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

PEDRO ABREU, Purdue University, USA
BENJAMIN DELAWARE, Purdue University, USA
ALEX HUBERS, The University of Iowa, USA
CHRISTA JENKINS, University of Iowa, USA
J. GARRETT MORRIS, The University of Iowa, USA
AARON STUMP, University of Iowa, USA

This paper proposes a new approach to writing and verifying divide-and-conquer programs in Coq. Extending the rich line of previous work on algebraic approaches to recursion schemes, we present an algebraic approach to divide-and-conquer recursion: recursions are represented as a form of algebra, and from outer recursions, one may initiate inner recursions that can construct data upon which the outer recursions may legally recurse. Termination is enforced entirely by the typing discipline of our recursion schemes. Despite this, our approach requires little from the underlying type system, and can be implemented in System F_ω plus a limited form of positive-recursive types. Our implementation of the method in Coq does not rely on structural recursion or on dependent types. The method is demonstrated on several examples, including mergesort, quicksort, Harper's regular-expression matcher, and others. An indexed version is also derived, implementing a form of divide-and-conquer induction that can be used to reason about functions defined via our method.

CCS Concepts: • **Theory of computation** → Type theory; **Algebraic semantics**; • **Software and its engineering** → *Functional languages*; **Recursion**.

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

ACM Reference Format:
Pedro Abreu, Benjamin Delaware, Alex Hubers, Christa Jenkins, J. Garrett Morris, and Aaron Stump. 2023. A Type-Based Approach to Divide-and-Conquer Recursion in Coq. *Proc. ACM Program. Lang.* 7, POPL, Article 3 (January 2023), 30 pages. https://doi.org/10.1145/3571196

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, to ensure logical soundness. This check is typically based on structural decrease of arguments to recursive calls, which covers many familiar examples from functional programming, including `map`, `filter`, `foldr`, and more. Many terminating programs use nonstructural recursions, however, including classic divide-and-conquer

Authors' addresses: Pedro Abreu, Purdue University, USA, pdacost@purdue.edu; Benjamin Delaware, Purdue University, USA, bendy@purdue.edu; Alex Hubers, The University of Iowa, USA, alexander-hubers@uiowa.edu; Christa Jenkins, University of Iowa, USA, cwjnkins@uiowa.edu; J. Garrett Morris, The University of Iowa, USA, garrett-morris@uiowa.edu; Aaron Stump, University of Iowa, USA, aaron-stump@uiowa.edu.

2475-1421/2023/1-ART3
https://doi.org/10.1145/3571196

algorithms. For example, `mergesort` splits its input list roughly in half, recurses on the halves, and then merges the results. The recursions on the two halves 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. This technique uses dependent types to connect the evidence to the recursive parameter. 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. Our approach relies 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. Somewhat surprisingly, our method does not even require dependent types (in the sense of types depending on terms): System F_ω plus a form of positive-recursive types is sufficient.

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 splitting phase of the algorithm is implemented as a subsidiary recursion, which constructs lists which may then be recursively sorted (by the outer recursion). Subsidiary recursions are provided with functions that are like constructors, except that they have different types from the actual constructors. Those types limit their use in a termination-preserving way.

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 requires adding two postulates to Coq's type theory: functional extensionality (a commonly added axiom), and also impredicative `Set`, which while supported natively by Coq, is somewhat more controversial. We will discuss this point further below (Section 7). 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 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 8). Proofs using this principle are demonstrated for the above examples, including that the sorting algorithms indeed sort.

An artifact with an implementation of these contributions in Coq is publicly available [Abreu et al. 2023].

To conclude this introduction, we would like to emphasize the theoretical contribution of our approach: we show how to derive an advanced recursion scheme suitable for divide-and-conquer recursion, using just F_ω plus a weakened form of positive-recursive types (Section 6.1). As these

features are available in Coq, we use it as our vehicle to present the method. As we shall see, however, our approach falls within the rich line of work on algebraic approaches to recursion schemes, and is thus not limited to our specific realization in Coq.

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: 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 the result is barely recognizable as a form of mergesort.

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. 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 that make writing well-founded recursions easier: `Function` [Barthe et al. 2006], `Program` [Sozeau 2006], and `Equations` [Sozeau and Mangin 2019]. These 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. From such code, they 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"]`. The implementation of `wordsby` 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 `Program`

Figure 2 on the right shows an implementation in Coq of a well-founded version of `wordsBy`, using `Program`. The `{measure (length l)}` annotation in Figure 2 tells `Program` to use the length of the list, ordered by the less-than relation, as a measure function to justify the recursive calls. We omit the obvious ports of `break` and `span` (of Figure 1). We also omit the short proofs of the termination obligations for the two recursive calls. A version using `Function` is essentially identical. By

```
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`. Proofs of obligations not shown.

design, the code accepted by `Program` (and `Function`) is very similar in style to what one would usually write in Coq for structural recursion. For space reasons, we omit a version using `Equations`, which supports a very different style of dependently typed programming, similar to Agda's.

`Function`, `Program`, and `Equations` all generate valid Coq implementations of `wordsBy` using well-founded recursion. As shown in Figure 3, though, these terms are much larger than the original starting codes, which are 8 lines with `Equations`, 11 with `Program` and `Function`. Also, quite a few auxiliary function definitions are introduced. To alleviate this complexity, `Function` and `Equations` both automatically derive reduction lemmas and induction principles, based on the pattern of the recursion. The goal is to hide the generated code completely from the programmer, with higher-level reasoning principles.

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

Fig. 3. The total lines of code (**LoC**) and number of auxiliary functions (**#Aux**) generated for `wordsBy`.

3.3 Performance of Generated Code

Performance of the generated terms *within Coq* can be asymptotically slow, because the machinery of well-founded recursion, which is erased during extraction of those terms to external languages, is not erased within Coq. Doing so would destroy decidability of type checking [Gilbert et al. 2019]. To measure the cost of well-founded recursion within Coq, 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 compute in (length (WORDSBY (Nat.eqb 0) (t2 ++ t1))).
```

This tests repetition of both branches of the split on `p hd`. The benchmarks all evaluate to 1.

We run these benchmarks with three instantiations of `WORDSBY`. The first is the one from Figure 2, which uses length as a measure. The second is the same, except that is uses a custom ordering `smallerList`, of type `list A -> list A -> Prop`, to compare lists structurally without any measure function. The third is the version using our divide-and-conquer recursion, which we present in Section 5.1 below.

