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

```
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

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

1.1 Summary of results

This paper presents a functor-generic derivation of terminating subsidiary recursion and induction in Coq. We emphasize that this is a derivation within the type theory of Coq, and requires no axioms or other modifications to Coq, except the `-impredicative-set` flag. Using this derivation, we present several example functions like `wordsBy`, and prove theorems about them. A nice example is a definition of run-length encoding using `span` as a subsidiary recursion, where we 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 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 input types of constructors 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. 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 $F\ (\mu F)$ is a retract of μF. Usually these types are isomorphic. Hence, we dub these **retractive-positive** recursive types. This weakening leads to noncanonical elements of μ, but we will see how to work around this. Our definition of retractive-positive recursive types makes essential use of impredicative quantification, and hence is not legal in predicative theories 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. 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 (cf. [22, 5, 11]). On this approach, one describes transformations to be performed on data as *algebras*, which can then be *folded* over data. The simplest form of algebras, namely F-algebras for functor F, are morphisms from $F\ A$ to A, for carrier object A. From a programming perspective, an F-algebra is given input of type $F\ A$, and must compute a result of type A. An example of F is the signature functor for lists, which we will use below:

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

Algebras for our subsidiary recursion are more complex than F-algebras. First, for reasons we will explain further below, the carrier of the algebra will be a functor `X : 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 A`, for some `A`). 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 recursion. Unlike with structural termination, though, the interface here is type-based and hence compositional.

As in previous work, we dub this interface a *recursion universe* [20]. As in other domains using the term "universe", we have a kind of space (here, `R`) with operations that keep one in that space (for other cases: the ordinal 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`. One can use `reveal` to leave, but then `eval` can no longer be used. Some points must still be explained: why `X` has type `Set -> Set`, and the definition of `FoldT`. Let us see these details next.

2.2 The interface in more detail

Let us consider two central files from our development.

2.2.1 Subrec.v

This file is parametrized by a signature functor `F` of type `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

```
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`)

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. For space reasons, we just give the example of lists. The file defines the signature functor `ListF`, already shown above. We have also the definitions of Figure 2. `List` is defined to be `Subrec`, with `F` instantiated to `ListF A`. This type `List` 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` (code elided). One direction uses Coq's structural recursion, the other uses subsidiary recursion, which we will see next.

2.3 Algebras for subsidiary recursion

`Subrec.v` also implements the notion of algebra we introduced informally above. 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`, which implements our retractive-positive recursive types (Section 4.1) at kind `KAlg`. Using `Alg` will require that `AlgF` only uses its parameter `Alg` positively. We will confirm this shortly.

`FoldT Alg C` is the type for fold functions which apply algebras (`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 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`. Reasoning parametrically, 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

```
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`)

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 last 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.

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

3 Examples of subsidiary recursion

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

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

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`; we cannot statically ensure that outer recursions on it are legal. But the second component will be a subdatum of the input list, and so can still have type `R`, enabling outer recursive calls. 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 `ListF A 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` (`X` will be implicit for the constructors):

```
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 suffix is available to the algebra in the subdata structure of type `ListF A R`.

3.1.1 The algebra for span

Figure 4 gives the algebra `SpanAlg` for computing `span`. 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.

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

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

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 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.

3.1.2 Defining `span` from `SpanAlg`

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

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

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

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

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

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

Figure 5 Functions derived from `SpanAlg` (`Span.v`)

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

The code for `WordsBy` is 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. The figure also shows the code for the subsidiary recursion `wordsByr`.

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

The Haskell library `Data.List.Extra` has a function `repeatedly`, defined essentially as in Figure 7, though we attempt a more informative name. This is like the standard `map` function on lists, except that 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.

To write this combinator using our infrastructure for subsidiary recursion, 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 WordsBy(p : A -> bool)(C : Set)
    : Alg (ListF A) C (Const (list (list A))) :=
    rollAlg (fun R reveal fo wordsBy xs =>
      match xs with
        Nil => []
      | Cons hd tl =>
        if p hd then
          wordsBy tl
        else
          let (w,z) := breakr fo p tl in
            (hd :: w) :: wordsBy z
      end).

