Coinductive definitional interpreters using the delay monad

Xavier Leroy

Collège de France and Inria, Paris

IFIP WG 2.8, May 2019, Bordeaux

Mechanized semantics for little languages...

... are naturally expressed in denotational style.

```
Fixpoint den (e: expr) : Z :=
  match e with
  | Const n => n
  | Add e1 e2 => den e1 + den e2
  | Mul e1 e2 => den e1 * den e2
  end.
```

or, more realistically:

```
Fixpoint den (e: expr) : mon machine_integer :=
  match e with
  | Const n => ret n
  | Add e1 e2 =>
     bind (den e1) (fun v1 => bind (den e2) (fun v2 => madd v1 v2)
  | Mul e1 e2 =>
     bind (den e1) (fun v1 => bind (den e2) (fun v2 => mmul v1 v2)
  end.
```

Mechanized semantics for little languages…

… are naturally expressed in denotational style.

```
Fixpoint den (e: expr) : Z :=
  match e with
  | Const n => n
  | Add e1 e2 => den e1 + den e2
  | Mul e1 e2 => den e1 * den e2
  end.
```

or, more realistically:

```
Fixpoint den (e: expr) : mon machine_integer :=
  match e with
  | Const n => ret n
  | Add e1 e2 =>
     bind (den e1) (fun v1 => bind (den e2) (fun v2 => madd v1 v2)
  | Mul e1 e2 =>
     bind (den e1) (fun v1 => bind (den e2) (fun v2 => mmul v1 v2)
  end.
```

Mechanized semantics for non-normalizing languages

No such simple translation to the meta-language.

Usual approach: consider finite prefixes of possibly-infinite executions.

- Reduction semantics.
- Scott domains.
- Definitional interpreters with “fuel”.

This talk: ideas for an alternate approach, based on a corecursive definitional interpreter.

Partial computations in type theory

(V. Capretta, *General recursion via coinductive types*, LMCS(1), 2005)

```
CoInductive delay {A: Type} : Type :=
  | now: A -> delay A
  | later: delay A -> delay A.
```

`delay` A represents computations that return a value of type A or diverge.

The `later` constructor represents one step of computation.

With an inductive definition of `delay`, terms of `delay A` are $\texttt{later}(\cdots(\texttt{later}(\texttt{now}(v)))\cdots)$. We're just counting the number of computation steps.

With the coinductive definition of `delay`, we can also represent infinitely many computation steps, that is, a nonterminating computation.

Partial computations in type theory

```
CoInductive delay {A: Type} : Type :=
  | now: A -> delay A
  | later: delay A -> delay A.
```

Here is the canonical diverging computation at type A:

```
CoFixpoint bottom (A: Type) : delay A := later (bottom A).
```

Terminating computations are characterized by an inductive predicate, diverging computations by a coinductive predicate.

```
Inductive terminates {A: Type} : delay A -> A -> Prop :=
  | terminates_now:
      forall v, terminates (now v) v
  | terminates_later:
      forall a v, terminates a v -> terminates (later a) v.

CoInductive diverges {A: Type} : delay A -> Prop :=
  | diverges_later:
      forall a, diverges a -> diverges (later a).
```

Partial computations in type theory

```
CoInductive delay {A: Type} : Type :=
  | now: A -> delay A
  | later: delay A -> delay A.
```

Here is the canonical diverging computation at type `A`:

```
CoFixpoint bottom (A: Type) : delay A := later (bottom A).
```

Terminating computations are characterized by an inductive predicate, diverging computations by a coinductive predicate.

```
Inductive terminates {A: Type} : delay A -> A -> Prop :=
  | terminates_now:
      forall v, terminates (now v) v
  | terminates_later:
      forall a v, terminates a v -> terminates (later a) v.

CoInductive diverges {A: Type} : delay A -> Prop :=
  | diverges_later:
      forall a, diverges a -> diverges (later a).
```

General recursion

We can define arbitrary general recursive functions with result type `delay A`, provided that all recursive calls are guarded by a `later` constructor.

✘
```
Fixpoint modulus (a b: N) : N :=
   if a <? b then a else modulus (a - b) b.
```

✘
```
CoFixpoint modulus (a b: N) : delay N :=
   if a <? b then now a else modulus (a - b) b.
```

✔
```
CoFixpoint modulus (a b: N) : delay N :=
   if a <? b then now a else later (modulus (a - b) b).
```

Reminder: recursion vs. corecursion

Recursive function definition (`Fixpoint`):

- Argument has an inductive type.
- `f x` can recursively call `f y` provided `y` is a strict sub-term of `x`.

