How to prove a coformula

Minao Kukita
minao.kukita@gmail.com

Kyoto University

Logic Seminar Series
The University of Melbourne
September 10, 2010

A formal system standing alone is an incomplete entity: it needs its interpretation.

—— Dana Scott, “Rules and derived rules.”

Background:

- Greg Restall's **coformulas**.
- A **gap** between logic and computation.

Our aim:

- To consider what kind of proof theory is suitable for coformulas by taking into account what coformulas are and should be in an intuitive sense, and in what context they are useful.

Outline

Stream

There are three viewpoints from which we can study streams:

- set-theoretic (static)
- computational (dynamic)
- category-theoretic (both)

Stream: from the set-theoretic point of view

Definition

Given a set A, $\mathit{Stream}(A) = A^\omega$ $(= \prod_{n\in\omega} A = (\omega \to A))$.

Note that

$$\mathit{Stream}(A) = A \times \mathit{Stream}(A) = A \times (A \times \mathit{Stream}(A)) = \dots$$

i.e., $\mathit{Stream}(A)$ satisfies the following equation:

$$\mathit{Stream}(A) = A \times \mathit{Stream}(A)$$

Stream: from the computational point of view

Definition

A stream s on the data type A is a machine $\mathcal{M} = (Q, q_0, \delta)$ where:

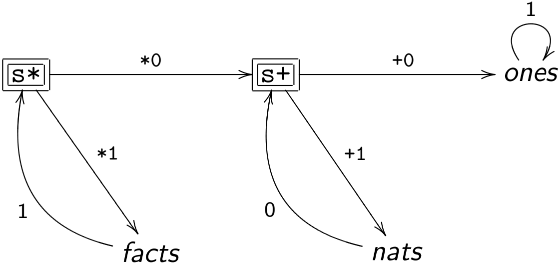
- Q is a set of the states of the machine,
- $q_0 \in Q$ is the initial state, and
- $\delta : Q \to A \times Q$ is a function from a state q to a pair (a, q') of an atomic data a and a state q'.

$\delta(q) = \langle a, q' \rangle$ means that a is the output of $\mathcal{M}$ in the state q, while q' is the state immediately after q. Call them **head** and **tail** respectively.

Examples

Name	Definition	Output
`ones`	`(1.ones)`	$1, 1, 1, 1, 1, 1, \ldots$
`nats`	`(0.(s+ ones nats))`	$0, 1, 2, 3, 4, 5, \ldots$
`facts`	`(1.(s* (tail nats) facts))`	$1, 1, 2, 6, 24, 120, \ldots$
`Fibs`	`(1.1.(s+ Fibs (tail Fibs)))`	$1, 1, 2, 3, 5, 8, \ldots$

where `s+` and `s*` stand for the stream operations of adding and multiplying componentwise respectively.


1
*0
+0
s*
s+
ones
*1
+1
1
0
facts
nats

Stream: from the category-theoretic point of view

Definition

Stream(A) is a final F_A-coalgebra, i.e., a terminal object in the category F_A-**CoAlg**, where $F_A : \mathbf{Set} \to \mathbf{Set}$ is a functor defined by:

$$F_A X = A \times X, \quad F_A f = \langle 1_A, f \rangle$$

and F_A-**CoAlg** consists of:

- Objects: functions $f : X \to F_A X$ in **Set**.
- Arrows: $\alpha : f \to g$ is a function from $\mathbf{dom}(f)$ to $\mathbf{dom}(g)$ such that $g \circ \alpha = F_A \alpha \circ f$.

Specifically, $\mathit{Stream}(A) = \langle \mathbf{head}, \mathbf{tail} \rangle : A^\omega \to A \times A^\omega$.

