Mechanized semantics, sixth lecture

Eternity is long…
Semantics for divergence:
domain theory and coinductive approaches

Xavier Leroy

2020-01-30

Collège de France, chair of software sciences

Why study diverging programs?

In languages without input/output (IMP, purely functional):

- No practical interest.
- Useful for theory: computability, program equivalences.

In real-world computing:

- Many programs are supposed to never stop! (OS kernels, Web servers, control-command codes, ...)
- Reactive divergence: finite computation between successive I/O operations.

How to formalize divergence?

Negatively:

- Diverging programs are those programs that do not terminate, neither normally nor by going wrong.

Positively, or even constructively: (parts 2 and 3)

- Coinductive characterizations of divergence. (Termination is fundamentally inductive..)

At the same time we formalize termination: (parts 1 and 3)

- One of the main goals of denotational semantics.
- Classically (domain theory) or constructively (partiality monad)

From a bounded interpreter to a denotational semantics

A reference interpreter

As seen in the first lecture: it is impossible to define the semantics of an IMP command as a function

$$\text{store “before”} \rightarrow \text{store “after”}$$

since this function would be partial (non-termination).

However, we can define an approximation of this function by bounding *a priori* the recursion depth, using a `fuel` parameter of type `nat`.

A bounded reference interpreter

```
Fixpoint cinterp (fuel: nat) (c: com) (s: store)
                                          : option store :=
  ...
```

A `Some s'` result mean that `c` is guaranteed to terminate on `s'`.

A `None` result is unconclusive: either `c` diverges, either we need more fuel to finish the execution of `c`.

These results form a monad ($\approx$ error monad):

```
Definition ret {A: Type} (v: A) := Some v.

Definition bind {A B: Type} (x: option A) (f: A -> option B) :=
  match x with None => None | Some v => f v end.
```

A bounded reference interpreter

```
Fixpoint cinterp (fuel: nat) (c: com) (s: store) : option store :=
  match fuel with
  | O => None
  | S n =>
      match c with
      | SKIP => Some s
      | ASSIGN x a => Some (update x (aeval a s) s)
      | SEQ c1 c2 => bind (cinterp n c1 s) (cinterp n c2)
      | IFTHENELSE b c1 c2 =>
          cinterp n (if beval b s then c1 else c2) s
      | WHILE b c1 =>
          if beval b s
          then bind (cinterp n c1 s) (cinterp n (WHILE b c1))
          else Some s
      end
  end.
```

Partial order over results

The order $r \sqsubseteq r'$, read "r' is more defined than r"

$$\texttt{None} \sqsubseteq r' \qquad\qquad \texttt{Some}(v) \sqsubseteq \texttt{Some}(v)$$

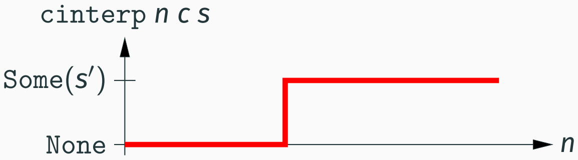
A crucial property: the interpreter is monotonically increasing.
(More fuel $\Rightarrow$ more defined result.)

$$i \leq j \quad \Rightarrow \quad \texttt{cinterp}\, i\, s\, c \sqsubseteq \texttt{cinterp}\, j\, s\, c$$

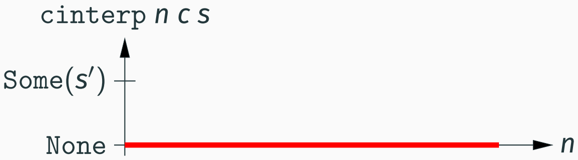
Towards a denotational semantics

What happens if “fuel goes to infinity” ?

For a command c that terminates when started in store s:



For a command c that diverges when started in store s:



Limits

Every increasing sequence $f : \mathtt{nat} \to \mathtt{option}\,A$ has a limit $\lim f$, equal to its supremum, and characterized by

$$\exists i, \forall j, i \leq j \Rightarrow f\ j = \lim f$$

This claim is not constructive: the Coq development (file `Divergence.v`) uses

- the excluded middle axiom to show that the limit exists (either $\forall i,\ f\ i = \mathtt{None}$ or $\exists i,\ f\ i \neq \mathtt{None}$);
- an axiom of description to define the limit $\lim f$ as a function of the sequence f.

