A Coinductive Monad for Prop-Bounded Recursion *

Adam Megacz
UC Berkeley
megacz@cs.berkeley.edu

Abstract

This paper develops machinery necessary to mechanically import arbitrary functional programs into Coq's type theory, manually strengthen their specifications with additional proofs, and then mechanicaly re-extract the newly-certified program in a form which is as efficient as the original program.

In order to facilitate this goal, the coinductive technique of [Cap05] is modified to form a monad whose operators are the *constructors* of a coinductive type rather than *functions* defined over the type. The inductive invariant technique of [KM03] is extended to allow optional "after the fact" termination proofs. These proofs inhabit members of Prop, and therefore do not affect extracted code.

Compared to [Cap05], the new monad makes it possible to directly represent unrestricted recursion without violating productivity requirements [Gim95], and it produces efficient code via Coq's extraction mechanism. The disadvantages of this technique include reliance on the JMeq axiom [McB00] and a significantly more complex notion of equality.

The resulting technique is packaged as a Coq library, and is suitable for formalizing programs written in any side-effect-free functional language with call-by-value semantics.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic*]: Lambda calculus and related systems; Proof theory

General Terms Verification

Keywords Type Theory, Coinductive Types

1. Introduction

The Calculus of Constructions [CH88] and related type theories provide a rich environment for proof development. Such environments are particularly well-suited to reasoning about total functional programs [Tur04] as they are capable of representing both the program and its correctness proof in the same formalism.

Unfortunately, most functional languages are not total; in practice unrestricted recursion and partial functions are the norm, and the ability to reason about such programs is important for a practical tool. Type theory cannot directly represent these functions because any term in the theory could potentially be part of a proof, and proofs must be strongly normalizing in order to preserve consistency.

To date, a number of encodings and techniques have been proposed to deal with this problem. Section 2 classifies these techniques into six categories, briefly touching on the advantages and disadvantages of each. Section 3 introduces an alternative to Capretta's encoding [Cap05] of partial functions as values of coinductive type. Section 4 introduces a pair of *inductive* predicates over the coinductive monad which can be used to prove termination of partial functions. These predicates are "after the fact" in the sense that the implementation of a function need involve forethought about the form of its termination proof. Moreover, these proof terms belong to members of Prop, and therefore do not clutter or degrade the performance of extracted code.

Section 5 describes the implementation of this technique as a Coq library, including the eval function. This function combines a partial function and its termination proof to yield a total function which is first-class within the type theory and can be evaluated within the proof assistant. A notable limitation of this implementation is that it currently supports only *call-by-value* evaluation in languages free from side effects; at the moment it is better suited to languages such as Ωmega[She05] or Id[Nik91] than to call-by-need languages such as Haskell, or languages with side effects such as ML.

Section 6 details several example programs and their formalization using the coinductive computation monad. Section 7 describes advanced applications of the coinductive monad, including composition of computations over different sets, higher-order computations, and first-class termination proofs. Section 8 compares this work to [Cap05]. Section 9 summarizes the paper and outlines directions for future work.

2. Existing Work

2.1 Syntactically Decreasing Calls

In the special case where every recursive invocation of a function is applied to an inductive value which is strictly smaller than the present argument, and where this fact is *syntactically* obvious[Gim98], Coq provides built-in machinery for translating self-referential values into equivalent well-founded terms. Unfortunately, this syntactic check often fails in situations where the function is, nonetheless, well-founded. Consider the following "fast exponentiation" algorithm, taken from [GBR06]:

$$2^n = \begin{cases} 1 & \text{if } n = 0 \\ 2 \cdot 2^{n-1} & \text{if } n \text{ is odd} \\ 2^{n/2} \cdot 2^{n/2} & \text{if } n \text{ is even} \end{cases}$$

* This material is based upon work supported by a National Science Foundation graduate fellowship.

Using the following auxiliary functions,

```
Definition square n := n * n.

Fixpoint divide_by_two (n:nat) : nat :=
  match n with
    | 0          => 0
    | 1          => 0
    | (S (S n')) => S (divide_by_two n')
  end.
```

A straightforward coding of the fast exponentiation algorithm in Coq would read like this:

```
(*
Fixpoint pow2 (n: nat) : nat :=
  match n with
  | 0      => 1
  | (S n') =>
    match even_odd_dec (S n') with
    | left  _ => square (pow2
                         (divide_by_two (S n')))
    | right _ => n * (pow2 n')
    end
  end.
 <<invalid>> *)
```

Unfortunately, it is not *syntactically* obvious to Coq that `divide_by_two (S n')` is less than `S n'`. Although it is not difficult to prove this fact, Coq is only interested in syntactic arguments.

