A Type-Based Approach to Divide-And-Conquer Recursion in Coq

ANONYMOUS AUTHOR(S)

This paper proposes a new approach to writing and verifying divide-and-conquer programs in Coq. Using ideas from advanced lambda-encodings, combinators are derived in Coq for divide-and-conquer recursion: from outer recursions, one may initiate inner recursions that can construct data upon which the outer recursions may legally recurse. Termination is enforced by the type system of Coq, using just the types of the Calculus of Constructions. In particular, the method does not rely on Coq's native structural recursion. The method is demonstrated on several examples, including mergesort, quicksort, Harper's regular-expression matcher, and others. An indexed version is also derived, implementing divide-and-conquer induction.

Additional Key Words and Phrases: Divide-and-conquer recursion, strong functional programming, well-founded recursion

1 RECURSION IN COQ

In interactive theorem provers such as Coq, Agda, and Lean, users may prove properties of strongly typed pure functional programs written in an ML-like language [de Moura et al. 2015; The Agda development team 2016; The Coq development team 2016]. In addition to typing requirements, these functions must pass a static termination check. This is due to the Curry-Howard isomorphism, where proofs are identified with programs. Without this check, users could prove arbitrary formulas using infinite loops.

Coq's termination checker (as well as Agda's and Lean's) is based on structural decrease of arguments at recursive calls. Structural termination covers many familiar examples from functional programming, including `map`, `filter`, `foldr`, and more. But many terminating programs use nonstructural recursions, notably classic divide-and-conquer algorithms. For example, `mergesort` splits the input list roughly in half, recurses on the halves, and then merges the results. The recursions on the half lists are not structural, because structural recursion prohibits recursive calls on the *results* of other computations, like splitting.

Several techniques for nonstructural recursion have been proposed previously (Section 2). The most prominent is well-founded recursion, where programs use explicit proof terms to justify recursive calls based on decrease in a well-founded ordering. Dependent typing is used to connect the evidence to the parameter of recursion. Cleverly, the evidence itself decreases structurally at recursive call sites, so programs written in this style satisfy the structural termination check.

This paper proposes a different solution to the problem of nonstructural recursions in type theory. The approach is based not on structural recursion, but rather on type-checking in Coq's core pure type system, the Calculus of Constructions (CC) [Coquand and Huet 1988]. We derive combinators in CC that are flexible enough to support divide-and-conquer programming. Typability implies that all programs written with them indeed terminate. In fact, we do not even need dependent types: System F_ω is sufficient. Our approach is thus applicable for strong functional programming, where the goal is to guarantee termination without dependent types [Turner 1995]. (We do make use of dependent types for the corresponding nonstructural induction principles.)

Our Coq development implements an interface for what we call *divide-and-conquer recursion*. From outer recursions, one may initiate *subsidiary* inner recursions that can construct data upon which the outer recursions may legally recurse. This is sufficient for examples like mergesort: the

2022. 2475-1421/2022/1-ART1 $15.00
https://doi.org/

splitting phase of the algorithm is implemented as a subsidiary recursion, which constructs lists upon which the outer recursion can then recurse. To construct such data, subsidiary recursions are provided with versions of the datatype's constructors with abstract types, which limit application of the constructors to ensure termination.

In addition to having a different foundation, our method results in a very different style of programming for nonstructural recursions. Well-founded recursion relies on including explicit proofs that arguments decrease at recursive call sites. So the method requires dependent typing, and proofs of decrease are interspersed in code. In contrast, our divide-and-conquer recursion does not use dependent types at all. Termination proofs are eliminated in favor of programming problems, namely coding up particular recursions against the interface we provide.

Finally, while the paper's focus is on the divide-and-conquer recursion scheme, we have also derived a dependently typed version, enabling divide-and-conquer induction. Along with example programs, we will consider example proofs of their behavior, defined in the same style. Our development does not require any extension to Coq's type theory, except postulating functional extensionality (a commonly added axiom), and also impredicative `Set` (enabled by a command-line option to Coq). The paper's specific contributions are:

(1) Formulation of an interface for divide-and-conquer recursion in Coq (Section 4). The formulation is generic for any signature functor, and so applies once and for all to a large, standardly used family of datatypes, including natural numbers, lists, binary and other forms of trees, and many other common examples. It does not make use of dependent types, and users of the interface do not prove statements showing that arguments decrease. Instead, the typing of CC is used to enforce termination.

(2) Realistic examples in Coq coded against this interface (Section 5), including quicksort, mergesort, run-length encoding, and a function `wordsBy`, which breaks a list into its maximal sublists whose elements do not satisfy a predicate `p`. Another example is Harper's regular expression matcher, which has been posed as a challenge problem for termination [Bove et al. 2016; Harper 1999].

(3) Derivation within Coq of an implementation of the interface (Section 6).

(4) Formulation and implementation of a dependently typed version of the interface, yielding a divide-and-conquer induction principle (Section 7). Proofs using this principle are demonstrated for the above examples, including that the sorting algorithms indeed sort.

Our supplementary material includes a full development of these contributions in Coq.

A further contribution is a detailed consideration of well-founded recursion in Coq (Section 3), as implemented by the commands `Function`, `Program`, and `Equations`. We point out several issues, to motivate the alternative approach we propose.

2 STANDARD APPROACHES TO NONSTRUCTURAL RECURSION

The problem of nonstructural recursion is well known within the theorem proving community [Bove et al. 2016; Owens and Slind 2008]. In this section, we survey several previous solutions.

2.1 Nonstandard structural recursions

Some nonstructurally recursive functions can be rewritten in a nonstandard way to become structurally recursive. For example, division by iterated subtraction is not structurally recursive, because it recurses on the result of a subtraction. Structural recursions must recurse only on pattern variables, and may not (in general) recurse on results of other function calls. In Coq's standard library (also Agda's) one finds a nonstandard implementation of division, using a four-argument function that fuses subtraction and division. This could also be written with nested recursions, where the inner recursion is essentially the subtraction function, and the outer is the loop for division. Either way, the functions must be fused in order to pass the structural termination check. This is a pity, as existing theorems about subtraction cannot be applied for reasoning about this formulation of division, as it does not actually invoke subtraction.

For another example, mergesort in Coq's standard library is expressed "using an explicit stack of pending mergings" [library Coq 2009]. The formulation is clever, and does not rely on nested recursions. But this comes at the cost of a very different – indeed, arguably barely recognizable – formulation of the algorithm.

While it is possible to find nonstandard implementations of nonstructurally recursive functions like division and mergesort, these implementations require ingenuity to craft, and concomitant effort to verify within Coq. It is thus desirable to have a way to write nonstructurally recursive functions in a style closer to their standard definitions.

2.2 Sized types

One technique supporting more direct expression of nonstructurally recursive functions is sized types [Hughes et al. 1996]. For example, a much more natural formulation of mergesort in the Agda type theory may be found in Copello et al. [2014]. The main algorithm is exactly as expected: split, recurse, then merge. The code is written using sized types, where datatypes are additionally indexed by static approximations of the sizes of the inhabiting data [Barthe et al. 2004a]. This method supports compositional termination checking for programs close to the standard definitions, but it has several costs. Users must work with sized versions of datatypes, and implementors must add support for sized types. In the case of Coq, while there has been a recent proposal for adding sized types, this has yet to be adopted [Chan and Bowman 2019].

2.3 Well-founded recursion

In constructive type theory, well-founded recursion is a widely used technique to represent nonstructural recursions as structural ones. Each function that recurses nonstructurally on some input `x` is augmented with an extra argument `acc : Acc R x`. This `acc` can be viewed as evidence that it is legal to recurse on any `y` which is less than `x`, according to relation `R`. From `acc`, one uses a proof of `R y x` to obtain evidence of `Acc R y`, which becomes the extra argument for the recursive call on `y`. This technique is implemented in theorem provers based on constructive type theory like Coq, Agda, and Lean. It is also used in provers with different logical foundations, including Dafny and Isabelle [Leino 2010; Nipkow et al. 2002], although without the explicit proof terms in code. The commonly used approach of adding a "fuel" argument to a function and then recursing on that may be viewed as a crude approximation.

In Coq, the type `Acc R X` is in `Prop`, and satisfies the so-called *singleton elimination* condition, a syntactic check on the form of the definition of inductive propositions that is intended to ensure no information can leak from that type to a computational type (cf. discussion in the introduction of Gilbert et al. [2019]). This allows proofs of `Acc R x` to be erased soundly during extraction from Coq to external languages like OCaml or Haskell. So well-founded recursion becomes a technical device to allow writing code that, under extraction, is exactly the desired nonstructural recursion. But within Coq itself (as opposed to via extraction), the situation with well-founded recursion is more complicated.

3 WELL-FOUNDED RECURSION IN COQ IN MORE DETAIL

In Coq, three commands exist to make writing well-founded recursions easier: `Function` [Barthe et al. 2006], `Program` [Sozeau 2006], and `Equations` [Sozeau and Mangin 2019]. `Program` and `Equations` aim to provide simpler ways to program in Coq with dependent types, and thus go beyond just providing well-founded recursion. All three commands can only be used at the top level, not locally within another term. They generate proof obligations for code which would otherwise not type-check in Coq due to failure of structural termination (among other things). From such code, they use use well-founded recursion to generate equivalent terms which Coq will accept. To see how these commands work, and provide points of comparison with the proposed new approach, let us consider how they handle a simple example.

