Strong Functional Pearl: Harper’s Regular-Expression Matcher in Cedille

AARON STUMP, University of Iowa, United States
CHRISTOPHER JENKINS, University of Iowa, United States of America
STEPHAN SPAHN, University of Iowa, United States of America
COLIN MCDONALD, University of Notre Dame, United States

This paper describes an implementation of Harper’s continuation-based regular-expression matcher as a strong functional program in Cedille; i.e., Cedille statically confirms termination of the program on all inputs. The approach uses neither dependent types nor termination proofs. Instead, a particular interface dubbed a recursion universe is provided by Cedille, and the language ensures that all programs written against this interface terminate. Standard polymorphic typing is all that is needed to check the code against the interface. This answers a challenge posed by Bove, Krauss, and Sozeau.

CCS Concepts: · Software and its engineering → Functional languages; Recursion; · Theory of computation → Regular languages.

Additional Key Words and Phrases: strong functional programming, regular-expression matcher, programming with continuations, recursion schemes

ACM Reference Format:

1 INTRODUCTION

Constructive type theory requires termination of programs intended as proofs under the Curry-Howard isomorphism [Sørensen and Urzyczyn 2006]. The language of types is enriched beyond that of traditional pure functional programming (FP) to include dependent types. These allow the expression of program properties as types, and proofs of such properties as dependently typed programs.

Pure FP seems to creep closer year by year to type theory, with the recent advent of dependent Haskell [Weirich et al. 2019, 2017], liquid types in Haskell [Vazou et al. 2014], hasochism [Lindley and McBride 2013], refinement and dependent types in Racket [Kent 2017], and more. A conceptual midpoint between type theory and pure FP—one not generally pursued in the works just cited—is strong functional programming as proposed by Turner [Turner 1995]. This programming discipline requires termination of all programs, but does not add new constructs (like dependent types) to the type system. Strong FP as originally proposed uses syntactic checks on recursively defined...
functions to ensure termination. It thus falls afoul of the same problems of noncompositionality (e.g., abstracting out a subexpression of a recursive function can easily lead to a violation of the syntactic check) that have been a technical challenge in type theory—one proposed solution to which has been type-based termination using sized types [Barthe et al. 2004].

In this functional pearl, we seek to advance the theory and practice of strong FP by using the type-based termination checking of the Cedille language to implement an interesting challenge program, namely Harper’s continuation-based regular-expression matcher [Harper 1999]. This pure functional program uses a continuation to check a suffix of the input string after consuming a prefix. Termination for this program is quite subtle, and has been considered in a number of works [Bove et al. 2016; Owens and Slind 2008; Xi 2002]. These are discussed more in Section 3 below. The most recent of these—a survey paper on partiality and recursion in interactive theorem provers (ITPs) by Bove, Krauss, and Sozeau—briefly sketches a solution in Coq using dependent types for the continuation to express decrease in the length of the suffix string, and notes that “there seems to be no easy way to achieve the same effect [i.e., expressing the critical termination condition] without the use of dependent types”. We are happy to report the perhaps surprising counter-result: our solution does not use dependent types, but rather a certain polymorphically typed interface we call a recursion universe, to control recursion. So while our approach is type-based, we emphasize that it does not use dependent, or size-indexed, or any other extension of basic polymorphic typing.

The use of a success continuation in Harper’s matcher is its distinctive characteristic. It also hides the fact that the function is actually nested recursive: the suffix produced by one recursive call is then fed into the next to search for the next matching prefix. Nested recursion is a challenge for termination [Krauss 2010], which we overcome here using Cedille’s type-based approach to termination. The contributions of this paper are:

- to present the concept of a recursion universe, as a typed interface enabling terminating recursion;
- a recursion universe for Mendler-style recursion, which is a more idiomatic form of terminating recursion, together with extensions for paramorphic and nested recursion;
- a tutorial on the realization of these ideas in Cedille, with comparison to a combinator-based approach in Haskell; and chiefly,
- using Cedille’s recursion universe, an implementation of Harper’s matcher as a strong functional program.

The paper is organized as follows. We begin with a version of Harper’s matcher in Haskell to orient readers before considering versions in Cedille. Section 3 discusses related work on termination of Harper’s matcher. Section 4 gives an extended tutorial introduction to Cedille’s interface (which we call a “recursion universe”) for writing recursive functions. Section 5 presents the implementation of Harper’s matcher in Cedille, using the recursion universe. Section 6 briefly describes Cedille’s termination guarantees. Finally, further related work is surveyed in Section 7.

## 2 HARPER’S MATCHER IN HASKELL

To set the stage for the rest of the paper, we recall Harper’s matcher in the form of a Haskell program, shown in Figure 1. We follow the approach of Korkut et al. [2016] in working with syntactically standard regular expressions, defined first in Figure 1. The intended semantics is that NoMatch does not match any string, MatchChar c matches the string consisting just of c, Or r1 r2 matches a string if either r1 or r2 does, Plus r matches a string if one or more concatenations of r does, and Concat r1 r2 matches a string if a prefix matches r1 and the corresponding suffix matches r2. Recall that String is defined as [Char] (list of characters) in Haskell.
module Matcher where

data StdReg = NoMatch | MatchChar Char | Or StdReg StdReg | Plus StdReg | Concat StdReg StdReg

type K = (String -> Bool)
type MatchT = K -> Bool

matchi :: StdReg -> Char -> String -> MatchT
matchi NoMatch c cs k = False
matchi (MatchChar c') c cs k = if c == c' then k cs else False
matchi (Or r1 r2) c cs k = matchi r1 c cs k || matchi r2 c cs k
matchi (Plus r) c cs k = matchi r c cs (\ cs -> k cs || matchh cs (Plus r) k)
matchi (Concat r1 r2) c cs k = matchi r1 c cs (\ cs -> matchh cs r2 k)

matchh :: String -> StdReg -> MatchT
matchh [] r k = False
matchh (c : cs) r k = matchi r c cs k

match :: String -> StdReg -> Bool
match s r = matchh s r null

Fig. 1. A version of Harper’s matcher, in Haskell

Harper’s paper works instead with general regular expressions including Kleene star and a pattern for the empty string, but imposes a semantic requirement that star cannot be applied to a regular expression r when the language of r contains the empty string. The syntactically standard version is simpler to work with, and general regular expressions can be put in syntactically standard form using a simple algorithm. We review this in Cedille code in Section 5.4. The restriction to standard regular expressions is to ensure that in the case of matchi for Plus, where a call is made to matchh with the same Plus-expression, the string at least has decreased, since the call to matchi in that case must consume at least one character; the regular expression r cannot match the empty string by definition of standard.

The code of Figure 1 has main entry point match, and helper functions matchh and matchi. The latter two are mutually recursive, and both accept a continuation of type K. This structuring into two mutually recursive functions will be useful for comparing with the Cedille version below. Intuitively, matchi recurses through the regular expression, and when it is time to attempt to proceed deeper into the string, it calls matchh. So recursive calls from matchh through matchi and back to matchh are guaranteed to decrease the length of the string. The continuation provides an elegant mechanism for implicitly maintaining the stack of regular expressions against which matchi will test the corresponding suffix of the current string after matching some prefix.

Just like the more general problem of membership for context-free grammars, the membership problem for regular expressions can be solved in polynomial time using dynamic programming (see [Holzer and Kutrib 2010] for a survey of related results). We note in passing that Harper’s matcher does not incorporate dynamic programming, and can take exponential time. An example that can be observed with the above Haskell code is checking strings of the form $a^n b$ against the regular expression Plus (Or (MatchChar 'a') (MatchChar 'a')). It takes time exponential in $n$.
for the matcher to determine failure, as it explores all branches of the exponential-size disjunctive
tree arising from the combination of Or and Plus.

