A Coinductive Monad for `Prop`-bounded Recursion

Adam Megacz
megacz@cs.berkeley.edu

PLPV'07
October 5th, 2007
Freiburg, Germany

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, reduction, equality

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
    - using built-in abstraction, reduction, equality
    - can prove properties of such functions

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, reduction, equality
  - can prove properties of such functions
  - can extract efficient implementations of such functions

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, reduction, equality
  - can prove properties of such functions
  - can extract efficient implementations of such functions
- **Goals:** we would like to

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, reduction, equality
  - can prove properties of such functions
  - can extract efficient implementations of such functions
- **Goals:** we would like to
  - Represent potentially-nonterminating computations in a manner which retains advantages above.

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, reduction, equality
  - can prove properties of such functions
  - can extract efficient implementations of such functions
- **Goals:** we would like to
  - Represent potentially-nonterminating computations in a manner which retains advantages above.
  - Allow optional termination proofs, which should:

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, reduction, equality
  - can prove properties of such functions
  - can extract efficient implementations of such functions
- **Goals:** we would like to
  - Represent potentially-nonterminating computations in a manner which retains advantages above.
  - Allow optional termination proofs, which should:
    - convert computations to functions.

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, reduction, equality
  - can prove properties of such functions
  - can extract efficient implementations of such functions
- **Goals:** we would like to
  - Represent potentially-nonterminating computations in a manner which retains advantages above.
  - Allow optional termination proofs, which should:
    - convert computations to functions.
    - be in `Prop`

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, reduction, equality
  - can prove properties of such functions
  - can extract efficient implementations of such functions
- **Goals:** we would like to
  - Represent potentially-nonterminating computations in a manner which retains advantages above.
  - Allow optional termination proofs, which should:
    - convert computations to functions.
    - be in `Prop`
    - be conventional (follow prose arguments)

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, reduction, equality
  - can prove properties of such functions
  - can extract efficient implementations of such functions
- **Goals:** we would like to
  - Represent potentially-nonterminating computations in a manner which retains advantages above.
  - Allow optional termination proofs, which should:
    - convert computations to functions.
    - be in `Prop`
    - be conventional (follow prose arguments)
    - not require advance planning; "after the fact"

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, reduction, equality
  - can prove properties of such functions
  - can extract efficient implementations of such functions
- **Goals:** we would like to
  - Represent potentially-nonterminating computations in a manner which retains advantages above.
  - Allow optional termination proofs, which should:
    - convert computations to functions.
    - be in `Prop`
    - be conventional (follow prose arguments)
    - not require advance planning; "after the fact"
- **This talk:** a coinductive type whose constructors are the operators of a monad.

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, reduction, equality
  - can prove properties of such functions
  - can extract efficient implementations of such functions
- **Goals:** we would like to
  - Represent potentially-nonterminating computations in a manner which retains advantages above.
  - Allow optional termination proofs, which should:
    - convert computations to functions.
    - be in `Prop`
    - be conventional (follow prose arguments)
    - not require advance planning; "after the fact"
- **This talk:** a coinductive type whose constructors are the operators of a monad.
  - Achieves goals above, except: sacrifices reduction and equality.

Entire Talk In One Slide

- Coq's type theory can directly represent side-effect free, obviously-terminating functions
  - using built-in abstraction, ~~reduction, equality~~
  - can prove properties of such functions
  - can extract efficient implementations of such functions
- **Goals:** we would like to
  - Represent potentially-nonterminating computations in a manner which retains advantages above.
  - Allow optional termination proofs, which should:
    - convert computations to functions.
    - be in `Prop`
    - be conventional (follow prose arguments)
    - not require advance planning; "after the fact"
- **This talk:** a coinductive type whose constructors are the operators of a monad.
  - Achieves goals above, except: sacrifices reduction and equality.

