



Subsidiary Recursion in Coq

Aaron Stump   

Computer Science Dept., The University of Iowa, USA

Alex Hubers 

Computer Science, The University of Iowa, USA

Christopher Jenkins  

Computer Science, The University of Iowa, USA

Benjamin Delaware  

Computer Science, Purdue University, USA

Abstract

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

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

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

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.

1 Introduction: subsidiary recursion

Central to interactive theorem provers like Coq, Agda, Isabelle/HOL, Lean and others are terminating recursive functions over user-declared inductive datatypes [8, 14, 17, 7]. Termination is usually enforced by a syntactic check for structural decrease, which is sufficient for many basic functions. For example, the `span` function from Haskell’s prelude (`Data.List`) takes a list and returns a pair of the maximal prefix whose elements satisfy a given predicate `p`, and the remaining suffix:

```
span :: (a -> Bool) -> [a] -> ([a],[a])
span _ []      = ([], [])
span p (x:xs) = if p x
                  then let (ys,zs) = span p xs in (x:ys,zs)
                  else ([],x:xs)
```

The sole recursive call is `span p xs`, and it occurs in a clause where the input list is of the form `x:xs`. Hence it is structurally decreasing. In the appropriate syntax, this definition can be accepted without additional effort by all the mentioned provers.

This paper is about a more expressive form of terminating recursion, called **subsidiary recursion**. While performing an outer recursion on some input `x`, one may initiate an inner recursion on `x` (or possibly some of its subdata), preserving the possibility of further invocations of the outer recursive function. Let us see a simple example. The function `wordsBy` (`Data.List.Extra`) breaks a list into its maximal sublists whose elements do not satisfy a predicate `p`. For example, `wordsBy isSpace " good day "` returns `["good","day"]`. Code is in Figure 1. Recall that `break p` is equivalent to `span (not . p)`. The first recursive call, `wordsBy p tl`, is structural. But in the second, we invoke `wordsBy p` on a value obtained



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

Interactive Theorem Proving 2022.

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

Leibniz International Proceedings in Informatics



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

XX:2 Subsidiary Recursion in Coq

```
wordsBy :: (a -> Bool) -> [a] -> [[a]]
wordsBy p [] = []
wordsBy p (hd:tl) =
  if p hd
  then wordsBy p tl
  else let (w,z) = break p tl in
       (hd:w) : wordsBy p z
```

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

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

41 1.1 Summary of results

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

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

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

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

69 2 Interface for subsidiary recursion

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

```

Definition List := Subrec ListF.
Definition inList : ListF List -> List := inn ListF.
Definition mkNil : List := inList Nil.
Definition mkCons (hd : A) (tl : List) : List := inList (Cons hd tl).
Definition toList : list A -> List.
Definition fromList : List -> list A.

```

■ **Figure 2** Some basics from `List.v`, specializing the functor-generic derivation of subsidiary recursion to lists (`List.v`)

2.1 The recursion universe

Our approach is within a long line of work using ideas from universal algebra and category theory to describe inductive datatypes and their recursion principles (cf. [22, 5, 11]). On this approach, one describes transformations to be performed on data as *algebras*, which can then be *folded* over data. The simplest form of algebras, namely *F*-algebras for functor *F*, are morphisms from *F A* to *A*, for carrier object *A*. From a programming perspective, an *F*-algebra is given input of type *F A*, and must compute a result of type *A*. An example of *F* is the signature functor for lists, which we will use below:

```

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

```

Algebras for our subsidiary recursion are more complex than *F*-algebras. Let us begin with an informal explanation. For reasons we will explain further below, the carrier of the algebra will be a functor $X : \mathbf{Set} \rightarrow \mathbf{Set}$. The algebra is presented with:

- a type $R : \mathbf{Set}$, which will be this recursion’s view of the datatype.
- a function `fold` : `FoldT Alg R`, which allows one to initiate subsidiary recursions over data of type *R*. We will present the type `FoldT Alg R` below.
- a function `rec` : $R \rightarrow X R$, to use for making recursive calls, on any value of type *R*.
- and a *subdata structure* $d : F R$, where *F* is the signature functor for the datatype.