3 PREVIOUS WORK ON HARPER’S MATCHER

After Harper’s original paper presenting the matcher and its termination proof [Harper 1999], there
have been a number of further works considering it. Xi [2002] proves termination of a version of
Harper’s matcher as an example of formalized termination reasoning in an extension of Dependent
ML. Xi’s version uses an explicit check in the continuation for the case of Kleene star, to make sure
that the length of the string has been decreased before recursing. The continuation is dependently
typed, to require that the string given to the continuation is smaller than the string given to
the matching function. Termination is established by automatically solving the size constraints
induced by recursions, thus requiring additional machinery beyond just basic type inference. (In
comparison, the approach of this paper does not require solving size constraints.) Xi additionally
claims verification of termination in his language for Harper’s matcher (without the length check),
though no details are given.

Independently, Danvy and Nielsen [2001] present a continuation-based matcher very similar to
Xi’s, including the length check in the case of Kleene star. They compare this with a defunctionalized
version using a stack of regular expressions for the remainder of the string.

Owens and Slind [2008] consider Harper’s matcher as an example for expressing functional
programs with difficult termination proofs, in the HOL-4 theorem prover. They verify termination
and semantic correctness of a first-order version of the matcher, without continuations. They
note that they can also verify termination for a slight variant of Xi’s length-testing version of the
higher-order algorithm. Verification of Harper’s matcher is not claimed.

Korkut et al. [2016] give a detailed description of full verification of Harper’s matcher in Agda,
as well as a defunctionalized version. They prove termination intrinsically, as part of the definition
of the matcher, using dependent types and well-founded recursion. Their matcher is typed as

\[
\text{match : } (C : \text{Set}) (r : \text{StdRegExp}) (s : \text{List Char}) \\
\rightarrow (k : \forall \{p \ s’\} \rightarrow p \ ++ \ s’ \equiv s \rightarrow p \ \in L^s \ r \rightarrow \text{Maybe } C) \\
\rightarrow \text{RecursionPermission } s \\
\rightarrow \text{Maybe } C
\]

The argument of type RecursionPermission s enables (in a standard way) well-founded recursion
on the length of the input string s. The type of the continuation expresses that the continuation
may be called with a prefix p and suffix s’ of the input string s, and that p really is in the language
(\in L) of the regular expression r that it matched against.

Bove et al. [2016] consider Harper’s matcher as a challenge problem for termination in interactive
theorem proving. They briefly sketch a solution in Coq, which again uses a dependent type for the
continuation, imposing different length-decrease requirements depending on the nullability of the
regular expression:

\[
\forall \ s’ : [A]. \\
(\text{if nullable } r \ \text{then length } s’ \leq \text{length } s \ \text{else length } s’ < \text{length } s) \rightarrow \text{Bool}
\]

As already noted, they conjecture that without dependent types there is no easy way to confirm
termination, for example in a language like HOL.

4 RECURSION UNIVERSES: INTERFACES FOR TERMINATING RECURSION

This section describes the method used to achieve static confirmation of termination for Harper’s
matcher in Cedille. The basic idea is that Cedille provides to the bodies of recursive functions
an interface, which we call a recursion universe. The term “interface” is used here in the usual
software-engineering sense, to mean a specification stating some types that should exist and the types of some required operations. In particular, the recursion universes we consider below provide to each recursive function an abstract type \( R \) (locally bound for that recursion and hence distinct from those for other recursions) together with various operations on it. The language then ensures that any functions written against this interface are guaranteed to terminate. Cedille translates code written using the recursion universe to the terminating fragment of its core type theory (“Cedille Core”) [Stump 2018c]. The details of that translation may be found in [Jenkins et al. 2019]. Section 6 below goes into more detail on Cedille’s termination guarantee.

One can implement interfaces for recursion as combinators within a host language. Indeed, previous work by Ahn and Sheard [2011] did this in Haskell for an overlapping set of interfaces to the ones we consider below. Termination properties are argued informally there, as Haskell does not enforce termination. In the current work, we obtain more formal guarantees by working within Cedille. This difference aside, another benefit of Cedille’s approach is that the language provides direct support for its recursion universe. So rather than make calls to combinators, one writes recursive functions in a more idiomatic style, invoking the operations the recursion universe provides in the bodies of recursive definitions.

The starting point for Cedille’s recursion universe is Mendler-style recursion. To this, Cedille adds support for several extensions, namely paramorphic, histomorphic (not needed for Harper’s matcher and hence not considered here), and nested recursion. Since these forms of recursion, particularly Mendler-style, are either not widely known or, for Mendler-style paramorphic and nested recursion, original in Cedille, in the rest of this section we give a tutorial introduction to Cedille’s recursion universe, at least for the operations needed for Harper’s matcher. We will consider various example recursive functions \( f \) of type \( D \rightarrow X \), where \( D \) is an inductive datatype. Code is given in both Cedille and Haskell, to introduce Cedille syntax comparatively. Before we begin, though, let us review a familiar classic approach to terminating recursion, namely catamorphic recursion. This will provide a good comparative point of departure for the Mendler-style recursion universe.

### 4.1 A Review of Catamorphic Recursion in Haskell

A well-known form for terminating recursive functions is the catamorphism. For the example of finite lists, catamorphisms are constructed using the fold-right combinator (\( \text{foldr} \) in the Haskell prelude). The construction is more general, though, and applies whenever the datatype \( D \) is defined from a functor \( F \). In this case, the catamorphism combinator accepts an \( F X \rightarrow X \) function (called an algebra, due to its generalization of the usual concept of an algebra as implementing some operations over some type, here \( X \)), and then produces a \( D \rightarrow X \) function. Requiring that \( F \) is a functor is similar to positivity requirements familiar from type theory, as implementing the operation (called \( \text{fmap} \) in the Haskell prelude) that takes an \( A \rightarrow B \) function to an \( F A \rightarrow F B \) function is generally not possible if \( F \) uses its input negatively. If the algebra is terminating and \( F \) is indeed a functor, then the catamorphism for that algebra will also be terminating.

To see an example of this in Haskell, we should first consider an explicitly functorial version of the list datatype. (Interested readers can find very similar developments, also in Haskell, in [Ahn and Sheard 2011] and [Swierstra 2008].) This is given in Figure 2. The functor is \( \text{List} f a \), for any type \( a \) for the elements of the list. It is indeed easily shown (via the instance declaration) to be an instance of Haskell’s standard \texttt{Functor} type class. Though Haskell allows lazy infinite lists, we will suppose here that we are dealing with just the inductive type of lists. This is generated using the \( \text{Mu} \) datatype, which can be thought of (for finite lists) as giving the least fixed-point of its parameter \( f \) (at least when \( f \) is a \texttt{Functor}). Then smart constructors \texttt{cons} and \texttt{nil} can be defined with the usual types by wrapping calls to the constructors of \texttt{List} with calls to \texttt{In}.
data Listf a x = Nil | Cons a x

instance Functor (Listf a) where
  fmap g Nil = Nil
  fmap g (Cons h t) = Cons h (g t)

data Mu f = In (f (Mu f))

type List a = Mu (Listf a)

cons :: a -> List a -> List a
cons h t = In (Cons h t)

nil :: List a
nil = In Nil

Fig. 2. Explicitly functorial definition of the list datatype (review)

type CataAlg f x = f x -> x

cata :: Functor f => CataAlg f x -> Mu f -> x
cata alg (In d) = alg (fmap (cata alg) d)

length :: List a -> Int
length = cata alg
  where alg :: CataAlg (Listf a) Int
       alg Nil = 0
       alg (Cons _ e) = 1 + e

Fig. 3. The catamorphism combinator cata and an example of using it (review)

Figure 3 then defines the cata combinator, which accepts an algebra alg of type f x -> x, and returns a Mu f -> x function, for any Functor f. The type for these algebras is CataAlg f x. The alg function is applied recursively throughout the subdata d (of type f (Mu f)) using fmap. This results in a value of type f x, to which the alg is then applied to obtain the desired type x. As an example, Figure 3 defines a length function using cata. The definition applies cata to an algebra alg, which is given a Listf data structure holding the results of all recursive calls (in this case, just the length e of the tail). The length of the next-bigger sublist is then just 1 + e. Intuitively, cata alg works by recursively applying cata alg to the tail of the input list, to obtain an Int. This Int is then packaged up in a value of type Listf a Int, and given to alg. This makes the results of recursion available to the algebra, which can then compute the length of the input list.