Many operations on Stream(A) can be defined as a mediating arrow to this terminal object. E. g.:

`ones`: $\mathit{Stream}(\omega)$

$$\begin{tikzcd}
\omega \times \omega^\omega & & \omega \times \mathbf{1} \arrow[ll, "\langle id_\omega, \llbracket \langle 1,! \rangle \rrbracket \rangle"'] \\
\omega^\omega \arrow[u, "\langle \mathbf{head},\mathbf{tail} \rangle"] & & \mathbf{1} \arrow[ll, "\llbracket \langle 1,! \rangle \rrbracket"] \arrow[u, "\langle 1,! \rangle"']
\end{tikzcd}$$

`interleave`: $\mathit{Stream}(A) \times \mathit{Stream}(A) \to \mathit{Stream}(A)$

$$\begin{tikzcd}
A \times A^\omega & A \times (A^\omega \times A^\omega) \arrow[l, "\langle id_A, \llbracket \phi \rrbracket \rangle"'] \\
A^\omega \arrow[u, "\langle \mathbf{head}, \mathbf{tail} \rangle"] & A^\omega \times A^\omega \arrow[l, "\llbracket \phi \rrbracket"] \arrow[u, "\phi"']
\end{tikzcd}$$

where $\phi = \langle \mathbf{head} \circ \mathbf{fst}, \langle \mathbf{snd}, \mathbf{tail} \circ \mathbf{fst} \rangle \rangle$.

Coformulas (cf. Restall, forthcoming)

For simplicity we think only of binary connectives.

Definition

Let Σ be a set of binary connectives. A **coformula** on Σ is a machine $\mathcal{M} = (Q, q_0, \delta)$ where:

- Q is a set of the states,
- q_0 is the initial state,
- $\delta : Q \to \Sigma \times Q \times Q$ is a function from a state q to a triple (σ, q', q'') of a connective $\sigma \in \Sigma$ and two next states q' and q''.

$\mathcal{M}$ is called finite if Q is. We call σ the **head** of this coformula, and q', q'' the **tails**.

Set-theoretically, a coformula is an infinite binary trees with each node labelled with a connective in Σ.

Category-theoretically, coformulas are a terminal object F_Σ-**CoAlg**, where $F_\Sigma : \mathbf{Set} \to \mathbf{Set}$ is defined by:

$$F_\Sigma X = \Sigma \times X \times X, \quad F_\Sigma f = \langle id_\Sigma, f, f \rangle$$

Specifically it is $\langle \mathbf{head}, \mathbf{tail}_0, \mathbf{tail}_1 \rangle : T_\Sigma \to \Sigma \times T_\Sigma \times T_\Sigma$, where T_Σ is the set of infinite binary trees with labels Σ.

Examples

Let → and ∧ be binary connectives. Then followings are coformulas:

- $Q = \{\star\},\ q_0 = \star,\ \delta(\star) = \langle \rightarrow, \star, \star \rangle.$
- $Q = \{\sharp, \natural\},\ q_0 = \sharp,\ \delta(\sharp) = \langle \rightarrow, \natural, \natural \rangle,\ \delta(\natural) = \langle \wedge, \sharp, \sharp \rangle.$

Definition

$R \subseteq T_\Sigma \times T_\Sigma$ is called **bisimulation relation** on T_Σ if for all $\mathcal{M}, \mathcal{M}' \in T_\Sigma$,

$$\mathcal{M} R \mathcal{M}' \Rightarrow \begin{cases} \textbf{head}(\mathcal{M}) = \textbf{head}(\mathcal{M}') \\ \textbf{tail}_0(\mathcal{M}) R \textbf{tail}_0(\mathcal{M}') \\ \textbf{tail}_1(\mathcal{M}) R \textbf{tail}_1(\mathcal{M}') \end{cases}$$

Coformulas $\mathcal{M}$ and $\mathcal{M}' \in T_\Sigma$ are said to be **bisimilar**, written $\mathcal{M} \simeq^B \mathcal{M}'$ if, $\textbf{head}(\mathcal{M}) = \textbf{head}(\mathcal{M}')$ and for some bisimilation relation R, $\mathcal{M} R \mathcal{M}'$.

Two bisimilar coformulas are observationally equal, i.e., we cannot distinguish one from the other by observing their behaviors.