2.2 `Set`-bounded recursion

Perhaps the most popular of the well-established techniques, `Set`-bounded recursion involves adding an additional argument to the function and calling each recursive invocation on some subcomponent of that argument. As long as the initial value provided for this argument is sufficiently large, the result will be the same as if the recursive function were evaluated directly.

```
Inductive res (A : Set) : Set :=
  | Ok : A -> res A
  | Timeout : res A.

Notation "var <-- e ; rest" :=
      (match e with
       | Ok x => (fun var => rest) x
       | Timeout => Timeout _ end)
      (at level 60, right associativity).

Fixpoint pow2_bounded (n:nat) (depth:nat)
                      {struct depth}
         : res nat :=
  match depth with 0 => Timeout | (S depth) =>
  match n with
  | 0      => Ok 1
  | (S n') =>
    match even_odd_dec (S n') with
    | left  _ => x <-- pow2_bounded
                       (divide_by_two (S n'))
                       depth
               ; Ok (square x)
    | right _ => x <-- pow2_bounded n' depth
               ; Ok (x*2)
    end
  end.
```

Note that because the original function returns a value in `Set`, the bounding argument cannot be in `Prop`, because the function decomposes the bounding argument in order to produce its return value, and a `Prop` cannot be decomposed in order to produce a `Set`. This is unfortunate, because it means that the extracted code will contain a great deal of extra computational content which cannot be optimized away by even a highly intelligent compiler:

```
(** val pow2_bounded : nat -> nat -> nat res **)

let rec pow2_bounded n = function
  | O -> Timeout
  | S depth0 ->
    match n with
    | O -> Ok (S O)
    | S n' ->
      match even_odd_dec (S n') with
      | Left ->
        match pow2_bounded
              (divide_by_two (S n'))
              depth0 with
        | Ok x0 -> Ok (square x0)
        | Timeout -> Timeout
        end
      | Right ->
        match pow2_bounded n' depth0 with
        | Ok x0 -> Ok (mult x0 (S (S O)))
        | Timeout -> Timeout
        end
      end
    end
  end
```

Furthermore, the additional `nat` argument can become cumbersome at times and adds a fairly unnatural element to proofs.

Niqui and Bertrot quantified the performance penalty of such constructs in a Haskell extraction and found[NB03]:

> *Later we modified the whole formalization and we used the* `Prop`*-sorted accessibility. Our tests showed a 25% to 30% decrease in both time and memory usage of the extracted algorithms.*

2.3 Ad-hoc Predicate-bounded Recursion

Bove [Bov01] suggests introducing a predicate over the domain of each partial function such that the predicate holds on that subset of the domain for which the function terminates. work on automatically generating such predicates has been explored in [Nor93, BC01]. Both of these developments were carried out in ALF[Nor88], which does not distinguish between computationally extractable and computationally irrelevant types.

Unfortunately, if the predicate type has more than one constructor, it will need to be in `Set` or `Type`, and will carry with it the disadvantages of the previous section. Forming ad-hoc predicates with a single constructor is often quite difficult, and requires careful planning of the termination proof *when writing the function*. Moreover, as [NB03] lament, “unfortunately the second approach [`Prop`-bounded recursion] is more technical and requires an advanced knowledge of the internals of *Coq*... a detailed study of the proof terms of *Coq* was necessary.”

2.4 Accessibility Predicates

As a result of the consequences of using a `Set` bound for computations, a number of different techniques have been developed for applying a `Prop` bound instead. Nearly all of them are in some way built on the basic technique employed by the `Coq.Init.Wf` library.

A programmer who wishes to use this library must structure any functions such that they recurse on a *primary argument*, and must choose some *well-founded relation* denoted by `<` under which each recursive call is applied to a *lesser* value than the argument to the callee.

Proving that the chosen relation is well-founded amounts to showing that every value in the domain of the primary argument is *accessible* under this relation. A value `x` is accessible if one of the following two cases holds:

1. No value is less than `x` under the chosen relation
2. All values less than `x` under the chosen relation are accessible

At first glance, this would appear to lend itself to an inductive definition with two constructors, one for each case; something approximately like this:

```
Inductive Acc (x:nat) : Prop :=
 | Acc_nopred  : ~ (exists y, y<x)        -> Acc x
 | Acc_haspred : (forall y, y<x -> Acc y) -> Acc x.
```

However, the first constructor, `Acc_nopred`, turns out to be superfluous; if `x` has no predecessors we can easily construct `Acc_haspred x` because the left hand side of its implication is equivalent to `forall y, False->Acc y`, which is trivially true.

Eliminating this first constructor leaves a *single-constructor Prop type*, which is a special case in which a `Prop` *may* be destructed as part of the process of constructing a value in `Set`. Indeed, this is the motivation for choosing this particular definition of accessibility in the first place. Note that this critera can be exploited by other techniques for representing recursion, but in those cases it is typically exposed to the user – that is, they bear the burden of working with an awkward single-constructor type.[1]

Once accessibility has been proven, the user must rewrite all functions in a particular form. Assume that the user's original function is in the following form

```
Fixpoint f (a:A) : (rt a) :=
   ... f a' ...
```

where `rt` is some function which computes the return `type` for a particular argument `a`, and where `a'` is some subexpression. The user must rewrite this function as:

```
Fixpoint f (a:A)
           (f':forall a':A, a'<a -> rt a')
         : (rt a) :=
   ... f' a' ...
```

Note the extra `f'` argument is passed to the function, and it is this argument (rather than the function itself) which is applied to the recursive argument. Also note that `f'` takes only a single argument, while `f` takes two.

Once the user's function has been rewritten in this manner, the library's `well_founded_induction` function can be used to transform the rewritten function and accessibility proof into a closed form function of type `forall a:A, rt a`.