A denotational semantics for IMP

Define $[\![c]\!]$, the denotation of command c, as the limit of c's executions by the reference interpreter:

$$[\![c]\!]\; s \quad \stackrel{def}{=} \quad \lim(\texttt{fun}\; n \Rightarrow \texttt{cinterp}\; n\; c\; s)$$

This definition satisfies the expected equations:

$$[\![\texttt{skip}]\!]\; s = \texttt{Some}(s)$$

$$[\![x := a]\!]\; s = \texttt{Some}(s\{x \leftarrow [\![a]\!]\; s\})$$

$$[\![c_1; c_2]\!]\; s = \texttt{bind}\; ([\![c_1]\!]\; s)\; [\![c_2]\!]$$

$$[\![\texttt{if}\; b\; \texttt{then}\; c_1\; \texttt{else}\; c_2]\!]\; s = \begin{cases} [\![c_1]\!]\; s & \text{if } [\![b]\!]\; s = \texttt{true} \\ [\![c_2]\!]\; s & \text{if } [\![b]\!]\; s = \texttt{false} \end{cases}$$

A denotational semantics for IMP: the case of loops

$$[\![\texttt{while}\ b\ \texttt{do}\ c]\!]\ s = \begin{cases} \texttt{bind}\ ([\![c]\!]\ s)\ [\![\texttt{while}\ b\ \texttt{do}\ c]\!] & \text{if } [\![b]\!]\ s = \texttt{true} \\ \texttt{Some}(s) & \text{if } [\![b]\!]\ s = \texttt{false} \end{cases}$$

Furthermore, $[\![\texttt{while b do c}]\!]$ is the smallest function $F : \texttt{store} \to \texttt{option store}$ solution of the equation

$$F\ s = \begin{cases} \texttt{bind}\ ([\![c]\!]\ s)\ F & \text{if } [\![b]\!]\ s = \texttt{true} \\ \texttt{Some}(s) & \text{if } [\![b]\!]\ s = \texttt{false} \end{cases}$$

Example: $[\![\texttt{while true do}\ c]\!]\ s = \texttt{None}$ because the function `fun s => None` is a solution of the equation.

Equivalence between natural and denotational semantics

$$c/s \Downarrow s' \quad \Rightarrow \quad [\![c]\!]\, s = \texttt{Some}(s')$$

$$[\![c]\!]\, s = \texttt{Some}(s') \quad \Rightarrow \quad \texttt{cinterp}\; n\; c\; s = \texttt{Some}(s') \quad \Rightarrow \quad c/s \Downarrow s'$$

(Proofs in file `Divergence.v`.)

Advanced topic: domain theory

A domain is a set A equipped with a partial order $\sqsubseteq$

$$x \sqsubseteq x \qquad \text{(reflexive)}$$

$$x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z \qquad \text{(transitive)}$$

$$x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y \qquad \text{(antisymmetric)}$$

that is ω-complete: every increasing sequence has a supremum.

$$u_0 \sqsubseteq u_1 \sqsubseteq \cdots \sqsubseteq u_n \sqsubseteq \cdots \quad \Rightarrow \quad \sup u \in A$$

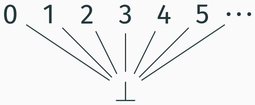
(In the literature, this is often called an ω-cpo or just a cpo.)

Examples of domains

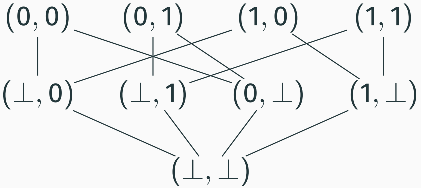
Flat domain
(≈ value of a base type)

$$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad \cdots$$

Pointed domain
(≈ computation of a base type)



Lazy pairs (≈ the OCaml type `bool lazy * bool lazy`)



Examples of domains

Stream of Booleans:

$$
\begin{tikzcd}
0 :: 0 :: \bot & & 0 :: 1 :: \bot & & 1 :: 0 :: \bot & & 1 :: 1 :: \bot \\
& 0 :: \bot \arrow[ul, no head] \arrow[ur, no head] & & & & 1 :: \bot \arrow[ul, no head] \arrow[ur, no head] & \\
& & & \bot \arrow[ull, no head] \arrow[urr, no head] & & &
\end{tikzcd}
$$

Combinations of domains:

- "Pointing" (adding a minimal element): $D_\bot = D \uplus \{\bot\}$
- Product $D_1 \times D_2$, sum $D_1 + D_2$.
- Continuous functions $[D_1 \to D_2]$.