3.1 The `wordsBy` function

Haskell's `Data.List.Extra` module includes a function `wordsBy`, which breaks a list into maximal sublists whose elements do not satisfy a predicate `p` [Mitchell 2021]. For example,

```
wordsBy isSpace " good day "
```

returns `["good","day"]`. Haskell code is in Figure 1. The cons clause of the definition has two recursive calls. The first, `wordsBy p tl`, is structural. The second invokes `wordsBy p` on a value obtained from another recursion, namely `break`, defined in terms of the function `span` in Figure 1. This recursive call is not structural, but it can be justified by well-founded recursion, as the value `z` produced by `break` will always have length less than or equal to `tl`.

```
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
```

```
span :: (a -> Bool) -> [a] -> ([a],[a])
span _ []    =  ([], [])
span p xs@(x:xs') =
  if p x
  then let (r,s) = span p xs' in
       (x:r,s)
  else ([],xs)
break p = span (not . p)
```

Fig. 1. Haskell code for `wordsBy` and its auxiliary `span` and `break` functions.

3.2 Implementation with `Function`, `Program`, and `Equations`

Figures 2 and 3 show implementations in Coq of well-founded versions of `wordsBy`, using `Program` and `Equations`, respectively. The code using `Function` is essentially identical to that of `Program`, so we show only the version with `Program`. The `{measure (length l)}` annotation in the version with `Program`, and `wf (length l) lt` for the one with `Equations`, tell the commands to use the length of the list, ordered by the less-than relation, as a measure function to justify the recursive calls.

The programs rely on the obvious ports (not shown) of `break` and `span` from Figure 1. Both figures omit the short proofs of the obligations for the two recursive calls:

- `length tl < length (hd :: tl)`
- `length z < length (hd :: tl)`, where `break p tl = (w,z)`

The latter requires a short structurally inductive lemma about `span` (also elided). By design, the code accepted by `Program` (and `Function`) is very similar in style to what one would usually write in Coq for structural recursion. In contrast, the code using `Equations` is quite different, as `Equations` aims to provide an interface for dependently typed programming similar to Agda's. One uses the `with` construct to extend a pattern-match with a new scrutinee, and `inspect` to retain an equation in the context between the scrutinee and the pattern it matches. Here, `inspect` is needed in order to have the necessary equation in scope when proving the obligations for the recursive calls. This

```
Program Fixpoint wordsByP (l : list A) { measure (length l) } : list (list A) :=
  match l with
  | []       => []
  | hd :: tl => if p hd
               then wordsByP tl
               else let '(w,z) := break p tl in
                      (hd :: w) :: (wordsByP z)
  end.
```

Fig. 2. Using Program to generate a well-founded version of wordsBy; the version with Function is essentially identical. Proofs of obligations not shown.

```
Equations? wordsByE (l : list A) : list (list A) by wf (length l) lt :=
 wordsByE [] := [] ;
 wordsByE (hd::tl) with p hd => {
   | true := wordsByE tl;
   | false with inspect (break A p tl) => {
     | exist ?(_) (w,z) e  := (hd::w)::(wordsByE z)
     }
   }.
```

Fig. 3. Using Equations to generate a well-founded version of wordsBy. Proofs of obligations not shown.

style of dependently typed programming is quite different from the usual format for structural recursions in Coq.

3.3 Size of generated terms

Function, Program, and Equations all generate valid Coq implementations of wordsBy using well-founded recursion. Figure 4 gives some statistics. For a short piece of starting code (8 lines with Equations, 11 with Program and Function), we see a substantial blow-up in the generated terms, with the introduction of quite a few auxiliary function definitions. The six definitions for Equations are generated following the nested matching structure of the Agda-style code.

Command	LoC	#Aux
Function	208	2
Program	60	4
Equations	50	6

Fig. 4. The total lines of code (**LoC**) and number of auxiliary functions (**#Aux**) generated for wordsBy.

How could one hope to prove theorems about terms this complicated? Well, Function and Equations both automatically derive reduction lemmas summarizing how the function executes depending on the form of the input, as well as induction principles following the structure of the recursion. The stated goal is to hide the generated code completely from the programmer, by providing higher-level reasoning principles. But if the underlying code is so complicated that the goal is to hide it completely, then perhaps there is room for improvement in the underlying scheme for nonstructural recursion!

3.4 Performance of generated code

Another issue is that performance of the generated terms can be poor, even asymptotically so. This is not the case for the *extractions* of those terms to external languages like OCaml or Haskell. After all, the type theory has been cleverly designed to make such extractions run with no overhead at all. But if one executes the code *within* Coq, the story is different, as the machinery of well-founded recursion remains. Furthermore, erasing this machinery within the type theory using definitional proof irrelevance would destroy decidability of type checking [Gilbert et al. 2019]. As a consequence, well-founded recursions will execute with overhead within Coq, for the foreseeable future.

To see how much overhead may occur, consider a family of examples, indexed by `NUM` and an implementation of `wordsBy`, of the following form:

```
Definition t1 := repeat 1 NUM.
Definition t2 := repeat 0 NUM.
Eval vm_compute in (length (WORDSBY (Nat.eqb 0) (t2 ++ t1))).
```

This has the effect of testing repetition of both branches of the split on `p hd` in the different versions of `wordsBy`, which instantiate `WORDSBY`. The benchmarks all evaluate to 1.

Figure 5 shows the results. The x-axis is labeled by `NUM/1000`, and the y-axis by seconds of time for Coq to evaluate the benchmark. Coq evaluation times can be somewhat variable, so we show the median of three runs for each timing. We omit the data for `Function`, where the graph soars quadratically off the chart: for the first value of `NUM` (2000), running time is already at 5.6s. The data for `Equations` are essentially identical to those for `Program`, so we just show the latter. We compare against an optimized version, explained below. There is a dip in the graph for the optimized version when `NUM=5500`, as Coq switches over (automatically), for both benchmarks, to a more efficient representation for `NUM`. We see that evaluation time for the version generated by `Program` (and `Equations`) appears to be quadratic in `NUM`. The benchmark should be executable in linear time.

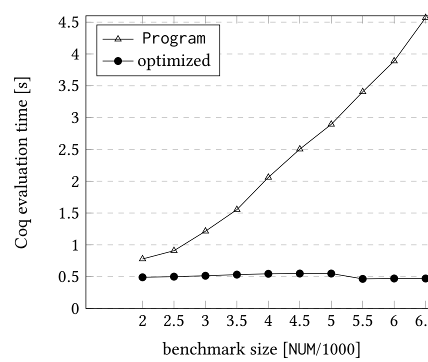

Fig. 5. Comparing two versions of `wordsBy` generated by `Program`. The optimized one uses a more efficient proof of well-foundedness of less-than. The versions generated by `Function` and `Equations` also exhibit quadratic running time.

The source of the quadratic behavior is that the measure function (`length`) gets evaluated on the input list for each recursion. In the case of the version for `Program`, this happens inside the proof of well-foundedness of less-than (`well_founded_ltof` from `Arith/Wf_nat.v` in the Coq standard library). So at each of the recursive calls, we call `length` on the list that is the input for that call. For this family of benchmarks, there are a linear number of recursive calls on lists of decreasing size, so calling `length` on each such list takes quadratic time.

Once we have identified this problem, we can solve it by a modified version of `well_founded_ltof`, which moves applications of the measure function behind an opaque definition. We spare the reader the details. Using this modified version, we can use `Program` to generate the optimized version of `wordsBy`, whose performance, shown in Figure 5, is satisfyingly linear. The same modification improves the version with `Equations` similarly. For the version with `Function`, one must also manually introduce opaque proofs for the decrease obligations, or else the behavior is still quadratic.

3.5 Implementation cost

There is a significant engineering cost, shown in Figure 6, to implementing the Coq commands for well-founded recursion. All three implementations include OCaml code depending on the internals of Coq. While `Program` and `Equations` aim to do more than just support well-founded recursions, `Function` does have just that purpose. Thus its rather heavy line count is concerning.

While the code for Program is much shorter, it also does less: it does not derive reduction lemmas or customized induction principles.

3.6 Discussion

We hope to have convinced the reader that the situation with well-founded recursion in Coq, and by extension similar theories, is somewhat complicated. For efficient extracted code, the technique relies on some special typing principles that allow proofs of well-foundedness to be erased upon extraction. For evaluation within Coq, though, such erasure is not possible [Gilbert et al. 2019]. To obtain versions with the expected runtime complexities, we have seen it is necessary to prevent measure functions referenced in proofs from being evaluated when the code is run within Coq. If opaque definitions were not available in the theory, the only technique we know to prevent asymptotically inefficient execution for examples like wordsBy is to introduce a custom ordering directly on the datatype; for the example of lists:

	Coq LoC	OCaml LoC
Function	64	9,544
Program	1,414	467
Equations	451	14,446

Fig. 6. Statistics on the implementations of Function, Program, and Equations

```
Inductive smallerList : list A -> list A -> Prop :=
| sl_nil  : forall (h : A) (t : list A), smallerList [] (cons h t)
| sl_cons : forall (h h' : A)(x y : list A),
            smallerList x y -> smallerList (cons h x) (cons h' y).
```

In our experiments, this approach also recovers linear-time execution of wordsBy. It is not too attractive, though, as it requires defining such an ordering and proving it well-founded, for each datatype. Well-foundedness proofs in particular are rather difficult to write, so this would require additional automation to be usable in practice.

Even with performance problems solved, there is still the fact that the Coq terms generated by these commands are large and complex. Reduction lemmas and custom induction principles are intended to hide them completely from the user. While this could be viewed as an admirable use of abstraction, less positively it points to issues with the underlying approach (which has to be hidden from the user to be manageable). Furthermore, these commands require substantial amounts of code to implement.