Corecursive function definition (`CoFixpoint`):

- Result has a coinductive type.
- `f x` can recursively call `f y` provided `f y` is a strict sub-term of `f x`.

(A.k.a. the productivity condition: the head constructor of `f x` can always be computed in finite time.)

General recursion

```
CoFixpoint modulus (a b: N) : delay N :=
    if a <? b then now a else later (modulus (a - b) b).
```

We can reason about termination or divergence of the function after we've defined it.

```
Theorem modulus_Euclid:
  forall a b, b > 0 ->
  exists q r, terminates (modulus a b) r ∧ r < b ∧ a = b*q+r.

Theorem modulus_divergence:
  forall a, diverges (modulus a 0).
```

General recursion

Another example where we literally have no clue when the function terminates, yet we can define it.

```
CoFixpoint Collatz (n: N): delay unit :=
  if n =? 1 then now tt
  else if N.even n then later (Collatz (n / 2))
  else later (Collatz (3 * n + 1)).

Conjecture Collatz_1:
  forall n, n >= 1 -> terminates (Collatz n) tt.

Conjecture Collatz_2:
  exists n, n >= 1 ∧ diverges (Collatz n).
```

Observational equivalence

A constructive definition of equitermination:

```
CoInductive equi {A: Type} : delay A -> delay A -> Prop :=
  | equi_terminates: forall x y v,
      terminates x v -> terminates y v -> equi x y
  | equi_later: forall x y,
      equi x y -> equi (later x) (later y).
```

Classically equivalent to

$$(\exists v,\ \texttt{terminates}\, x\, v \land \texttt{terminates}\, y\, v) \ \lor\ (\texttt{diverges}\, x \land \texttt{diverges}\, y)$$

but constructively stronger. (No need to “know in advance” whether both computations diverge or both terminate.)

The delay monad

`delay` is a monad, with `now` as the unit operation, and the bind operation being the sequencing of two computations:

```
CoFixpoint bind {A B: Type}
                (a: delay A) (f: A -> delay B) : delay B :=
  match a with
  | now v => later (f v)
  | later a' => later (bind a' f)
  end.
```

We have the expected properties of sequencing, e.g. `bind a f` diverges iff `a` diverges or `a` terminates on `v` and `f v` diverges.

The three monadic laws hold, up to `equi`:

```
equi (bind (now v) f) (f v)
equi (bind a now) a
equi (bind (bind a f) g) (bind a (fun x => bind (f x) g))
```

A definitional interpreter in the delay monad

Consider lambda-calculus with constants:

```
Inductive term : Type :=
  | Const (n: Z)
  | Var (x: var)
  | Lam (x: var) (a: term)
  | App (a b: term).
```

Can we define a definitional interpreter as a function

```
CoFixpoint eval (a: term) : delay (option term) := ...
```

(`option` because terms can get stuck).

Productivity problem

```
  CoFixpoint eval (a: term) : delay (option term) :=
    match a with
    | Const n => now (Some (Const n))
    | Var x => now None
    | Lam y b => now (Some (Lam y b))
    | App b c =>
✘     bind (eval b) (fun r =>
          match r with
          | Some (Lam x d) => eval (subst x c d)
          | _, _ => now None
          end))
    end.
```

`eval b` is not a strict sub-term of `eval a`. Hence not productive!

The free monad to the rescue!

(A use of the trick described by N. A. Danielsson in *Beating the Productivity Checker Using Embedded Languages*, 2010.)

Work around the productivity problem by making the problematic function `bind` into a constructor of a coinductive type.

This coinductive type has 3 constructors corresponding to the 3 operations of the delay monad: `ret`, `bind`, `later`.

```
CoInductive mon: Type -> Type :=
  | Ret: forall {A: Type}, A -> mon A
  | Later: forall {A: Type}, mon A -> mon A
  | Bind: forall {A B: Type}, mon A -> (A -> mon B) -> mon B
```

A.k.a. the free monad (plus `later`).

A.k.a. an AST for Moggi's monadic metalanguage (plus `later`).