Continuous functions

A function $f : D_1 \to D_2$ is Scott-continuous if it preserves the supremum of increasing sequences:

$$\sup\ f(u_i) = f(\sup u_i)$$

All continuous functions are increasing. The converse is false.

Example: the function finite stream $\mapsto 0$, infinite stream $\mapsto 1$ is increasing yet discontinuous. Besides, it is not computable.

Theorem (Scott's fixed point theorem)

If D is a pointed domain, any continuous function $F : D \to D$ has a least fixed point $\mu F = \sup_n F^n(\bot)$.

Recursive types

To interpret recursive data types as domains, we need to solve (up to isomorphism) equations between domains, such as:

- Integer lists: $D_{list} \simeq \{nil\} + Nat \times D_{list}$
- Pure lambda-terms: $D_\infty \simeq [D_\infty \to D_\infty]$

This can be done if we use algebraic domains, also called Scott domains, where every element is the limit of a sequence of compact elements ($\approx$ finitely described elements).

(See Plotkin's lecture notes in references.)

Coinductive predicates and natural semantics for divergence

Predicates defined by axioms and inference rules

$$P(\texttt{skip}, s) \qquad\qquad \frac{c/s \to c'/s' \qquad P(c', s')}{P(c, s)}$$

Up to now, we have interpreted such definitions of predicates in an inductive manner:

- as the least fixed point of an operator;
- in terms of finite derivations.

Another interpretation exists, the coinductive interpretation:

- as the greatest fixed point of an operator;
- in terms of infinite or finite derivations.

Operator associated with a definition

$$P(\texttt{skip}, s) \qquad\qquad \frac{c/s \to c'/s' \qquad P(c', s')}{P(c, s)}$$

To this axiom and this inference rule, we associate the operator

$$F(X) = \{(\texttt{skip}, s)\} \cup \{(c, s) \mid c/s \to c'/s' \wedge (c', s') \in X\}$$

Intuitively: $F(X)$ is the set of all facts that we can deduce by assuming the facts X and by applying one axiom or one inference rule.

The F operator is increasing, therefore it has a least fixed point and a greatest fixed point.

Smallest fixed point

$$F(X) = \{(\mathtt{skip}, s)\} \cup \{(c, s) \mid c/s \to c'/s' \wedge (c', s') \in X\}$$

The smallest fixed point is

$$\mu F \stackrel{def}{=} \bigcap\{X \mid F(X) \subseteq X\}$$

It is the limit of the increasing sequence $\emptyset, F(\emptyset), \ldots, F^n(\emptyset), \ldots$

In the example, $F^n(\emptyset)$ is the set of (c, s) that reduce to $\mathtt{skip}$ in at most n reductions. Hence, μF is the set of (c, s) that terminate ($c/s \xrightarrow{*} \mathtt{skip}/s'$).

Finite derivations

A derivation = a tree with axioms at the leaves and inference rules at the nodes.

The inductive interpretation μF corresponds to facts that are conclusion of a derivation tree where all branches are finite.

(If all rules have finitely many premises, these derivations are the finite trees.)

Finite derivations

An example of a finite derivation:

$$\dfrac{c_1/s_1 \to c_2/s_2 \qquad \dfrac{c_2/s_2 \to c_3/s_3 \qquad \dfrac{\dfrac{c_n/s_n \to \texttt{skip}/s_{n+1}}{\vdots}}{P(c_3, s_3)}}{P(c_2, s_2)}}{P(c_1, s_1)}$$

$P(c, s)$ can be derived by a tree of height n if and only if c/s reduces to `skip` in n step.

Greatest fixed point

$$F(X) = \{(\texttt{skip}, s)\} \cup \{(c, s) \mid c/s \to c'/s' \wedge (c', s') \in X\}$$

The greatest fixed point is

$$\nu F \stackrel{def}{=} \bigcup \{X \mid X \subseteq F(X)\}$$

It's the limit of the decreasing sequence $U, F(U), \ldots, F^n(U), \ldots$ where U is the universe of all pairs (c, s).