In the rest of this paper, we propose an alternative to well-founded recursion, which we call divide-and-conquer recursion. The approach does not rely on the structural recursion of CIC at all. Rather, it makes use of the powerful, and compositional, termination properties imposed directly by typing in the Calculus of Constructions. It can be applied within terms (not just as top-level commands), does not blow up code at all, leads to execution with the expected asymptotic complexities without any tweaking, and does not require any special features of CIC and Coq's implementation, such as opaque definitions, singleton elimination, or the Prop-Set distinction. Our approach also represents a completely different style of terminating programming: instead of writing proofs about decrease of arguments, one programs against an interface, with no dependent types, that enforces termination. Our approach is less general than well-founded recursion, as it applies just to the specific – but inarguably very important – class of divide-and-conquer algorithms. We do not have an independent characterization of this class, but will hope to convince the reader it has broad scope through the diversity of examples (Section 5).

4 THE INTERFACE FOR DIVIDE-AND-CONQUER RECURSION

In this section, we describe the interface our development provides to programmers for writing divide-and-conquer recursions. The implementation is explained in Section 6. 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. [Cockett and Spencer 1992; Hagino 1987; Traytel et al. 2012]). For readers not so familiar with this approach, we begin with a short tutorial.

4.1 The Algebraic Approach to Datatypes

The simplest form of algebras, namely F-algebras for functor F, are categorically morphisms from $F\ X$ to X, for carrier object X. From a programming perspective, an F-algebra with carrier X is given input of type $F\ X$, and must compute a result of type X. F is called the *signature functor* for the datatype. An example is the signature functor for lists, parametrized by the type `A` of elements:

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

This is similar to the list datatype, except that for the second argument to `Cons`, one supplies an argument of type `X` rather than the tail list. For good introductions to the functorial view of datatypes in functional programming, see Swierstra [2008] and Bird and de Moor [1997].

An example `ListF`-algebra, written in Coq and using constructors `O` and `S` for natural numbers, is the following for computing the length of a list:

```
Definition lengthAlg(d : ListF nat) : nat :=
  match d with
    Nil       => O
  | Cons x xs => S xs
  end.
```

This code type checks, because `xs` has type `nat`.

Once one has an algebra of type $F\ X \longrightarrow X$, it can be converted, by a function traditionally called `fold`, to a catamorphism of type $\mu F \longrightarrow X$. Here, μF is the least fixed-point of F, which corresponds to the actual datatype of interest. So μ(`ListF A`) will represent lists with elements of type `A`. The catamorphism applies the algebra throughout the input, to compute a result of type X.

The type `list A` from Coq's standard library will be distinct, in our development, from μ(`ListF A`), which we will denote `List A`. We include conversion functions between these types: `toList` goes from `list` to `List`, and `fromList` does the reverse. With our approach, one first converts from `list` to `List`, and then applies our recursion scheme. From the user's perspective, `List A` is just used behind the scenes to do divide-and-conquer recursion on a Coq value of type `list A`. Usually these recursions can compute a `list` as their output directly, and so no conversion back from `List` is needed.

Finally, we note that the algebraic approach uses a single constructor called "`in`", of type $F\ \mu F \longrightarrow \mu F$. Specialized to `ListF` and eliding the parameter `A`, this function takes a `ListF List` to a `List`. So it takes in a `ListF` data structure where the second argument of `Cons` is indeed a list, namely the tail; and produces a list. It thus captures in one function the usual `nil` and `cons` constructors.

4.2 Algebras for Divide-and-Conquer Recursion

Algebras in our development are more complex than the basic F-algebras just recalled. Where a `ListF`-algebra with carrier X is presented with input of type `ListF` X and must produce a value of type X, our divide-and-conquer algebras are given a toolbox of inputs that we call a *recursion universe*, following Stump et al. [2020]. Furthermore, we have two different kinds of algebras: `Alg` is for outermost recursions like `wordsBy`, and `SAlg` is for subsidiary (inner) recursions like `span`. `Alg` and `SAlg` are similar, but only subsidiary algebras are given abstract versions of the datatype constructors. The reason for this difference is discussed in Section 6 on the implementation.

This section presents the interfaces for `Alg` and `SAlg` by going through the toolbox of inputs each is given. Both kinds of algebra generalize the type of the carrier X from just `Set` to `Set` $\to$ `Set`. Additionally, the carrier must be a functor satisfying the usual functor identity law. We denote this functoriality requirement `Functor` X. Both `SAlg` and `Alg` have kind `KAlg`, defined as:

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

The input of kind `Set` $\to$ `Set` is the carrier for the algebra.

The types `Alg` and `SAlg` are recursive. But they have a special form, called *positive-recursive* (cf. Section 2.2 of [Coquand and Paulin 1988]): every recursive occurrence in the type appears in the domain part of an even number of function types. We will see in detail how to take a fixed-point of this type within Coq in Section 6.1. An important intuition for the definitions following is that each defines its own universe for recursion, where elements are of an abstract type R, and recursive calls are allowed only on data of type R.

4.2.1 *Interface for* `SAlg` *for subsidiary recursion.*

An `SAlg X`, where `X` of type `Set` $\to$ `Set` is the carrier, is a function which will be called with the following inputs, to perform a recursion that is subsidiary to some outer recursion. We call that outer recursion the *parent* recursion.

- `P`, the abstract type for the parent recursion.
- `R`, the abstract type for this subsidiary recursion.
- `up` of type `R` $\to$ `P`, for sending data from the current recursion up to the parent recursion.
- `sfo` of type `FoldT SAlg R`, for spawning yet a further subsidiary recursion (subsidiary to this one); we will consider the definition of `FoldT` shortly. Note that `SAlg` is used recursively here; we will discuss this further when defining `FoldT`.
- `abstIn` of type F `R` $\to$ `P`; this is an abstracted version of the datatype's (sole) constructor `in`. Recall that `in` has type $F\ \mu F \to \mu F$. In the type for `abstIn`, the first μF is abstracted to `R`, and the second to `P`. The typing says that applying this abstracted constructor to data in the current recursion universe constructs data in the parent recursion universe.
- `rec` of type `R` $\to$ `X R`, which is used for making recursive calls on any value of type R. Note that the carrier `X` is applied here to `R`. This means that data produced by a recursive call could, potentially, be eligible for a further recursive call with this algebra. So the interface supports nested recursion [Krauss 2010].
- `d` of type F `R`. We call this the *subdata structure*. It presents a value of the algebra's datatype (unfolded from μF to $F\ \mu F$), but using `R` for subdata. This means that the subdata are eligible for recursive calls using `rec`.

Given these inputs, an `SAlg` must produce an output of type `X P`. Please note the subtlety here: the carrier `X` is applied to the abstract type `P` of the *parent* algebra, as opposed to the abstract type `R` of the current algebra. This is what will enable outer recursions to recurse on results produced by subsidiary ones, and what necessitates the generalization of the type of the carrier to `Set` $\to$ `Set`.

4.2.2 *Definition of* `FoldT`.

The `sfo` component of an `SAlg`'s toolbox has type `FoldT SAlg R`, where `R` is the abstract type for the `SAlg`. The Coq definition is:

```
Definition FoldT(A : KAlg)(R : Set) : Set :=
  forall (X : Set -> Set) (FunX : Functor X), A X -> R -> X R.
```

The definition is parametrized by the type `A` of algebra to use (either `Alg` or `SAlg`). Given a functorial carrier `X` and an algebra of type `A X` (again, this will either be `SAlg X` or `Alg X`), and given an `R`, an `X R` is returned. Having `sfo` of type `FoldT SAlg R` (so instantiating `A` to `SAlg`) means that we can recurse on any `R` from the current recursion, using a subsidiary algebra, and obtain a result of type `X R`. Again, the argument `R` to the carrier `X` is critical here: invoking a subsidiary recursion produces data upon which the current recursion may legally recurse.

For our implementation (Section 6), it will be crucial that `SAlg` is a positive recursive type. To see this, note that `FoldT SAlg R` is convertible with

```
forall (X : Set -> Set) (FunX : Functor X), SAlg X -> R -> X R.
```

The occurrence of `SAlg` is negative, as it is in the domain part of just one function type. But `FoldT SAlg R` is the type for an input to an `SAlg`. So in the definition of `SAlg`, the sole occurrence of `SAlg` appears in the domain part of two function types. Hence, `SAlg` is positive-recursive.

4.2.3 *Interface for* `Alg` *for outer recursion.*

The type for algebras for outer recursion is `Alg X`, where again `X` is the carrier, of type `Set` → `Set`. An algebra is a function that takes arguments similar to those for subsidiary algebras above, except that there is no parent algebra. So there are no `up` or `abstIn` inputs, as these only make sense if there is a parent. Also, the algebra returns a value of type `X R`. Finally, it turns out that to implement the interface (Section 6), we must add a function `fo` to fold an `Alg`, in addition to `sfo` for folding an `SAlg`. Inclusion of `fo` makes `Alg` a positive-recursive type, like `SAlg`. We end up with the following components:

- `R`, the abstract type for this recursion.
- `fo` of type `FoldT Alg R`, for spawning an inner recursion using an `Alg` instead of an `SAlg`.
- `sfo` of type `FoldT SAlg R`, for spawning subsidiary recursions.
- `rec` of type `R` → `X R`, for making recursive calls.
- `d` of type F `R`, the subdata structure as for `SAlg`.