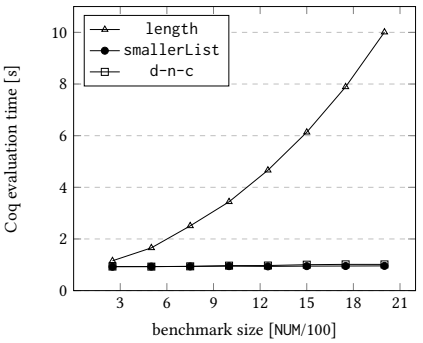

Fig. 4. Comparing three versions of `wordsBy`: `length` uses the length of the list as a measure for well-founded decrease with `Program`; `smallerList` uses `smallerList` as a custom ordering directly on the list, again with `Program`; `d-n-c` uses our divide-and-conquer recursion.

For each version of `wordsBy`, Figure 4 shows the median time of three Coq evaluations for each benchmark. The `length` version is quadratic in `NUM`, while the almost coinciding versions `smallerList` and `d-n-c` are linear (with `d-n-c` just slightly slower). Not shown, `Function` and `Equations` also exhibit quadratic running time. The source of the quadratic behavior is that the proof of well-foundedness of less-than in the Coq standard library evaluates the measure function on the input for each recursive call. Careful rewriting of that proof yields a version that also runs in linear time.

3.4 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; but such such erasure is not possible within Coq [Gilbert et al. 2019]. The standard approach to well-founded recursion using measure functions can be asymptotically slow; this can be corrected using a custom ordering, at the cost of an extra well-foundedness proof, or by carefully rewriting the underlying combinator for well-founded recursion with measures. Automatically generated terms for well-founded recursion are large, requiring further generation of reduction lemmas and custom induction principles to be usable.

The rest of this paper proposes an alternative, which we call divide-and-conquer recursion. Instead of the structural recursion of CIC, 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 typing features like singleton elimination. 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]). 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 a function from inputs of type $F\ X$ to a result of type X. F is called the *signature functor* for the datatype. As an example, the `ListF` type shown on the right is the signature functor for lists, parametrized by the type `A` of elements. This is similar to the standard definition of lists in functional languages, except that the second argument of `Cons` is of type `X` rather than `ListF X`. The `lengthAlg` definition is an example (`ListF A`)-algebra in Coq. This algebra is used to compute the length of a list. Note that the `Cons` case in `lengthAlg` type checks because `xs` has type `nat`. For good introductions to the functorial view of datatypes in functional programming, see Swierstra [2008] and Bird and de Moor [1997].