The algebra is then required to produce a value of type $X R$.

We will use Coq inductive types for the signature functors *F* of various datatypes, thus enabling recursions to use Coq’s pattern-matching on the subdata structure *d*. So the style of coding against this interface retains a similar feel to structural recursion. Unlike with structural termination, though, the interface here is type-based and hence compositional.

We have previously dubbed this interface a *recursion universe* [20]. As in other domains using the term “universe”, we have a kind of space (here, *R*), which one cannot escape using certain operations. Other examples are the ordinal ϵ_0 and ω^- , and the physical universe and traveling at the speed of light. Staying in the recursion universe is good, because we may recurse (via `rec`) on any value of type *R*. Some points must still be explained: why X has type $\mathbf{Set} \rightarrow \mathbf{Set}$, and the definition of `FoldT`. Let us see these details next.

2.2 Types for subsidiary recursion (`Subrec.v`, `List.v`)

The type over which one can recurse with our scheme of subsidiary recursion is called `Subrec`. It is parametrized by a signature functor *F* of type $\mathbf{Set} \rightarrow \mathbf{Set}$. `Subrec` comes with

XX:4 Subsidiary Recursion in Coq

```

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

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

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

Definition Alg : KAlg := MuAlg AlgF.

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

```

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

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

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

113 2.3 Algebras for subsidiary recursion

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

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

124 Let us look at `AlgF`. As noted already, each recursion is based on an abstract type `R`,
 125 representing the data upon which we will recurse. This is the first argument to a value of
 126 type `AlgF Alg X`. Reasoning parametrically, an algebra can assume nothing about `R` except

```

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

```

■ **Figure 4** Computation law for subsidiary recursion, stated as a theorem

127 that it supports the following operations. Next we have a local `fold` function, which will
 128 allow us to fold another algebra over data of type `R`. We will use `fold` to initiate subsidiary
 129 recursions. Then there is `rec`, for recursive calls on data of type `R`.

130 As noted already, for subsidiary recursion, algebras have a carrier `X` which depends
 131 (functorially) on a type. When we fold an algebra using a fold function (either global or local)
 132 of type `FoldT Alg C`, (i) recursive calls may compute a result of type `X R`, mentioning the
 133 abstract type `R` for that recursion; and (ii) outside that recursion, the result will have type
 134 `X C`. Having a functor for the carrier of the algebra gives us the flexibility to type results
 135 inside a recursion with the abstract type `R`, but view those results as having the type `C`
 136 outside the recursion. The function `fold` in the figure initiates top-level folds. We also can
 137 have functions between `Alg` and its `Algf`-unfolding. We will return to the code for `Subrec.v`
 138 in Section 4.

139 Finally, for a recursion scheme, one would like to see not just the typed interface, but
 140 also the computation law. This is shown as a theorem in Figure 4. Intuitively, it states that
 141 folding an algebra over constructed data `inn d` is equal to invoking the algebra on `fold` for
 142 the fold function; an invocation of `fold` with the algebra for the `rec` function; and `d` for the
 143 subdata structure.

144 3 Examples of subsidiary recursion

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

146 3.1 The span function (`Span.v`)

147 This first example does not invoke subsidiary recursions, but will itself be used as a subsidiary
 148 recursion in other examples to follow. Given a predicate `p : A -> bool`, and a value of
 149 type `List A`, we would like to compute a pair of type `list A * List A`, where the first
 150 component is the maximal prefix whose elements satisfy `p`, and the second is the remaining
 151 suffix. This is the typing for a top-level recursion. More generally, though, given a type
 152 `R : Set` along with a fold function for that type (i.e., of type `FoldT (Alg (ListF A)) R`),
 153 we would like to map an input list of type `R` to a pair of type `list A * R`. The first component
 154 of this pair is going to be built up from scratch, and so cannot have type `R`; we cannot
 155 statically ensure that outer recursions on it are legal. But the second component will be a
 156 subdatum of the input list, and so can still have type `R`, enabling outer recursive calls. So we
 157 want:

```

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

```

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

XX:6 Subsidiary Recursion in Coq

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

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

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

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

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

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

171 3.1.1 The algebra for `span`

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

```

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

Definition spanr{R : Set}(fo:FoldT (Alg (ListF A)) R)
  (p : A -> bool)(xs : R) : list A * R
:= match spanhr fo p xs with
  SpanNoMatch => ([],xs)
  | SpanSomeMatch l r => (l,r)
end.

Definition breakr{R : Set}(fo:FoldT (Alg (ListF A)) R)
  (p : A -> bool)(xs : R) : list A * R :=
  spanr fo (fun x => negb (p x)) xs.