The example from the previous section would be rewritten as follows:

```
(* pre-packaged proof lt_wf:(well_founded lt) *)
Require Import Coq.Arith.Wf_nat.

Definition pow2_wf0 :
  forall n : nat,
          (forall y : nat, y < n -> nat) -> nat.
  refine (
    fun n => fun pow2_wf =>
      match n with
      | 0      => 1
      | (S n') =>
        match even_odd_dec (S n') with
        | left  _ => square (pow2_wf
                              (divide_by_two
                               (S n')) _)
        | right _ => 2 * (pow2_wf n' _)
        end
      end
  ).
  (* proof omitted *)

Definition pow2_wf :=
   @well_founded_induction
        nat lt lt_wf (fun _ => nat) pow2_wf0.
```

In reality the development is more complex – the omitted proof forces the body of the `refine` tactic to be rewritten using quite a few `return` clauses. Also, had the relation not been the less-than relation on `nat`s, the user would have had to prove `well_founded` manually.

Perhaps the most serious shortcoming, however, is the fact that the implementation of the function is entangled with its termination argument. It is not possible to write the function first, and then worry about termination later, which is necessary in order to be able to mechanically import and export code written in other languages.

Several other techniques act as "front ends" to this library (Coq's `RecursiveDefinition` and others), and indeed the inner machinery of the `eval` function introduced in section 5 has a very similar structure.

2.5 Extensions to the Type Theory

Another branch of work involves directly extending the type theory to include *partial objects* [CS87, CS93, Smi88, Smi95, Aud91, GPZ99]. However, this technique restricts recursion to *admissible* types.

Dybjer [Dyb00] suggests a more modest change, adding the ability to introduce *simultaneous inductive-recursive definitions*. This technique makes it possible to define a recursive function over an inductive predicate in which the definition of the inductive predicate depends upon the function.

[1] An important advantage of the technique presented in this paper is that the single-constructor type is completely concealed from the user, and is used only internally to pass proofs between the `Terminates` predicate and `eval` function.

2.6 Representing Computations as Coinductive Values

In [Cap05], Capretta introduced a coinductive type for representing potentially-nonterminating computations. The type proposed in that paper is essentially equivalent to:

```
CoInductive Computation (A:Set) : Type :=
  | Return  :  A -> #A
  | Step    : #A -> #A
    where "# A" := (Computation A).
```

Capretta's paper is focused primarily on analysis of program *execution traces*, for which these constructors are quite well-suited. However expressing programs using this datatype can be difficult, particularly when branching computations are involved.

This paper extends Capretta's type to one whose constructors form a monad directly. As will be shown in the next section, the new type for the second constructor is able to support *nested recursion* directly.

3. The Coinductive Computation Monad

This paper proposes representing computations as values of the following coinductive type:

```
CoInductive Computation (A:Set) : Type :=
  | Return  : A              -> #A
  | Bind    : (A->#A) -> #A -> #A.
    where "# A" := (Computation A).
```

The `Return` constructor is used to simply lift a normal value into the trivial computation which always terminates with that value.

The `Bind` constructor is used in the same way as the `>>=` operator in Haskell – it takes a computation and a function which produces a second computation from the first computation's return value. For example, the following computation is equivalent to `Return 5`:

```
Bind (fun x => Return (x+2)) (Return 3)
```

The following notation will be used as a convenience:

```
Notation "var <- c ; rest" :=
  (Bind (fun var => rest) c)
  (at level 60, right associativity).

(* ocasionally required to satisfy productivity *)
Notation "'call' c" :=
  (c >>= (fun x=>(Return x)))
  (at level 60, right associativity).
```

Using this new monad and notation, the fast exponentiation program can be written as:

```
CoFixpoint pow2 (n:nat) : #nat :=
  match n with
  | 0      => Return 1
  | (S n') =>
    match even_odd_dec (S n') with
    | left  _ => x <- pow2 (divide_by_two (S n'))
               ; Return (square x)
    | right _ => x <- pow2 n'
               ; Return (x*2)
    end
  end.
```

Coq will accept this definition, and we can use the `bounded_eval` function to test the result of evaluating this function to a particular number of evaluation steps:

```
Eval compute in (bounded_eval 10 (pow2 8)).
(* Some 256 *)
```

The use of a *co*-inductive type serves two purposes:

- It permits the representation of infinitely long computations as infinitely large coinductive objects.
- It lets the user write a function which directly references itself, without the syntactic restrictions of `Fixpoint`.

4. Termination Proofs

We can prove that a particular computation terminates by demonstrating the existence of an *inductive* object whose size bounds the length of the computation. Because inductive object must be of finite size, the existence of such an object is proof that the corresponding computation is of finite length.

4.1 A First Attempt

A first attempt at such a type is given below:

```
Inductive Terminates : #A -> Prop :=
| TerminateReturn :
    forall (a:A),
      Terminates (Return a)

| TerminateBind :
   forall (f:A->#A) (c:#A),
     Terminates c
     -> (forall (a:A), Terminates (f a))
     -> Terminates (Bind f c)
```

The two constructors (`TerminateReturn` and `TerminateBind`) correspond to the two constructors of the `Computation` (`#`) type.

The first constructor, (`TerminateReturn c`), can be used for any `c` which is a `Return`. This is simply stating that every `Return` terminates, and no additional information is required in order to produce a proof of its termination.