4.2 A Mendler-Style Recursion Universe

Next, we consider the first step in our progression towards Cedille’s recursion universe. This is a recursion universe for so-called Mendler-style recursion, an approach to terminating recursion originated by Mendler [Mendler 1991], and studied subsequently in a number of other works [Ahn and Sheard 2011; Firsov et al. 2018; Uustalu and Vene 1999, 2000]. The basic idea proposed by
Mendler is to change the form of algebras from $F X \rightarrow X$ to $\forall R. (R \rightarrow X) \rightarrow F R \rightarrow X$. The algebra is not given a data structure full of the results of recursive calls (i.e., the input of type $F X$). Instead, the data structure it is given is populated with elements of some abstracted type $R$, together with a function that converts $R$ to $X$. It is this type $R$, with its operations, which we are thinking of as the “recursion universe” (and hence, since it is absent from catamorphic recursion, we do not consider that to be based on a recursion universe, except perhaps in some degenerate sense). So the recursion universe consists of:

- an abstract type $R$,
- a function—let us call it $\text{eval}$—of type $R \rightarrow X$, and
- a subdata structure of type $F R$.

How should this be implemented? There are actually several possibilities. We could use $X$ for $R$. Then the subdata structure is of type $F X$ just as for a catamorphism, and the $\text{eval}$ function should then just be the identity. With this choice, all recursive results will have to be precomputed to present in the subdata structure to the algebra. So while programs will still be closer to idiomatic, since recursive calls are now explicit, performance will be no better than catamorphic recursion, at least for eager evaluation.

The more interesting choice is to take the inductive datatype $D$ itself for $R$. Then the subdata structure is of type $F D$, which is exactly what the data of type $D$ already contains (as we see in the type of $\text{In}$ in Figure 3). So the subdata structure is just extracted from the input data, rather than being computed by recursion through the entire input data. The $\text{eval}$ function will then, of course, have real work to do, as it must recursively evaluate an input of type $D$ to obtain an $X$. But this is just what we want for idiomatic recursion! The algebra is allowed to make recursive calls selectively on subdata. In contrast, catamorphic algebras do not make explicit recursive calls; all the results of recursion are presented to them in the subdata structures. Mendler-style recursion opens the possibility for asymptotically more efficient programs, at least under eager semantics. For some computation may not require the recursive results for all subdata of some input piece of data, and with Mendler-style recursion an algebra implementing this computation has the possibility not to recurse on those subdata (whereas catamorphic algebras implicitly recurse on all subdata). Of course, with the lazy semantics of Haskell, we would not expect a difference: unneeded results of catamorphic recursion will not be normalized. Like catamorphic recursion, Mendler-style recursions terminate if their algebras do. This follows, for example, from the equivalence of inductive types with catamorphism and Mendler-style inductive types proved by Uustalu and Vene [1999].

4.2.1 The Mendler-Style Recursion Universe in Haskell. Figure 4 defines the combinator $\text{mcata}$ (following Ahn and Sheard [2011] for the name) with its form of algebra $\text{McataAlg}$. The definition of $\text{mcata}$ does not require that $f$ is a $\text{Functor}$, although we will not take advantage of this flexibility in this paper. The higher-rank type of $\text{mcata}$ is critical to the approach (and requires the RankNTypes language extension of ghc Haskell). The type variable $r$ is instantiated (implicitly) with $\text{Mu } f$ in the body of $\text{mcata}$, where $\text{alg}$ is called.
data Treef a x = Leaf a | Node x x

type Tree a = Mu (Treef a)

extremum :: Bool -> Tree a -> a
extremum b = mcata alg
  where alg :: McataAlg (Treef a) a
        alg eval (Leaf a) = a
        alg eval (Node t1 t2) = eval (if b then t1 else t2)

Fig. 5. An example function implemented using Mendler-style recursion

Figure 5 then gives an example for binary trees with data at the leaves. Say we wish to find the datum at the end of either the leftmost or rightmost path in an input tree, depending on some additional boolean input. The function extremum, written using mcata does this. The helper function alg, unlike the alg of Figure 3, takes in first a function eval of type \( r \to a \) to use for making recursive calls, and then the subdata structure of type Treef a r, for abstract type r. The introduction, from McataAlg, of abstract type r sets eval and the subdata up as lock and key: the only inputs we can possibly provide to eval are the subdata. But, as noted above, we do not need to make all possible recursive calls, and indeed, extremum recurses into only one subtree, depending on the boolean input.

This simple example shows a neat benefit of Mendler-style recursion over structural recursion, which one finds in type theories like Coq and Agda, and which was proposed originally for strong FP by Turner [1995] (see further discussion in Section 7 below). Namely, termination is enforced based on the typing. Therefore it is perfectly fine to call eval on the if-then-else expression, since the latter has the expected type \( r \). In contrast, in a recent version of Agda (2.6.0.1), such a recursive call is flagged as unsafe by the termination checker, since it is not on a pattern variable from the initial pattern-match.

Let us see a simple example showing how Mendler-style recursion prevents nontermination, in Figure 6. This code is attempting to inhabit the empty type Empty with an infinite loop, as eval will be instantiated by mcata with mcata \( \text{loopAlg} \) again, and hence eval \( 1 \) will just be again mcata \( \text{loopAlg} \). ghc rejects this code with the following type error:

- Occurs check: cannot construct the infinite type: \( r \to \text{Listf} a r \)
  
Expected type: \( (r \to \text{Empty}) \to \text{Listf} a r \to \text{Empty} \)

Actual type: \( (\text{Listf} a r \to \text{Empty}) \to \text{Listf} a r \to \text{Empty} \)

We see that the problem is that eval expects an input of type \( r \), but ghc has deduced that it is being applied to an input of type \( \text{Listf} a r \). So type checking rejects this code as falling outside the terminating sublanguage delimited by Mendler-style recursion, and hence possibly (here, actually) nonterminating.

4.2.2 The Mendler-Style Recursion Universe in Cedille. Let us see, finally, our first example of how Cedille supports more idiomatic recursion by implementing the Mendler-style recursion universe as part of the language. Figure 7 shows the code for extremum in Cedille. First (to show the syntax) we have datatype definitions for the booleans and the binary tree type. Note that Cedille does not require type argument \( A \) to Tree in the definition of Tree, because \( A \) has been declared as a parameter of the datatype. Parameters are written in parentheses immediately after the name introduced for the datatype. Otherwise, applications to type arguments are written using center dot, as in Tree \( \cdot \) A.
data Empty

loop :: List a -> Empty
loop = mcata alg
  where alg :: McataAlg (Listf a) Empty
        alg eval l = eval l

Fig. 6. This code has a type error at eval 1, showing how Mendler-style recursion leverages type checking to catch termination errors.