4.2.4 *The* `Dc` *type, and folding algebras.*

Given a functor F, our development provides a type `Dc`, for data supporting divide-and-conquer recursion using algebras of the kinds discussed above. `Dc` has this constructor:

```
Definition inDc : F Dc -> Dc.
```

There are also `fold` and `sfold` functions for folding algebras (`Alg`) and subsidiary algebras (`SAlg`), respectively, over values of type `Dc`:

```
Definition fold  : FoldT Alg Dc.
Definition sfold : FoldT SAlg Dc.
```

Our development also provides functions `rollAlg` and `rollSAlg` for creating an `Alg` or an `SAlg`, respectively, from a term whose type is the one-step unrolling of the recursive definition. A final function we will use below is

```
Definition out{R : Set}(sfo : FoldT SAlg R) : R -> F R
```

This function takes an `sfo` function, and uses it to turn an `R` into an `F R` by a trivial subsidiary recursion. It can be used to perform nested pattern-matching on a term of type `R`, preserving the possibility of recursing. For the central example of lists, we instantiate F with `ListF A` (for parameter `A : Set`), which is proven to be a functor. We use `inDc` to write constructors `mkNil` and `mkCons`, and use them to convert from `list` to `List`. Our development can derive much of this boilerplate automatically; see Section 6.4.

5 EXAMPLES OF PROGRAMS USING THE DIVIDE-AND-CONQUER INTERFACE

In this section, we consider a number of examples written using the interface for divide-and-conquer recursion described in the previous section. We will elide the obvious proofs of functoriality for the carriers of the algebras for these examples. We include Haskell code to help clarify some of the

```
Definition SpanSAlg(p : A -> bool) : ListSAlg A SpanF :=
 rollSAlg (fun P R up sfo abstIn span xs =>
   match xs with
     Nil        => ([], abstIn xs)
   | Cons x xs' =>
     if p x
     then let (r,s) := span xs' in (x :: r, up s)
     else ([], abstIn xs)
   end).
```

Fig. 7. Subsidiary algebra for span

algorithms, and for comparison with the Coq versions which are the contribution of the section. All examples in this section have been evaluated within Coq and exhibit runtimes compatible with the expected asymptotic complexities.

5.1 The `wordsBy` function

Our first example is the `wordsBy` function (Section 3.1).

5.1.1 Implementing `span`. To implement `wordsBy`, we first need to implement `span` (Figure 1) as a subsidiary recursion. Since `wordsBy` recurses on the second component of the pair returned by `span` (via `break`), we need to ensure that this second component is typed at the abstract type of `span`'s parent recursion. So the carrier of the `SAlg` for `span` is:

```
Definition SpanF(X : Set) : Set := list A * X.
```

Figure 7 shows the code for subsidiary algebra `SpanSAlg`. It makes calls to `up` and `abstIn`, because the body of `SpanPAlg` is supposed to have type `SpanF P`, where `P` is the abstract type for the parent recursion. Invoking `up` and `abstIn` is the only way to get a value of type `P` within the body of the `SAlg`. The calls to `abstIn` are well-typed because `xs` is the subdata structure for the algebra, of type `ListF R`, which is exactly the input type for `abstIn`. For the type of `SpanSAlg`, we use a slight abbreviation `ListSAlg` to specialize `SAlg` to the `ListF` functor.

As a convenience when invoking `SpanSAlg` from another recursion, we define:

```
Definition spanr{R : Set}(sfo : ListSFoldT A R)
                (p : A -> bool)(xs : R) : SpanF R :=
  sfo SpanF SpanFunctor (SpanSAlg p) xs.
```

The type of `sfo` is another slight abbreviation, for `FoldT (SAlg (ListF A))`. So `sfo` can initiate recursions on `List`s with `SAlg`s. Here, `spanr` calls `sfo` to initiate a recursion on `xs` using `SpanSAlg`. The point is that `spanr` hides the argument `SpanFunctor`, which proves that the carrier `SpanF` of `SpanSAlg` is indeed a functor. Limitations of Coq's system of implicit arguments prevents this from being inferred automatically.

5.1.2 Implementing `wordsBy`. Using `breakr`, defined similarly to `spanr` but negating the predicate `p`, we can write an `Alg` for `wordsBy`, in Figure 8. The `Alg` is given a function `sfo` of type `FoldT SAlg R` for initiating subsidiary recursions, and it passes this function to `breakr`. The call to `breakr` then has type `list A * R`, since the carrier of `SpanSAlg` is `SpanF`, and `FoldT` says that `sfo` will return a result of type `X R` for an algebra with carrier `X`. The code of Figure 8 is otherwise very similar to the starting-point code of Figure 1. We have chosen to implement this function as an `Alg`, and to have it return a `list`. So `WordsByAlg` has constant carrier `Const (list A)`, which is the constant functor which only returns `list A`. We may now define

```
Definition WordsByAlg(p : A -> bool) : ListAlg (Const (list (list A))) :=
  rollAlg (fun R fo sfo wordsBy xs =>
    match xs with
      Nil         => []
    | Cons hd tl => if p hd
                    then wordsBy tl
                    else let (w,z) := breakr sfo p tl in
                           (hd :: w) :: wordsBy z
    end).
```

Fig. 8. Algebra for wordsBy

```
Definition WordsBy(p : A -> bool)(xs : List A) : list (list A).
Definition wordsBy(p : A -> bool)(xs : list A) : list (list A).
```

The former uses the `fold` function described above (Section 4.2.4), with `WordsByAlg`. The latter converts a `list` to a `List` and then invokes `WordsBy`. The type of `wordsBy` is expressed solely in terms of standard types from the Coq prelude. So the `List` type and the machinery for divide-and-conquer recursion are completely internal to `wordsBy`.

5.2 Combinator-based run-length encoding

Run-length encoding is a basic data-compression algorithm where maximal sequences of n occurrences of element e are summarized by the pair (n, e) [Salomon and Motta 2009]. In this section, we demonstrate a concise implementation of run-length encoding using a recursion combinator we call `mapThrough`. The implementation also makes use of `span` (Section 5.1.1), a nice demonstration of the obvious benefit of compositionality that our approach enjoys over nonstandard structural recursion. Suppose we had implemented `wordsBy` using a syntactically nested or fused recursion, as discussed in Section 2.1, where `span` is the inner recursion. It is not so obvious how to do that. But if done, we would not be able to reuse `span` here for `rle`, as it could not be abstracted out of the code for `wordsBy`. With our approach, the code for `span` is separate from that for `wordsBy`, and can be invoked from another function (`rle`).

5.2.1 The `mapThrough` *combinator.* Haskell code for `mapThrough` is given in Figure 9. It works like `map` on lists, except that the function being mapped takes both the head and tail of the list, and returns a pair of an element to include in the output list, and a suffix of the list on which the recursive `mapThrough` should continue. The Haskell library `Data.List.Extra` defines this function with the name `repeatedly` [Mitchell 2021]. To write `mapThrough` using divide-and-conquer recursion in Coq, we will use this type for functions that will be applied to lists:

```
Definition mappedT(A B : Set) : Set :=
  forall(R : Set)(sfo : ListSFoldT A R), A -> R -> B * R.
```

Such a function is given the head of the list at type `A`, and then the tail at abstract type `R`. We need to supply `mappedT` functions with the `sfo` function to use for initiating subsidiary recursions. This

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

Fig. 9. mapThrough in Haskell

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

Fig. 10. The algebra `MapThroughAlg`, and the type of `mapThrough`

`sfo` function will come from `mapThrough`'s recursion universe. The mapped function must then return a pair whose second component has type `R`, and hence must be (hereditarily) a tail of the input. Because of this typing, `mapThrough` will be able to recurse on that second component.

Given this definition, Coq code for `mapThroughAlg` is shown in Figure 10. The algebra is parametrized by the function `f` to be mapped through lists. The code is similar to the Haskell version (Figure 9), though when we call `f`, we must supply the abstract type `R` and fold function `sfo`. For simplicity, we choose to define `MapThroughAlg` as an `Alg` rather than an `SAlg`, and just return a `list B`. From the definition of `mappedT`, we have that `b : B` and `c : R`, so we may indeed invoke `mapThrough : R -> list B` on `c`.

5.2.2 Implementing run-length encoding with `mapThrough`*.* Now we may implement run-length encoding (`rle`) by mapping a function `compressSpan` through the input list. A Haskell implementation is in Figure 11. 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 12. The function `compressSpan` is written at the type `mappedT A (nat * A)` that will be required by `mapThrough`. This type is equivalent to:

```
forall(R : Set)(sfo:ListSFoldT A R), A -> R -> (nat * A) * R.
```

The code for `mapThroughAlg` will invoke `compressSpan` with the function `sfo` that is part of the recursion universe for `mapThroughAlg`. Then `compressSpan` will extract from the tail at type `R` (second input) the suffix upon which `mapThrough` should recurse (second component of the output pair). The algebra `RleAlg` is then obtained by supplying `compressSpan` to `MapThroughAlg` (Figure 10). We also define functions `Rle` and `rle` for top-level recursions, using `fold` and, in the latter case, `toList`.

```
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)
```

Fig. 11. Run-length encoding in Haskell, using `mapThrough` and `span`

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

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

Fig. 12. The function `rle` for run-length encoding, and the algebra `RleAlg` defining it in terms of `MapThroughAlg` of Figure 10

5.2.3 Comparison with well-founded recursion. It is straightforward to use `Program` to obtain a well-founded version of `mapThrough`, which we will call `mapThroughWf`. We consider this here, to see how the intertwining of proof terms in a well-founded recursion impacts combinator programming. The code is in Figure 13. The big difference is that here, we must change the type of functions that will be mapped through, so that they now produce a proof that the output list (on which `mapThroughWf` will then recurse) has length less than or equal to the length of the input list. This proof is then used to satisfy the proof obligation for the recursive call to `mapThroughWf`.