The second constructor, (`TerminateBind f c`), is more complex. It states that in order to prove that (`Bind f c`) terminates, we must prove:

1. That `c` terminates
2. For all values `a` in the domain of `f`, the computation (`f a`) terminates

4.2 Problems

This inductive predicate does in fact accomplish its task – it proves that the corresponding computation terminates. However, the condition is too strong. Consider this function:

```
CoFixpoint diverge_on_odd_numbers (n:nat) : #nat :=
 call
  match n with
    | 0         => Return 0
    | 1         => diverge_on_odd_numbers 1
    | (S (S n)) => diverge_on_odd_numbers n
  end.
```

Note that this function terminates for even arguments and diverges for odd arguments. An attempt to prove the following fairly straightforward theorem runs in to problems, however:

```
Theorem two_terminates :
  Terminates ((Return 2) >>= diverge_on_odd_numbers).
```

The result is the unsatisfiable proof obligation

```
forall a : nat, Terminates (diverge_on_odd_numbers a)
```

Clearly this goal is false when `a=1`, for example.

4.3 Weakening the Termination Condition

The problem is that the requirements for proving that `(Bind f c)` terminates are too strong. It is not necessary to prove that `(f a)` terminates *for all* `a` – we only need to prove that it terminates *for values of* `a` *which could have been produced by* `c`. This property is described as an *inductive invariant* in [KM03]:

> *The inductive invariants we introduced in this paper are specific invariants of the functional. They are predicates on $A \Rightarrow B$ that are defined by input-output relations (predicates on $A \times B$). We have found that the extra information about the function being defined that needs to be known in order to prove termination is invariably in the form of an invariant relation.*

Indeed, after extending the computation structure to include nested recursion as a first-class concept, proving termination of the "outer" nested call often requires proving facts about the *value* returned by the "inner" nested call.

This concept can be formalized with an auxiliary predicate called `TerminatesWith c a`, which acts as a proof that computation `c` (of type `#A`) terminates *and furthermore, terminates with the value* `a` (of type `A`).

```
Inductive TerminatesWith : #A -> A -> Prop :=
| TerminateReturnWith :
  forall (a:A),
    TerminatesWith (Return a) a

| TerminateBindWith :
  forall (a:A) (a':A) (f:A->#A) (c:#A),
    (TerminatesWith c a)
    -> TerminatesWith (f a) a'
    -> TerminatesWith (Bind f c) a'
```

This type is fairly straightforward; `Return a` always terminates with `a`, and `(Bind f c)` terminates with `a` if `(f a')` terminates with `a` and `c` terminates with `a'`.

Armed with this new auxiliary predicate, it is possible to rephrase the weakened termination condition:

```
Inductive Terminates : #A -> Prop :=
| TerminateReturn :
  forall (a:A),
    Terminates (Return a)

| TerminateBind :
  forall (f:A->#A) (c:#A),
    Terminates c
    -> (forall (a':A),
          (TerminatesWith c a')
          -> Terminates (f a'))
    -> Terminates (Bind f c)
```

Note in particular the last hypothesis of the second constructor. This type indicates that in order to prove that `(Bind f c)` terminates, one must prove that *for every value a' which* `c` *could possibly have terminated with,* `(f a')` *terminates.*

It is interesting to note that the `TerminatesWith` predicate never appears as a proof obligation for users of the predicate. Because it occurs negatively in the `TerminateBind` constructor, `TerminatesWith` appears only as a *hypothesis* for the user to take advantage of – typically by using `inversion` to prove `False` in situations where a value of `a'` has arisen which could not have been returned by `c`.

It should also be noted that the `Terminates` predicate is technically unnecessary in the presence of `TerminatesWith`: the predicate `(Terminates c)` logically equivalent to `(exists a, TerminatesWith c a)`, and each can be used to derive the other. Indeed, they exist as distinct predicates for largely historical and pedagogical reasons, and the `Terminates` predicate may be deprecated in future versions of the library.

5. Evaluating Computations

As shown earlier, a computation whose termination is dubious can be evaluated to a certain number of evaluation steps using the library's `bounded_eval` function:

```
bounded_eval : forall A:Set, nat -> #A -> option A
```

However, if a termination proof is available, one can instead coerce a computation of type `A->#A` to a simple function of type `A->A`. This is done using the `eval` function:

```
eval : forall (A:Set) (c:#A), Terminates c -> A
```

5.1 The `InvokedBy` Predicate

The `eval` function is perhaps the most interesting part of the library. Before diving into the details of how `eval` works, it is necessary to first explain a predicate `InvokedBy`, which is internal to the library – the user never encounters it.

Intuitively, `InvokedBy a b` means that `a` is a subcomputation of `b`. If `b` is in the form `Return _`, then it has no subcomputations, so one will never encounter a predicate of the form `InvokedBy _ (Return _)`. Rather, the `InvokedBy` predicate serves to enumerate the two possible subcomputations of a `Bind`. For any `c` and `f`,

- We know that `c` is a subcomputation of `Bind f c`; that is, `InvokedBy c (Bind f c)`
- Assuming `c` terminates with the value `a`, we know that `(f a)` is a subcomputation of `Bind f c`; that is, `InvokedBy (f a) (Bind f c)`.