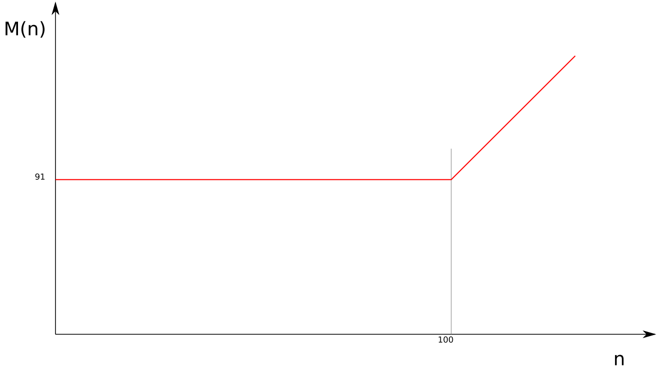
Running Example: McCarthy's Function

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

Running Example: McCarthy's Function

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

Actual behavior:



First Attempt: Direct Representation

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

```
Fixpoint mccarthy (n:nat) {struct n} : nat :=
  match le_gt_dec n 100 with
    | left  _ => mccarthy (mccarthy (11+n))
    | right _ => n-10
  end.
```

First Attempt: Direct Representation

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

```
Fixpoint mccarthy (n:nat) {struct n} : nat :=
  match le_gt_dec n 100 with
    | left  _ => mccarthy (mccarthy (11+n))
    | right _ => n-10
  end.
```

- problem: `11+n` is not structurally smaller than `n`

First Attempt: Direct Representation

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

```
Fixpoint mccarthy (n:nat) {struct n} : nat :=
  match le_gt_dec n 100 with
    | left  _ => mccarthy (mccarthy (11+n))
    | right _ => n-10
  end.
```

- problem: `11+n` is not structurally smaller than `n`
- problem: `mccarthy (11+n)` is not structurally smaller than `n`

First Attempt: Direct Representation

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

```
Fixpoint mccarthy (n:nat) {struct n} : nat :=
  match le_gt_dec n 100 with
    | left  _ => mccarthy (mccarthy (11+n))
    | right _ => n-10
  end.
```

- problem: `11+n` is not structurally smaller than `n`
- problem: `mccarthy (11+n)` is not structurally smaller than `n`
- problem: any decreasing metric will more complex than the function itself

Solution #1: Set-bounded recursion

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

```
Fixpoint mccarthy (  n:nat) {struct n} :         nat :=
    match le_gt_dec n 100 with
     | left  _ => mccarthy        (mccarthy    (11+n))
     | right _ =>       (n-10)
    end.
```

Solution #1: Set-bounded recursion

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

```
Notation "a =<< b" :=
     (match b with None => None | Some x => a x end)
     (at level 100).

Fixpoint mccarthy (d n:nat) {struct d} : option nat :=
  match d with
   | O       => None
   | (S d') =>
    match le_gt_dec n 100 with
     | left  _ => mccarthy d' =<< (mccarthy d' (11+n))
     | right _ => Some (n-10)
    end end.
```

Typical Solution: `Set`-bounded evaluation

- good: determination of recursion bound is separated from function definition

Typical Solution: `Set`-bounded evaluation

- good: determination of recursion bound is separated from function definition
- bad: recursion bound is in `Set`; will be included in extracted code

Typical Solution: `Set`-bounded evaluation

- good: determination of recursion bound is separated from function definition
- bad: recursion bound is in `Set`; will be included in extracted code

> *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.*
>
> *– Niqui and Bertot (2003)*

Other Approaches

- Domain predicate
  - multi-constructor type (in `Set`) [Bove, Capretta 2001]
  - often with Dybjer's simultaneous inductive-recursive definitions
- Accessibility predicate
  - single-constructor predicate (in `Prop`) [`Coq.Init.Wf`]
- Extensions to the type theory (Y, bar types, etc)

Proposed Solution: A Coinductive Computation Monad

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

Programming `mccarthy` Using the Monad

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

- Use of `CoFixpoint` circumvents usual syntactic check of recursive references.
- Making monad operators *constructors* of the coinductive type ensures generativity.