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

Definition lengthAlg (A : Set)
 (d : ListF A nat) : nat :=
  match d with
    Nil       => 0
  | Cons x xs => 1 + xs
  end.
```

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

In our development, the type `list A` from Coq's standard library is distinct from μ(`ListF A`), which we denote as `List A`. We include conversion functions between these types: `toList` goes from `list` to `List`, and `fromList` does the reverse. To use our approach, one first converts from

`list` to `List`, and then applies our recursion scheme. From a user's perspective, `List A` is just used behind the scenes to do divide-and-conquer recursion on a standard `list A`. Usually our recursions can compute a `list` as their output directly, and no conversion back from `List` is needed.

The algebraic approach to datatypes uses a single constructor called `in`, of type $F\ \mu F \to \mu F$. Specialized to `ListF A`, this function converts a `ListF A (List A)` to a `List A`. In other words, `in` takes a `ListF A` 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 described. Where a (`ListF A`)-algebra with carrier X is given an input of type `ListF A X` and produces a value of type X, our divide-and-conquer algebras are equipped with an additional 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. Section 6 discusses the reason for this difference.

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 → Set`. The carrier must be a functor satisfying the usual functor identity law. We denote this functoriality requirement `Functor X`. Thus, both `SAlg` and `Alg` have kind $(* \Rightarrow *) \Rightarrow *$, where the input of kind $* \Rightarrow *$ is the carrier for the algebra.

Both `Alg` and `SAlg` are recursive types of a special form, called *positive-recursive*: every recursive occurrence in such a type appears in the domain part of an even number of function types [Mendler 1991, Section 2.1]. Section 6.1 details how the fixed-points of these types are built. For now, an important intuition is that both kinds of algebras define their own universe for recursion, whose elements are of an abstract type `R`. Recursive calls are allowed only on data of type `R`.

Interface for subsidiary recursion. A subsidiary algebra defines a recursion that may be called by an outer recursion, which we call the *parent* recursion. The type for subsidiary algebras with carrier `X` is defined as:

$$\begin{aligned}\texttt{SAlgF}\ (\texttt{X} :: * \Rightarrow *) \quad := \quad & \forall \texttt{P} :: *.\ \forall \texttt{R} :: *. \\ & \underbrace{(\texttt{R} \to \texttt{P})}_{\texttt{up}} \to \underbrace{(\texttt{FoldT}\ \ \texttt{SAlg}\ \ \texttt{R})}_{\texttt{sfo}} \to \underbrace{(F\ \texttt{R} \to \texttt{P})}_{\texttt{abstIn}} \to \underbrace{(\texttt{R} \to \texttt{X}\ \ \texttt{R})}_{\texttt{rec}} \to \underbrace{F\ \texttt{R}}_{\texttt{d}} \to \texttt{X}\ \ \texttt{P}\end{aligned}$$

Each of the components of this type provides a piece of the toolbox that allows a subsidiary algebra to interact with its recursion universe `R`, as well as the universe of its parent recursion `P`:

- `P`, the abstract type for the parent recursion.
- `R`, the abstract type for this subsidiary recursion.
- `up` of type `R → 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 $F\ \texttt{R} \to \texttt{P}$, 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`. Thus, applying this abstracted constructor to data in the current recursion universe constructs data in the parent recursion universe.

- `rec` of type $\mathtt{R} \longrightarrow \mathtt{X\ R}$, used to make recursive calls on values of type `R`. Note that the carrier `X` is applied to `R` here. This allows data produced by a recursive call to be eligible for a further recursive call with this algebra.
- `d` of type $F\ \mathtt{R}$. 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`.

Supplied with these inputs, a `SAlgF X` will 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 the results of subsidiary recursions, and what necessitates the generalization of the kind of the carrier to $* \longrightarrow *$.

Definition of `FoldT`. The `sfo` component of the `SAlgF` toolbox has the type `FoldT SAlg R`, where `R` is the abstract type of the recursion. The definition is

$$\mathtt{FoldT\ (A :: KAlg)(R :: *)} \quad := \quad \forall X : * \Rightarrow *.\ \mathtt{Functor\ X} \longrightarrow \mathtt{A\ X} \longrightarrow \mathtt{R} \longrightarrow \mathtt{X\ R}$$

The parameter `A` is the type of algebra to use. The instantiation (`FoldT SAlg R`) unfolds to:

$$\forall X : * \Rightarrow *.\ \mathtt{Functor\ X} \longrightarrow \mathtt{SAlg\ X} \longrightarrow \mathtt{R} \longrightarrow \mathtt{X\ R}$$

Thus, the `sfo : FoldT SAlg R` component of a subsidiary algebra can be used to recurse on any `R` from the current recursion, using a further subsidiary algebra, to obtain a result of type `X R`, for any functorial `X`. Again, the use of `R` in the result type of `FoldT` is critical here: invoking a subsidiary recursion produces data upon which the current recursion may legally recurse.

For the implementation described in Section 6, it will be crucial that `SAlg` is a positive-recursive type. To see this, note that the occurrence of `SAlg` is negative in `SFoldT` above, as it is in the domain part of just one function type. In `SAlgF` itself, however, `FoldT SAlg R` is the type of an input. Thus, in the definition of a subsidiary algebra, the sole occurrence of `SAlg` is in the domain part of two function types. Hence, `SAlg` is positive-recursive.

Interface for outer recursion. The type of algebras for outer recursion is `Alg X`, where again `X` is the carrier of type `KAlg`. These algebras are built from functions that take similar arguments to those of subsidiary algebras, except that there is no parent algebra:

$$\mathtt{AlgF\ (X :: * \Rightarrow *)} \quad := \quad \forall \mathtt{R} :: *.\ \underbrace{\mathtt{(FoldT\ Alg\ R)}}_{\mathtt{fo}} \longrightarrow \underbrace{\mathtt{(FoldT\ SAlg\ R)}}_{\mathtt{sfo}} \longrightarrow \underbrace{\mathtt{(R \longrightarrow X\ R)}}_{\mathtt{rec}} \longrightarrow \underbrace{F\ \mathtt{R}}_{\mathtt{d}} \longrightarrow \mathtt{X\ R}$$

Accordingly, an outer algebra returns a value of type `X R` and there are no `up` or `abstIn` inputs, as these only make sense in the presence of a parent recursion universe. Finally, in order to implement this interface, we add a function `fo` to fold an `Alg`, in addition to `sfo` for folding an `SAlg`. The inclusion of `fo` makes an outer algebra a positive-recursive type, like `SAlg`. To summarize the components required to build an outer algebra:

- `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 $\mathtt{R} \longrightarrow \mathtt{X\ R}$, for making recursive calls.
- `d` of type $F\ \mathtt{R}$, the subdata structure.

Additional definitions. Several functions will be helpful in our upcoming discussion. Their signatures are in Figure 5. Given a functor F, our development provides a type, `Dc`, for data supporting divide-and-conquer recursion using the two algebras discussed above.

The function inDc is used as the (sole) categorical constructor for type Dc. Specializing F to ListF A in inDc allows us to define the constructors mkNil and mkCons for List A, for example. We can then use these constructors to define functions to convert from list A to List A and back (fromList and toList); see Section 6.4 for automatic derivation of boilerplate like this. Our examples will also use the sfold and fold functions to fold outer and subsidiary algebras over values of Dc, respectively. To create algebras and subsidiary algebras from values of AlgF and SAlgF, our development provides the functions rollAlg and rollSAlg. Intuitively, such terms corresponds to a one-step unrolling of Alg and SAlg. The final function we will use below is out, which uses the sfo component of a subsidiary algebra to convert R values into F R values via a trivial subsidiary recursion. Subsidiary algebras can use out to perform nested pattern-matching on a term of type R, while still preserving the possibility of recursing on subdata.

$$
\begin{aligned}
\texttt{inDc} &: F\ \texttt{Dc} \to \texttt{Dc} \\
\texttt{toList} &: \texttt{list A} \to \texttt{List A} \\
\texttt{fromList} &: \texttt{List A} \to \texttt{list A} \\
\texttt{fold} &: \texttt{FoldT Alg Dc} \\
\texttt{sfold} &: \texttt{FoldT SAlg Dc} \\
\texttt{rollSAlg} &: \forall \texttt{X} :: * \to *.\ \texttt{SAlgF SAlg X} \to \texttt{SAlg X} \\
\texttt{unrollSAlg} &: \forall \texttt{X} :: * \to *.\ \texttt{SAlg X} \to \texttt{SAlgF SAlg X} \\
\texttt{rollAlg} &: \forall \texttt{X} :: * \to *.\ \texttt{AlgF Alg X} \to \texttt{Alg X} \\
\texttt{unrollAlg} &: \forall \texttt{X} :: * \to *.; \texttt{Alg X} \to \texttt{AlgF SAlg X} \\
\texttt{out} &: \Lambda \texttt{R} :: *.\ \texttt{FoldT SAlg R} \to \texttt{R} \to \texttt{F R}
\end{aligned}
$$

Fig. 5. Signatures of auxiliary functions used in Section 5.

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

This section presents several example programs that use 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 algorithms, and for comparison with the Coq versions which are the contribution of the section.

5.1 Implementing wordsBy

Our first example is the wordsBy function from Section 3.1. The first step is to implement span (Figure 1) using the SAlg shown in Figure 6. As a subsidiary algebra, this program uses two abstract types R and P corresponding to its recursion universe and that of its parent. In order to allow wordsBy to recurse on the second component of the pair returned by span (via break), we need to ensure that this part of the pair is typed at P. Note that the *only* way to get a value of type P within the body of SpanSAlg is to invoke either up and abstIn. Thus, the second component of all the result values contains an application of one of these components. The calls to abstIn are well-typed because xs is the subdata structure for the algebra, of type ListF A R, which is exactly the input type for abstIn. Similarly, observe that the type of (local variable) span guarantees it can only be applied to subdata

```
Definition SpanF(X : Set) : Set := list A * X.
Definition SpanSAlg (p : A -> bool)
 : SAlg (ListF A) SpanF :=
  rollSAlg (fun (P R : Set)
                (up : R -> P)
                (sfo : FoldT (SAlg (ListF A)) R)
                (abstIn : ListF A R -> P)
                (span : R -> (list A * R))
                (xs : ListF A R) =>
   match xs return SpanF P 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. 6. Subsidiary algebra for span

of xs. Modulo these conversion functions, the body of SpanSAlg is quite similar to the Haskell definition in Figure 1.

As a convenience when invoking SpanSAlg from another recursion, Figure 7 defines helper functions spanr and breakr. The type of the sfo parameter of spanr says that sfo can be used to initiate recursions on values of type R using SAlgs (over functor (ListF A)). The definition of spanr calls sfo to initiate a recursion on xs using SpanSAlg. This helper function hides the argument SpanFunctor, which witnesses that the carrier SpanF of SpanSAlg is indeed a functor.