The definition of extremum is given in the form \( f : T = t \), where \( T \) is the type to check the definition \( t \) of \( f \) against. Cedille uses a form of local type inference to infer missing types on binders and missing type arguments [Jenkins and Stump 2018]. But explicit term-level binders for type variables are required, so the defining term for extremum begins with \( \Lambda A \) to introduce the type variable \( A \). Then we introduce the boolean \( b \) and tree \( t \). The value of type \( A \) is then computed by a \( \mu \)-term, which is a Mendler-style recursion over some inductive data. The syntax \( \mu f . t \{ \text{cases} \} \) initiates this recursion over inductive datum \( t \), introducing \( f \) as the name to use for recursive calls within the cases.

The two cases in the \( \mu \)-term of Figure 7 are similar to those of \( \text{alg} \) in Figure 5. An obvious difference is the use of Cedille’s \( \sigma \)-term to do a simple (non-recursive) case split on the boolean \( b \), instead of \( \text{if-then-else} \) in the Haskell version. A more subtle one is that here, because Cedille implements the recursion universe at the language level, we do not need to use a combinator like \( \text{mcata} \), and hence do not need to take the function to use for recursion as an explicit argument. Within the cases, Cedille introduces the following components of the recursion universe:

- a type variable \( \text{Type/extremum} \) of kind \( \star \); this is the abstract type \( R \)
- a term variable \( \text{extremum} \), of type \( \text{Type/extremum} \to A \); this is the eval function of the recursion universe
- the subdata, at type \( \text{Type/extremum} \); for the Node cases the subdata introduced with this type are the subtrees \( t1 \) and \( t2 \)

So even though the code looks like an idiomatic general recursion, with recursive calls being made explicitly using the name introduced by the \( \mu \)-term, it is in fact a Mendler-style recursion, using Mendler’s crucial insight that introducing an abstract type allows us to restrict the applicability of the eval function to the subdata. And unlike the Haskell version, this extremum is guaranteed by the language to terminate on all inputs. Note that Cedille’s compilation to Cedille Core makes all this completely explicit, as it defines functors that are then used as the basis for a least fixed-point construction (in particular, using impredicative type-quantification).

4.3 A Mendler-Style Paramorphic Recursion Universe

A well-known limitation of catamorphic recursion is that it lacks direct access to the subdata as it recurses, and hence operations like computing the tail of a list require linear time instead of the expected constant time (as one must catamorphically rebuild the subdata to return them; cf. [Hutton 1999]). Mendler-style recursion is no better off on this point: the fact that algebras may make explicit (and hence selective) recursive calls is no cure for lack of direct access to subdata. This is because Mendler-style algebras are presented with a subdata structure of type \( F R \). There is no way for them to compute the subdata of type \( D \) from this without recursing, as they are not allowed to treat \( R \) as \( D \). Indeed, we saw above that one option for implementing the Mendler-style recursion universe is to take \( X \) for \( R \), and so the subdata of type \( D \) really would not be available.
data Bool : ⋆ =
  True : Bool
| False : Bool.

data Tree(A : ⋆) : ⋆ =
  Leaf : A → Tree
| Node : Tree → Tree → Tree.

extremum : ∀ A : ⋆ . Bool → Tree · A → A =
    λ A . λ b . λ t .
    μ extremum . t
    { Leaf a → a
    | Node t1 t2 → extremum (σ b { True → t1 | False → t2 }) }.

Fig. 7. Extremum example in Cedille

Constant-time access to subdata is needed for practical programming, otherwise many programs
will be asymptotically slower than expected. Recursive functions with direct access to subdata
are called paramorphisms [Meertens 1992; Meijer et al. 1991]. Note that the distinction between
catamorphism and paramorphism is essentially that from logic between iteration and primitive
recursion; for a type-theoretic study see [Matthes 1999]. It seems previous works did not consider
a paramorphic version of Mendler-style recursion.

We may add support for paramorphic recursion to our Mendler-style recursion universe by
changing the type of algebras yet again (where D is the datatype):

∀R. (R → X) → (R → D) → F R → X

So algebras will be given:
- an abstract type R,
- a function eval of type R → X, and
- a subdata structure of type F R;

and additionally:
- a function reveal of type R → D.

This reveal function allows algebras to convert from the abstract type R to the real datatype D
(thus “revealing” that R was D all along). Note that now we no longer have (it seems) the choice
discussed in Section 4.2 above, between X or D for the implementation of R. To implement reveal,
we cannot take X for R in general, as this would require computing the input value of type D that
produced a particular output value of X (in other words, program inversion).

4.3.1 The Mendler-Style Paramorphic Recursion Universe in Haskell. Figure 8 implements a com-
binator mcatap for this in Haskell. This is the same as mcata of Figure 4 except that the type for
algebras now includes the type for the reveal function. The definition of mcatap implements
reveal simply as the identity function (second argument to alg in the definition of mcatap), since
the abstract type r in the definition of McatapAlg is being instantiated with Mu f (and hence id
has the required type Mu f → Mu f).

As an example of using mcatap, the figure gives code for a function drop (similar to the one in
Haskell’s prelude) that drops the first n elements from a list, and returning the empty list if there
are fewer than n elements. The helper alg, of type McatapAlg (Listf a) (Nat → List a),
type McatapAlg f x = (forall r . (r -> x) -> (r ->Mu f) -> f r -> x)

mcatap :: McatapAlg f x -> Mu f -> x
mcatap alg (In d) = alg (mcatap alg) id d

drop :: List a -> Nat -> List a
drop = mcatap alg
  where
    alg :: McatapAlg (Listf a) (Nat -> List a)
    alg drop reveal Nil _ = nil
    alg drop reveal (Cons h t) Zero = cons h (reveal t)
    alg drop reveal (Cons h t) (Suc n) = drop t n

Fig. 8. Combinator mcatap for paramorphic Mendler-style recursion, and an example of using it takes in the eval function for recursive calls, namely drop; and the reveal function, which is used when the input Nat is Zero and the subdata structure is a Cons. In this case, t from the subdata structure has abstract type r, and reveal converts this to List a as required.

4.3.2 The Mendler-Style Paramorphic Recursion Universe in Cedille. Paramorphic Mendler-style recursion is also directly supported by Cedille (with justifying translation to Cedille Core). Figure 9 shows the code, beginning (to be self-contained) with definitions for the Nat and List datatypes. The drop function again uses a µ-term to fold a paramorphic Mendler-style algebra over the input list l. The case-splitting structure of the code is the same as that of the Haskell version except that, as for extremum in Cedille, we use Cedille’s simple case-splitting construct σ to match on the input Nat (as we do not wish to initiate some new recursion over that Nat, but just decompose it). In the case where the Haskell code above applied reveal, here the Cedille code has to/List -isType/drop t, to be explained next. The code is otherwise similar except for supporting a direct recursive call drop t n instead of invocation of an explicitly supplied eval function.

To explain the to/List call: in Cedille, every datatype definition introduces a few symbols into the context for use with the recursion universe. For a simple example, Figure 10 shows these for the Nat type. There is a type variable Is/Nat of kind ⋆ → ⋆. This is meant to apply to types N that are Nat-like in supporting two eliminating operations: values of such a type N can be cast to Nat, and they can also be decomposed, using σ, with patterns for the constructors of Nat. Critically, however, there is no reverse conversion from Nat to a Nat-like N. Also, at present one cannot initiate new recursions on data of Nat-like type.