Termination Proofs

```
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'
```

Termination Proofs

```
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'
```

Termination Proofs

```
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'
```

Termination Proofs

```
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'
```

Termination Proofs

```
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'
```

Termination Proofs

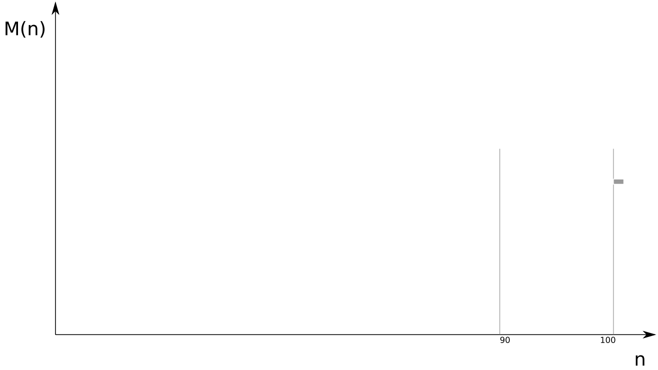
```
Inductive TerminatesWith : #A -> A -> Prop :=
| TerminateReturnWith :
  forall (a:A),
    TerminatesWith (Return a) a

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

Note that facts about the return value (a) of one computation (c) may be used in the termination argument for some other computation (`f a`).

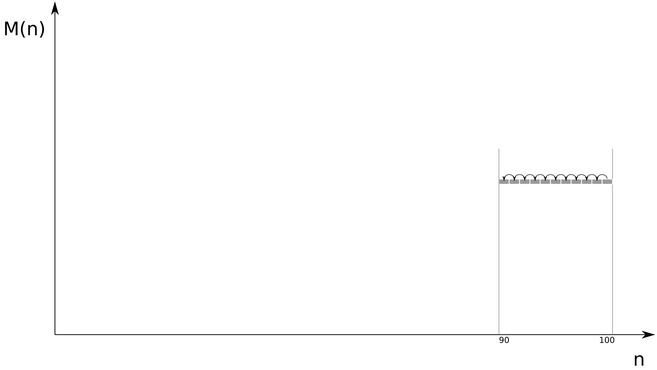
Base Case: `M(101)` terminates

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



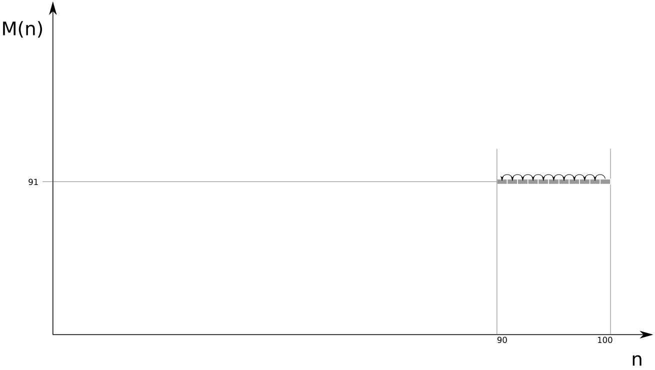
First Induction: $90 \leq n \leq 100 \Rightarrow M(n) = M(n+1)$

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



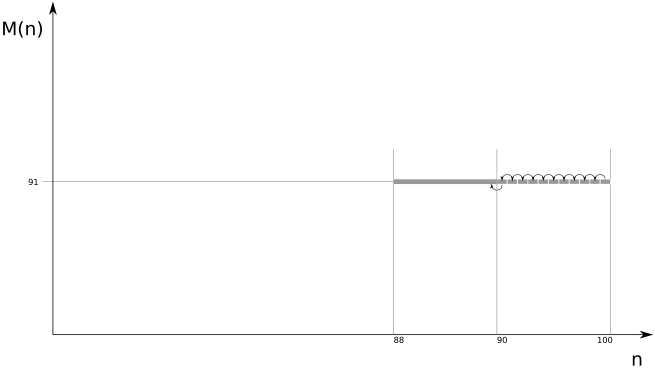
Second Induction: $90 \leq n \leq 100 \Rightarrow M(n) = 91$

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



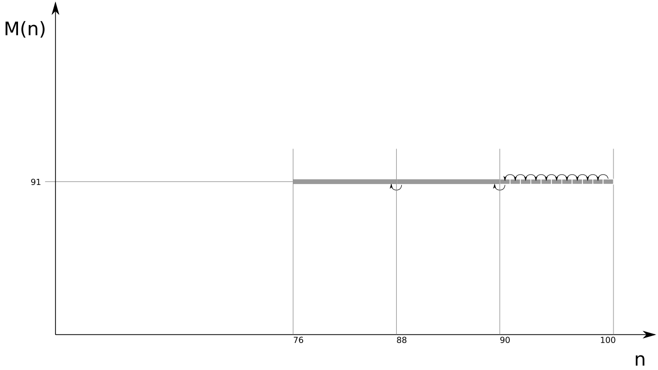
