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, 9, 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,

Interactive Theorem Proving 2022.
Editors: June Andronick and Leonardo da Moura; Article No. ; pp. :1–:14
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
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

using also `break` (`Data.List`). Recall that `break p` is equivalent to `span (not . p)`: it splits its input into the maximal prefix whose elements do not satisfy `p`, and the following suffix. The first recursive call, `wordsBy p tl`, is structural. But in the second, we invoke `wordsBy p` on a value obtained from another recursion, namely `span`. This is not allowed under structural termination, but will be permitted by subsidiary recursion as derived below.

1.1 Summary of results

This paper presents a functor-generic derivation of terminating subsidiary recursion and induction in Coq. We should emphasize that this is a derivation of this recursion scheme within the type theory of Coq. No axioms or other modifications to Coq of any kind are required. Based on this derivation, we present several example functions like `wordsBy`, and prove theorems about them. For example, we prove the expected property that the sublists returned by `wordsBy` consist of elements satisfying `not . p`. For another, we give a definition of run-length encoding as a subsidiary recursion using `span`, and prove that encoding and then decoding returns the original list. Our approach applies to the standard datatypes in the Coq library, and does not require switching libraries or datatype definitions.

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

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

All presented derivations have been checked with Coq version 8.13.2, using command-line option `-impredicative-set`. The code may be 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 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 : Set -> 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 : Set`, which will be this recursion's view of the datatype.
- a function `reveal : R -> 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 -> 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* [11]. 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 `Set -> 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.

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

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`)

2.2.1 Subrec.v

This file is parametrized by a signature functor `F` of type `Set -> Set`. It provides the implementation of subsidiary recursion. Two crucial values are `Subrec : Set`, which is the type to use for subsidiary recursion; and `inn : F Subrec -> Subrec`, which is to be used as a constructor for that type. An important point, however, is that `Subrec.v` does not provide an induction principle based on `inn`. Induction is derived later (Section 5). `Subrec.v` makes critical use of retractive-positive recursive types, to take a fixed-point of a construction based on `F`. We present these recursive types in Section 4.1 below.

2.2.2 List.v

This file specializes the development in `Subrec.v` to the case of lists (parametrized by the type `A` of elements). In general, to use our development to get subsidiary recursion over some datatype, one will have a similar "shim" file. The file defines the signature functor `ListF`, shown in Figure 2. Using `Subrec`, we then get a type `List`. This is not to be confused with the type `list` of lists in Coq's standard library. As noted previously, our development is meant to be used in extension of existing inductive datatypes, not replacing them. The figure also shows constructors `mkNil` and `mkCons` for `List`, and types for conversion functions between `List` and `list` (see Section 4 for the code).

2.3 Algebras for subsidiary recursion

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

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

The variable `Alg` occurs only positively (but not strictly positively) in `AlgF`, because it occurs negatively in `FoldT Alg R` which occurs negatively in `AlgF Alg C X`. So we can

```
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`)

indeed take a fixed-point of `AlgF` to define the constant `Alg`.

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

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

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

3 Examples of subsidiary recursion

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

3.1 The span function (Span.v)

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

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

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

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

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

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

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

Figure 4 gives the algebra `SpanAlg` for computing `span`, and the code for `spanr`. We elide the proof `SpanFunctor` that `SpanF` is indeed a `Functor`, and make `X` implicit in the constructors of `SpanF`. The type of `SpanAlg p C` is

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

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

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

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

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

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

```
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)

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

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

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

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

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

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

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

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

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

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

We can write this combinatory using our infrastructure for subsidiary recursion. For this to work, we need to supply the mapped function with the fold function for `mapThrough`'s recursion. This is so that the mapped function can initiate a subsidiary recursion, returning a value in the abstract 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.
```

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

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

```
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

3.4 Run-length encoding

Using `mapThrough`, we can write a quite concise function for *run-length encoding*, a basic data-compression algorithm where maximal sequences of n occurrences of element e are summarized by the pair (n, e) [10]. In Haskell, invoking `span` and `mapThrough` (defined above), the code is simply

```
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)
```

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

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

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

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

4 Derivation of subsidiary recursion

4.1 Retractive-positive recursive types

As we have seen, our definitions require a form of positive-recursive types, to allow algebras to accept fold functions that themselves require algebras, and also for the definition of `Subrec`. But as recalled already, full positive-recursive types are incompatible with Coq's type theory [3]. It is worth noting that one can impose some restrictions on large eliminations which then allow positive-recursive types [2]. This approach would require changing the underlying theory. To avoid this, we here take a different approach, exploiting Coq's impredicative polymorphism.

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

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

which satisfies the identity-preservation law for functors:

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

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

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

This is exactly what is used in approaches to modular datatypes in functional programming, like Swierstra's [12]. But this definition is (rightly) rejected by Coq, as instantiations of `F` that are not strictly positive would be unsound.

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

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

Instead of taking in `F Mu`, constructor `mu` accepts an input of type `F R`, for some type `R` for which we have a function of type `R -> Mu`. The impredicative quantification of `R` is essential

```
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

here: we instantiate it with `Mu` itself in the definition of `inMu` (Figure 9). So this approach would not work in a predicative theory like Agda's. The quantification of `R` can be seen as applying a technique due to Mendler, of introducing universally quantified variables for problematic type occurrences, to a datatype constructor. We will review this in Section 7.

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

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

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

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

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

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

Noncanonicity leads to some issues, as we turn next to the problem of inductive reasoning about subsidiary recursions. With some care, however, we can avoid pitfalls, leaving us with a form of positive-recursive type that enables our definitions to go through.

5 Interface for subsidiary induction

6 Examples of subsidiary induction

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 [8]. This recursor has type

$$\forall X. (\forall R. (R \to X) \to F\ R \to X) \to \mu\ F\ \to X$$

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 [13]. Others have explored the possibility of using it with negative type schemes [1].

References

1. Ki Yung Ahn and Tim Sheard. A hierarchy of mendler style recursion combinators: Taming inductive datatypes with negative occurrences. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 234–246, New York, NY, USA, 2011. ACM.
2. Frédéric Blanqui. Inductive types in the calculus of algebraic constructions. *Fundam. Informaticae*, 65(1-2):61–86, 2005. URL: `http://content.iospress.com/articles/fundamenta-informaticae/fi65-1-2-04`.
3. 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`.
4. 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`.
5. The Agda development team. *Agda*, 2021. Version 2.6.2.1. URL: `https://agda.readthedocs.io/en/v2.6.2.1/`.

6 Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL*. URL: https://isabelle.in.tum.de/doc/functions.pdf.

7 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2021. Version 8.13.2. URL: http://coq.inria.fr.

8 N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1):159 – 172, 1991.

9 Wolfgang Naraschewski and Tobias Nipkow. Isabelle/hol, 2020. URL: http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

10 David Salomon and Giovanni Motta. *Handbook of Data Compression*. Springer, 2009.

11 Aaron Stump, Christopher Jenkins, Stephan Spahn, and Colin McDonald. Strong functional pearl: Harper's regular-expression matcher in cedille. *Proc. ACM Program. Lang.*, 4(ICFP):122:1–122:25, 2020. URL: https://doi.org/10.1145/3409004, doi:10.1145/3409004.

12 Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008. URL: https://doi.org/10.1017/S0956796808006758, doi:10.1017/S0956796808006758.

13 Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic J. of Computing*, 6(3):343–361, September 1999.