```

■ **Figure 6** Functions derived from SpanAlg (Span.v)

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

185 3.1.2 Defining span from SpanAlg

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

194 3.2 The wordsBy function (WordsBy.v)

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

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

XX:8 Subsidiary Recursion in Coq

```
Definition WordsBy(p : A -> bool)
  : Alg (ListF A) (Const (list (list A))) :=
  rollAlg (fun R fo wordsBy xs =>
    match xs with
    | Nil => []
    | Cons hd tl =>
      if p hd then
        wordsBy tl
      else
        let (w,z) := breakr fo p tl in
        (hd :: w) :: wordsBy z
    end).
Definition wordsByr{R : Set}(fo:FoldT (Alg (ListF A)) R)
  (p : A -> bool)(xs : R) : list (list A) :=
  fo (Const (list (list A))) (FunConst (list (list A))) (WordsBy p) xs.
```

■ **Figure 7** Functions `wordsBy` and `wordsByr`, and the algebra they fold (`WordsBy.v`)

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

■ **Figure 8** The `mapThrough` function in Haskell

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

212 Here we see the benefit of our approach. From Figure 6, the return type of `breakr`
213 is `list A * R`, where `R` comes from the type `FoldT (ListF A) Alg R` of `fo`, from the
214 definition of `AlgF` in Figure 3 (instantiating the functor with `ListF A`). This means that
215 from the invocation of `breakr`, we get `w : list A` and `z : R`. And so we can indeed apply
216 `wordsBy : R -> list (list A)` to `z` to recurse. The figure also shows the code for the
217 subsidiary recursion `wordsByr`.

218 3.3 The `mapThrough` function (`MapThrough.v`)

219 In this example, we see how to write a combinator that factors out a subsidiary recursion.
220 The Haskell library `Data.List.Extra` has a function `repeatedly`, defined essentially as in
221 Figure 8, though we attempt a more informative name. This is like the standard `map` function
222 on lists, except that the function `f` that we are mapping (or “mapping through”) takes in
223 not just the current element `a`, but also the tail `as`. It then returns the value `b` to include in
224 the output list, and whatever other list it wishes, upon which `mapThrough` will recurse.

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

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



```

Definition MapThroughAlg{B : Set}(f:mappedT A B)
  : Alg (ListF A) (Const (list B)) :=
  rollAlg (fun R fo mapThrough xs =>
    match xs with
    | Nil => []
    | Cons hd tl =>
      let (b,c) := f R fo hd tl in
      b :: mapThrough c
    end).
Definition mapThroughr{R : Set}(fo:FoldT (Alg (ListF A)) R)
  {B : Set}(f:mappedT A B) : R -> list B.
Definition mapThrough{B : Set}(f:mappedT A B) : List A -> list B.

```

■ **Figure 9** The algebra `MapThroughAlg` defining function `mapThrough` and `mapThroughr`; the code for those follows the pattern of `wordsBy` and `wordsByr` (Figure 7), so we omit it (`MapThrough.v`)

```

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

```

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

229 This type is more informative than the Haskell type, since it shows that the second component
 230 of the returned value must have type `R`, and hence must be (hereditarily) a tail of the input.
 231 Given this definition, the code is in Figure 9. `MapThroughAlg` is similar to the Haskell
 232 code above, though when we call `f`, we must supply the abstract type `R` and fold function
 233 `fo`. Then, from the definition of `mappedT`, we have that `b : B` and `c : R`. So we may indeed
 234 invoke `mapThrough : R -> list B` on `c`. Note that as we are building up a new list from
 235 scratch (rather than just extracting some tail of the input list), we just return `list B`; we
 236 cannot perform further subsidiary recursion on the output.