In the example, νF comprises

- all the (c, s) that terminate: $c/s \xrightarrow{*} \texttt{skip}/s'$
- all the (c, s) that diverge: $c/s \xrightarrow{*} c_n/s_n \to \cdots$

Infinite derivations

The coinductive interpretation νF corresponds to the facts that are conclusion of a finite or infinite derivation tree.

An example of infinite derivation:

$$
\cfrac{c_1/s_1 \to c_2/s_2 \qquad \cfrac{c_2/s_2 \to c_3/s_3 \qquad \cfrac{\cfrac{c_n/s_n \to c_{n+1}/s_{n+1} \qquad \vdots}{\vdots}}{P(c_3, s_3)}}{P(c_2, s_2)}}{P(c_1, s_1)}
$$

Divergence, co-inductively

$$\frac{c/s \to c'/s' \qquad \mathtt{div}(c', s')}{\mathtt{div}(c, s)}$$

Inductive interpretation: always false! (there are no axioms…)

Coinductive interpretation (double horizontal line):
characterizes the existence of an infinite sequence of reductions.

In Coq:

```
CoInductive div: com * state -> Prop :=
  | div_intro: forall c s c' s',
      red (c, s) (c', s') -> div (c', s') ->
      div (c, s).
```

Coinduction principle

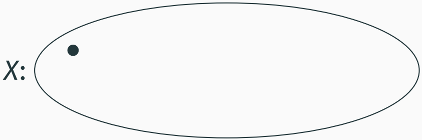
By definition of the greatest fixed point $\nu F = \cup\{X \mid X \subseteq F(X)\}$, any X such that $X \subseteq F(X)$ is contained in νF.

Hence: if the predicate $X : \texttt{com * store} \to \texttt{Prop}$ satisfies

$$\forall c, \forall s,\ X\,(c, s) \Rightarrow \exists c', \exists s',\ c/s \to c'/s' \wedge X\,(c', s')$$

then $X\,(c, s)$ implies $\texttt{div}\,(c, s)$.

Graphically:



Coinduction principle

By definition of the greatest fixed point $\nu F = \cup\{X \mid X \subseteq F(X)\}$, any X such that $X \subseteq F(X)$ is contained in νF.

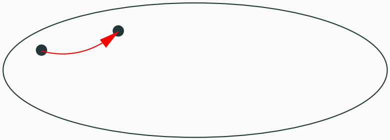
Hence: if the predicate $X : \texttt{com} * \texttt{store} \to \texttt{Prop}$ satisfies

$$\forall c, \forall s,\ X\,(c, s) \Rightarrow \exists c', \exists s',\ c/s \to c'/s' \wedge X\,(c', s')$$

then $X\,(c, s)$ implies $\texttt{div}\,(c, s)$.

Graphically:

X:



Coinduction principle

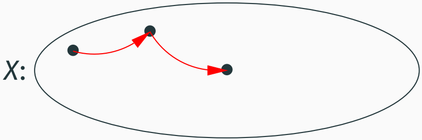
By definition of the greatest fixed point $\nu F = \cup \{X \mid X \subseteq F(X)\}$, any X such that $X \subseteq F(X)$ is contained in νF.

Hence: if the predicate $X : \texttt{com * store} \rightarrow \texttt{Prop}$ satisfies

$$\forall c, \forall s,\ X\ (c, s) \Rightarrow \exists c', \exists s',\ c/s \rightarrow c'/s' \wedge X\ (c', s')$$

then $X\ (c, s)$ implies $\texttt{div}\ (c, s)$.

Graphically:



Coinduction principle

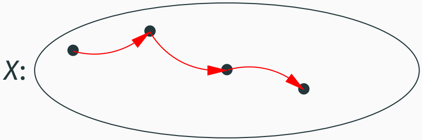
By definition of the greatest fixed point $\nu F = \cup\{X \mid X \subseteq F(X)\}$, any X such that $X \subseteq F(X)$ is contained in νF.

Hence: if the predicate $X : \texttt{com * store} \to \texttt{Prop}$ satisfies

$$\forall c, \forall s,\ X\ (c, s) \Rightarrow \exists c', \exists s',\ c/s \to c'/s' \wedge X\ (c', s')$$

then $X\ (c, s)$ implies $\texttt{div}\ (c, s)$.