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

Definition breakr {R : Set}
  (sfo : ListSFoldT A R) p :=
    spanr sfo (fun x => negb (p x)).
```

Fig. 7. Functions based on SpanAlg

Using breakr, we can write the Alg for wordsBy shown in Figure 8. The algebra is given a function sfo of type FoldT SAlg R for initiating subsidiary recursions, which it passes 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 in 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 A. Thus, the carrier of WordsByAlg is the constant functor that always returns list A, Const (list A).

With this algebra in hand, we can define the implementation of wordsBy on the standard type of lists from the Coq prelude, list A. This function first converts the input argument List and then applies WordsByAlg using the fold function from Figure 5. Thus, the List A type and the machinery for divide-and-conquer recursion are completely internal to wordsBy.

```
Definition WordsByAlg(p : A -> bool)
  : Alg (ListF A) (Const (list (list A))) :=
  rollAlg
    (fun (R : Set)
         (fo : FoldT (Alg (ListF A)) R)
         (sfo : FoldT (SAlg (ListF A)) R)
         (wordsBy : R -> (list (list A)))
         (xs : (ListF A) R) =>
       match xs return list (list A) 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).

Definition wordsBy (p : A -> bool)
  (xs : list A) : list (list A) :=
    fold _ _ _ (WordsByAlg p) (toList xs).
```

Fig. 8. Algebra for wordsBy

5.2 Nested Recursion

Nested recursion allows nested calls to a recursive function, as in f (f x) [Krauss 2010]. Our scheme can be seen as generalizing this pattern, so that recursive calls of the form f (g x) are allowed. Here, g could be f, or another recursively defined function, as long as it returns only subdata of the original argument of f. A simple example is the following (total) program from Krstić and Matthews [2003], which recursively reduces its input argument to zero:

```
zeroOut x = if x == 0 then 0 else zeroOut (zeroOut (x-1))
```

We can encode this function via the (subsidiary) algebra in Figure 9, which uses the identity functor as its carrier. Defining zeroSAlg as an SAlg equips it with two key operations: up and abstIn. The base case uses abstIn to inject Zero' into the parent universe, while the recursive

```
Definition zeroSAlg : SAlg NatF Id :=
  rollSAlg (fun P R up sfo abstIn zero n =>
    match n with
    | Zero' => abstIn Zero'
    | Succ' p => up (zero (zero p))
   end).
```

```
Definition ZeroOut (n : Nat) : Nat :=
  sfold NatF _ _ zeroOutAlg n.
```

Fig. 9. Nested-recursive zero function as an SAlg

case uses up to do the same for the results of the nested recursive call. ZeroOut uses sfold to apply zeroSAlg, for the final function.

As an example that is rejected by our typing discipline, consider the following (partial) variant of zeroOut

```
oneOut x = if x == 0 then 1 else oneOut (oneOut (x-1))
```

This function is undefined on all nonzero inputs, as oneOut 1 ≡ oneOut (oneOut 1). When attempting to write a corresponding algebra using our recursion scheme, the offending call to abstIn (Succ' Zero') in the base case is (rightly) rejected, because the argument has the type NatF (NatF R), and *not* the expected type of inputs to abstIn, namely NatF R.

5.3 Higher-Order Recursion

Higher-order recursive functions feature occurrences of the recursive function that are not fully applied. The following is an example, as mirror is used on the right-hand side as an argument to map, instead of being applied to an argument:

```
data Tree a = Node a [Tree a]
mirror (Node a ts) = Node a (reverse (map mirror ts))
```

While mirror can also be defined in Coq using its Fixpoint command, this is only possible because the termination checker is smart enough to unfold the definition of map to check for structural recursion. If we mark map as opaque or switch the order of map and rev, the termination check fails. Our interface, however, is able to also handle both of these scenarios.

```
Definition mirrorAlg : SAlg TreeF Id :=
  rollSAlg (fun P R up sfo abstIn mirr t =>
    match t with
    | NodeF a xs =>
      abstIn (NodeF a (map mirr (rev xs)))
    end).
```

5.4 Combinator-Based Run-Length Encoding

Run-length encoding is a basic data-compression algorithm for lists in which 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 spanr from Section 3.1, a nice demonstration of the benefits of compositionality that our approach enjoys over nonstandard structural recursion.

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

Fig. 10. mapThrough in Haskell

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

```
Context {A : Set}. (* Type of list elements *)
Context eqb : A -> A -> bool. (* Equal As *)

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

Definition RleAlg := MapThroughAlg compressSpan.

Definition rle (xs : list A) : list (nat * A) :=
    fold _ _ _ RleAlg (toList xs).
```

Fig. 12. Run-length encoding in Haskell and using divide-and-conquer recursion in Coq.

The Haskell implementation of the `mapThrough` combinator is given in Figure 10. It is similar to the standard `map` operation 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, we use this type for the function that will be applied to the input list:

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

This is similar to the type `a -> [a] -> (b,[a])` from the Haskell version, except that the occurrences of `[a]` have been replaced with the abstract type `R`. This is a type-based way of expressing the fact that `mapThrough` expects the function it is given to return only sublists of its input list. Accordingly, `mapThrough` will be able to recurse on the second component. In order to manipulate terms of the abstract type `R`, functions of type `mappedT` are also given a function, `sfo`, for initiating subsidiary recursions. This function will be supplied by `mapThrough`.

A Coq implementation of `mapThroughAlg` that uses this definition is shown in Figure 11. The code is similar to the Haskell version shown in Figure 10, though the function `f` being mapped must be also be applied to the abstract type `R` and fold function `sfo`. For simplicity, we choose to define `MapThroughAlg` as an `Alg` rather than an `SAlg`, and to simply return a `list B` (as before, we use a constant functor for the carrier of the algebra). Inside the body of `mapThroughAlg`, we have that `b : B` and `c : R`, so the algebra may indeed make a recursive call by applying `mapThrough : R -> list B` to `c`.

```
Definition MapThroughAlg {B : Set}
    (f : mappedT A B)
  : Alg (ListF 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 :=
    fold _ _ _ (MapThroughAlg f).
```

Fig. 11. The `MapThroughAlg` algebra and the `mapThrough` function.

The Haskell implementation of run-length encoding simply maps a compression function through the input list, as shown on left side of Figure 12. The `compressSpan` helper function gathers up all elements at the start of the tail `tl` that are equal to the head `hd`, using `(== hd)` to test for equality with `hd`. This prefix is returned

as p, with the remaining suffix as s; the function then returns the pair (1 + length p, hd) to summarize hd : p.

The right side of Figure 12 presents an analogous implementation in Coq which uses our divide-and-conquer interface. Our helper function compressSpan has type mappedT A (nat * A), as required by mapThrough. The code for mapThroughAlg invokes compressSpan using the sfo function over the recursion universe of mapThroughAlg. compressSpan applies this function to the tail of the list tl to extract a suffix s upon which mapThrough can recurse further. The algebra RleAlg is then obtained by supplying compressSpan to MapThroughAlg The final implementation of rle is obtained using fold and the toList conversion function.

5.4.1 Comparison with Well-Founded Recursion. It is straightforward to use Program to obtain an implementation of mapThrough that uses well-founded recursion. To see how the intertwining of proof terms in a well-founded recursion impacts combinator programming, consider the implementation of mapThrough shown in Figure 13 (mapThroughWf). The big difference is that this implementation 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.