The value is/Nat asserts the triviality that Nat is Nat-like. The value to/Nat implements the first eliminating operation, namely conversion of Nat-like values to Nat. Though it does not factor into this paper, we show (for faithfulness) the ⇒ arrow in the type for to/Nat, which is for erased inputs. Here, it indicates that the conversion to Nat does not use the evidence that the type is Nat-like at run time. Application to erased arguments is written with a ~ sign, as in the call under consideration: to/List -isType/drop t. (Erased arguments are important for dependently typed programming, which Cedille supports though it is not used in this pearl; see [Barras and Bernardo 2008] for an approach similar to Cedille’s, or [Stump 2017, 2018a].)

For isType/drop: within the cases of a µ-term, an additional new variable is introduced, which asserts that the abstract type R is D-like. For the case of drop, we have isType/drop of type Is/List · A · Type/drop. This can be used with the to/List function to cast from the abstract type Type/drop to List · A, which is exactly what is needed for this base case of drop. Having a
data Nat : ⋆ =
  Zero : Nat
| Suc : Nat → Nat.

data List(A : ⋆) : ⋆ =
  Nil : List
| Cons : A → List → List.

drop : ∀ A : ⋆. List · A → Nat → List · A =
  λ A . λ l .
    μ drop . l {
      Nil → λ _ . Nil · A
      | Cons h t → λ n .
        σ n {
          Zero → Cons h (to/List -isType/drop t)
          | Suc n → drop t n
        }.
    }.

Fig. 9. The drop function in Cedille, showing the use of to/List for reveal

Is/Nat: ⋆ → ⋆
is/Nat: Is/Nat · Nat
to/Nat: ∀ X : ⋆. Is/Nat · X ⇒ X → Nat

Fig. 10. Extra declarations added automatically by Cedille at the Nat datatype declaration

cast function to/List separate from evidence isType/drop of castability may seem unnecessarily
general. Just introducing an \( R \rightarrow D \) function in the cases of \( μ \)-terms would be simpler, certainly. 
Cedille uses this somewhat heavier mechanism because it also works nicely for histomorphic
recursion, though discussing this would take us too far from the present pearl. Regardless, it would
be quite desirable to have Cedille automatically insert these \( R \rightarrow D \) coercions; this might be added
in a future version of Cedille with support for subtyping.

A note on semantics: intuitively, there should be no risk to termination by adding the \( \text{reveal} \)
function, and indeed there is not. The translation of this definition to Cedille Core uses positive
recursive types, which are known to preserve termination [Jenkins and Stump 2020; Parigot 1989].
The occurrence of \( D \) in the type of \( \text{reveal} \) turns into a positive-recursive occurrence in the type to
which \( D \) is translated.

4.4 The Final Extension: Nested Recursion

For Harper’s matcher, we will use Mendler-style paramorphic recursion, just considered, but with
one final extension. We need the ability to make nested recursive calls; i.e., calls like \( f (f a) \) in the
recursive definition of \( f \). Harper’s matcher does not explicitly do this. Rather, it uses a success
continuation to continue recursing. This is nested recursion in disguise, as recursive calls are made
on subdata produced by earlier recursive calls. This is not possible with the recursion universes
presented so far.

To extend the recursion universe to support nested recursion (including via continuations), we
must make one basic change to the general setup. For the type \( R \rightarrow X \) of the \( \text{eval} \) function is not
type McatapnAlg f x = (forall r . (r -> x r) -> (r -> Mu f) -> f r -> x r)

mcatapn :: McatapnAlg f x -> Mu f -> x (Mu f)

mcatapn alg (In d) = alg (mcatapn alg) id d

Fig. 11. The combinator for nested Mendler-style paramorphic recursion

rich enough to support nested recursion, as the type X cannot possibly mention R for scoping reasons (R is introduced after X). To address this, we raise the kind X of the algebra from ⋆ to ⋆ → ⋆. The eval function can then be given the type R → X R, giving the algebra the possibility of instantiating X with a type-level function that uses its argument R. The recursion universe is just as before, except for this richer type for eval:

- an abstract type R,
- a subdata structure of type F R,
- a function eval of type R → X R, and
- a function reveal of type R → D.

An algebra presented with this interface is required now to produce a value of type X R. And the recursion combinator, which in our previous version returned a value of type X, now returns a value of type X D. One important restriction: X must use its type input only positively to ensure termination. If not, one can construct, for example, an algebra that takes in an inverse to reveal, which we already noted would allow nontermination.

4.4.1 Nested Mendler-Style Paramorphic Recursion in Haskell. Figure 11 gives code in Haskell for the recursion combinator implementing the above recursion universe. Note that the return type of mcatapn is x (Mu f), corresponding to X D in the preceding discussion. And in the definition of McatapnAlg, x is applied to the type r (corresponding to X R).

Using this combinator, it is possible to write nested recursive functions, and functions that pass subdata of the input to a continuation for further recursion. Harper’s matcher needs this feature due to its use of continuations to continue recursive matching on suffixes of the input string. As a simpler example, we can define a continuation-based natural-number remainder operation. The preferred method for this in Cedille is to use histomorphic recursion (not covered in this paper), but nested recursion also works.

The rough idea is that to compute the remainder when dividing x by y, we will repeatedly subtract y from x, returning what is left of x when it falls below y. We implement this in a more interesting way using continuations; see Figure 12. The main function mod strips off initial successors from x and y (if possible) and then calls a helper function modh defined using mcatapn. This function, and its underlying McatapnAlg, accepts two continuations, k1 and k2. The details are somewhat baroque (see Appendix A). Here we just focus on the typing. The crucial point is that k1 must accept a part of the original input x, on which it may recurse. Hence, k1 needs to take in the abstract type r. To realize this with mcatapn, we use a definition of ModhT of Haskell kind ⋆ → ⋆. We see that the input type r is indeed used as the input type in (r -> Nat), for the first continuation (k1).

4.4.2 Nested Recursion in Cedille. Cedille supports nested recursion as just discussed by allowing the return type of the recursive function f to mention Type/f (the abstract type of the recursion universe for f) positively. See Figure 13. The return type must be explicitly given with @ in the µ-term, with its dependence on the recursion universe explicitly specified using Type/modh:

\[
\mu \text{modh} \cdot n @ (\lambda _ : \text{Nat} . \text{Bool} \rightarrow \text{Nat} \rightarrow (\text{Type/modh} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat})
\]
newtype ModhT r =
    ModhT { unModhT :: Bool -> Nat -> (r -> Nat) -> (Nat -> Nat) -> Nat }

modh :: Nat -> ModhT Nat
modh = mcatapn alg
    where alg :: McatapnAlg Natf ModhT
          alg = ...

mod :: Nat -> Nat -> Maybe Nat
mod _ (In Zero) = Nothing
mod (In Zero) (In (Suc _)) = Just zero
mod (In (Suc x)) (In (Suc y)) = Just (unModhT (modh x) True y id id)

Fig. 12. Remainder operation implemented with continuations using mcatapn; see Appendix A for elided details

    λ n .
    μ modh . n
    @ (λ _ : Nat . Bool → Nat → (Type/modh → Nat) → (Nat → Nat) → Nat)
    { ... }.

mod : Nat → Nat → Maybe · Nat =
    λ n . λ m .
    σ n { Zero → Nothing · Nat
        | Suc x →
        σ m { Zero → Just Zero
            | Suc y → Just (modh x True y (id · Nat) (id · Nat)))}
    }.