Graphically:



Coinduction principle

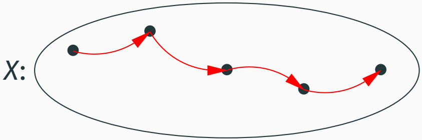
By definition of the greatest fixed point $\nu F = \cup\{X \mid X \subseteq F(X)\}$, any X such that $X \subseteq F(X)$ is contained in νF.

Hence: if the predicate $X : \texttt{com * store} \to \texttt{Prop}$ satisfies

$$\forall c, \forall s,\ X\,(c, s) \Rightarrow \exists c', \exists s',\ c/s \to c'/s' \wedge X\,(c', s')$$

then $X\,(c, s)$ implies $\texttt{div}\,(c, s)$.

Graphically:

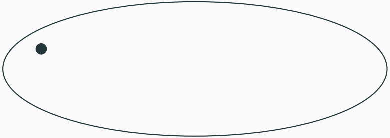


Coinduction principle

From this coinduction principe, we can derive another, very useful principle, using $\xrightarrow{+}$ (one or several reductions) instead of $\rightarrow$ (one reduction):

If $\forall c, \forall s,\ X\ c\ s \Rightarrow \exists c', \exists s',\ c/s \xrightarrow{+} c'/s' \wedge X\ c'\ s'$,
then $X\ c\ s$ implies `div` $c\ s$.

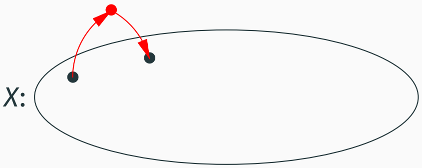
X:



Coinduction principle

From this coinduction principe, we can derive another, very useful principle, using $\xrightarrow{+}$ (one or several reductions) instead of $\rightarrow$ (one reduction):

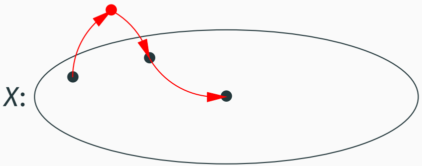
If $\forall c, \forall s,\ X\ c\ s \Rightarrow \exists c', \exists s',\ c/s \xrightarrow{+} c'/s' \wedge X\ c'\ s'$,
then $X\ c\ s$ implies `div` $c\ s$.



Coinduction principle

From this coinduction principe, we can derive another, very useful principle, using $\xrightarrow{+}$ (one or several reductions) instead of $\rightarrow$ (one reduction):

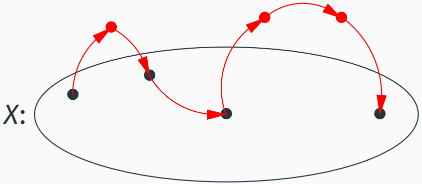
If $\forall c, \forall s,\ X\ c\ s \Rightarrow \exists c', \exists s',\ c/s \xrightarrow{+} c'/s' \wedge X\ c'\ s'$,
then $X\ c\ s$ implies `div` $c\ s$.



Coinduction principle

From this coinduction principe, we can derive another, very useful principle, using $\xrightarrow{+}$ (one or several reductions) instead of $\rightarrow$ (one reduction):

If $\forall c, \forall s,\ X\ c\ s \Rightarrow \exists c', \exists s',\ c/s \xrightarrow{+} c'/s' \wedge X\ c'\ s'$,
then $X\ c\ s$ implies `div` $c\ s$.



Back to natural semantics

In the first lecture, we introduced natural semantics as a way to structure the reductions to $\texttt{skip}$.

For example, if command $c; c'$ terminates, its reduction sequence must have the following shape:

$$\begin{aligned}(c; c')/s \to (c_1; c')/s_1 \to \cdots \to (\texttt{skip}; c')/s' \\ \to c'/s' \to \cdots \to \texttt{skip}/s''\end{aligned}$$

This structure is reflected by the natural semantics rule for sequences:

$$\frac{c/s \Downarrow s' \qquad c'/s' \Downarrow s''}{c; c'/s \Downarrow s''}$$

A structure for infinite sequences of reductions

Likewise, if command $c; c'$ diverges, its infinite sequence of reductions must have one of the following two shapes:

$$(c; c')/s \to \cdots \to (c_n; c')/s_n \to \cdots \tag{1}$$

$$(c; c')/s \xrightarrow{*} (\texttt{skip}; c')/s' \to c'/s' \to \cdots c'_n/s'_n \to \cdots \tag{2}$$

In case (1): c diverges, c' does not get to run.
In case (2): c terminates, then c' diverges.

