThroughAlg f R).

Definition mapThrough{B : Set}(f:mappedT A B) : List A -> list B :=
  mapThroughr (fold (ListF A)) f.
```

■ **Figure 7** The `mapThrough` and `mapThroughr` functions, with their defining algebra

283 3.4 Run-length encoding (Rle.v)

284 Using `mapThrough`, we can quite concisely implement *run-length encoding*, a basic data-
285 compression algorithm where maximal sequences of n occurrences of element e are summarized
286 by the pair (n, e) [11]. In Haskell, invoking `span` and `mapThrough` (defined above), the code
287 is simply

```
288 rle :: Eq a => [a] -> [(Int,a)]
289 rle = mapThrough compressSpan
290   where compressSpan a as =
291         let (p,s) = span (== a) as in
292         ((1 + length p, a),s)
```

293 (Recall that `(== a)` is a Haskell *section* testing its input for equality with `a`.) The
294 `compressSpan` helper function gathers up all elements at the start of the tail `as` that
295 are equal to the head `a`. This prefix is returned as `p`, with the remaining suffix as `s`. The pair
296 `(1 + length p, a)` is returned to summarize `a :: p`. We then use `mapThrough` to iterate
297 `compressSpan` through the suffix `s`.

298 Assuming `A : Set` and an equality test `eqb : A -> A -> bool` on it, code for run-length
299 encoding using our infrastructure is listed in Figure 8. The function `compressSpan` is written
300 at the type `mappedT A (nat * A)` that will be required by `mapThrough`. Unfolding the
301 definition of `mappedT`, `compressSpan` has type:

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

303 It will be invoked by the code for `mapThrough` with a fold function `fo` with anchor type
304 `R`, and then has the responsibility of mapping the tail at type `R` (second input) to a result
305 upon which `mapThrough` should recurse (second component of the output pair). Then we
306 define an algebra `RleAlg` by supplying `compressSpan` as the function to map through, to
307 `MapThroughAlg` (Figure 7). Following the pattern we have seen in all the examples above,
308 we may then define function `mapThroughr` for subsidiary recursions, and `mapThrough` for
309 top-level recursions.

```

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 8** The function `rle` for run-length encoding, and the algebra `RleAlg` defining it in terms of `MapThroughAlg` (Figure 7)

310 4 Derivation of subsidiary recursion

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

313 4.1 Retractive-positive recursive types

314 As we have seen, our definitions require a form of positive-recursive types, to allow algebras
 315 to accept fold functions that themselves require algebras, and also for the definition of **Subrec**
 316 (which we will see in more detail in the next section). As already noted, full positive-recursive
 317 types are incompatible with Coq’s type theory [3]. One can impose some restrictions on large
 318 eliminations which then enable positive-recursive types [2]. This approach would require
 319 changing the underlying theory. To avoid this, we here take a different approach, exploiting
 320 Coq’s impredicative polymorphism.

321 This is done in a file `Mu.v`, whose central definitions are in Figure 9. The development is
 322 parametrized by `F : Set -> Set` which is assumed to have an `fmap` function (morphism
 323 part of the functor) of type

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

325 which satisfies the identity-preservation law for functors:

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

327 Let us consider the code in Figure 9. The critical idea is embodied in the definition of `Mu`.
 328 Ideally, we would like to have a definition like

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

330 This is exactly what is used in approaches to modular datatypes in functional programming,
 331 like Swierstra’s [13]. But this definition is (rightly) rejected by Coq, as instantiations of `F`
 332 that are not strictly positive would be unsound.