237 3.4 Run-length encoding (`Rle.v`)

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

246 Assuming `A : Set` and an equality test `eqb : A -> A -> bool` on it, we port this code
 247 to our Coq infrastructure in Figure 11. The function `compressSpan` is written at the type
 248 `mappedT A (nat * A)` that will be required by `mapThrough`. Unfolding the definition of
 249 `mappedT`, `compressSpan` has type:

```

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

```

XX:10 Subsidiary Recursion in Coq

```
Definition compressSpan : mappedT A (nat * A) :=
  fun R fo hd tl =>
    let (p,s) := spanr fo (eqb hd) tl in
    ((succ (length p),hd), s).

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

■ **Figure 11** The function `rle` for run-length encoding, and the algebra `RleAlg` defining it in terms of `MapThroughAlg` of Figure 9 (`Rle.v`)

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

256 4 Derivation of subsidiary recursion

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

259 4.1 Retractive-positive recursive types (`Mu.v`)

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

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

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

267 which satisfies the identity-preservation law for functors:

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

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

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

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

```

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

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

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

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

```

■ **Figure 12** Derivation of retractive-positive recursive types (Mu.v)

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

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

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

```

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

```

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

```

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

```

291 Since $\text{fmap } f (\text{NilF A})$ equals NilF B for $f : A \rightarrow B$, if we apply outMu (of Figure 12)
 292 to mkNil' or mkNil , we will get $\text{NilF} (\text{List A})$. But critically, mkNil and mkNil' are not
 293 equal, neither definitionally nor provably. One can define a function that puts Mu values in
 294 normal form by folding inMu over them. Then mkNil and mkNil' will have the same normal
 295 form, and be equivalent in that sense. But the fact that they are not provably equal is what
 296 we term noncanonicity.

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

XX:12 Subsidiary Recursion in Coq

```
Definition SubrecF(C : Set) :=  
  forall (X : Set -> Set) (FunX : Functor X), Alg X -> X C.  
Definition Subrec := Mu SubrecF.  
Definition roll: SubrecF Subrec -> Subrec.  
Definition unroll: Subrec -> SubrecF Subrec.
```

■ **Figure 13** Definition of `Subrec` as a fixed-point of `SubrecF` (`Subrec.v`)

300 4.2 The implementation of `Subrec` (`Subrec.v`)

301 The type `Subrec` is defined in Figure 13, as a fixed-point of `SubrecF : Set -> Set`. We
302 take this fixed-point with `Mu`, discussed in the previous section, and obtain `roll` and `unroll`
303 functions between `SubrecF Subrec` and `Subrec`. Unrolling `Subrec` gives us the type

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

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

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

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

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

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

328 5 Interface for subsidiary induction (`Subreci.v`)

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

```

Definition fold : FoldT Alg Subrec :=
  fun X FunX alg d => unroll d X FunX alg.

Definition inn : F Subrec -> Subrec :=
  fun d => roll (fun X xmap alg =>
    unrollAlg alg Subrec fold (fold X xmap alg) d).

Definition out{R:Set}(fo:FoldT Alg R) : R -> F R :=
  fo F FunF (rollAlg (fun R' _ _ d => d)).

```

■ **Figure 14** The rest of the interface for Subrec (Subrec.v)

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

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

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

354 6 Examples of subsidiary induction

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

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

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

XX:14 Subsidiary Recursion in Coq

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

Definition AlgFi(A : KAlgi)(X : kMo -> kMo) : Set :=
  forall (R : kMo)
    (fo : (forall (d : Subrec), FoldTi A R d))
    (ih : (forall (d : Subrec), R d -> X R d))
    (d : Subrec),
    Fi R d -> X R d.

Definition Algi := MuAlgi Subrec AlgFi.

Definition SubrecFi(C : kMo) : kMo :=
  fun d => forall (X : kMo -> kMo) (xmap : fmapiT Subrec X), Algi X -> X C d.
Definition Subreci := Mui Subrec SubrecFi.

Definition foldi(i : Subrec) : FoldTi Algi Subreci i.
Definition inni(i : Subrec)(fd : Fi Subreci i) : Subreci i.
```