Definition wordsByr{R : Set}(fo:FoldT (Alg (ListF A)) R)
                   (p : A -> bool)(xs : R) : list (list A) :=
  fo (Const (list (list A))) (FunConst (list (list A))) (WordsBy p R) xs.
```

Figure 6 Functions functions `wordsBy` and `wordsByr`, and the algebra they fold (`WordsBy.v`)

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

Figure 7 The `mapThrough` function in Haskell

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

Definition mapThroughr{R : Set}(fo:FoldT (Alg (ListF A)) R)
                      {B : Set}(f:mappedT A B) : R -> list B.
Definition mapThrough{B : Set}(f:mappedT A B) : List A -> list B.
```

Figure 8 The algebra `MapThroughAlg` defining function `mapThrough` and `mapThroughr`; the code for those follows the pattern of `wordsBy` and `wordsByr` (Figure 6), so we omit it (`MapThrough.v`)

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

Figure 9 Run-length encoding in Haskell, using `mapThrough` and `span`

```
Definition 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.

Given this definition, the code is in Figure 8. `MapThroughAlg` is similar to the Haskell code above, 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.

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

Using `mapThrough`, we can quite concisely implement *run-length encoding*, a basic data-compression algorithm where maximal sequences of n occurrences of element e are summarized by the pair (n, e) [19]. Haskell code is in Figure 9. Recall that `(== a)` tests 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`. The `mapThrough` combinator then iterates `compressSpan` through the suffix `s`.

Assuming `A : Set` and an equality test `eqb : A -> A -> bool` on it, we port this code to our Coq infrastructure in Figure 10. The function `compressSpan` is written at the type

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

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

Figure 10 The function `rle` for run-length encoding, and the algebra `RleAlg` defining it in terms of `MapThroughAlg` of Figure 8 (`Rle.v`)

`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 extracting from the tail at type `R` (second input) 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 8). Following the pattern seen above, we define function `rle` for top-level recursions using `fold` (we could also define a subsidiary version `rler`).

4 Derivation of subsidiary recursion

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

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

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` (which we will see in more detail in the next section). Full positive-recursive types are incompatible with Coq's type theory [6]. One can impose some restrictions on large eliminations which then enable positive-recursive types [3], but this requires changing the underlying theory. Here we exploit Coq's impredicative polymorphism for a different solution.

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
```

which satisfies the identity-preservation law for functors:

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

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

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

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

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

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

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

Figure 11 Derivation of retractive-positive recursive types (Mu.v)

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

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

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

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

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

For a simple example: suppose we instantiate `F` with `ListF` (of Figure 2). Our derivation uses a different type that wraps `F`, but this will show the issue in a simple form. Let us temporarily define `List A` as `Mu (ListF A)` (again, for subsidiary recursion do not use 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 11) 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

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

Figure 12 Definition of `Subrec` as a fixed-point of `SubrecF` (`Subrec.v`)

form, and be equivalent in that sense. But the fact that they are not provably equal is what we term noncanonicity.

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

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

The type `Subrec` is defined in Figure 12, 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 $\forall\ X.\ \mathit{Alg}\ X \to X$ for the Church encoding, where $\mathit{Alg}\ X$ is $F\ X \to X$. 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 13. 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`, we `unroll` the definition of `Subrec` and apply 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 this section). This value takes in a carrier `X`, its fmap function `xmap`, and an algebra `alg` with that carrier. Note that the anchor type of this algebra is `Subrec`. It will then call `alg` (after `unrolling` it) with implementations for the components of the recursion universe (cf. Section 2.1, also Figure 3):

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

Finally, Figure 13 defines `out` as a subsidiary recursion, given any fold function with its anchor type `R`. Outside the recursion, `d` has type `F R`; inside the recursion it has type `F R'`

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

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

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

Figure 13 The rest of the interface for `Subrec` (`Subrec.v`)

where `R'` is the abstract type of the subsidiary recursion. So `out` implements the idea that unfolding an abstract type one step is just a trivial case of subsidiary recursion.

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

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

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

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

6 Examples of subsidiary induction

For proving the main theorem about run-length encoding, we need several lemmas about `span`, shown in Figure 16. For lack of space, we just state the properties. The first says that appending the results of a call to span returns the original list (module some conversions to `list` from `List`). The second uses the inductive type `Forall` from Coq's standard library

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

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

Definition Algi := MuAlgi Subrec AlgFi.