Let's try to reflect this structure as rules for a predicate $c/s \Uparrow$, "command c diverges started in store s".

Natural semantics for divergence

$$\frac{c_1/s \Uparrow}{c_1; c_2/s \Uparrow} \qquad \frac{c_1/s \Downarrow s' \quad c_2/s' \Uparrow}{c_1; c_2/s \Uparrow}$$

$$\frac{[\![b]\!]\, s = \mathtt{true} \quad c_1/s \Uparrow}{(\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2)/s \Uparrow} \qquad \frac{[\![b]\!]\, s = \mathtt{false} \quad c_2/s \Uparrow}{(\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2)/s \Uparrow}$$

$$\frac{[\![b]\!]\, s = \mathtt{true} \quad c/s \Uparrow}{(\mathtt{while}\ b\ \mathtt{do}\ c)/s \Uparrow}$$

$$\frac{[\![b]\!]\, s = \mathtt{true} \quad c/s \Downarrow s' \quad (\mathtt{while}\ b\ \mathtt{do}\ c)/s' \Uparrow}{(\mathtt{while}\ b\ \mathtt{do}\ c)/s \Uparrow}$$

An example of diverging execution

The loop $c \stackrel{def}{=} \texttt{while true do x := x + 1}$ diverges.

$$\dfrac{x := x+1/s_0 \Downarrow s_1 \qquad \dfrac{x := x+1/s_1 \Downarrow s_2 \qquad \dfrac{x := x+1/s_2 \Downarrow s_3 \qquad \vdots}{c/s_2 \Uparrow}}{c/s_1 \Uparrow}}{c/s_0 \Uparrow}$$

(Where $s_i = s_0[x \leftarrow s_0(x) + i]$.)

Equivalence with reduction semantics

Theorem

If $c/s \Uparrow$, then c/s reduces infinitely.

Proof (constructive).

We show $c/s \Uparrow \ \Rightarrow \exists c', \exists s', c/s \xrightarrow{+} c'/s' \wedge c'/s' \Uparrow$.
We conclude by the second coinduction principle applied to the set
$X = \{(c, s) \mid c/s \Uparrow\}$. □

Theorem

If c/s reduces infinitely, then $c/s \Uparrow$.

Proof (classical).

By coinduction and case analysis on the shape of reduction sequences.
We need excluded middle: either c/s reduces finitely to `skip`, or c/s reduces infinitely. □

Application to compiler verification

In lecture #2, we used natural semantics to show the correctness of the compiled code for a terminating IMP command:

```
Lemma compile_com_correct_terminating:
  forall s c s', cexec s c s' ->
  forall C pc σ, code_at C pc (compile_com c) ->
  transitions C
      (pc, σ, s)
      (pc + codelen (compile_com c), σ, s').
```

An induction on the derivation of `cexec s c s'` led to a rather simple proof.

Paradise lost: this simple proof did not extend to diverging commands.

Application to compiler verification

Paradise regained: natural coinductive semantics leads to a rather simple proof of compiler correctness for diverging commands.

Consider the set of machine configurations corresponding to diverging commands:

$$X \stackrel{def}{=} \{(pc, \sigma, s) \mid \exists c,\ c/s \Uparrow\ \wedge \texttt{code\_at}\ C\ pc\ (\texttt{compile\_com}\ c)\}$$

We show that this set is “productive”:

$$\forall (pc, \sigma, s) \in X,\ \exists (pc', \sigma', s') \in X,\ (pc, \sigma, s) \xrightarrow{+} (pc', \sigma', s')$$

We conclude that, when started in a configuration that belongs to X, the machine performs infinitely many transitions.

Partiality monad and coinductive reference 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 produce a value of type *A* if they terminate.

The `later` constructor materializes one step of computation.