```
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 defined via Program.

From the 13 line program shown in Figure 13, Program generates 270 lines of code using well-founded recursion. This expansion is worse than we saw for wordsBy in Section 3. 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 can hide these using Coq's abstract tactical, or by manually introducing intermediary lemmas; but the proofs will still remain in the code in some form. Another source of blow-up is packing and unpacking the components of the dependent pair produced by the mapped function.

5.5 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. 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 presents the implementation of Harper's matcher as an Alg. We define K T as the type for a continuation expecting an input of type T, and MatchT as the type for a function expecting such a continuation. The algebra defining the matcher has the functorial carrier MatcherF. There is an inner recursion, matchi, on the regular expression; and an outer recursion, matcherAlg, on

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

Definition matchi(T : Set)(matcher : T -> Regex -> MatchT T)
  : Regex -> Ascii.ascii -> T -> MatchT T :=
  fix matchi (r : Regex) : Ascii.ascii -> T -> MatchT T :=
  match r with
    NoMatch      => fun c cs k => false
  | MatchChar c' => fun c cs k =>
                      if (Ascii.eqb c c') then k cs else false
  | Or r1 r2     => fun c cs k =>
                      (matchi r1 c cs k) || (matchi r2 c cs k)
  | Plus r       => fun c cs k =>
                      matchi r c cs
                        (fun cs => (k cs) || (matcher cs (Plus r) k))
  | Concat r1 r2 => fun c cs k =>
                      matchi r1 c cs
                       (fun cs' => matcher cs' r2 k)
  end.

Definition MatcherF(X : Set) := Regex -> MatchT X.
Definition matcherAlg : Alg (ListF Char) MatcherF :=
  rollAlg
    (fun R fo sfo matcher s =>
       match s with
         Nil       => fun r k => false
       | Cons c cs => fun r k =>
                       matchi R matcher r c cs k
       end).
```

Fig. 14. Harper's matcher as an `Alg`, in our divide-and-conquer recursion scheme.

the string. The inner recursion `matchi` recurses through the given regular expression, modifying the continuation `k` as it goes. This continuation is invoked on the suffix of the string to be matched, after a prefix is found satisfying the regular expression. Interestingly, `matchi` is a simple structural recursion on the regular expression (i.e., not a subsidiary recursion).

Our approach is needed in defining the outer recursion, to allow the `matcher` local variable of `matcherAlg` to be passed as an argument to `matchi`, in the `Cons` case of `matcherAlg`. Furthermore, this `matcher` can be invoked with an input of type `R`, to produce a result of type `K R -> bool`. In `matchi`, this function is pulled into continuations when recursing into regular expressions, something that is not allowed with structural recursion. It is worth noting that actually, as challenging as this example has been considered to be in previous works, it does not need the full power of our approach: the inner recursion does not need the ability to construct new data upon which the outer recursion will recurse, and indeed can just be a simple structural recursion.

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 the inner and outer recursions. For this merged code, `Program` is used to implement lexicographic decrease of the pair of the regular expression and the string. In contrast, our approach is able to keep the functions separate, and for `matchi` to use structural recursion on the regular expression. The outer recursion `matcherAlg` does need to use our scheme, in order to support embedding recursive calls to `matcher` into continuations

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

Definition mergeSortH (xs : List A) :=
  fold _ _ _ MergeSortAlg xs.

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

```
Definition SplitF (X : Set) : Set := X * X.

Definition SplitAlg
 : SAlg (ListF 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).

Definition splitr {R : Set}
  (sfo: ListSFoldT A R) : R -> SplitF R :=
  sfo SplitF FunSplitF SplitAlg.
```

Fig. 15. An implementation of `mergesort` and its subsidiary `split` function.

inside `matchi`. Once again, we do not need to use dependent types or any other technique to prove that strings are smaller when `matcher` is invoked, relying instead on the typing of CC to enforce this. This keeps the types simpler and eliminates explicit reasoning about termination.

5.6 Mergesort

Figure 15 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 this list is nonempty, it peels off its head and passes it as an argument to `mergeSortH`, which starts the initial outer recursion. This function first evenly divides the input list into two lists, which are each recursively processed. This continues until each recursive branch has only the additional element that was passed in. It then constructs a singleton from that argument, and the resulting lists are merged on their way up, as expected. Since `mergeSortH` has return type `list A`, we are able to reuse the implementation of `merge` in Coq's standard library to combine the results of the recursive calls.

Because we recurse on values obtained from splitting the input list, we write `split` as the subsidiary algebra `SplitAlg` that appears on the right of Figure 15. Since the parent recursion wishes to recurse on both the outputs of this algebra, `SplitAlg` has the carrier `SplitF`. It thus returns a value of type `P * P`. When splitting a list, we try to match two levels down into it, in order to find the first two elements (if there are that many). Note that we cannot do this nested pattern matching directly, however, as the type of the input list `xs` is `ListF A R`. Thankfully, we are able to deploy the `out` helper function described in Section 4 to convert the tail of the list from type `R` to type `ListF A R`, on which we can then perform the second pattern match. The two elements are then used to construct the left and right components of the result pair, respectively, via an application of `abstIn`, which injects both lists back into `P`, the parent recursion universe.

```
Context A : Set. (*Type of list elements *)
Context ltA : A -> A -> bool. (* Ordering *)

Definition PartF (X : Set) : Set :=
  A -> (X * X).

Definition PartSAlg : SAlg (ListF A) PartF :=
  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).
```

```
Definition partr {R : Set}
  (sfo : ListSFoldT A R)
  : R -> A -> R * R :=
    sfo PartF FunPartF PartSAlg.

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

Fig. 16. Algebras for quicksort and partitioning a list based on a bound.

What would go wrong if one attempted to recurse improperly inside `MergeSortAlg`? Our typing discipline ensures that the results of `splitalg` are safe to recurse on, as the type of `abstIn` ensures the results of `SplitAlg` stay inside the recursion universe of `MergeSortAlg`. So we would get a type error if we tried break out of this universe; e.g., by returning `Cons hd (Cons hd ys)` for the first component. This is like the hypothetical `oneOutAlg` from Section 5.2. Alternatively, suppose one tried to call `mergesort mkNil A` in the `Nil` case of `MergeSortAlg`, thus causing the algebra to diverge. This is also prevented by our typing discipline, since the typing of the local variable `mergesort` provided by our interface ensures it can only be applied to inputs of type `R`, and `mkNil A` does not have this type.