Third Induction: each “block of eleven” same

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



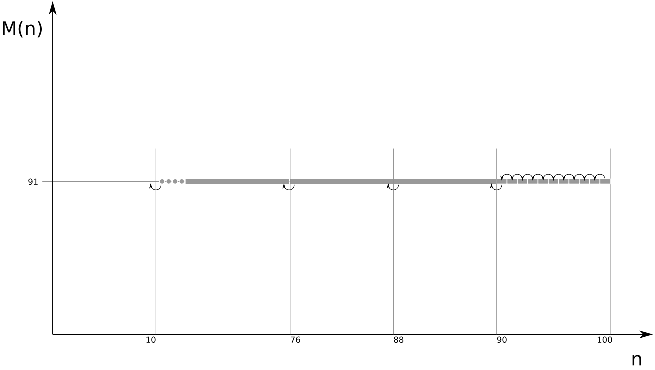
Third Induction: each “block of eleven” same

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



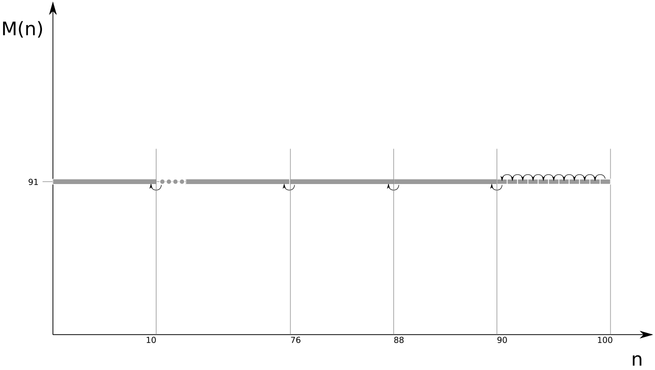
Third Induction: each "block of eleven" same

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



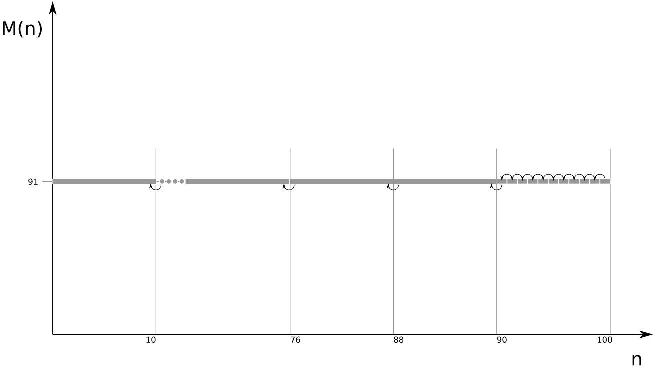
Third Induction: each “block of eleven” same

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



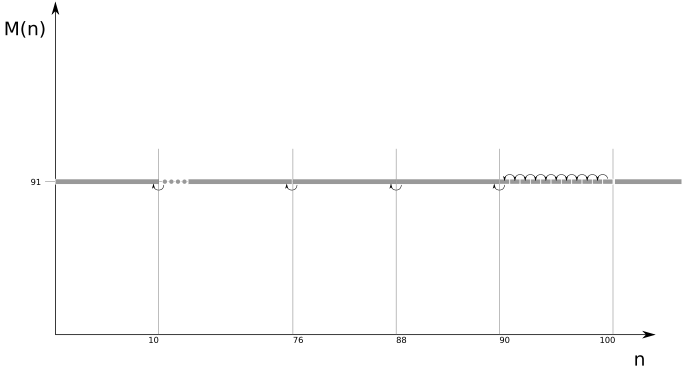
Therefore $M(n)$ terminates for $n \leq 100$

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



Termination for $n > 100$ is immediate

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



Summary of Termination Argument for `mccarthy`

- 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.

Summary of Termination Argument for `mccarthy`

- 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.

Unlike metric-based techniques, the proof of termination using a coinductive monad can follow the conventional prose argument.

Proving `forall n, Terminates (mccarthy n)`

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

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).