Fig. 13. Remainder operation in Cedille; see Appendix A for elided details

We see that Type/modh is used here as the domain type for the type of the first continuation. Note that Cedille requires the return type specified to have kind D → ⋆ for use with dependent eliminations; in this paper, since we eschew dependent types for purposes of strong FP, the D input is ignored (hence the input to the motive is declared as _ : Nat). The code, given in the Appendix, is otherwise a port of the Haskell version.

5 HARPER’S CONTINUATION-BASED MATCHER VIA NESTED RECURSION

Here we present our implementation of Harper’s matcher. The main functions are in Figure 15. The code is quite similar to the Haskell version we considered above (Section 2), except that Cedille’s type system statically ensures that the matcher terminates on all inputs. Cedille’s definition of syntactically standard regular expressions is in Figure 14. As for the Haskell version, Cedille works with a String type defined as a list of characters (i.e., List · Char).

5.1 Structure of the Main Functions (Figure 15)

The code of Figure 15 begins with type-level functions K and matchT similar to the Haskell version above. Next comes mutually recursive definitions of functions matchi and match (the latter
data StdReg : ⋆ =
    NoMatch : StdReg
  | MatchChar : Char → StdReg
  | Or : StdReg → StdReg → StdReg
  | Plus : StdReg → StdReg
  | Concat : StdReg → StdReg → StdReg.

Fig. 14. Syntactically standard regular expressions as defined by Korkut, Trifunovski, and Licata

combining matchh and match of the Haskell version). Cedille does not (currently) provide direct support for mutual recursion, so we take advantage of the Mendler aspect of Cedille’s recursion universe to break the cycle of the mutual recursion. For this, the type of matchi says that for any type T, given a function of type T → StdReg → matchT · T, matchi will then consume a Char and a T and return a matchT · T (i.e., it will next take in a continuation and produce the ultimate Bool answer). The function that match will provide for the first term argument to matchi will be (the µ-bound) match itself. This works because, as we have seen above, Mendler-style algebras allow recursive calls on the abstract type T independently of the code for the recursive function. So we can pass match to a helper function, as we are doing here, and this poses no problems for termination checking. In contrast, it would be impossible in languages like Coq or Agda without the use of sized types.

5.2 The Functions Provided by the Recursion Universe
To understand the code further, it is helpful to see what Cedille’s Emacs mode also shows the programmer, namely the instances of the recursion universe for the main functions. These are shown in Figure 16, first for matchi and then for match.

As explained in Section 4.2.2, the types Type/... are the abstract types of the recursion universe (R, in terms of Section 4.2). The istType/... values are used to cast from the abstract types to the concrete data type (so String for match and StdReg for matchi). This is necessary in a couple places in matchi.

Finally, we have the variables match and matchi, which are used for recursion inside the body of the µ-terms. The central technical innovation of the paper appears in the type for match. This type uses Type/match not just as its input type, but (positively) in its output type. This enables nested recursion, which as we noted earlier is mediated in Harper’s matcher by a continuation. The type of this continuation (unrolling matchT · Type/match) is Type/match → Bool. Since match invokes matchi with Type/match as the instantiation of the type T (in matchi’s type), both matchi and match are using continuations of type Type/match → Bool.

5.3 Final Code Notes for Figure 15
Both in the Haskell version above and in this Cedille one, matchi takes in the head character of the string as well as the tail (in the Cedille version, at abstract type). This design, while optional for the Haskell version, is rather forced in Cedille because, in the recursion for match, the original string is not available at the abstract type needed by matchi (to enable nested recursion). We can see what is happening better by seeing part of the context (which the Cedille emacs mode displays) for the cons case of match, in Figure 17.

We can confirm that s, the string given to the lambda-term defining the global match is not available at abstract type. This is necessary, of course, since it should not be legal to recurse on that string itself. We can only recurse on its tail cs, which we see (in Figure 17) is available at

\begin{align*}
\text{matchT} : \forall T : ⋆. (S → StdReg → \text{matchT} · T) \rightarrow \\
\text{StdReg} \rightarrow \text{Char} \rightarrow T \rightarrow \text{matchT} · T = \\
\lambda T . \text{matchi} . \lambda r . \\
\text{NoMatch} \rightarrow \lambda c . \lambda cs . \lambda k . \text{False} \\
\text{MatchChar} c' \rightarrow \lambda c . \lambda cs . \lambda k . \\
\sigma (\text{eqChar} c c') \{ \text{True} \rightarrow k cs \mid \text{False} \rightarrow \text{False} \} \\
\text{Or} r1 r2 \rightarrow \lambda c . \lambda cs . \lambda k . \\
\text{matchi} r1 c cs k \\
\text{matchi} r2 c cs k \\
\text{Plus} r \rightarrow \lambda c . \lambda cs . \lambda k . \\
\text{matchi} r c cs \\
(\lambda cs . \\
\text{or} (k cs)) \\
\text{matchi} r c cs k \\
\text{Concat} r1 r2 \rightarrow \lambda c . \lambda cs . \lambda k . \\
[r2c = \text{to/StdReg -isType/matchi}] - \\
\text{matchi} r1 c cs \\
(\lambda cs' . \text{matchi} r2 c cs k).
\end{align*}

\text{match} : \text{String} → \text{StdReg} → \text{Bool} = \\
\lambda s . \lambda r . \\
\text{matchi} . s @ (\lambda _ : \text{List} · \text{Char} . \\
\text{StdReg} \rightarrow \text{matchT} · \text{matchi}) \\
\{ \text{nil} \rightarrow \lambda r . \lambda k . \text{False} \\
\text{cons} c cs \rightarrow \lambda r . \lambda k . \\
\text{matchi} \text{match} r c cs k \\
\}
\text{r} \\
(\text{isNil} · \text{Char}).

Fig. 15. Main functions for Harper’s matcher in Cedille

\begin{align*}
\text{Type/matchi} : ⋆ \\
\text{isType/matchi} : \text{Is/StdReg} · \text{Type/matchi} \\
\text{matchi} : \text{Type/matchi} → \text{Char} → T → \text{matchT} · T
\end{align*}

\begin{align*}
\text{Type/match} : ⋆ \\
\text{isType/match} : \text{Is/List} · \text{Char} · \text{Type/match} \\
\text{match} : \text{Type/match} → \text{StdReg} → \text{matchT} · \text{Type/match}
\end{align*}

Fig. 16. The instances of the recursion universe for matchi and match
c: Char
s: Type/match
k: K · Type/match
r: StdReg
s: String

Fig. 17. Some typings for the cons case of match of Figure 15

type Type/match. If we wished to make use of cons c cs at type String, we would need to cast the tail to String, which the recursion universe allows, and then rebuild the cons-term. Note also that the μ-term initiates a fold over s; it does not directly define a function with input s. So rebuilding cons c cs will equal s only at the outermost point of the fold. As the fold proceeds into s, cons c cs will be bound to suffixes of s.

5.4 Handling General Regular Expressions
To keep the paper self-contained, we include in Figure 18 the datatype Reg of (not necessarily standard) regular expressions, a function matchesEmpty that checks whether such an expression can match the empty string, and a function standardize that converts a Reg to a StdReg that is equivalent except for not matching the empty string. This development is a port of similar code in [Harper 1999; Korkut et al. 2016]. The one interesting difference in the Cedille version is that the code for standardize needs to use the conversion to/Reg to convert from the recursion universe of standardize back to Reg in order to call matchesEmpty. Armed with the definitions of Figure 18, we can define the matcher on general regular expressions in Figure 19.