The formula-coformula pair is one of many dual notions of the same kind:

Algebra $\Longleftrightarrow$ Coalgebra
Recursion $\Longleftrightarrow$ Corecursion
List $\Longleftrightarrow$ Stream
Formula $\Longleftrightarrow$ Coformula

So to think of coformulas seems a natural step.

But the following questions also seem as natural:

- What, if any, can they stand for?
- Is there any context in which they are useful?

—Yes.

- Some of them can be thought of as **recursive types**,
- They can **fill a gap between logic and computation**.

Curry-Howard isomorphisms

Logic		Typed λ calculus
Propositional logic	$\Longleftrightarrow$	Simply typed λ calculus
1st-order logic	$\Longleftrightarrow$	Dependant type theory
2nd-order prop. logic	$\Longleftrightarrow$	Polymorphic type theory
	etc.	
Proposition	$\Longleftrightarrow$	Type
$\rightarrow$-I ($\forall$-I)	$\Longleftrightarrow$	Abstraction
$\rightarrow$-E ($\forall$-E)	$\Longleftrightarrow$	Function application
Proof	$\Longleftrightarrow$	Term
Normalization	$\Longleftrightarrow$	Reduction
	etc.	

Difference between logic and computation

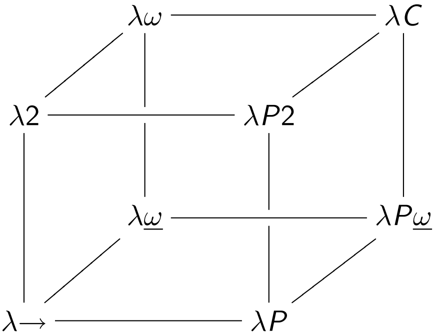
Hence the cliché:

“To construct a proof is to write a program.”

But not vice versa:

“To write a program is NOT to construct a proof.”

Cf. Nordström et al. *Programming in Martin-Löf Type Theory*:
“In type theory it is also possible to write specifications of programming tasks as well as to develop provably correct programs. **Type theory is therefore more than a programming languages**, and **it should not be compared with programming languages**, but with formalized programming logics such as LCF and PL/CV.” (my emphases)



$\lambda\omega$ λC

$\lambda 2$ $\lambda P2$

$\lambda\underline{\omega}$ $\lambda P\underline{\omega}$

$\lambda{\to}$ λP

The λ cube: they are all strongly normalising,
therefore not Turing complete.

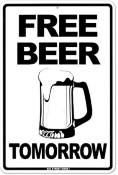
$$\begin{tikzcd}
 & \lambda\omega \arrow[rr, dash] \arrow[dl, dash] \arrow[dd, dash] & & \lambda C \arrow[dl, dash] \arrow[dd, dash] \\
\lambda 2 \arrow[rr, dash] \arrow[dd, dash] & & \lambda P2 \arrow[dd, dash] & \\
 & \lambda\underline{\omega} \arrow[rr, dash] \arrow[dl, dash] & & \lambda P\underline{\omega} \arrow[dl, dash] \\
\lambda{\to} \arrow[rr, dash] & & \lambda P &
\end{tikzcd}$$

The λ cube: they are all strongly normalising, therefore not Turing complete.

On the other hand, most practical programming languages do not have the SN property.

An example of non-terminating program:

while (now.isToday()) {

    System.out.print();

}

We need such constructions particularly in order to allow recursive definitions.

Consider how the following definition works:

$$f(n) = \begin{cases} 1 & \text{if n} = 0 \\ n * f(n-1) & \text{otherwise} \end{cases}$$