333 Instead, the definition of `Mu` in Figure 9 weakens this ideal definition to a strictly positive
 334 approximation:

```
335 Inductive Mu : Set :=
336   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 9** Derivation of retractive-positive recursive types

337 Instead of taking in $F\ Mu$, constructor `mu` accepts an input of type $F\ R$, for some type R for
338 which we have a function of type $R \rightarrow Mu$. The impredicative quantification of R is essential
339 here: we instantiate it with Mu itself in the definition of `inMu` (Figure 9). So this approach
340 would not work in a predicative theory like Agda’s. The quantification of R can be seen
341 as applying a technique due to Mendler, of introducing universally quantified variables for
342 problematic type occurrences, to a datatype constructor. We will review this in Section 7.

343 Returning to Figure 9, we have functions `inMu` and `outMu`, which make $F\ Mu$ a retraction
344 (`outIn`) of Mu : the composition of `outMu` and `inMu` is (extensionally) the identity on $F\ Mu$.
345 But the reverse composition cannot be proved to be the identity, because of the basic problem
346 of **noncanonicity** that arises with this definition.

347 For a simple example of noncanonicity, suppose we instantiate F with `ListF` (of Figure 2).
348 Please note that as Mu is used in our derivation of subsidiary recursion, we will not instantiate
349 this F with the signature functor of a datatype directly; but this will show the issue in
350 a simple form. Let us temporarily define `List A` as $Mu\ (ListF\ A)$ (again, for subsidiary
351 recursion we use a different functor than just `ListF` directly). The canonical way to define
352 the empty list would be, implicitly instantiating F to `ListF A`,

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

354 But given this, there are infinitely many other equivalent definitions. For any $Q : Set$, we
355 could take

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

357 Since `fmap f (NilF A)` equals just `NilF B` for $f : A \rightarrow B$, if we apply `outMu` (of Figure 9)
358 to `mkNil'` or `mkNil`, we will get `NilF (List A)`. But critically, `mkNil` and `mkNil'` are not
359 equal, neither definitionally nor provably. One can define a function that puts Mu values in
360 normal form by folding `inMu` over them. Then `mkNil` and `mkNil'` will have the same normal
361 form, and be equivalent in that sense. But the fact that they are not provably equal is what
362 we term noncanonicity.

363 Noncanonicity must be handled carefully when reasoning about functions defined with
364 our interface. We will see examples in Section 6. First, though, let us complete the exposition
365 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 10** Definition of `Subrec` as a fixed-point of `SubrecF`

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

The type `Subrec` is defined in Figure 10, as a fixed-point of `SubrecF : Set -> Set`. We take this fixed-point with `Mu`, discussed in the previous section, and obtain `roll` and `unroll` functions between `SubrecF Subrec` and `Subrec`. Unrolling `Subrec` gives us the type

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

So we see that `Subrec` is the type of functions which, for all algebras with anchor type `Subrec` and functorial carrier `X`, compute a value of type `X Subrec`. This is a generalization of the functor-generic type for the Church encoding:

$$\forall X. Alg X \rightarrow X$$

where $Alg X$ is $F X \rightarrow X$. We elide the implementation of the `roll` and `unroll` functions, but note that `unroll` makes use of functoriality of carriers `X`.

The rest of the interface for `Subrec` is shown in Figure 11. We have `fold`, which is a fold function with anchor type `Subrec`. To fold an algebra `alg` with carrier `X` (with `fmap` function given by `FunX`) over `d : Subrec`, `fold` unrolls the definition of `Subrec` and applies that to the algebra (with its carrier).

More interesting is the definition of `inn`, which is the critical point where the recursion universe is implemented. To create a value of type `Subrec` from data of type `F Subrec`, the definition of `inn` rolls a value of type `SubrecF Subrec` (we saw this type unfolded at the start of Section 4.2). 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`.
- the identity function is passed as the value for `reveal : R -> Subrec` (since `R` has been instantiated to `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` (again, since we are instantiating `R` with `Subrec`).

Finally, Figure 11 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.

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 11** The rest of the interface for Subrec

```
Fixpoint listFold(l : list A){X : Set}(alg : ListF X -> X) : X :=
  match l with
  | nil => alg Nil
  | cons hd tl => alg (Cons hd (listFold tl alg))
  end.