Corecursive functions in the free monad

```
CoFixpoint eval (a: term) : mon (option term) :=
  match a with
  | Const n => Ret (Some (Const n))
  | Var x => Ret None
  | Lam y b => Ret (Some (Lam y b))
  | App b c =>
✔    Bind (eval b) (fun r =>
        match r with
        | Some (Lam x d) => eval (subst x c d)
        | _, _ => Ret None
        end))
  end.
```

This function is productive!

From free monad to computations

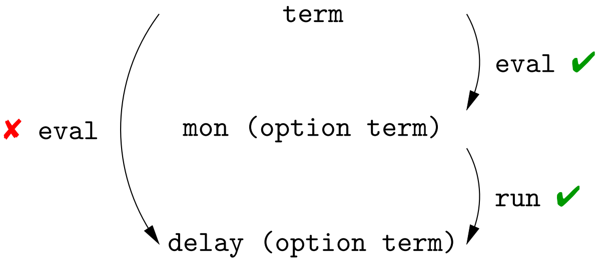
A term of type `mon A` describes a computation of type `delay A`.

```
CoFixpoint run {A: Type} (m: mon A) : delay A :=
  match m with
  | Ret v => now v
  | Later m => later (run m)
  | Bind (Ret v) f => later (run (f v))
  | Bind (Later m) f => later (run (Bind m f))
  | Bind (Bind m f) g =>
      later (run (Bind m (fun x => Bind (f x) g)))
  end.
```

This function is productive!!

Note the use of the first and third monadic laws “on the fly”.

What?



Productivity is a syntactic approximation. It is not compositional.

Properties of `run` as a denotational semantics

`run` is actually a denotational semantics for Moggi's monadic metalanguage, mapping syntax (type `mon A`) to meanings (type `delay A`).

We expect `run` to satisfy a number of equivalences:

✔	`later` denotation	`equi (run (Later m)) (later (run m))`
?	`bind` denotation	`equi (run (Bind m f))` `(bind (run m) (fun x => run (f m))`
✔	1st monadic law	
?	2nd monadic law	
✔	3rd monadic law	

(? means I could not prove it, not that it is false.)

The Monadic Abstract Machine (MAM)

An alternative to `run`, using a continuation explicitly represented as a list of functions `A -> mon B`.

```
Inductive continuation: Type -> Type -> Type :=
  | K0: forall {A: Type}, continuation A A
  | Kbind: forall {A B C: Type} (f: A -> mon B) (k: continuation B C),
           continuation A C.

CoFixpoint mam {A B: Type} (m: mon A) (k: continuation A B): delay B :=
  match m with
  | Ret v =>
      match k with
      | K0 => now v
      | Kbind f k => later (mam (f v) k)
      end v
  | Later m =>
      later (mam m k)
  | Bind m f =>
      later (mam m (Kbind f k))
  end.
```

A run based on the MAM

```
Definition runk {A: Type} (m: mon A) : delay A := mam m KO.
```

Enjoys the expected properties:

✔	later denotation	`equi (runk (Later m)) (later (runk m))`
✔	bind denotation	`equi (runk (Bind m f))` `(bind (runk m) (fun x => runk (f m))`
✔	1st monadic law	
✔	2nd monadic law	
✔	3rd monadic law	
?	same denotations	`equi (run m) (runk m)`
✔		`equi (run (Bind m Ret)) (runk m)`

Back to the definitional interpreter

```
Definition dinterp (a: term): delay (option term) := runk (eval a).
```

Satisfies some classic properties of denotational semantics, e.g. compatibility with reductions:

> *If* $a \to_\beta a'$ *then* `equi` (`dinterp` a) (`dinterp` a').

Is executable to some extent:

```
Fixpoint exec {A: Type} (x: delay A) (n: nat) : option A :=
  match x, n with
  | now v  , _   => Some v
  | _      , O   => None
  | delay x, S n => exec x n
  end.

Definition dexec (a: term) (nsteps: nat) :=
  exec (dinterp a) nsteps.
```

Is this a good approach?

Unclear at this point; possibly not.

+ A nice flavor of denotational semantics.
+ Executability (to some extent).
- Heavy reasoning modulo the `equi` relation.
- Proving that a term diverges requires lower-level reasoning.
  E.g. $\texttt{dinterp}(\delta\ \delta) = \texttt{later}(\cdots(\texttt{dinterp}(\delta\ \delta))\cdots)$
  not just $\texttt{equi}\ (\texttt{dinterp}(\delta\ \delta))\ (\texttt{dinterp}(\delta\ \delta))$.