From the 11 lines of Figure 13, `Program` generates 270 lines of code using well-founded recursion. This expansion is worse than we saw for `wordsBy` above. Part of the blow-up is due to the fact that tactic-based proofs, such as one would naturally wish to write for the obligations that arguments decrease at recursive-call sites, can generate large proof terms. Of course, we could hide these through the use of Coq's `abstract` tactical, or by manually introducing intermediary lemmas; but the fact that the proofs appear in the code is an issue. Another source of blow-up is packing and unpacking the components of the dependent pair produced by the mapped function.

5.3 Harper's regular-expression matcher

Harper's matcher is a continuation-based matcher for regular expressions [Harper 1999]. It has been considered as a challenge problem for termination in a number of previous works [Bove et al. 2016; Korkut et al. 2016; Owens and Slind 2008; Xi 2002]. Bove et al. [2016] sketch a solution in Coq, and conjecture that there is no easy way to solve the problem without dependent types.

```
Definition mappedT(A B : Set) : Set :=
  A -> forall (xs : list A),
       B * sig (fun xs' : list A => length xs' <= length xs).

Program Fixpoint mapThroughWf{A B : Set}(f:mappedT A B)(xs : list A)
  { measure (length xs) } : list B  :=
  match xs with
    []       => []
  | hd :: tl =>
    let '(b, exist _ c pf) := f hd tl in
      b :: mapThroughWf f c
  end.
```

Fig. 13. A well-founded version of `mapThrough`, using `Program`

```
Definition K(T : Set) : Set  := T -> bool.
Definition MatchT(T : Set)   := K T -> bool.
Definition MatcherF(X : Set) := Regex -> MatchT X.
```

Fig. 14. Type definitions for the `matcher` function

Surprisingly, Stump et al. [2020] found such a solution, using the Cedille prover. Cedille's theory is quite different from Coq's, so it is a further surprise that their code can be ported to Coq using our divide-and-conquer recursion scheme. Like the original, this port does not use dependent types or reason about decrease of arguments.

Figure 14 shows the type definitions used for the `matcher` function. Harper's matcher uses a success continuation, which is invoked on the suffix of the string to be matched, after a prefix is found that matches the current regular expression (`Regex`). If the current prefix does not match, the matcher will return `false`. `K T` is the type for a continuation expecting an input of type `T`, and `MatchT` is the type for a function expecting such a continuation. The algebra defining the matcher has functorial carrier `MatcherF`.

Figure 15 shows the types for four short functions, totalling less than forty lines of code, for the matcher. As the code is essentially the same as in Stump et al. [2020], we focus here just on the types, to see how it fits into our divide-and-conquer recursion scheme in Coq. There is an inner recursion, `matchi`, on the regular expression; and an outer recursion, `matcher`, on the string. The inner recursion `matchi` recurses through the given regular expression, modifying the continuation as it goes. In calls `matchi R matcher r c t k`, the type `R` is the abstract type we are using for the tail `t` of the string, whose head character is `c`; `r` is the regular expression; and `k` is the continuation. This `R` is the abstract type for `matcherAlg`, which calls `matchi`. Interestingly, `matchi` does not need to be a subsidiary recursion, based on an `SAlg`. Instead, it is just a structural recursion on the regular expression. It is given the `rec` function available inside `matcherAlg`, for making recursive calls on a value of type `R`. This function is pulled into continuations when recursing into regular expressions, something that would certainly not be allowed with structural recursion.

This is an example of the compositionality enabled by our type-based approach: because there is no syntactic check on recursions beyond typing, we can perform arbitrary (well-typed) computation with the function to use for recursive calls. Hence the `matcher` function can pass itself to the recursion `matchi`, which would not be allowed with structural recursion. The higher-kinded carrier `MatcherF` also plays an important role. Inside `matcherAlg`, recursive calls produce values of type `MatcherF R`, where `R` is the abstract type for the recursion. But in `matcherh`, where we do a `fold` with the `matcherAlg`, the result has type `MatcherF String`, where `String` abbreviates `List char` (and similarly, `string` abbreviates `list char`). So we can shift our view from the abstract type inside the recursion, which is needed so the continuation can accept a value of abstract type; to the actual concrete `String` type outside the recursion. The final `matcher` function converts from `string` to `String` and supplies a base-case continuation.

Bove et al. [2016] describe an implementation of Harper's matcher in Coq, using dependent types for the type of the continuation, and a merged version of `matchi` and `matcher`. For this merged code, `Program` is used to implement lexicographic decrease of the pair of the regular expression and the string. In contrast, we are able to keep the functions separate, and for `matchi`, just use structural recursion on the regular expression. The outer recursion `matcher` then needs to use our scheme, to support embedding recursive calls to `matcher` into continuations inside `matchi`. We do not need to use dependent types or any other technique to prove that strings are smaller

```
Definition matchi(T : Set)(matcher : T -> Regex -> MatchT T)
  : Regex -> Ascii.ascii -> T -> MatchT T.
Definition matcherAlg : ListAlg Char MatcherF.
Definition matcherh(r : Regex)(s : String) : MatchT String.
Definition matcher(r : Regex)(s : string) : bool.
```

Fig. 15. The types for Harper's matcher, using our divide-and-conquer recursion scheme

when `matcher` is invoked. Instead, we rely on the typing of CC to enforce this. This keeps the types simpler and eliminates explicit reasoning about termination.

5.4 Mergesort

Figure 16 gives our implementation of `mergeSort`, using a helper function `mergeSortH`, which invokes an algebra `MergeSortAlg`. The outermost `mergesort` function expects a regular `list A`. If the list is nonempty, it peels off the first head and passes it as argument to `mergeSortH`, which starts the initial outer recursion. `MergeSortAlg` then passes the first head to the left recursive branch, and the second head to the right. This continues until each recursive branch has only the additional head it carries. It then constructs a singleton from that head, and the singletons are merged on their way up, as expected.

Because `mergeSortH` has return type `list A`, we are able to re-use the `merge` implementation from Coq's standard library to merge the results of the recursive calls. The novelty here is that we recurse on `ys` and `zs`, which are formed from splitting the tail in half in the `Cons` case. Because we will recurse on values obtained from `split`, we must write `split` as a subsidiary algebra (Figure 17).

We wish to recurse on both splits, so `SplitAlg` has `SplitF` as carrier. The body of `SplitAlg` should return a value of type `P * P`. To split a list, we try to match two levels down into it, to find the first two elements (if there are that many). We will then deal these elements to the left and right components of the returned pair, respectively. To pattern-match two levels deep, we use

```
Definition MergeSortAlg : ListAlg A (Const (A -> list A)) :=
  rollAlg (fun R fo sfo mergesort xs a =>
      match xs with
        Nil        => [a]
      | Cons hd tl =>
          let (ys, zs) := sfo SplitF FunSplitF SplitAlg tl in
            merge (mergesort ys a) (mergesort zs hd)
      end).

Definition mergeSortH (xs : List A) (x : A) :=
  fold (ListF A) (Const (A -> list A))
      (FunConst (A -> list A)) MergeSortAlg xs x.

Definition mergeSort (xs : list A) : list A :=
  match xs with
  | []       => []
  | hd :: tl => mergeSortH (toList tl) hd
  end.
```

Fig. 16. An implementation of `mergesort` with destructed input.

```
  Definition SplitF(X : Set) : Set := X * X.
  Definition SplitAlg : ListSAlg A SplitF :=
    rollSAlg (fun P R up sfo abstIn split xs =>
        match xs with
          | Nil => (abstIn Nil, abstIn Nil)
          | Cons hd tl => match (out (ListF A) sfo tl) with
            | Nil => (abstIn (Cons hd tl), abstIn Nil)
            | Cons hd' tl' => let (ys, zs) := (split tl') in
                     (abstIn (Cons hd ys), abstIn (Cons hd' zs))
            end
        end).
```

Fig. 17. A subsidiary algebra for split.

out (discussed in Section 4.2.4), which converts the tail of type R to type ListF A R. This can be destructed by pattern matching. SplitAlg calls abstIn to create data of type P for each component of the returned pair, thus satisfying its return type of P * P.

5.5 Quicksort

We show (pure functional) quicksort as a final example, because it requires a different subsidiary recursion, namely partitioning, to divide the list. Code is given in Figure 19. We just show the algebras, from which the functions of interest are then concisely written using fold (Section 4.2.4). The code is parametrized by the type A of list elements, and a function ltA : A -> A -> bool for strict total ordering of elements. The code for PartitionAlg, as for SplitAlg above, makes use of abstIn to construct data of the abstract type P of the parent recursion. Since the carrier of PartitionAlg has return type X * X, where we call partition in the body of PartitionSAlg we get a value of type R * R. We must then return one of type P * P, which we can do using abstIn, and also using up to convert, in each case of the if expression, one of the components of type R directly to type P. Besides these calls to abstIn and up, the code is exactly as we would expect. The code for QuicksortAlg also has exactly the form we would like: partition, recurse, and combine the results (just by appending).

```
Definition PartitionF (X : Set) : Set := A -> (X * X).
Definition PartitionSAlg : ListSAlg A PartitionF :=
  rollSAlg (fun P R up sfo abstIn partition d bound =>
       match d with
         Nil => (abstIn d, abstIn d)
       | Cons x xs =>
         let (l, r) := partition xs bound in
           if ltA x bound then
             (abstIn (Cons x l), up r)
           else
             (up l, abstIn (Cons x r))
       end).
```

Fig. 18. Subsidiary algebra for partitioning a list based on a bound. The first list in the returned pair consists of elements less than the bound, and the second list of elements greater than or equal to the bound.

```
Definition QuicksortAlg : ListAlg A (Const (list A)) :=
  rollAlg (fun R fo sfo qsort xs =>
       match xs with
         Nil       => []
       | Cons p xs =>
         let (l,r) := partitionr sfo xs p in
           qsort l ++ p :: qsort r
       end).
```

Fig. 19. Algebra for quicksort, invoking `partitionr` defined from `PartitionAlg` of Figure 18.