Definition toList (xs : list A) : List := listFold xs (inn ListF).

Definition listIn(d : ListF (list A)) : list A :=
  match d with
  | Nil => []
  | Cons hd tl => hd :: tl
  end.

Definition fromList : List -> list A :=
  fold ListF (Const (list A)) (FunConst (list A))
    (rollAlg (fun R reveal fo eval fr => listIn (fmap eval fr))) .
```

■ **Figure 12** Additional functions for the List type (Subrec ListF)

402 4.3 Converting between list and List (List.v)

403 Figure 12 gives the definitions (specified in Figure 2 above) for conversion functions between
404 `list` (the list type of Coq's standard library) and our `List` type for subsidiary recursion
405 on lists. To convert a `list` to a `List`, we use `listFold` to recursively apply the function
406 `inn` (Figure 11) throughout a list. This `listFold` provides a functorial view of the
407 usual fold-right function on lists. To convert a `List` to a `list`, we use our `fold` function
408 (Figure 11) to fold an algebra over the input `List`. This algebra does an `fmap` (for `ListF`)
409 with `eval : R -> list A`. The resulting value has type `ListF (list A)`. We may then
410 apply `listIn` to produce a `list A`. This `listIn` function converts from `ListF (list A)` to
411 `list A` in the obvious way, converting `Nil` to the empty list, and `Cons` to an application of
412 `::`. These functions were not necessary in implementing examples of subsidiary recursion
413 above, but they will be needed for reasoning about them. We turn to this topic next.

```

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 13** Interface for subsidiary induction

414 5 Interface for subsidiary induction (Subreci.v)

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

422 The central definitions for the type **Subreci** : **Subrec** -> **Prop** are given in Figure 13.
 423 Where having a value **x** of **Subrec** entitles us to define subsidiary recursions to inhabit types
 424 **X Subrec**, a value of type **Subreci x** lets us prove properties of **x** by subsidiary induction.
 425 Briefly: **kMo** is the kind for *motives*, namely predicates on **Subrec** [8]. **KAlgi** is the kind for
 426 indexed algebras. **FoldTi** is the indexed version of **FoldT**: it expresses provability of **X C**
 427 for **d**, based on an indexed algebra and a value of type **C d**, where **C** is the anchor type, now
 428 indexed. **AlgFi** and **Algi** are indexed versions of the algebras we saw for recursion. The
 429 **eval** function (Figure 3) has now become an induction hypothesis: given any **d** where **R d**
 430 holds, **ih** proves **X R d**. A value of type **R d** is thus a license to induct on **d**. Finally, the
 431 algebra is given a subdata structure indexed by **d** : **Subrec**, and must produce a proof of
 432 **X R d**. **Subreci** is defined as the suitably indexed fixed-point of **SubrecFi**, which is the
 433 natural indexed version of **SubrecF**.

434 For lists, we instantiate **Fi** with **ListFi**, shown in Figure 14. This is just the indexed
 435 version of **ListF**. Given a list **A**, **toListi** returns a value of type **Listi** (**toList xs**).
 436 This can be understood as saying that for any list (from Coq's standard library), we can