The definition of `InvokedBy` simply formalizes this:

```
Inductive InvokedBy : #A -> #A -> Prop :=
| invokesPrev : forall
    (c:#A)
    (f:A->#A),
      InvokedBy c (c >>= f)
| invokesFunc : forall
    (c:#A)
    (f:A->#A)
    (a':A)
    (_:TerminatesWith c a'),
      InvokedBy (f a') (c >>= f).
```

The remainder of the development parallels quite closely the development of the well_founded_induction function in Coq.Init.Wf, and InvokedBy plays a role very similar to that of the relation (<).

5.2 The Safe Predicate

Once the user supplies a computation c:#A and a proof Terminates c, the library internally will use that termination proof to construct an object of type Safe c:

```
Inductive Safe : #A -> Prop :=
  Safe_intro :
    forall (c:#A),
       (forall (c':#A),
           InvokedBy c' c -> Safe c')
       -> Safe c.
```

Note the similarity between Safe and the Acc predicate from Coq.Init.Wf. The key feature to observe is that the Safe predicate has only one constructor; and therefore can be destructed in the process of creating a value in Set. This is precisely what eval does:

```
Theorem termination_is_safe :
  forall (A:Set) (c:#A) (t:Terminates A c), Safe A c.
   (* omitted *)

Notation "! c" :=
    {a:A|TerminatesWith c a} (at level 5).
Definition eval' C c (s:Safe C c)
    : {a:C|TerminatesWith c a}.
   (* omitted *)

Definition eval (A:Set) (c:#A) (t:Terminates c)
           : A :=
  match eval' c (termination_is_safe A c t) with
    | exist x pf => x
  end.
```

Utilizing the eval function, one can execute a computation "to completion" provided the termination proof:

```
Definition pow2_eval (n:nat) :=
  eval (pow2 n) (pow2_terminates n).

(* note return type is nat, not #nat *)
Check pow2_eval.
(* pow2_eval : nat -> nat *)

Eval compute in (pow2_eval 4).
(* 16 *)
```

5.3 Extracting Code

Extracting the function pow2_eval gives the following:

```
eva' :: (Computation a1) -> a1
eva' c =
  case c of
    Return x -> x
    Bind f cn -> eval (f (eval cn))

pow2 :: Nat -> Computation Nat
pow2 n =
  case n of
    O -> Return (S O)
    S n' ->
      case even_odd_dec (S n') of
       Left -> Bind
                (\x -> Return (square x))
                (pow2 (divide_by_two (S n')))
       Right -> Bind
                (\x -> Return (mult x (S (S O))))
                (pow2 n')

pow2_eval :: Nat -> Nat
pow2_eval n =
  eval (pow2 n)
```

It is fairly easy to see that the only difference between the programmer's intent and the extracted function is the Bind and Return constructors.

Indeed, if these constructors are (syntactically) removed and local beta reduction is performed on what is left, the results are *precisely* what the programmer wrote. It is reasonable to suspect that a more sophisticated extraction could remove them to produce a program which exactly matches the user's intent.

6. Examples

This section details two simple examples which apply the technique.

6.1 Euclid's GCD Algorithm

Consider a simple definition of Euclid's GCD function:

```
(* Euclid's GCD Function *)
CoFixpoint euclid_gcd (a b:nat) : #nat :=
  match a with
  | O      => Return b
  | (S a') =>
    match b with
    | O      => Return a
    | (S b') =>
      match le_gt_dec a b with
      | left  pf => call (euclid_gcd a     (b-a))
      | right pf => call (euclid_gcd (a-b) b    )
      end
    end
end.
```

The termination proof for this function is easy with the help of a general double strong induction lemma:

```
Theorem double_strong_induction :
  forall (P:nat->nat->Prop),
    (P 0 0)
    -> (forall (a b:nat),
         (forall (a' b':nat),
                ((a'<a /\ b'<=b)
                 \/(a'<=a /\ b'<b))
                   ->(P a' b'))
       -> P a b)
    -> (forall (a b:nat), P a b).
(* proof omitted *)
```

Using this lemma, the termination proof is quite short:

```
Theorem euclid_gcd_terminates :
  forall a b:nat, Terminates (euclid_gcd a b).

  apply double_strong_induction.

  (* uncomp is a tactic which acts like unfold *)
  uncomp euclid_gcd; constructor.

  intros; uncomp euclid_gcd.
  destruct a;
      try destruct b;
      try constructor.
  destruct (le_gt_dec (S a) (S b));
    constructor;
    try apply H;
    try omega;
    intros;
    constructor.
Qed.
```

Termination for this program can be demonstrated easily using most of the techniques from section 2 because care was taken when writing the GCD function to ensure that the recursion did not reverse the order of the arguments – even though reversing them would not affect the termination or correctness of the function.