5.6 Discussion

We have seen a diverse group of example programs written using our interface for divide-and-conquer recursion. These examples all make use of nonstructural recursion. They can all also be written using well-founded recursion, using proofs that arguments decrease at recursive calls. Those proofs are included, by `Function`, `Program`, and `Equations`, within the definitions of the functions. In contrast, with our divide-and-conquer recursion, there are no such proofs, neither within the programs nor somehow external to them. Termination is enforced, as described in the introduction, in a totally different way, namely by the typing of CC. Instead of proofs that arguments decrease, one achieves termination with our approach by coding the program in question against our non-dependently typed interface. There is some syntactic overhead in the resulting programs: subsidiary algebras must call `abstIn` and `up` functions, to communicate data up to outer recursion in a way that preserves the ability to recurse. But otherwise, these terms look as one would expect. In contrast, code written with `Function`, `Program`, or `Equations`, while appearing as expected on the surface, is elaborated into a complicated term possibly an order of magnitude larger.

6 IMPLEMENTING THE INTERFACE FOR DIVIDE-AND-CONQUER RECURSION

Having introduced our interface for divide-and-conquer recursion, and demonstrated it through a diverse group of examples, we turn to the intricacies of its implementation in Coq. Recall that the types `Alg` and `SAlg` of the interface are positive recursive. If we attempted to define them as inductive types within Coq, we would get an error, as Coq (and Agda, and Lean) restrict inductive types to satisfy a requirement known as *strict positivity*: in the type T of any argument to any constructor of the datatype, the inductive type may not be used in the domain part of an arrow type in T. The first technical problem is how to take a fixed-point for positive functors within Coq.

6.1 Retractive-positive recursive types

Coquand and Paulin [1988] proved that full positive-recursive inductive types are incompatible with Coq's type theory. One can recover these types by some subtle changes to the type theory [Blanqui 2005]. Here we give a different solution, dubbed *retractive-positive* recursive types. These are a weakened, but still sufficient, form of positive-recursive types, which do not require any changes to the underlying type theory. Our starting point is a functor `F : Set -> Set`. As a functor, `F` comes with an `fmap` function of type

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

We require only that `fmap` satisfy the identity-preservation law:

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

We do not require preservation of compositions. Ideally, we would like to use this definition:

```
  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.
```

Fig. 20. Derivation of retractive-positive recursive types

This formulation is exactly what is used in many approaches to modular datatypes in functional programming, like Swierstra [2008]. But it is (rightly) rejected by Coq, as instantiations of `F` that are not strictly positive would be unsound.

Figure 20 defines `Mu` as a strictly positive approximation to this ideal `Mu'`. Instead of taking in `F Mu`, the constructor `mu` accepts an input of type `F R`, for some type `R` with a function of type `R -> Mu`. Impredicativity is essential here: we will instantiate `R` with `Mu` itself in the definition of `inMu` (Figure 20). 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 [Mendler 1991]. This trick works here because `Mu` occurs strictly positively in the type for `mu`.

Returning to Figure 20, we have functions `inMu` and `outMu`, and a proof `outIn` that they make `F Mu` a retraction of `Mu`: the composition of `outMu` and `inMu` is (extensionally) the identity on `F Mu`. 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 A` (from Section 4). Our development actually uses a different type that wraps `F`, but using `ListF A` suffices to demonstrate the issue. Let us temporarily define `List A` as `Mu (ListF A)`. The canonical way to define the empty list would be:

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

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

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

With the `fmap` function for `ListF`, `fmap f (NilF A)` equals `NilF B` for any `f : A -> B`. So if we apply `outMu` from Figure 20 to `mkNil'` or `mkNil`, we will get `NilF (List A)`. But critically, `mkNil` and `mkNil'` are not equal, neither definitionally nor provably. Of course, one could define a function that puts `Mu` values in canonical form by folding `inMu` over them. Then `mkNil` and `mkNil'` would be equivalent. But they would still not be provably equal, which is the problem of noncanonicity. This is the price we are paying to get a form of positive-recursive type in Coq. We will see below (Section 7.3) how to work around noncanonicity in proofs.

To use these ideas to define the recursive types `Alg` and `SAlg`, we need a higher-kinded version of `Mu`, to account for the carrier of the algebra. But this is an easy adaptation. So, having accepted noncanonicity, we gain recursive versions of `Alg` and `SAlg`. We use `rollAlg`, `unrollAlg`, etc. to roll and unroll the recursive type expressions.

6.2 Implementing the Dc type

Let us see now how to implement the Dc type, and its fold and sfold functions (Section 4.2.4). Our development is generic in a functor F. As noted earlier, our approach is based on ideas from lambda-encodings of data. We define Dc recursively from DcF as follows:

```
Definition DcF(C : Set) :=
  forall (X : Set -> Set) (FunX : Functor X), Alg X -> X C.
Definition Dc := Mu DcF.
```

So a value of type Dc is a function which, for any algebra with functorial carrier X, produces a result of type X Dc. Functoriality of the carrier is required so that the occurrence of Dc in X Dc is positive, making DcF a positive-recursive type. The straightforward proof that DcF is a functor requires Coq's often asserted axiom of functional extensionality, in order to formalize the argument that functoriality of X implies functoriality of DcF.

Using inMu and outMu, we define functions rollDc and unrollDc, witnessing that DcF Dc is a retraction of Dc. Defining fold is then trivial, by construction:

```
Definition fold : FoldT Alg Dc := fun X FunX alg d => unrollDc d X FunX alg.
```

A Dc value is exactly a function that can be used to fold an algebra.

Not at all trivial, however, is the definition of sfold, which recurses over a Dc value using an SAlg. To understand why, let us attempt the definition of the constructor inDc for Dc. This function must look like this, for some values of R?, fo?, sfo?, and rec?, which are the components of the recursion universe that the (unrolled) alg is expecting:

```
Definition inDc : F Dc -> Dc :=
  fun d => rollDc (fun X xmap alg =>
                     unrollAlg alg R? fo? sfo? rec? d).
```

The choices of all of these are clear, except for sfo?:

- R? should be Dc.
- fo? is supposed to have type FoldT Alg R?, which is satisfied by fold : FoldT Alg Dc.
- rec? is supposed to have type R? -> X R?. We can achieve this using the term fold X xmap alg, as this has type Dc -> X Dc due to the typing of fold (recalling the definition of FoldT in Section 4.2.2).
- d has the correct type F Dc for the subdata structure.

To instantiate sfo?, we will define sfold, for folding an SAlg over a value of type Dc. It was trivial to define fold, because a Dc value is essentially its own fold function for *algebras*. But SAlg is a different interface, and cannot be folded directly with a Dc value.

To solve this problem, we define a function promote that converts an SAlg to an Alg, which may then be folded by a Dc. The definition of promote, shown in Figure 21, is the most intricate part of our derivation. The first subtlety is that to fold an SAlg, it turns out that we need to know that the abstract type R of the Alg can be mapped to Dc. So instead of carrier X, the Alg constructed by promote has carrier RevealT X, which adds a function type R -> Dc to the original carrier. The code for promote names this function "reveal", as it reveals the identity of R to be Dc. This revelation is trivial outside the Alg, because in the definition of sfold at the end of the figure, where we do a fold, the return type is RevealT Dc. This means that our requirement of a function R -> Dc becomes the trivial requirement of a function of type Dc -> Dc. This is met by the identity function, at the end of the definition of sfold.

Within the algebra constructed by promote, however, this reveal function has type R -> Dc, where R is the abstract type of the algebra. This is not a trivial tool to add to the toolbox. Let us see

```
Definition RevealT(X : Set -> Set) : Set -> Set := fun R => (R -> Dc) -> (X Dc).
Definition promote : forall (X : Set -> Set)(FunX : Functor X),
                              (SAlg X) -> Alg (RevealT X) :=
  fun X funX salg =>
    rollAlg (fun R fo sfo rec fr reveal =>
               let abstIn := fun fr =>
                   rollDc (fun X funX alg =>
                       fmap reveal (unrollAlg alg R fo sfo (fo X funX alg) fr))
               in let rec' := sfo X funX salg
               in unrollSAlg salg Dc R reveal sfo abstIn rec' fr).

Definition sfold : FoldT SAlg Dc :=
  fun X funX salg x =>
    fold (RevealT X) (FunRevealT X funX) (promote X funX salg) x (fun x => x).
```

Fig. 21. Code for `promote`, which converts an `SAlg` into an `Alg`

what we need in order to use the `salg` in the body of the algebra created by the call to `rollAlg` in the figure. We must have:

`unrollSAlg salg P? R? up? sfold? abstIn? rec? fr`

We choose these instantiations:

- P? is `Dc`.
- R? is the abstract type `R` of the algebra (i.e., the one we are constructing).
- up? is `reveal`, as this has type `R -> Dc` (matching `R? -> P?`).
- `sfold`? is the `sfo` function of the algebra.
- `rec`? is `sfo X funX salg`, which has type `R -> X R`, thanks to the type of `sfo`.