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

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

Figure 14 Interface for subsidiary induction (`Subreci.v`)

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

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

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

Figure 15 The indexed version `ListFi` of `ListF` (`List.v`)

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

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

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

Figure 16 Statements of three lemmas about `span` (directory `SpanPfs`)

to state that all the elements of the prefix returned by `span` satisfy `p`. These lemmas are proved using an indexed algebra where the indexed anchor type is not used (so the carriers are constant indexed functors returning the types shown). But `GuardPresF` uses the indexed anchor type (its argument `S`), to express that whenever `spanh` returns a suffix `r`, that suffix satisfies the indexed anchor type. This enables us to invoke an outer induction hypothesis on this suffix, when reasoning subsidiarily about `span`. Using these lemmas, we can write a short proof by subsidiary induction of the following, where `rld : list (nat * A) -> list A` is the obvious decoding function:

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

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

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 `List`s. 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`. The statement that one can prove is shown in Figure 17.

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

Figure 17 A statement of the property that `span` returns the empty suffix, computed using `getNil` to avoid noncanonicity problems, if all elements satisfy `p`

7 Related Work

Termination. In some tools, like Coq, Agda, and Lean, termination is checked statically, based on structural decrease. Others, like Isabelle/HOL, allow one to write recursions first, and prove (possibly with automated help) their termination afterwards [12]. These tools all support well-founded recursion, but in constructive type theory, evidence of well-foundedness then propagates through code. In contrast, our approach here, while less general, does not clutter code with proofs. Subsidiary recursion can be seen as a generalization of *nested recursion*, which allows recursive calls of the form `f (f x)` [13]. In subsidiary recursion, these are generalized to the form `f (g x)`, where `g` could be `f` or another recursively defined function. For more on partiality and recursion in theorem provers, see [4].

Our work contributes to the program proposed by Owens and Slind, of broadening the scope of functional programs that can be accommodated in ITPs [18]. The goal of terminating recursion has been advocated in the literature on programming languages under the name *strong functional programming* [23]. Uustalu and Vene developed a categorical view of a recursion scheme allowing one level of subsidiary recursion, and illustrated it in Haskell with an artificial example [25]. In contrast, our scheme allows arbitrary finite nestings of recursion, and we illustrate it in Coq with realistic examples. Like them, we find that subsidiary recursion subsumes course-of-value recursion. The technique of sized types is similar to our method in providing a type-based method for termination [2]. This technique enriches datatypes with abstract sizes, which must then be propagated through code, using dependent types. Our `Subrec` does not require dependent types, resulting in just polymorphically typed code for the examples we saw (`Subreci`, for proofs, of course does use dependent types).

Mendler-style recursion. Mendler introduced the idea of using universal abstraction to support compositional termination checking [16]. He proposed a functor-generic recursor of type $\forall X. (\forall R. (R \to X) \to F\ R \to X) \to \mu\ F\ \ \to X$. 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, and showed how to reduce it to basic catamorphisms (i.e., structural recursion) [24]. Another considered its use with negative type schemes [1]. Previous work from our group showed how to derive inductive datatypes in Cedille using extensions of the Mendler encoding [9, 10]. Here, we do not derive inductive types using these methods, but rather apply them to justify a terminating recursion scheme for existing datatypes.

8 Conclusion

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

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*

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`.

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

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

18 Scott Owens and Konrad Slind. Adapting functional programs to higher order logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008. URL: `https://doi.org/10.1007/s10990-008-9038-0`, `doi:10.1007/s10990-008-9038-0`.

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

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

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

22 Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 596–605. IEEE Computer Society, 2012. URL: `https://doi.org/10.1109/LICS.2012.75`, `doi:10.1109/LICS.2012.75`.

23 D. A. Turner. Elementary Strong Functional Programming. In *Proceedings of the First International Symposium on Functional Programming Languages in Education*, FPLE '95, page 1–13, Berlin, Heidelberg, 1995. Springer-Verlag.

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

25 Tarmo Uustalu and Varmo Vene. The Recursion Scheme from the Cofree Recursive Comonad. *Electron. Notes Theor. Comput. Sci.*, 229(5):135–157, 2011. URL: `https://doi.org/10.1016/j.entcs.2011.02.020`, `doi:10.1016/j.entcs.2011.02.020`.