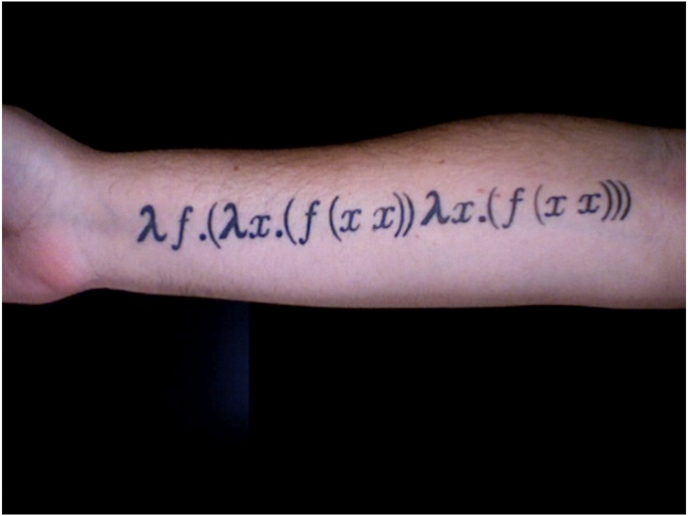
What we do here is to compute the least fixed point of the functional $\Phi : (N \rightharpoonup N) \to (N \rightharpoonup N)$ such that:

$$\Phi(f)(n) = \begin{cases} 1 & \text{if n} = 0 \\ n * f(n-1) & \text{otherwise} \end{cases}$$

where $N \rightharpoonup N$ is the set of partial functions on natural numbers.

But how do we find the least fixed point of such functionals?

Curry's fixed-point combinator



Definition

$x \in A$ is called a **fixed point** of $f : A \to A$ if $f(x) = x$.
$F : (A \to A) \to A$ is called a **fixed-point operator** if for all $f : A \to A$, $f(F(f)) = F(f)$.

Let Y be $\lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$. Then for any term M,

$$\begin{aligned} YM &\to_\beta M((\lambda x.M(xx))(\lambda x.M(xx))) \\ &\to_\beta M(M((\lambda x.M(xx))(\lambda x.M(xx)))) \\ M(YM) &\to_\beta M(M((\lambda x.M(xx))(\lambda x.M(xx)))) \end{aligned}$$

$\therefore M(YM) =_\beta YM$.

Definition of factorial using a fixed-point operator:

```
(define Y
    (lambda (f)
        ((lambda (x) (f (x x))) (lambda (x) (f (x x))))))

(define (F f)
    (lambda (n)
        (if (= n 0) 1
            (* n (f (- n 1))))))

((Y F) 5)

>> 120
```

Note the following facts:

Fact

- *Y has no βnf, therefore $\nvdash_{\lambda\star} Y : \sigma$, for any pure type system $\lambda\star$ enjoying the SN property.*
- *$\lambda\!\rightarrow$ with Y (or any other mechanism for finding the least fixed point of an arbitrary term) is Turing complete.*

So such a mechanism is the major factor that divides logics (typed λ calculi) and programming languages.

1 Coformulas

2 Relation between logic and computation

3 Coformulas as recursive types

4 Proofs

Reflexive domain

Definition

A type D with terms $\phi_D : D \to (D \to D)$ and $\psi_D : (D \to D) \to D$ is called a **reflexive domain** if

$$\phi_D \circ \psi_D = I_{D \to D}$$

where $I_{D \to D} : (D \to D) \to (D \to D)$ is the identity function on $(D \to D)$.

Rules for reflexive domains:

$$\frac{M : D \to D}{\psi_D M : D}\ \text{fold} \qquad \frac{M : D}{\phi_D M : D \to D}\ \text{unfold}$$

$$\frac{M : D \to D}{\phi_D(\psi_D M) = M : D \to D}\ \text{reflexive identity}$$

Y combinator

$$
\dfrac{
\dfrac{
\dfrac{
y : D \to D \qquad
\dfrac{
\dfrac{x : D}{\phi x : D \to D}\,\text{unfold} \qquad x : D
}{\phi x x : D}
}{y(\phi x x) : D}
}{\lambda x^{D}.y(\phi x x) : D \to D}
\qquad
\dfrac{
\dfrac{
\dfrac{
y : D \to D \qquad
\dfrac{
\dfrac{x : D}{\phi x : D \to D}\,\text{unfold} \qquad x : D
}{\phi x x : D}
}{y(\phi x x) : D}
}{\lambda x^{D}.y(\phi x x) : D \to D}
}{\psi(\lambda x^{D}.y(\phi x x)) : D}\,\text{fold}
}{(\lambda x^{D}.y(\phi x x))(\psi(\lambda x^{D}.y(\phi x x))) : D}
}{\lambda y^{D \to D}.(\lambda x^{D}.y(\phi x x))(\psi(\lambda x^{D}.y(\phi x x))) : (D \to D) \to D}
$$

A reflexive domain $D \to D$ can be defined by means of more general notion of recursive types.