This leaves `abstIn`? to define. The `SAlg` interface says it should have type `F R? -> P?`, which becomes `F R -> Dc` with the instantiations we have made for `R`? and `P`?. Let us walk through the definition of `abstIn` in the body of `promote` (Figure 21). It takes in `fr : F R`, and must produce a value of type `Dc`. To do this, it applies `rollDc` to a function taking in an algebra `alg` with functorial carrier `X`. That function must then return a value of type `X Dc`. (This is the definition of a `Dc` value, as a function that applies an `X`-algebra to obtain a value of type `X Dc`.) We apply the (unrolled) `alg` to instantiate the recursion universe for `alg`. The components are all inherited from the algebra that `promote` is defining, except that we use `fo X funX alg` for the `rec : R -> F R` function expected by the `alg`. Since we are supplying `R` as the value for the `alg`'s abstract type, the whole application `unrollAlg alg ...` has type `X R`. We can then obtain the required type `X Dc` by applying `fmap reveal`, which has type `F R -> F Dc`.

6.3 Discussion

The above construction is intricate, but explains some facets of the interface. We can see now why `Alg` requires a `fo` function in addition to an `sfo` function: we need that `fo` function where we apply the `alg` in the definition of `abstIn`. Without it, the definition of `promote` could not be completed. We can also see why two types of algebras are needed. If we just had `SAlg`, then we would get stuck trying to define `inDc`: there, we need to apply an algebra (hypothetically, an `SAlg`) to the components of the recursion universe that it requires. But an `SAlg` requires an abstract version of the `inDc` function itself! How could we provide this in the middle of the definition of `inDc`? The above construction manages to do so, by breaking the circularity in stages: first we define `abstIn` (in the code for `promote`) assuming we have a function `sfo`, and then we use `promote` to define the real `sfold`. The cost of this technique is requiring two types of algebra. `RevealT` may seem unnecessary, as we could just build in the `reveal` function to the recursion universe. We found that doing so results in a definition of `abstIn` that cannot be proved extensionally equivalent to `inDc`. This makes it impossible to prove motive-preservation lemmas for subsidiary recursions, a crucial technique we will see in Section 7.

6.4 Functorializing Datatypes

While the above definitions are defined once and for all, they must be instantiated with the concrete functor being used in a divide-and-conquer recursion. As alluded to previously, inductive datatypes defined by Coq's `Inductive` datatype command (e.g., `list A`) are different from their functorial representations (e.g., `List A`) that divide-and-conquer recursions operate over. In order to apply our approach, a user must define the functor corresponding for the datatype being recursed over. In addition to the representation of the functor as a datatype (e.g., `ListF A`), users must also provide an implementation of `fmap`, a proof of the `fmapId` identity law, and functions for recursing over an encoded datatype via an algebra. The definitions of all these are largely boilerplate, and our Coq implementation includes a library for automatically generating each of them from a user-specified datatype. Our library is built on IDT [Ye and Delaware 2022], a Coq library for automatically generating exactly these sorts of boilerplate definitions via a combination of tactic-based metaprogramming and the MetaCoq framewok [Sozeau et al. 2020]. This library also automatically generates a variety of convenient definitions for users, including type aliases (e.g., `ListSFoldT`), constructors for functorial encodings of datatypes, and conversion functions between a datatype and its functorial representation (e.g. `toList` and `fromList`).

7 DIVIDE-AND-CONQUER INDUCTION, WITH EXAMPLES

It is a straightforward exercise, in the spirit of Bernardy and Lasson [2011], to implement an indexed version of our interface for divide-and-conquer recursion. The development is parametrized by index type `I : Set`. If one were doing dependently typed programming with an indexed type like `vector`, then `I` would be instantiated with the type for the indices (so `nat` for `vector`). Due to Coq's universe system, we need different versions of the development depending on whether the underlying sort for carriers of algebras is `Set` or `Prop`. We show just a version with `Prop`, suitable for divide-and-conquer inductions. The index type `I` is (implicitly) instantiated with `Dc`.

To provide a glimpse of the indexed development, consider the type for indexed algebras. Carriers have kind `(I -> Prop) -> (I -> Prop)`, generalizing the kind `Set -> Set` of nonindexed algebras by adding in the index type `I`. We see also the change to use `Prop` instead of `Set`. We abbreviate the kind `I -> Prop` as `kMo`, because it is the kind for *motives*, in the sense of McBride [2002]. So the carrier for an indexed algebra is a motive-transformer `X : kMo -> kMo`. The type of indexed algebras is `Algi`, specifying indexed versions of the components given to `Alg`:

- `R : kMo`, the abstract motive for the indexed recursion
- `fo : forall (d : I), FoldTi Algi R d`, the indexed fold function. For the case we are considering here, of divide-and-conquer induction, this allows initiating a subsidiary induction given a proof of `R d` for any `d`.
- There is similarly a version of `sfo` that uses an `SAlgi` instead of an `Algi`.
- `ih : forall (d : I), R d -> X R d`. Given a proof that the abstract motive `R` holds of `d`, this allows one to conclude that the motive `X R` holds of `d`. Invoking this function corresponds to applying the induction hypothesis.

- `d : I` and `fd : Fi R d`. This `fd` can be thought of as containing proofs of the abstract motive for various indices, and itself has index `d`.

The indexed algebra is then required to prove `X R d`. We denote the indexed version of `Dc` as `Dci`. A value of type `Dci d` can be understood as evidence that we may prove properties of `d` by divide-and-conquer induction.

For lists, the indexed functor is the following, where `lkMo` abbreviates `List A -> Prop`:

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

This looks just like the nonindexed `ListF` functor, except that the return types of the constructor are indexed by values of type `List A`. Again, following Bernardy and Lasson [2011], we can see this as the realizability translation of the nonindexed `ListF A`. We also derive the following indexed conversion function:

```
Definition toListi(xs : list A) : Listi (toList xs)
```

This converts a list `xs` from Coq's standard library into evidence that it is legal to prove properties about the `List` version of `xs` (namely `toList xs`) by divide-and-conquer induction.

To prove a theorem, we apply an indexed algebra using an indexed version of `fold`:

```
Definition foldi(i : I) : FoldTi Algi Dci i.
```

With `I` instantiated to `List A`, and expanding the definition of `FoldTi`, the return type becomes:

```
forall (X : kMo -> kMo) (xmap : Functori I X), Algi X -> Dci i -> X Dci i.
```

This says that given `i : List A`, an indexed algebra with carrier `X`, and a proof of `Dci i`, we can derive `X Dci i`. Again, this shows `Dci i` acting as permission to perform divide-and-conquer induction, in this case to prove `X Dci` about `i`.

7.1 Decoding property for `rle`

Using indexed algebras, it is possible to reason about the behavior of divide-and-conquer recursions. As an example, suppose we wish to show decoding the run-length encoding of a list results in the original list, where `rld : list (nat * A) -> list A` is the obvious decoding function:

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

Proving this theorem requires the three lemmas about `span` formulated in Figure 22. 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 proposition `Forall` from Coq's standard library to state that all the elements of the prefix returned by `span` satisfy `p`. These lemmas are proved using indexed algebras with constant (indexed) carriers. In contrast, `MotivePresF` is not constant: it expresses that `span` preserves motives from the input to the returned suffix `r`. When the abstract motive of the outer recursion holds of a value, we may invoke the induction hypothesis. So motive-preservation of `span` is the key to invoking our outer induction hypothesis on the returned suffix, when reasoning subsidiarily about `span`.

Using these lemmas, we can write a short (10 lines) proof of `RldRle` using subsidiary induction. This proof invokes 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 `MotivePresF` takes in the indexed fold function `foi` from the outer induction (for `RldRle`), to show that the abstract motive `R` applies to the suffix `r` returned by `span`. This enables the outer induction hypothesis (for `RldRle`) to be applied.

```
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 MotivePresF(p : A -> bool)(R : List A -> Prop)(xs : List A) : Prop :=
  forall (l : list A)(r : List A),
    span p xs = (l,r) -> R r.
```

Fig. 22. Formulations of three lemmas about span

```
Definition MotivePresF(R : List A -> Prop) (l : List A) :=
  let ret := Split A l in
    R (fst ret) /\ R (snd ret).
```

Fig. 23. Carrier for proving that `Split` preserves motives

We can reuse the lemmas about `span` for other proofs. For example, proving that all the lists returned by `wordsBy p` consist of elements where the predicate `p` does not hold uses two of these lemmas. This would not have been possible if we were relying on nonstandard structural recursions.

7.2 The sorting functions indeed sort

To prove that `mergeSort` sorts requires just one helper lemma, namely that the `Split` function for splitting a list preserves abstract motives. The carrier for the indexed algebra is shown in Figure 23. With this proved, verifying mergesort proceeds easily by divide-and-conquer induction: the abstract motive for that proof is preserved by `Split`, and hence we may invoke the induction hypothesis on the lists `Split` returns. We may then apply a theorem from Coq's standard library, that merging sorted lists yields a sorted list.