```
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.
```

Proving `forall n, Terminates (mccarthy n)`

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

By a second induction we can show that $M(n) = 91$ over this range.

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

Proving `forall n, Terminates (mccarthy n)`

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

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.

```
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.
```

Proving `forall n, Terminates (mccarthy n)`

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

Therefore the function terminates for $n \leq 100$.

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

Proving `forall n, Terminates (mccarthy n)`

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

Termination for $n > 100$ is immediate from the definition of the function.

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

Proving `forall n, Terminates (mccarthy n)`

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

Therefore, `mccarthy` is total.

```
Theorem mccarthy_terminates :
  forall n:nat,
     -> Terminates (mccarthy n).
```

Evaluation and Extraction

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

Functions produced by `eval` yield efficient extractions; the `Terminates` term (`Prop` bound) is completely omitted.

```
bounded_eval :
   forall (A:Set) (c:#A) (n:nat), option A
```

Summary So Far

- We can represent potentially-nonterminating computations
- We can write proofs about the properties (such as termination) of such computations,
  - Proofs can be used to convert a computation to a function (via `eval`)
  - Proofs are in `Prop`
  - Proofs are conventional (follow prose)
  - Proofs are "after the fact"
- Coq's extraction mechanism produces efficient code for applications of `eval`.

Prior work: [Capretta 2005]

Different encoding of computations as coinductive values; closer connection to operational semantics:

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


CoFixpoint bind (A B:Set)(f:A->#B)(x:#A) : #B :=
  match x with
    | Return a => f a
    | Step x'  => Step (bind A B f x')
  end.
```

Prior work: [Capretta 2005]

Different encoding of computations as coinductive values; closer connection to operational semantics:

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

- Disadvantage: `bind` is no longer a constructor, so programs with nested recursion fail the generativity requirement.

Prior work: [Capretta 2005]

Different encoding of computations as coinductive values; closer connection to operational semantics:

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

- Disadvantage: `bind` is no longer a constructor, so programs with nested recursion fail the generativity requirement.

Generalizing to Different Range/Domain

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

Generalizing to Different Range/Domain

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

- Proof of `eval` safety requires `JMeq` axiom [McB00] in this case

Mutual Recursion

```
CoFixpoint isEven (isOdd:nat->#bool) (n:nat) : #bool :=
  match n with
    | 0      => Return true
    | (S n') => x <- isOdd n';
                Return (negb x)
  end.

CoFixpoint isOdd (isEven:nat->#bool) (n:nat) : #bool :=
  match n with
    | 0      => Return false
    | (S n') => x <- isEven n';
                Return (negb x)
  end.

CoFixpoint isEven' := (kludge isEven) isOdd'
with       isOdd'  := (kludge isOdd)  isEven'
with       kludge  := (fun x=>x).
```

Higher-Order Computations

```
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.
```

First-Class Termination Proofs

```
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).
```

Thank you