XX:16 Subsidiary Recursion in Coq

```
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 14** 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 15** Statements of three lemmas about span

437 reason by subsidiary induction to prove properties of `toList xs`. We also introduce an
438 abbreviation `ListFoldTi` for the type of indexed fold functions over lists.

439 6 Examples of subsidiary induction

440 For proving the main theorem about run-length encoding, we need several lemmas about
441 `span`, shown in Figure 15. For lack of space, we just state the properties. The first says that
442 appending the results of a call to `span` returns the original list (module some conversions to
443 `list` from `List`). The second uses the inductive type `Forall` from Coq's standard library
444 to state that all the elements of the prefix returned by `span` satisfy `p`. These lemmas are
445 proved using an indexed algebra where the indexed anchor type is not used (so the carriers
446 are constant indexed-functor returning the types shown). But for the third, `GuardPres`, we
447 are showing that whenever `spanh` (see Figure 4) returns a suffix `r`, that suffix satisfies the
448 indexed anchor type. This enables us to invoke an outer induction hypothesis on this suffix,
449 when using `span` subsidiarily.

450 Using these lemmas, we can write a short proof by subsidiary induction of:

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

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

7.1 Termination

In some tools, like Coq, Agda, and Lean, termination is checked statically, based on structural decrease at recursive calls. Others, like Isabelle/HOL, allow one to write recursions first, and prove (possibly with automated help) their termination afterwards [6].

It has not escaped the notice of designers of ITPs that structural recursion is not the only form of terminating recursion. All the mentioned tools provide support for well-founded recursion, where for recursive calls, one must show that the parameter of recursion has decreased in some well-founded order.

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

7.2 Mendler encoding

Mendler introduced the basic idea of using universal abstraction to support compositional termination checking; an accessible source is [9]. 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 work explored the categorical perspective on Mendler-style recursion [14]. Others have explored the possibility of using it with negative type schemes [1].

8 Conclusion

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>.

- 492 **3** Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and
493 Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn,*
494 *USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages
495 50–66. Springer, 1988. URL: https://doi.org/10.1007/3-540-52335-9_47, doi:10.1007/
496 3-540-52335-9_47.
- 497 **4** Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and program-
498 ming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction -*
499 *CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July*
500 *12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages
501 625–635. Springer, 2021. URL: https://doi.org/10.1007/978-3-030-79876-5_37, doi:
502 10.1007/978-3-030-79876-5_37.
- 503 **5** The Agda development team. *Agda*, 2021. Version 2.6.2.1. URL: [https://agda.readthedocs.](https://agda.readthedocs.io/en/v2.6.2.1/)
504 [io/en/v2.6.2.1/](https://agda.readthedocs.io/en/v2.6.2.1/).
- 505 **6** Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL*. URL: [https://isabelle.](https://isabelle.in.tum.de/doc/functions.pdf)
506 [in.tum.de/doc/functions.pdf](https://isabelle.in.tum.de/doc/functions.pdf).
- 507 **7** The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2021.
508 Version 8.13.2. URL: <http://coq.inria.fr>.
- 509 **8** Conor McBride. Elimination with a motive. In Paul Callaghan, Zhaohui Luo, James McKinna,
510 and Robert Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES*
511 *2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes*
512 *in Computer Science*, pages 197–216. Springer, 2000. URL: https://doi.org/10.1007/3-540-45842-5_13, doi:10.1007/3-540-45842-5_13.
- 513 **9** N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus.
514 *Annals of Pure and Applied Logic*, 51(1):159 – 172, 1991.
- 515 **10** Wolfgang Naraschewski and Tobias Nipkow. Isabelle/hol, 2020. URL: [http://www.cl.cam.](http://www.cl.cam.ac.uk/research/hvg/Isabelle/)
516 [ac.uk/research/hvg/Isabelle/](http://www.cl.cam.ac.uk/research/hvg/Isabelle/).
- 517 **11** David Salomon and Giovanni Motta. *Handbook of Data Compression*. Springer, 2009.
- 518 **12** Aaron Stump, Christopher Jenkins, Stephan Spahn, and Colin McDonald. Strong func-
519 tional pearl: Harper’s regular-expression matcher in cedille. *Proc. ACM Program. Lang.*,
520 4(ICFP):122:1–122:25, 2020. URL: <https://doi.org/10.1145/3409004>, doi:10.1145/
521 3409004.
- 522 **13** Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008. URL:
523 <https://doi.org/10.1017/S0956796808006758>, doi:10.1017/S0956796808006758.
- 524 **14** Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic J. of*
525 *Computing*, 6(3):343–361, September 1999.
- 526