Verifying that `Quicksort` truly sorts is more involved, as one must prove first that the `partition` function whose `Alg` we saw above really does partition the list. This is proven with an indexed subsidiary algebra, so that we may then apply the outer induction hypothesis to the results of partitioning. A further subsidiary induction is required to show that the sorted lists are still partitioned, so that appending them, with the pivot element in the middle, is indeed sorted.

7.3 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` that computes an empty list from `xs`. The statement that one can prove is shown in Figure 24.

```
Definition SpanForall2F(p : A -> bool)(xs : List A) : Prop :=
  Forall (fun a => p a = true) (fromList xs) ->
  span p xs = (fromList xs, getNil xs).
```

Fig. 24. Motive stating if all elements of a list satisfy `p`, then `span` returns the empty suffix, where the latter is computed using `getNil` to avoid noncanonicity problems.

8 FURTHER RELATED WORK

Our work contributes to the program proposed by Owens and Slind, of broadening the class of functional programs that can be accommodated in theorem provers [Owens and Slind 2008]. Our recursion scheme generalizes *nested recursion*, where recursive calls of the form `f (f x)` are allowed [Krauss 2010]. Here, 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 Bove et al. [2016].

Our method is similar to the technique of sized types, in providing a type-based method for termination [Barthe et al. 2004b; Hughes et al. 1996]. With sized types, datatypes are indexed with abstract sizes, which must then be propagated through code, using dependent types. In contrast, our approach relies just on polymorphism, and does not require dependent types.

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 [Uustalu and Vene 2011]. In contrast, our scheme allows arbitrary finite nestings of recursion, and enables abstract application of constructors. We illustrated it in Coq with realistic examples. It seems that generalizing the carriers of algebras to functors is the critical step enabling such examples.

Mendler introduced the idea of using universal abstraction to support compositional termination checking [Mendler 1991]. Previous work explored the categorical perspective on Mendler-style recursion [Uustalu and Vene 1999]. It has also been considered for negative type schemes [Ahn and Sheard 2011]. Previous work on the Cedille proof assistant showed how to derive inductive datatypes using extensions of the Mendler encoding [Firsov et al. 2018; Firsov and Stump 2018]. We do not derive inductive types, but rather a terminating recursion scheme for existing datatypes.

9 CONCLUSION AND FUTURE WORK

We have seen how to implement an interface for divide-and-conquer recursion in Coq, using just the typing of the Calculus of Constructions to enforce termination. We demonstrated our technique on a diverse range of examples, including classic divide-and-conquer algorithms like `mergesort` and `quicksort`, as well as challenge problems like Harper's regular-expression matcher. We motivated our interest in an alternative to well-founded recursion, by a detailed evaluation of Coq's `Function`, `Program`, and `Equations` commands. We sketched also an indexed version of the development, and showed how it can be used to write proofs about our example programs.

We envision future work in two directions. First, there is more to do to automate parts of the approach. For example, lemmas that subsidiary recursions preserve motives are essential to our approach to divide-and-conquer induction. These lemmas closely follow the form of the original program, and so it should be possible to produce them automatically. A second direction is to capitalize on the fact that our approach does not require dependent types, and so is suitable for strong functional programming in the sense of Turner [1995]. The research problem is to design a language providing native support for divide-and-conquer recursion, something which has not been previously achieved.

REFERENCES

Ki Yung Ahn and Tim Sheard. 2011. 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* (Tokyo, Japan) *(ICFP '11)*. ACM, New York, NY, USA, 234–246.

Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. 2006. Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant. In *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3945)*, Masami Hagiya and Philip Wadler (Eds.). Springer, 114–129. https://doi.org/10.1007/11737414_9

Gilles Barthe, Maria João Frade, Eduardo Giménez, Luís Pinto, and Tarmo Uustalu. 2004a. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science* 14, 1 (2004), 97–141. https://doi.org/10.1017/S0960129503004122

Gilles Barthe, Maria João Frade, Eduardo Giménez, Luís Pinto, and Tarmo Uustalu. 2004b. Type-based termination of recursive definitions. *Math. Struct. Comput. Sci.* 14, 1 (2004), 97–141. https://doi.org/10.1017/S0960129503004122

Jean-Philippe Bernardy and Marc Lasson. 2011. Realizability and Parametricity in Pure Type Systems. In *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6604)*, Martin Hofmann (Ed.). Springer, 108–122. https://doi.org/10.1016/j.tcs.2006.12.042

Richard S. Bird and Oege de Moor. 1997. *Algebra of programming*. Prentice Hall.

Frédéric Blanqui. 2005. Inductive types in the Calculus of Algebraic Constructions. *Fundam. Informaticae* 65, 1-2 (2005), 61–86. http://content.iospress.com/articles/fundamenta-informaticae/fi65-1-2-04

Ana Bove, Alexander Krauss, and Matthieu Sozeau. 2016. Partiality and recursion in interactive theorem provers - an overview. *Mathematical Structures in Computer Science* 26, 1 (2016), 38–88. https://doi.org/10.1017/S0960129514000115

Jonathan Chan and William J. Bowman. 2019. Practical Sized Typing for Coq. *CoRR* abs/1912.05601 (2019). arXiv:1912.05601 http://arxiv.org/abs/1912.05601

Robin Cockett and Dwight Spencer. 1992. Strong Categorical Datatypes I. In *International Meeting on Category Theory 1991 (Canadian Mathematical Society Proceedings)*, R. A. G. Seely (Ed.). AMS.

Ernesto Copello, Alvaro Tasistro, and Bruno Bianchi. 2014. Case of (Quite) Painless Dependently Typed Programming: Fully Certified Merge Sort in Agda. In *Programming Languages - 18th Brazilian Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8771)*, Fernando Magno Quintão Pereira (Ed.). Springer, 62–76.

T. Coquand and G. Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2-3 (1988), 95–120.

Thierry Coquand and Christine Paulin. 1988. Inductively defined types. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings (Lecture Notes in Computer Science, Vol. 417)*, Per Martin-Löf and Grigori Mints (Eds.). Springer, 50–66. https://doi.org/10.1007/3-540-52335-9_47

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

Denis Firsov, Richard Blair, and Aaron Stump. 2018. Efficient Mendler-Style Lambda-Encodings in Cedille. In *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 (Lecture Notes in Computer Science, Vol. 10895)*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer, 235–252.

Denis Firsov and Aaron Stump. 2018. Generic derivation of induction for impredicative encodings in Cedille. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 215–227.

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional proof-irrelevance without K. *Proc. ACM Program. Lang.* 3, POPL (2019), 3:1–3:28. https://doi.org/10.1145/3290316

Tatsuya Hagino. 1987. *A Categorical Programming Language*. Ph.D. Dissertation. University of Edinburgh.

Robert Harper. 1999. Proof-directed debugging. *Journal of Functional Programming* 9, 4 (1999), 463–469. https://doi.org/10.1017/S0956796899003378

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 410–423. https://doi.org/10.1145/237721.240882

Joomy Korkut, Maksim Trifunovski, and Daniel Licata. 2016. Intrinsic Verification of a Regular Expression Matcher. Unpublished, available from Licata's web site.

Alexander Krauss. 2010. Partial and Nested Recursive Function Definitions in Higher-order Logic. *J. Autom. Reasoning* 44, 4 (2010), 303–336. https://doi.org/10.1007/s10817-009-9157-2

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Standard library Coq. 2009. Sorting/Mergesort.v.

The Agda development team. 2016. *Agda*. http://wiki.portal.chalmers.se/agda/pmwiki.php Version 2.5.1.

The Coq development team. 2016. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.5.

Conor McBride. 2002. Elimination with a Motive. In *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers (Lecture Notes in Computer Science, Vol. 2277)*, Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack (Eds.). Springer, 197–216.

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

Neil Mitchell. 2021. `Data.List.Extra`. https://hackage.haskell.org/package/extra-1.7.10/docs/Data-List-Extra.html

Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

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

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

Matthieu Sozeau. 2006. Subset Coercions in Coq. In *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4502)*, Thorsten Altenkirch and Conor McBride (Eds.). Springer, 237–252. https://doi.org/10.1007/978-3-540-74464-1_16

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* 64, 5 (2020), 947–999. https://doi.org/10.1007/s10817-019-09540-0

Matthieu Sozeau and Cyprien Mangin. 2019. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.* 3, ICFP (2019), 86:1–86:29. https://doi.org/10.1145/3341690

Aaron Stump, Christopher Jenkins, Stephan Spahn, and Colin McDonald. 2020. Strong Functional Pearl: Harper's Regular-Expression Matcher in Cedille. *Proc. ACM Program. Lang.* 4, ICFP, Article 122 (Aug. 2020), 25 pages. https://doi.org/10.1145/3409004

Wouter Swierstra. 2008. Data Types à La Carte. *J. Funct. Program.* 18, 4 (July 2008), 423–436.

Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. 2012. 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*. IEEE Computer Society, 596–605. https://doi.org/10.1109/LICS.2012.75

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

Tarmo Uustalu and Varmo Vene. 1999. Mendler-style Inductive Types, Categorically. *Nordic J. of Computing* 6, 3 (Sept. 1999), 343–361.

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

Hongwei Xi. 2002. Dependent Types for Program Termination Verification. *Higher-Order and Symbolic Computation* 15, 1 (March 2002), 91–131. https://doi.org/10.1023/A:1019916231463

Qianchuan Ye and Benjamin Delaware. 2022. Scrap your boilerplate definitions in 10 lines of Ltac!. In *The Eighth International Workshop on Coq for Programming Languages*.