■ **Figure 15** Interface for subsidiary induction (Subreci.v)

```
Definition lkMo := List -> Prop.

Inductive ListFi(R : lkMo) : lkMo :=
  nilFi : ListFi R mkNil
| consFi : forall (h : A)(t : List), R t -> ListFi R (mkCons h t).

Definition Listi := Subreci ListF ListFi.
Definition toListi(xs : list A) : Listi (toList xs) := listFoldi xs Listi inni.
Definition ListFoldTi(R : List -> Prop)(d : List) : Prop :=
  FoldTi ListF (Algi ListF ListFi) R d.
```

■ **Figure 16** The indexed version ListFi of ListF (List.v)

```

Definition SpanAppendF(p : A -> bool)(xs : List A) : Prop :=
  forall (l : list A)(r : List A) ,
    span p xs = (l,r) ->
    fromList xs = l ++ (fromList r).

```

```

Definition spanForallF(p : A -> bool)(xs : List A) : Prop :=
  forall (l : list A)(r : List A),
    span p xs = (l,r) ->
    Forall (fun a => p a = true) l.

```

```

Definition GuardPresF(p : A -> bool)(S : List A -> Prop)(xs : List A) : Prop :=
  forall (l : list A)(r : List A),
    spanh p xs = SpanSomeMatch l r ->
    S r.

```

■ **Figure 17** Statements of three lemmas about `span` (directory `SpanPfs`)

```

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

```

■ **Figure 18** A statement of the property that `span` returns the empty suffix, computed using `getNil` to avoid noncanonicity problems, if all elements satisfy `p`

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

```

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

```

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

378 7 Related Work

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

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

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

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

411 8 Conclusion

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

References

- 1 Ki Yung Ahn and Tim Sheard. A hierarchy of mendler style recursion combinators: Taming inductive datatypes with negative occurrences. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP '11*, pages 234–246, New York, NY, USA, 2011. ACM.
- 2 Gilles Barthe, Maria João Frade, Eduardo Giménez, Luís Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Math. Struct. Comput. Sci.*, 14(1):97–141, 2004. URL: <https://doi.org/10.1017/S0960129503004122>, doi:10.1017/S0960129503004122.
- 3 Frédéric Blanqui. Inductive types in the calculus of algebraic constructions. *Fundam. Informaticae*, 65(1-2):61–86, 2005. URL: <http://content.iospress.com/articles/fundamenta-informaticae/fi65-1-2-04>.
- 4 Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive theorem provers - an overview. *Mathematical Structures in Computer Science*, 26(1):38–88, 2016. URL: <https://doi.org/10.1017/S0960129514000115>, doi:10.1017/S0960129514000115.
- 5 Robin Cockett and Dwight Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. AMS, 1992.
- 6 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988. URL: https://doi.org/10.1007/3-540-52335-9_47, doi:10.1007/3-540-52335-9_47.
- 7 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. URL: https://doi.org/10.1007/978-3-030-79876-5_37, doi:10.1007/978-3-030-79876-5_37.
- 8 The Agda development team. *Agda*, 2021. Version 2.6.2.1. URL: <https://agda.readthedocs.io/en/v2.6.2.1/>.
- 9 Denis Firsov, Richard Blair, and Aaron Stump. Efficient mendler-style lambda-encodings in cedille. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 235–252. Springer, 2018.
- 10 Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in cedille. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 215–227. ACM, 2018.
- 11 Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.
- 12 Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL*. URL: <https://isabelle.in.tum.de/doc/functions.pdf>.
- 13 Alexander Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Autom. Reasoning*, 44(4):303–336, 2010. URL: <https://doi.org/10.1007/s10817-009-9157-2>, doi:10.1007/s10817-009-9157-2.
- 14 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2021. Version 8.13.2. URL: <http://coq.inria.fr>.
- 15 Conor McBride. Elimination with a motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes*

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