5.5 Concluding Discussion
Writing Harper’s matcher in Cedille is straightforward and follows the functional structure of other versions of the matcher in the literature (as surveyed in Section 3). If we consider the effort required to make use of Cedille for statically ensuring termination, it is almost zero. We could say that the helper function matchi is forced to take in the head and tail of the list separately (as opposed to in a single string argument), so it can accept that tail at the abstract type needed for subsequent recursive calls. But this is really the only change. Otherwise, the code is a direct port of the Haskell code of Figure 1, requiring no ingenuity.

This should be contrasted with previous approaches (Section 3) that view Harper’s matcher as an interesting challenge problem. They either rely on type-checking augmented with automated theorem proving to dispatch termination conditions relating to the length of the string [Xi 2002]; fail to verify termination of the original higher-order matcher [Owens and Slind 2008]; rely on well-founded recursion and explicit (though not onerous) proofs in Agda [Korkut et al. 2016]; or else again explicit proofs, in Coq [Bove et al. 2016]. With the strong FP approach, no termination reasoning at all is required of the programmer. Instead, one must fit one’s recursion into the structure imposed by the recursion universe. For Harper’s matcher, this can be done with little change to the code.

6 CEDILLE’S NORMALIZATION GUARANTEE
Since this paper presents Harper’s matcher as a strong functional pearl, we should consider briefly how Cedille enforces termination and the basis for trusting that it is doing so correctly.

The realizability semantics presented in the original paper on Cedille’s theory ensures that every inhabitant of a function type is β-equal to a λ-abstraction [Stump 2017]. While Cedille’s
data Reg : ⋆
  = RegNoMatch : Reg
  | RegEmpty : Reg
  | RegChar : Char → Reg
  | RegConcat : Reg → Reg → Reg
  | RegOr : Reg → Reg → Reg
  | RegStar : Reg → Reg
.

matchesEmpty : Reg → Bool
= λ r. μ matchesEmpty. r {|
  | RegNoMatch → ff
  | RegEmpty → tt
  | RegChar _ → ff
  | RegConcat r1 r2 →
    and (matchesEmpty r1) (matchesEmpty r2)
  | RegOr r1 r2 →
    or (matchesEmpty r1) (matchesEmpty r2)
  | RegStar r → tt
}. 

standardize : Reg → StdReg
= λ r. μ standardize. r {
  | RegNoMatch → NoMatch
  | RegEmpty → NoMatch
  | RegChar c → MatchChar c
  | RegConcat r1 r2 →
    [r1' = standardize r1] -
    [r2' = standardize r2] -
    [t = to/Reg -isType/standardize] -
    σ (matchesEmpty (t r1)) {
      | ff →
        σ (matchesEmpty (t r2)) {
          | ff → Concat r1' r2' | tt → Or r1' (Concat r1' r2') }
      | tt →
        σ (matchesEmpty (t r2)) {
          | ff → Or r2' (Concat r1' r2') | tt → Or r1' (Or r2' (Concat r1' r2')) }
    }
  | RegOr r1 r2 → Or (standardize r1) (standardize r2)
  | RegStar r → Plus (standardize r)
}. 

Fig. 18. Standardizing regular expressions Reg

current theory has changed (mostly by drastic simplification) from this original one, the realizability semantics has been kept in sync with the changes [Stump 2018b]. As a corollary we have:
matchReg : String → Reg → Bool = λ s. λ r. 
  or (and (matchesEmpty r) (isNil s)) 
    (match s (standardize r)) .

Fig. 19. Using match and standardization to match general regular expressions

**Theorem 6.1 (Call-by-name normalization of functions).** Suppose \( t : T \), \( t \) is closed, and there exists a closed term \( t' \) that erases to \( \lambda x. x \) and whose type is \( T \to \Pi x : T_1. T_2 \) for some \( T_1, T_2 \). Then \( |t| \) is call-by-name normalizing.

Erasure means erasing type annotations and erased function inputs throughout terms. This theorem expresses the idea that if one can cast (i.e., transform via an identity function) \( T \) to a \( \Pi \)-type, then all closed terms \( t \) of type \( T \) are cbn-normalizing. The result follows from the semantics by standardization for untyped lambda calculus [Kashima 2001].

Since Cedille can translate from its full source language (including datatypes and recursion) to its core pure typed lambda calculus (Cedille Core, with no such constructs), and that translation maps data to types \( T \) that can (as they are just polymorphic function types) indeed be cast to (non-polymorphic) function types, normalization of full source programs is assured. This result holds both by the currently unpublished theoretical analysis of the translation [Jenkins et al. 2019], and by checking the elaborations with Cedille’s core checker [Stump 2018c].

For readers interested in foundational guarantees (i.e., carefully identified and minimized formal artifacts justifying computational tasks like certifying termination): our approach, while certifying termination of Harper’s matcher and similar pure functional programs utilizing Cedille’s recursion universe, is currently not foundational, in the sense that it depends on correctness of the elaborator from full Cedille down to Cedille Core, and also the correctness of Cedille Core. Both algorithms have been proven correct on paper only [Jenkins et al. 2019; Stump 2018c].

**7 OTHER RELATED WORK**

**Previous work on strong FP.** Turner advocates strong FP in several papers [Turner 1995, 2004]. His primary motivation is to ensure that the seemingly close fit between functional programming and mathematical reasoning is not undermined by the possibility that an expression of type \( T \) might not denote a value of type \( T \). Related to this is work of Cockett and Spencer [1995] on Charity, a terminating programming language based on a category-theoretic treatment of datatypes and co-datatypes. Charity implements a catamorphic recursion universe.

The main difference with this paper is that in these works, functions are required to be structurally recursive, and nested recursion is not supported. Remarkably, Cockett [1996, Section 3.2.5] found a clever (but arguably not too natural) way to express nonstructural recursions like mergesort in Charity, as a combination of inductive and coinductive programming. Essentially, these are a-priori bounded hylomorphisms (so that the result can be inductive). In contrast, our proposal does not require use of coinductive types.

An interesting example of a form of strong FP applied to parsing is Danielsson [2010]’s library of total parser combinators, which guarantees that parsers built using the given combinators terminate for all input strings.

**Termination in proof assistants** The work already cited of Bove et al. [2016] gives a thorough overview of the problem of termination—and its diverse solutions—in proof assistants. We note particularly Krauss [2010]’s work on partial and nested recursions in HOL, where the main concern is allowing the function to be expressed and certain properties proved about it before requiring
a termination proof. It is also worth noting that proof assistants may soundly allow possibly nonterminating expressions, as long as these do not pollute the fragment of the language intended to be sound as a logic (cf. [Constable and Smith 1987]).

**Automated termination analysis.** There is a large literature on termination analysis for various programming paradigms (imperative, declarative, functional). One can view these approaches as much more powerful versions of the simple syntactic check for structural recursion proposed by Turner (indeed Telford and Turner [2000] went beyond this simple scheme). The goal of these methods is generally to increase the class of systems that can be automatically judged terminating (or proven nonterminating), and reduce the time needed to do so. There is an annual Termination Competition aimed at stimulating such improvements [Giesl et al. 2019]. These methods have some drawbacks that would hinder their use for strong FP:

- compositionality is generally not considered;
- understanding why a given program was rejected (or accepted) by the termination checker requires understanding complex algorithms; and,
- state-of-the-art tools are big [Giesl et al. 2017]. For example, they rely on SMT solvers, which are typically in the hundreds of thousands of lines of code. It is unattractive to require a dependency on such complex software as a termination checker.

In contrast, the methods we describe here can be implemented with very modest line counts, and are compositional since they are type-based. Understanding their scope is a matter of understanding a single interface (to the recursion universe), and the basics of the ambient type system.