5.7 Quicksort

For our final example, we show (functional) quicksort, to see how our approach deals with quicksort's partitioning of the input list into a one list with elements less than some bound, and another with elements greater than or equal to the bound. The algebra for this is shown on the left of Figure 16. As with `SplitAlg` above, `PartitionAlg` uses `abstIn` to construct data of the abstract type `P` of the parent recursion from a value of `ListF A R`. In contrast to `SplitAlg`, however, it only does so for one component of the result pair, depending on the result of the bound comparison. The algebra uses `up` to inject the other, unchanged result of the recursive call into the parent universe. Besides these calls to `abstIn` and `up`, the code is exactly as expected. `QuicksortAlg` then has the desired form: partition, recurse, and combine the results using list concatenation.

5.8 Discussion

We have seen a diverse group of example programs written using our interface, all of which make use of nonstructural recursion. These examples cover a range of interesting recursion patterns from the literature [Bove et al. 2016], including nested recursion, higher-order recursion, and recursive calls in a continuation. Our approach provides functions to enable subsidiary recursions to propagate values, even newly constructed ones, to an outer recursion, which may then recurse on them. This is all without any use of dependent types in the definition of clients, as termination is guaranteed

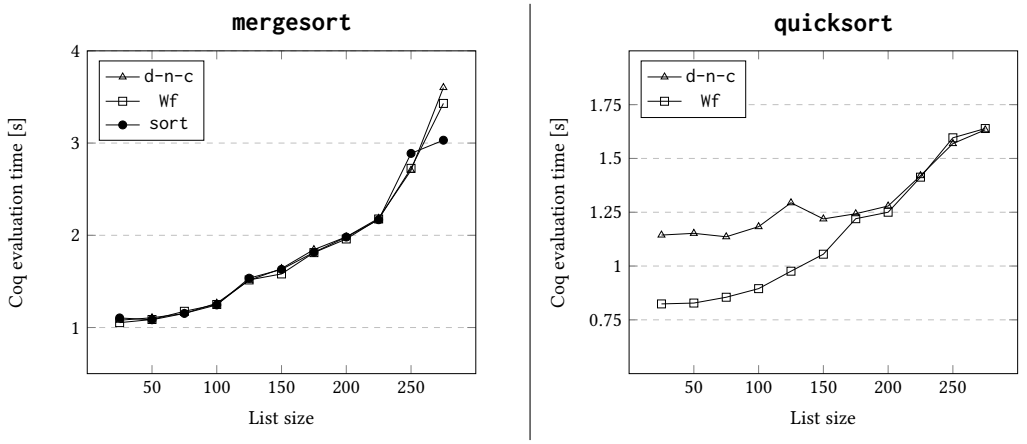

Fig. 17. Comparing implementations of mergesort and quicksort. `d-n-c` uses our divide-and-conquer recursion; `Program` uses program; and `sort` is the implementation from Coq's standard library.

by the typing rules of CC itself. These examples can also be written using well-founded recursion, by placing proofs certifying arguments decrease at each recursive call site.for the mirror example, As we saw in Section 5.4.1, such proofs *are* inserted by `Function`, `Program`, and `Equations` into the definitions of the functions. Of course, our approach also introduces some syntactic overhead in the resulting programs, as subsidiary algebras must call `abstIn` and `up` functions to communicate data up to outer recursion in a way that preserves the ability to recurse. Outside of these conversion functions, however, the resulting programs look quite similar to their expected implementations. The performance of functions written using our interface appears compatible with the expected asymptotic complexity of $O(n \log n)$, and is comparable to that of versions written using well-founded recursion. Figure 17 presents times for `mergesort` and `quicksort` on pseudo-randomly generated lists, using the two approaches. As recursive calls are made on lists of exponentially decreasing size, the asymptotically suboptimal performance for well-founded recursion that we noted in Section 4 above is not observable.

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 of Matthes

Coquand and Paulin [1988] proved that full positive-recursive inductive types are incompatible with Coq's type theory, although they can be recovered by some subtle changes to the type theory [Blanqui 2005]. Here, we instead make use of an idea of Matthes, which gives a weakened, but still sufficient, form of positive-recursive types, without requiring any changes to the underlying type theory [Matthes 2009]. We summarize Matthes's approach here. He did not introduce a name for his weakened form of positive-recursive type; we propose *retractive-positive* recursive types, because, as we will see, they make the unfolding of the recursive type `Mu` a retraction of `Mu`. Note

that the point of using a recursive type instead of another impredicative encoding is to ensure that unfolding the recursive type can be done in constant time.

We start with 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'.
```

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 18 defines `Mu` as a strictly positive approximation to this ideal `Mu'`, in the manner proposed by Matthes. 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, as we instantiate `R` with `Mu` itself in the definition of `inMu`. `Mu` is a legal inductive type in Coq, because `Mu` occurs strictly positively in the type for `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. 18. Derivation of retractive-positive recursive types

Figure 18 lists functions `inMu` and `outMu`, and a theorem `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 A`, `fmap f (NilF A)` equals `NilF B` for any `f : A -> B`. So if we apply `outMu` from Figure 18 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 one pays with Matthes's technique. We will see below (Section 8.3) 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. In fact, this is what Matthes used, for studying nested datatypes. This gives us 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 from Figure 5. 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).
- 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 19, 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.

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

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 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 19). 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