However, had the function been written by a programmer who did not have provability in mind at the time, the result might have been a program like this:

```
(* Euclid's GCD Function *)
CoFixpoint euclid_gcd (a b:nat) : #nat :=
  match a with
  | 0      => Return b
  | (S a') =>
    match b with
    | 0      => Return a
    | (S b') =>
      match le_gt_dec a b with

                 (* note reversal of arguments *)
      | left  pf => call (euclid_gcd (b-a) a    )
      | right pf => call (euclid_gcd (a-b) b    )
      end
    end
end.
```

Nonetheless, the termination of this "reversed" GCD function can be established with the same proof shown above, simply by using a stronger induction lemma:

```
Theorem double_strong_commutative_induction :
  forall (P:nat->nat->Prop),
    (P 0 0)
    -> (forall (a b:nat),
         (forall (a' b':nat),
                 ((a'<a /\ b'<=b)
                  \/(a'<=a /\ b'<b)
                  \/(a'<b  /\ b'<=a)
                  \/(a'<=b /\ b'<a))
                    ->(P a' b'))
        -> P a b)
    -> (forall (a b:nat), P a b).
(* proof omitted *)
```

6.2 McCarthy's Function

Termination of the previous example can be established just as easily using most of the other techniques described in section 2. However, some functions are more difficult. One example function whose termination is difficult to prove using metric-based methods is McCarthy's Function [Knu77], defined over the naturals as:

$$M(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ M(M(n+11)) & \text{if } n \leq 100 \end{cases}$$

The actual behavior of this function is that it returns 91 for all $n \leq 101$ and $n - 10$ for all $n > 101$. However, its termination is quite difficult to prove by standard methods, chiefly due to the use of nested recursion.

Examples such as this are easy to formalize using the coinductive monad technique described earlier. First, we define the partial function in Coq:

```
CoFixpoint mccarthy (n:nat) : #nat :=
  match le_gt_dec n 100 with
    | left  _ => n' <- mccarthy (11+n)
               ; mccarthy n'
    | right _ => Return (n-10)
  end.
```

Note that `le_gt_dec` is the Coq library function to decide if one natural is less than another; it returns `left` if this is the case and `right` otherwise, supplying a proof in either case.

Unlike other techniques, the proof of termination using a coinductive monad can follow the conventional prose argument, which is approximately this: we can show by *downward* induction that for $90 \leq n \leq 100$, $M(n) = M(n+1)$ (taking $M(100) = M(101)$ as the base case). By a second induction we can show that $M(n) = 91$ over this range. By a third downward induction we can show that $M(n) = 91$ holds for each chunk of eleven integers less than 100, using the initial chunk $90 \leq n \leq 100$ as the base case. Therefore the function terminates for $n \leq 100$. Termination for $n > 100$ is immediate from the definition of the function.

A formalization of this fact involves five lemmas, each of which corresponds exactly to one of the five previous sentences; thus the formalization takes exactly the same shape as the most straightforward verbal proof:

```
Lemma mccarthy_is_m_of_n_plus_1_for_90_n_100 :
  forall n k:nat,
    90 <= n <= 100
    -> TerminatesWith (mccarthy (n+1)) k
    -> TerminatesWith (mccarthy (n  )) k.

Lemma mccarthy_n_is_91_for_90_n_100 :
  forall n:nat,
    90 <= n <= 100
    -> TerminatesWith (mccarthy n) 91.

Lemma mccarthy_n_is_91_for_blocks_of_11 :
  forall k:nat,
    100 > k*11
    -> forall n:nat,
      90-k*11 <= n <= 100-k*11
      -> TerminatesWith (mccarthy n) 91.
```

```
Lemma mccarthy_terminates_for_n_le_100 :
  forall n:nat,
     n <= 100
     -> TerminatesWith (mccarthy n) 91.

Lemma mccarthy_terminates_for_n_gt_100 :
  forall n:nat,
     n > 100
     -> Terminates (mccarthy n).

Theorem mccarthy_terminates_for_all_n :
  forall n:nat,
     Terminates (mccarthy n).
```

7. Advanced Applications

7.1 Composing Computations over Different Sets

The definition of the coinductive type presented in section 3 is actually a simplified version of the full library definition; the complete definition is more polymorphic and includes a bind operator which is capable of composing two computations over different result sets:

```
CoInductive Computation (A:Set) : Type :=
  | Return   :                    A            -> #A
  | Bind     : forall (B:Set), (B->#A) -> #B -> #A.
    where "# A" := (Computation A).
```

Note that there is no size issue here: #A belongs to Type, and therefore may quantify over Sets such as B. However, this construction does considerably complicate the proof that Terminates is sufficient to establish the safety of eval, and seems to require the JMeq axiom [McB00]. Fortunately, this complicated proof is completely encapsulated within the library, and users of the library need not be aware of it.

7.2 Higher-Order Computations

The updated definition makes it possible to define higher-order operations on computations; for example, the fold operation with a computation-producing function:

```
CoFixpoint foldc
    (A B:Set)(la:list A)(b:B)(f:A->B->(#B)) : #B :=
  match la with
    | nil            => Return b
    | (cons a la')   => b' <- f a b
                      ; foldc A B la' b' f
  end.
```

7.3 First-Class Termination Proofs

Making termination proofs first class objects opens up the possibility of termination proofs paramterized over computations. This makes it possible for a functional to be packaged with a termination proof parameterized over its argument and a proof that its argument terminates under suitable circumstances.

For example, the foldc function from the previous section might be packaged with the following lemma, which offers easier proof obligations when termination condition is immune to duplication, rearrangement, and deletion of list elements and independent of the accumulative argument.

```
Lemma foldc_termination :
  forall
    (A B:Set)
    (la:list A)
    (b0:B)
    (f:A->B->#B),

    (forall (a:A)(b:B),
      (In a la) ->
      (Terminates (f a b)))

    -> Terminates (foldc la b0 f).
```

The lemma above (whose proof is included in the library) states that for any function f of two arguments, foldc list b f will terminate if it can be shown that f terminates whenever its first argument is drawn from list. Clearly this is a sufficient but not necessary condition for termination, but it may be much easier to prove.