The `delay` type being coinductive, we can have infinitely many steps of computation, that is, a non-terminating computation.

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

Partial computations

```
CoInductive delay (A: Type) : Type :=
  | now: A -> delay A
  | later: delay A -> delay A.
```

Terminating computations are characterized inductively; diverging computations, coinductively.

```
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 general recursive functions with `delay` result type, provided all recursive calls are guarded by `later`.

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

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

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

Recursion and corecursion in Coq

Recursive function definition (`Fixpoint`):

- The argument has an inductive type.
- Guard condition: $f\ x$ can call $f\ y$ recursively provided the argument y is a strict sub-term of the argument x.

Corecursive function definition (`CoFixpoint`):

- The result has a coinductive type.
- Productivity condition: $f\ x$ can call $f\ y$ recursively provided the result $f\ y$ is a strict sub-term of the result $f\ x$.
  ($f\ x$ is $f\ y$ wrapped inside one or several constructors.)

General recursion

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

We can reason about termination or divergence of the function after its definition.

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

Theorem remainder_divergence:
  forall a, diverges (remainder a 0).
```

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).
```

In classical logic, this is equivalent to

$(\exists v,\ \texttt{terminates}\ x\ v \wedge \texttt{terminates}\ y\ v) \vee (\texttt{diverges}\ x \wedge \texttt{diverges}\ y)$

but it is stronger in constructive logic. (No need to "guess in advance" whether the two computations terminate or diverge.)

The partiality monad

The `delay` type is a monad, with constructor `now` as the `ret` operation, and the `bind` operation defined as 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 for a sequencing, e.g. `bind a f` diverges iff `a` diverges or `a` terminates with `v` and `f v` diverges.

The partiality monad

The three monadic laws hold up to observational equivalence ($\texttt{equi}$, written $\approx$ from now on):

$$\begin{aligned} \texttt{bind}\ (\texttt{now}\ v)\ f &\approx f\ v \\ \texttt{bind}\ a\ \texttt{now} &\approx a \\ \texttt{bind}\ (\texttt{bind}\ a\ f)\ g &\approx \texttt{bind}\ a\ (\texttt{fun}\ x \Rightarrow \texttt{bind}\ (f\ x)\ g) \end{aligned}$$

Furthermore,

$$\texttt{bind}\ a\ f \approx \texttt{bind}\ a'\ f' \quad \text{if } a \approx a' \text{ and } \forall x, f\ x \approx f'\ x$$

An interpreter in the partiality monad

Using the partiality monad, let's try to write a general recursive interpreter for IMP.

```
CoFixpoint cinterp (c: com) (s: store) : delay store :=
  match c with
  | SKIP => now s
  | ASSIGN x a => now (update x (aeval a s) s)
  | SEQ c1 c2 => bind (cinterp c1 s) (cinterp c2)
  | IFTHENELSE b c1 c2 =>
      later (cinterp (if beval b s then c1 else c2) s)
  | WHILE b c =>
      if beval b s then bind (cinterp c s) (cinterp (WHILE b c))
                   else ret s
  end.
```

Problem: this definition is not productive!

Going through the free monad

(An application of N. A. Danielsson's technique, *Beating the Productivity Checker Using Embedded Languages*, 2010)

We can work around the problem by presenting the monad as a coinductive type whose constructors are the monad operations: `ret`, `bind`, and `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
```

In other words: the free monad (plus `later`).

In other words: an abstract syntax for Moggi's monadic metalanguage (plus `later`).

An interpreter in the free monad

```
CoFixpoint cinterp (c: com) (s: store) : mon store :=
  match c with
  | SKIP => Ret s
  | ASSIGN x a => Ret (update x (aeval a s) s)
  | SEQ c1 c2 => Bind (cinterp c1 s) (cinterp c2)
  | IFTHENELSE b c1 c2 =>
      Later (cinterp (if beval b s then c1 else c2) s)
  | WHILE b c =>
      if beval b s then Bind (cinterp c s) (cinterp (WHILE b c))
                   else Ret s
  end.
```

This definition is productive!