```
Theorem ComputationLaw :
  forall (X : Set -> Set) (FunX : Functor X)(alg : Alg X)(d : F Dc),
    fold X FunX alg (inDc d) =
    unrollAlg alg Dc fold sfold (fold X FunX alg) d.
```

Fig. 20. Computation law, showing how an elimination (`fold`) of an introduction (`inDc d`) computes.

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 what we call *motive-preservation lemmas*, which we will see (Section 8) play a crucial role in reasoning about subsidiary recursions.

Finally, algebraic approaches to recursion usually include a computation law, expressing how folding an algebra over a constructed value computes. With the above definitions, we can derive the law shown in Figure 20, which says that folding an algebra over `inDc d` invokes the algebra with the various expected values for the operations in the recursion universe, and `d` for the subdata structure.

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 framework [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 ASSUMPTIONS AND LIMITATIONS

Before moving on to a discussion of how to reason about functions defined using this approach, we reflect on the requirements our divide-and-conquer interface and implementation place on the underlying type system. While the interfaces of both regular and subsidiary algebras provide several components, the most advanced feature needed is the sort of type polymorphism found in most functional languages. The *implementation* of our algebras, on the other hand, is more demanding. Our implementation requires a fixpoint operator on functors that cannot be defined directly in languages that enforce strict-positivity restrictions on inductive datatype definitions. Happily, we may apply Matthes's approximation of this operator to work around this issue [Matthes 2009].

Impredicative Set. Less happily, this form of impredicative typing is not in line with many modern proof assistants, including Lean and Agda, which only support predicative computationally relevant universes (Lean, of course, has an impredicative computationally irrelevant universe at the bottom of its type hierarchy). This design decision, which Coq itself adopted after version 8.0, was driven by a desire for compatibility with classical axioms, in particular excluded-middle and the axiom of description. Thus, our Coq implementation relies on the `impredicative-set` option, which restores this former functionality. While not standard, this choice is in line with other recent Coq developments, including mechanizations of modular metatheory [Delaware et al. 2013a,b], the coq-of-ocaml project [Claret 2021], a formalization of monad transformers [Affeldt and Nowak 2021], the development of a model of relational parametricity for System F_ω by Atkey [2012], and Matthes' aforementioned work [Matthes 2009].

Thankfully, this option is not completely incompatible with classical logic. Only classical logic stated in impredicative set is inconsistent with `impredicative-set`, so it is possible to use the 'standard' law of excluded middle for propositions: $\forall A : Prop, A \vee \neg A$. While care must still be taken when combining axioms of choice with LEM or predicate extensionality, many classical developments are compatible with `impredicative-set`.

Applicability. Recursive functions which do not recurse on subdata of the input argument are not directly supported by our interface. A famous artificial example is McCarthy's 91 function:

```
f91 n = if n > 100 then n - 10 else f91 (f91 (n + 11))
```

Since the first recursive call is on a larger input, this function falls outside the scope of our approach.

On the other hand, we can handle functions using nonstructural lexicographic recursion. An example is the Euclidean algorithm for greatest common divisor. The Haskell implementation is shown in Figure 21; we elide the port of this to our combinators, to focus just on the form of the recursion. The usual approach to lexicographic recursion applies: split a two-argument lexicographic recursion into two nested one-argument recursions. The `gcd'` function is the outer recursion, recursing on the first argument; `inner` recurses on the second. The `gcd'` function passes itself to `inner`, thus allowing us to separate these nested recursions: usually one would write `inner` as a syntactic subterm of `gcd'` (the outer recursion). Our version in Coq is to separate these functions in the same way, thanks to the compositionality our approach inherits from typing.

```
inner gcd x O = (S x)
inner gcd x (S y) =
 if (lte x y) then
   inner gcd x (sub y x)
 else
   gcd (sub x y) (S y)

gcd' O y = y
gcd' (S x) y = inner gcd' x y
```

Fig. 21. An implementation of gcd in Haskell, decomposed into two recursions

We also note that some advanced encodings of recursive functions that use dependent types to guarantee totality can also easily represent *partial* functions, which our interfaces do not directly support. See Section 9 for a more detailed comparison.

8 DIVIDE-AND-CONQUER INDUCTION, WITH EXAMPLES

In this section, we consider an induction principle for reasoning about divide-and-conquer recursions. It turns out that a nice way to derive this is as an indexed version of the recursion principle, parametrized by index type `I : Set` (cf. Bernardy and Lasson [2011]). Here, `I` will be instantiated to `Dc` (the type for divide-and-conquer recursions). 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`, to support induction. A version retaining `Set` is also

possible, for indexed programming (e.g., with length-indexed vectors). 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(A : Set)(R : lkMo) : lkMo :=
  nilFi : ListFi A R mkNil
| consFi : forall (h : A)(t : List A), R t -> ListFi A R (mkCons h t).
```

This looks just like the nonindexed `ListF A` 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 A (toList xs)
```

This converts a list `xs` from Coq's standard library into evidence that it is legal to prove properties about the `List A` 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.
```

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 : I`, 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`.

8.1 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 (modulo 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:

```
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

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 divide-and-conquer 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). 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.

8.2 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.

8.3 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 Lists. For example, suppose we wish to prove that if all elements of a list satisfy p, then the suffix returned by span is empty. It is dangerous to phrase this as "the suffix equals mkNil", because for a noncanonical input xs, span will return that same noncanonical xs as the suffix (and so it may be a noncanonical empty list, not

```
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.

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.

8.4 A Comparison with Well-Founded Induction

In this section, we compare two proofs about mergesort. The first is the one already introduced above (Section 8.2), using our approach. The second is about a version `mergeSortWf` whose central helper function `msHWf` is written using `Function`. The code is shown in Figure 25. The proof that this helper function produces sorted output uses an induction principle generated by `Function`, to mirror the pattern of `msHWf`'s recursive calls. The induction principle is shown in Figure 25, and states that to prove a property `P` of a list `l`, element `a`, and output `msHWf l a`, it suffices to prove a base case for empty `l`, and a step case for nonempty `l`, where one may assume the property holds of the results of recursive calls on the components returned by `split`. This is a very convenient principle to have proven automatically, and makes the final proof just 13 lines long, invoking a lemma from the Coq standard library that merging sorted lists results in a sorted list. The proof of `msHWf_ind` that `Function` generates, however, runs to a somewhat hefty 100 lines.

The proof using divide-and-conquer induction also applies that lemma about merging sorted lists, and is just a little longer, weighing in at 16 lines. Several of these lines perform some extra control of conversion, to pass from explicit applications of `fold` and `sfold` to the functions like `mergesortH` and such that are named in the theorem. Instead of an automatically derived induction principle, the proof makes use of divide-and-conquer induction, which allows us to invoke the induction hypothesis on the results of `split`, as long as we can prove that `split` preserves abstract motives. This property, shown in Figure 23, was discussed already. Its proof is 29 lines long. Because it exactly follows the structure of `split`, and indeed can be seen as nothing more than lifting split from simple to indexed typing, we anticipate it could be generated automatically, as an extension to the library we discussed in Section 6.4 above.

So to compare, the well-founded proof and divide-and-conquer proof are of similar lengths, though there is some extra work in the latter to convert from explicit applications of `fold` and `sfold`. The well-founded proof rests on an automatically generated induction principle whose proof is just under 100 lines long, while the divide-and-conquer proof relies on a lemma showing that `split` preserves abstract motives, whose proof is 29 lines long. The divide-and-conquer proof uses less familiar machinery, of course, and so qualitatively may be argued to be somewhat more difficult than the well-founded version. Nevertheless, the methods seem comparable, thus providing evidence that the divide-and-conquer approach could be developed as a viable alternative to the standard well-founded approach.