7.4 Dependently Typed Computations

This section closes by noting that nothing prevents the use of dependently typed computations. Consider the following simple implementation of exponentiation, and a simple arithmetic lemma about its behavior:

```
Fixpoint slow_pow2 (n:nat) : nat :=
  match n with
    | 0      => 1
    | (S n') => 2*(slow_pow2 n')
  end.

(* (2^((2+n)/2))^2 = 2^(n+2) if n is even *)
Lemma exponentiation_lemma : forall (n:nat),
  even n ->
    square (slow_pow2 (divide_by_two (S (S n))))
    = slow_pow2 (S (S n)).
  (* proof omitted *)
```

Below is an example of a version of pow2 which uses Coq's specification predicates and syntax to define a dependently-typed computation whose type strongly specifieds its partial correctness with respect to the simpler slow_pow2 implementation.

```
Definition pow2 :
  forall (n:nat), #{x|x=(slow_pow2 n)}.

  refine (
  cofix pow2(n:nat) : #{x|x=(slow_pow2 n)} :=
    match n return #{x|x=(slow_pow2 n)} with
     | 0      => Return (exist _ 1 _)
     | (S n') => match even_odd_dec (S n') with
          | left  pf =>
               x <- pow2 (divide_by_two (S n'))
               ; match x with
                 | exist x' pf' =>
                   Return (exist _ (square x') _)
                 end
          | right pf => x <- pow2 n'
                ; match x with
                  | exist x' pf' =>
                    Return (exist _ (2*x') _)
                  end
          end
     end).
  simpl; auto.
  induction n';
    inversion pf; inversion H0; subst.
  apply lemma; auto.
  simpl; omega.
Qed.
```

8. Comparison to [Cap05]

As [Cap05] is the work most closely related to this paper, the following section examines the differences in detail.

Capretta's monad can be directly applied to non-nested recursion. For dealing with nested recursion, that paper offers three approaches.

The first approach, as proven by Theorem 4.2 of [Cap05], shows that any recursive function can be represented in type theory, but must be extensively manipulated. These manipulations alter the structure of the program and make subsequent proofs about its properties much more difficult.

The second approach called "the devil's nest." This approach requires that recursive functions be rephrased in terms of an auxiliary combinator $\mathsf{devil}_{\mathsf{aux}}$, a form which is often unnatural. Additionally, in the form given in the paper and its accompanying Coq formalization [Cap07] cannot handle functions which involve a variable number of nestings, although the paper suggests a modification to deal with this case. It is not clear there is a way to express mutually recursive functions and higher-order recursive functions without unnatural transformations of the code.

The third approach involves creating a monad not from the constructors of the coinductive type, as this paper does, but rather by defining them as coinductive functions over the type of computations.

The formalization [Cap07] which accompanies [Cap05] declares its coinductive type like this:

```
CoInductive Partial: Set :=
  rtrn : A -> Partial
| step : Partial -> Partial.

Inductive Value : Partial -> A -> Prop :=
  value_return : forall a:A, Value (rtrn a) a
| value_step   : forall (x:Partial)(a:A),
                   Value x a -> Value (step x) a.
```

And the paper describes the monad's *bind* operator as:

$$(f^* x) = \mathsf{Cases}\ \ x\ \ \mathsf{of} \begin{cases} \ulcorner a \urcorner & \mapsto\ \ (f\ \ a) \\ \triangleright x' & \mapsto \triangleright(f^* x') \end{cases}$$

Unfortunately, this combinator is the only component of the paper which is absent from the formalization. The proper formalization should be something along the lines of:

```
CoFixpoint bind
   (A B:Set)(f:A->Partial B)(x:Partial A)
   : Partial B :=
  match x with
    | rtrn a  => f a
    | step x' => step (bind f x')
  end.
```

Indeed, it is possible to prove a theorem which is essentially equivalent to the TerminateBind constructor:

```
Theorem capretta_termination_proofs_compose :
 forall (A B:Set)
        (pa:Partial A)(a:A)
        (pb:A -> Partial B)(b:B),
        (Value pa a) ->
        (Value (pb a) b) ->
        (Value (bind pb pa) b).
```

But unfortunately it is *not* possible to write programs with unrestricted recursion. For example, the McCarthy function presented earlier would be written as:

```
CoFixpoint mccarthy (n:nat) : Partial nat :=
  match le_gt_dec n 100 with
    | left  _ => bind mccarthy (mccarthy (11+n))
    | right _ => rtrn (n-10)
  end.
```

Unfortunately this program does not meet the criteria that "each recursive call in the definition must be protected by at least one constructor, *and only by constructors*" [Coq06] in order to ensure productivity. Indeed, no amount of step-insertion will fix this problem, which is fundamental because bind *is not a constructor*. This problem appears to be fundamental to the monad-as-functions approach.