Interpreting the free monad

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.
```

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

Productivity is surprising

$$
\begin{tikzcd}
\texttt{com * store} \arrow[d, "\texttt{cinterp}\ \checkmark"] \arrow[dd, bend right=60, "\text{✘}\ \texttt{cinterp}"'] \\
\texttt{mon store} \arrow[d, "\texttt{run}\ \checkmark"] \\
\texttt{delay store}
\end{tikzcd}
$$

The productivity condition is a syntactic approximation.
It is not compositional.

The `run` function as a denotational semantics

The `run` function can be viewed as a denotational semantics for the monadic metalanguage:

$$\texttt{run}: \text{ syntax (type } \texttt{mon A}) \to \text{meaning (type } \texttt{delay A}).$$

Equivalences satisfied by `run`:

$$\texttt{run}\ (\texttt{Later}\ m) \approx \texttt{later}(\texttt{run}\ m) \qquad (\texttt{Later}\text{ denotation})$$

$$\texttt{run}\ (\texttt{Bind}\ m\ f) \approx \texttt{bind}\ (\texttt{run}\ m)\ (\texttt{fun}\ x \Rightarrow \texttt{run}\ (f\ x)) \qquad (\texttt{Bind}\text{ denotation})$$

$$\texttt{run}\ (\texttt{Bind}\ (\texttt{Ret}\ v)\ f) \approx \texttt{run}\ (f\ v) \qquad (\text{first monadic law})$$

$$\texttt{run}\ (\texttt{Bind}\ m\ \texttt{Ret}) \approx \texttt{run}\ m \qquad (\text{second monadic law})$$

$$\texttt{run}\ (\texttt{Bind}\ (\texttt{Bind}\ m\ f)\ g) \approx \texttt{run}\ (\texttt{Bind}\ m\ (\texttt{fun}\ x \Rightarrow \texttt{Bind}\ (f\ x)\ g)) \qquad (\text{third monadic law})$$

The coinductive interpreter as a denotational semantics

Define the denotation of a command c as

$$[\![c]\!]\ s \stackrel{def}{=} \texttt{run}\,(\texttt{cinterp}\ c\ s) \qquad \text{(with type } \texttt{delay}\ \ \texttt{store}\text{)}$$

This definition satisfies the expected equations:

$$\begin{aligned}
[\![\texttt{skip}]\!]\ s &\approx \texttt{now}(s) \\
[\![x := a]\!]\ s &\approx \texttt{now}(s\{x \leftarrow [\![a]\!]\ s\}) \\
[\![c_1; c_2]\!]\ s &\approx \texttt{bind}\ ([\![c_1]\!]\ s)\ [\![c_2]\!] \\
[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\ s &\approx \begin{cases} [\![c_1]\!]\ s & \text{if } [\![b]\!]\ s = \texttt{true} \\ [\![c_2]\!]\ s & \text{if } [\![b]\!]\ s = \texttt{false} \end{cases} \\
[\![\texttt{while}\ b\ \texttt{do}\ c]\!]\ s &\approx \begin{cases} \texttt{bind}\ ([\![c]\!]\ s)\ [\![\texttt{while}\ b\ \texttt{do}\ c]\!] & \text{if } [\![b]\!]\ s = \texttt{true} \\ \texttt{now}(s) & \text{if } [\![b]\!]\ s = \texttt{false} \end{cases}
\end{aligned}$$

Summary

Summary

Coinduction is a fundamental tool to reason about divergence, from the trivial (infinite sequences of reductions) to the subtle (natural semantics for divergence, partiality monad).

Denotational semantics require an appropriate mathematical structure. Classically, it's Scott domains; constructively, it could be the quotient type $\mathtt{delay}\, A / \approx$.

For the time being, the approaches outlined in this lecture do not scale to "big languages" as well as transition semantics.

References

References

Domain theory and its mechanization:

- G. Plotkin, *Domains*, 1983,
  `http://homepages.inf.ed.ac.uk/gdp/publications/Domains_a4.ps`
- N. Benton, A. Kennedy, C. Varming, *Some Domain Theory and Denotational Semantics in Coq*, TPHOLs 2009.

Coinductive natural semantics:

- X. Leroy et H. Grall, *Coinductive big-step operational semantics*, Inf&Comp 207(2), 2009.

Partiality monads and denotational semantics:

- V. Capretta, *General recursion via coinductive types*, LMCS(1), 2005.
- N. A. Danielsson, *Operational Semantics Using the Partiality Monad*, ICFP 2012,