**Sized types.** The idea of type-based termination may bring to mind the idea of sized types, for which the name “type-based termination” is also used [Barthe et al. 2004]. There, termination is established using types that are indexed by (approximations to) the sizes of the data being manipulated. This requires complex additional machinery in the type system, and is well outside the usual Hindley-Milner algorithm for static type inference. In contrast, our proposal does not make use of indexed types. Indeed, types are unmodified from standard polymorphic FP. The change comes in the typing of recursive functions.

**Recursion schemes.** A number of works in applied category theory and functional programming consider recursion schemes, which abstract patterns of recursion into higher-order combinators. Perhaps the most cited work is [Meijer et al. 1991]. Establishing algebraic laws for these combinators and uniqueness of solutions of their defining equations are the primary considerations. While this work is related and provides important background (for example, the idea of datatypes as initial algebras), the focus is on algebraic reasoning about programs, not on program termination. Program termination is established for some basic recursion schemes like catamorphisms (which correspond to structural recursions), but for more powerful schemes like hylomorphisms, termination is guaranteed only under certain conditions [Adámek et al. 2007; Hinze 2013]. For strong FP, in contrast, the main goal is to ensure termination while preserving a natural style of functional programming.

8 CONCLUSION

In this pearl we have described an implementation of Harper’s continuation-based regular expression matcher as a strong functional program in Cedille. Where recent research has appeared mostly to show pure FP drawing inspiration from type theory, we hope that this work can serve as an example of the reverse. Strong FP has the potential not only to be a useful extension of pure FP, but also for adoption in type theory. Indeed, though we only used its functional-programming fragment, Cedille is furthermore a dependent type theory in which proofs of properties of programs like the ones considered here can be carried out. This remains to future work. Further future work...
includes pursuing further extensions of Cedille’s recursion universe to support additional patterns of recursion. A painful omission here, as well as in standard approaches such as those implemented in type theories like Coq and Agda, is for divide-and-conquer algorithms like mergesort. The divide step typically creates new data that, while smaller than the input datum by some measure, are not strictly speaking subdata; Cedille’s recursion universe cannot help, nor can the usual type-theoretic structural-recursion checks. Sized types are applicable [Copello et al. 2014], but these fall outside the realm of strong functional programming. A strong FP solution for divide-and-conquer thus remains as an interesting challenge.

ACKNOWLEDGMENTS

Many thanks to Richard Eisenberg for shepherding the paper, and the anonymous ICFP reviewers, for very detailed and constructive criticism that resulted in a major revision of the original submission and numerous smaller improvements. The paper would not have come together in its current form without their generous aid. Thanks also to Garrett Morris for many conversations about recursion combinators in Haskell. We gratefully acknowledge NSF support under award 1524519, and DoD support under award FA9550-16-1-0082 (MURI program).

A THE CONTINUATION-BASED REMAINDER EXAMPLE, IN HASKELL AND CEDILLE

Figure 20 gives the full code for the natural-number remainder operation defined with continuations discussed in Sections 4.4.1 and 4.4.2. Calls to helper function modl are of the form modl x start y k1 k2, ignoring some newtype machinery that was needed for type inference to succeed. At a high level, the continuation k1 is called to start a new subtraction of y when the current one finishes. The continuation k2 is called when we reach the end of x but not yet the end of y. This represents the case of a non-zero remainder, and k2’s job is to add back on successors to its input to correspond to the part of the original y already consumed by the current attempt to subtract off y. In more detail, and connecting to the code of Figure 20, we have the following semantics for modl x start y k1 k2 (again, ignoring the newtype accessor):

- **start** indicates whether we are beginning a subtraction of y (True) or are in the middle of a subtraction (False)
- **if** x and y become Zero simultaneously, return zero; y evenly divides x (see (a) in the code)
- **if** x becomes Zero strictly before y does, then return k2 (Suc zero); this is the case where we have a non-zero remainder, and the job of k2 is to compute it from 1 (see (b)).
- **if** y becomes Zero strictly before x, then if we just started our subtraction we are effectively taking the remainder when y is 1; so we should return 0 since there 1 evenly divides any number (see (c)). Alternatively, if we are in the middle of a subtraction, we call k1 with the predecessor of x; we have successfully subtracted off the original y one time from x, and the job of k1 is the continue the recursion by subtracting again (see (d)).
- **Finally**, if neither y nor x is Zero, we recurse, setting start to False (since we are definitely in the middle of a subtraction with this recursion). We update k1 so that if we are just starting a subtraction, then when k1 is called with some x’ (less than x), we recurse with the original y (see (e)); and if start is False, we call k1 (see (f)). k2 is updated to tack on an extra Suc, so that we account for the part of y we subtract off if we reach the base case above for a non-zero remainder.

This intricate control-flow aside, the most critical detail, as noted in the main text above, is the definition of ModlT of Haskell kind * -> * . The input type r is used as the input type for the first continuation (k1). This is what justifies calling k1 with x and x’ in the second clause for a1g; those
newtype ModhT r =
    ModhT { unModhT :: Bool -> Nat -> (r -> Nat) -> (Nat -> Nat) -> Nat }

modh :: Nat -> ModhT Nat
modh = mcatapn alg
where alg :: McatapnAlg Natf ModhT
    alg modh _ Zero =
        ModhT (
            _ _ k1 k2 -> case y of
                In Zero -> zero -- (a)
                _ -> k2 (suc zero)) -- (b)
    alg modh _ (Suc x) =
        ModhT (
            start y k1 k2 -> case y of
                In Zero ->
                    if start then zero -- (c)
                    else k1 x -- (d)
                In (Suc y) ->
                    unModhT (modh x) False y
                    (\ x' -> if start then
                        unModhT (modh x') True
                        (suc y) k1 k2 -- (e)
                        else k1 x') -- (f)
                    (suc . k2) )

mod :: Nat -> Nat -> Maybe Nat
mod _ (In Zero) = Nothing
mod (In Zero) (In (Suc _)) = Just zero
mod (In (Suc x)) (In (Suc y)) = Just (unModhT (modh x) True y id id)

Fig. 20. Remainder operation implemented with continuations using mcatapn

variables have abstract type r, and k1 accepts inputs of type r. The first continuation effectively makes a nested recursive call at label (e).

The full implementation in Cedille is shown in Figure 21 (eliding standard definitions of id and compose). Thanks to the combination of the dependence of the return type on the abstract type Type/modh and the Mendler-style recursion allowing recursive calls on any input of that type, we get an idiomatic recursion.
λ n . μ modh . n @ (λ _ : Nat . Bool → Nat → (Type/modh → Nat) → (Nat → Nat) → Nat)
{ Zero → λ _ . λ p . λ _ . λ k2 .
  σ p {
    Zero → Zero -- (a)
    | Suc _ → k2 (Suc Zero) } -- (b)
    | Suc x → λ start . λ p . λ k1 . λ k2 .
  σ p {
    Zero → σ start {
      True → Zero -- (c)
      | False → k1 x } -- (d)
    | Suc y →
    modh x False y
    (λ x' . σ start {
      True → modh x' True
      (Suc y) k1 k2 -- (e)
      | False → k1 x' })} -- (f)
  (compose Suc k2) }
}.

mod : Nat → Nat → Maybe · Nat =
λ n . λ m .
σ n { Zero → Nothing · Nat
  | Suc x →
  σ m { Zero → Just Zero
    | Suc y → Just (modh x True y (id · Nat) (id · Nat)))}.

Fig. 21. Remainder operation in Cedille

REFERENCES