9. Conclusion and Future Work

One of the major disadvantages to the use of coinductives is that Coq currently will not automatically reduce the application of a cofix expression, nor will the cbv tactic force such a reduction. The usual workaround, which involves a "decomposition equality theorem" is cumbersome to use, although for most common cases this can be automated with an Ltac script. Hopefully future versions of Coq will extend the built-in reduction mechanism to eliminate the need for such workarounds.

An additional disadvantage of this paper's technique is that computations which produce identical results are often not equal under Coq's notion of equality. Using the coinductive monad representation, they are only equal when their execution traces are *bisimilar*, which limits the use of the rewrite tactic in most cases. Furthermore, in order to satisfy the monad laws [Wad93] a weaker notion of equality must be introduced,[2] namely

```
Inductive computationally_equivalent
  (A:Set)(ca1:#A)(ca2:#A)
  : Prop :=

  | computationally_equivalent_intro :
    (forall a,
        TerminatesWith ca1 a
        <->
        TerminatesWith ca2 a)

  where "x =#= y" := (computationally_equivalent x y)
                     (at level 200).
```

The proof that the Terminates predicate is a sufficient condition for safe use of eval requires the JMeq axiom [McB00] when the two-set-polymorphic version of the monad from Section 7.1 is

[2] note that, as of this writing, proofs of the monad laws and that =#= is an equivalence relation are not yet included in the library

used. It may be possible to substitute `-fimpredicative-set` for this axiom.

As mentioned in the abstract, the long-term goal of this work it to use Coq for purposes similar to [ABB+05] – that is, to be able to mechanically import programs into the language of the theorem prover, manually strengthen their specifications, and then mechanically extract the newly-certified programs without loss of efficiency.

The Coq library described in this paper can be downloaded from:

`http://www.cs.berkeley.edu/~megacz/computation/`

Acknowledgments

I would like to thank Adam Chlipala for his helpful comments and guidance in the early stages of this paper. I would also like to thank the reviewers for their helpful and exceptionally detailed comments.

References

[ABB+05] Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. Verifying haskell programs using constructive type theory. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 62–73, New York, NY, USA, 2005. ACM Press.

[Aud91] Philippe Audebaud. Partial objects in the calculus of constructions. In *Proceedings 6th Annual IEEE Symp. on Logic in Computer Science, LICS'91, Amsterdam, The Netherlands, 15–18 July 1991*, pages 86–95. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[BC01] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. *Lecture Notes in Computer Science*, 2152:121+, 2001.

[Bov01] Ana Bove. Simple general recursion in type theory. *Nordic J. of Computing*, 8(1):22–42, 2001.

[Cap05] Venanzio Capretta. General recursion via coinductive types. *LMCS-1*, 2:1, 2005.

[Cap07] Venanzio Capretta. Formalization accompanying *general recursion via coinductive types* (`rec_coind.v`). Downloaded 31-July-2007, 2007.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[Coq06] Development Team For Coq. *The Coq Proof Assistant Reference Manual*. LogiCal Project, 2006. Version 8.1.

[CS87] Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *LICS*, pages 183–193. IEEE Computer Society, 1987.

[CS93] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Theoretical Computer Science*, 121(1–2):89–112, 1993.

[Dyb00] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log*, 65(2):525–549, 2000.

[GBR06] David Pichardie Gilles Barthe, Julien Forest and Vlad Rusu. Defining and reasoning about recursive functions: a practical tool for the coq proof assistant. In *Proc. of 8th International Symposium on Functional and Logic Programming (FLOPS'06)*, number 3945 in Lecture Notes in Computer Science, pages 114–129. `http://www.springer.de/comp/lncs/index.html`Springer-Verlag, 2006.

[Gim95] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *TYPES '94: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 39–59, London, UK, 1995. Springer-Verlag.

[Gim98] Eduardo Gimenez. Structural recursive definitions in type theory. In *Automata, Languages and Programming*, pages 397–408, 1998.

[GPZ99] Herman Geuvers, Erik Poll, and Jan Zwanenburg. Safe proof checking in type theory with y. In *CSL*, pages 439–452, 1999.

[KM03] S. Krstic and J. Matthews. Inductive invariants for nested recursion, 2003.

[Knu77] D. E. Knuth. Algorithms. 236(4):63–66, 69–72, 79–78, 80, April 1977.

[McB00] Conor McBride. Elimination with a motive. In *TYPES*, pages 197–216, 2000.

[NB03] M. Niqui and Y. Bertot. Qarith: Coq formalisation of lazy rational arithmetic, 2003.

[Nik91] R. S. Nikhil. Id language reference manual (version 90.1). Technical Report 284-2, 1991.

[Nor88] B. Nordström. Terminating general recursion. *BIT*, 28(3):605–619, 1988.

[Nor93] Bengt Nordström. The ALF proof editor. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 253–266, Nijmegen, 1993.

[She05] Tim Sheard. Putting curry-howard to work. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, New York, NY, USA, 2005. ACM Press.

[Smi88] Scott Fraser Smith. Partial objects in type theory. Technical Report TR88-938, 1988.

[Smi95] Scott F. Smith. Hybrid partial-total type theory. *Int. J. Found. Comput. Sci*, 6(3):235–263, 1995.

[Tur04] D. A. Turner. Total functional programming. 10(7):751–768, 2004. `http://www.jucs.org/jucs_10_7/total_functional_programming`.

[Wad93] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.