9 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]. Bove et al. [2016] present a detailed survey of partiality and recursion in theorem provers. Several works in this direction have also considered how to accomodate partial functions, in addition to the non-structurally recursive total functions handled by our solution. One solution is to use

```
Function msHWf (l:list A)(a : A)
  {measure length l} : list A :=
   match l with
   | [ ]       => [a]
   | (x :: xs) =>
     let ret := split xs in
      merge (msHWf (fst ret) a)
            (msHWf (snd ret) x)
    end.
...
Qed.

Definition mergeSortWf
  (l : list A) : list A :=
  match l with
  | [ ] => []
  | hd :: tl => msHWf tl hd
  end.
```

```
msHWf_ind
 : forall P : list A -> A -> list A -> Prop,
   (forall (l : list A) (a : A),
            l = [] -> P [] a [a]) ->
   (forall (l : list A) (a x : A)
           (xs : list A),
      l = x :: xs ->
      let ret := split xs in
        P (fst ret) a (msHWf (fst ret) a) ->
        P (snd ret) x (msHWf (snd ret) x) ->
        P (x :: xs) a (merge
                         (msHWf (fst ret) a)
                         (msHWf (snd ret) x))) ->
   forall (l : list A) (a : A),
      P l a (msHWf l a)
```

Fig. 25. Well-founded version of mergesort and its induction principle, written using `Function`.

dependent types to restrict the domains of the function inputs to precisely those on which the function is defined, thus converting a partial function into a total one. As one example, Bove and Capretta [2005] present a technique to automatically derive an inductive predicate characterizing the domain of a recursive function. They then automatically construct a total version of the function that recurses over such a proof. Totality is expressed as the property that such a witness can be constructed for all inputs.

Charguéraud [2010] propose an alternative approach in the form of a general fixed point combinator for Coq which supports partial functions in addition to higher-order and nested recursion/corecursion. This approach does not require dependent types, but the combinator relies on Hilbert's epsilon operator and propositional extensionality, axioms that must be added to the underlying type theory and together make the underlying logic classical [Bell 1993]. Furthermore, to guarantee termination and productivity, users of this library must prove a contraction condition [Matthews 1999], a generalization of the accessibility predicate of well-founded recursion. An interesting point of comparison with our work is their implementation of Harper's continuation-based regular-expression matcher [Charguéraud 2021]. Formulated as a partial function, their proof that it is well-defined for normal regular expressions consists of ~230 LoC (including comments and some redundancy which is shown to be amenable to refactoring). In contrast, our implementation is well-defined on the same domain *by construction*.

Similar results to Charguéraud [2010] have been obtained for Isabelle/HOL, using different foundations. An impressively rich set of nested recursive and corecursive functions, including nonuniformly indexed ones, are derived definitionally from the unmodified axioms of Isabelle/HOL [Biendarra et al. 2017]. Breitner et al. [2021] axiomatize an approach to modeling partial Haskell functions in Coq so that they can be extrinsically proved terminating on their domain; they observe that this result is an instance of the recursive combinator of Charguéraud [2010].

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 [2011] developed a categorical view of a recursion scheme allowing one level of subsidiary recursion, and illustrated it in Haskell with an artificial example. 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.

10 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful criticism, which greatly improved this paper. Thanks also to Denis Firsov for early discussions on the idea of divide-and-conquer recursion based on λ-encodings. We thank Robert Dickerson and Qianchuan Ye for their assistance in preparing the accompanying artifact, and Eric Bond for his feedback on the final version of this paper. This material is based upon work partially supported by the Purdue Graduate School under a Summer Research Grant.

REFERENCES

Pedro Abreu, Benjamin Delaware, Alex Hubers, Christa Jenkins, J. Garrett Morris, and Aaron Stump. 2023. A Type-Based Approach to Divide-And-Conquer Recursion in Coq: POPL'23 Artifact. Zenodo. https://doi.org/10.5281/zenodo.7305612

Reynald Affeldt and David Nowak. 2021. Extending Equational Monadic Reasoning with Monad Transformers. In *26th International Conference on Types for Proofs and Programs (TYPES 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 188)*, Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:21. https://doi.org/10.4230/LIPIcs.TYPES.2020.2

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.

Robert Atkey. 2012. Relational Parametricity for Higher Kinds. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs, Vol. 16)*,

Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 46–61. https://doi.org/10.4230/LIPIcs.CSL.2012.46

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

J. L. Bell. 1993. Hilbert's epsilon-Operator and Classical Logic. *Journal of Philosophical Logic* 22, 1 (Feb 01 1993), 1. http://login.proxy.lib.uiowa.edu/login?url=https://www.proquest.com/scholarly-journals/hilberts-epsilon-operator-classical-logic/docview/1292914019/se-2 Last updated - 2013-02-22.

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

Julian Biendarra, Jasmin Christian Blanchette, Aymeric Bouzy, Martin Desharnais, Mathias Fleury, Johannes Hölzl, Ondrej Kuncar, Andreas Lochbihler, Fabian Meier, Lorenz Panny, Andrei Popescu, Christian Sternagel, René Thiemann, and Dmitriy Traytel. 2017. Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic. In *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10483)*, Clare Dixon and Marcelo Finger (Eds.). Springer, 3–21. https://doi.org/10.1007/978-3-319-66167-4_1

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 and Venanzio Capretta. 2005. Modelling general recursion in type theory. *Mathematical Structures in Computer Science* 15, 4 (2005), 671–708. https://doi.org/10.1017/S0960129505004822

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

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, Joshua M. Cohen, and Stephanie Weirich. 2021. Ready, Set, Verify! Applying hs-to-coq to real-world Haskell code. *J. Funct. Program.* 31 (2021), e5. https://doi.org/10.1017/S0956796820000283

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

Arthur Charguéraud. 2010. The Optimal Fixed Point Combinator. In *Interactive Theorem Proving*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 195–210.

Arthur Charguéraud. 2021. *The TLC Coq Library*. https://github.com/charguer/tlc

Guillaume Claret. 2021. Coq of Ocaml. https://github.com/clarus/coq-of-ocaml. Accessed: 2021-09-09.

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

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013a. Meta-Theory à La Carte. *SIGPLAN Not.* 48, 1 (jan 2013), 207–218. https://doi.org/10.1145/2480359.2429094

Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. 2013b. Modular Monadic Meta-Theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 319–330. https://doi.org/10.1145/2500365.2500587

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

Sava Krstić and John Matthews. 2003. Inductive Invariants for Nested Recursion. In *Theorem Proving in Higher Order Logics*, David Basin and Burkhart Wolff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 253–269.

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.

Ralph Matthes. 2009. An induction principle for nested datatypes in intensional type theory. *J. Funct. Program.* 19, 3-4 (2009), 439–468. https://doi.org/10.1017/S095679680900731X

John Matthews. 1999. Recursive Function Definition over Coinductive Types. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1690)*, Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry (Eds.). Springer, 73–90. https://doi.org/10.1007/3-540-48256-3_6

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*.

Received 2022-07-07; accepted 2022-11-07