Recursive types are type expressions of the form

$$\mathbf{rec}\, t.\tau$$

where τ is a type expression and t is a type variable.

Roughly, $\mathbf{rec}\, t.\tau$ is a type which satisties the equation:

$$\mathbf{rec}\, t.\tau = \tau[\mathbf{rec}\, t.\tau/t]$$

Therefore, a reflexive domain $D \to D$ is defined as $\mathbf{rec}\, t.t \to t$.

Type expressions

Definition

Let **TVar** be a set of type variables (ranged over by t). The set **TExp** of type expressions (ranged over by τ) is defined by the following grammer:

$$\tau ::= t \mid \tau\sigma\tau \mid \mathbf{rec}\,t.\tau$$

where $\sigma \in \Sigma$.

A type expression is called **coformulaic** if it is closed and does not contain subexpressions of the form $\mathbf{rec}\,t.t'$ (whether $t = t'$ or not). We denote the set of coformulaic expressions by $\mathbf{TExp}^0$.

We define a function $F : T_{\Sigma}^{\text{fin}} \rightarrow \textbf{TExp}^0$ that transforms any finite coformula into coformulaic expression as follows:

Given a finite coformula $\mathcal{M} = \langle Q, q_0, \delta \rangle$, we can construct a corresponding $\textbf{TExp}^0$ as follows: Let $Q = \{q_k : 0 \leq k \leq n\}$. Let

$$S_k = t_{i(k,0)} \sigma_k t_{i(k,1)}$$

where $\delta(q_k) = \langle \sigma_k, q_{i(k,0)}, q_{i(k,1)} \rangle$ and $t_{i(k,0)}$'s are distinct type variables for $0 \leq k \leq n$.

Define $\hat{S}_k (0 \leq k \leq n)$ inductively as follows:

$$\begin{array}{rcl} \hat{S}_n & = & \mathbf{rec} t_n . S_n \\ \hat{S}_{k-1} & = & \mathbf{rec} t_{k-1} . (S_{k-1}[\hat{S}_n / t_n] \ldots [\hat{S}_k / t_k]) \end{array}$$

Then let $\hat{S}_0$ be $F\mathcal{M}$.

For example, consider the following coformula $\mathcal{M} = \langle Q, q_0, \delta \rangle$:

$$\begin{array}{rcl} Q & = & \{q_0, q_1, q_3\}, \\ \delta(q_0) & = & \langle \rightarrow, q_1, q_2 \rangle, \\ \delta(q_1) & = & \langle \wedge, q_2, q_0 \rangle, \\ \delta(q_2) & = & \langle \vee, q_0, q_1 \rangle. \end{array}$$

Applying F to $\mathcal{M}$, we get

$$\mathbf{rec}\, t_0.(\mathbf{rec}\, t_1.(\mathbf{rec}\, t_2.(t_0 \vee t_1) \wedge t_0) \rightarrow \mathbf{rec}\, t_2.(t_0 \vee \mathbf{rec}\, t_1.(\mathbf{rec}\, t_2.(t_0 \vee t_1) \wedge t_0)))$$

$$\begin{aligned}
S_2 &= t_0 \vee t_1 \\
S_1 &= t_2 \wedge t_0 \\
S_0 &= t_1 \to t_2
\end{aligned}$$

$$\begin{aligned}
\hat{S}_2 &= \mathbf{rec}\, t_2.(t_0 \vee t_1) \\
S_1[\hat{S}_2/t_2] &= \mathbf{rec}\, t_2.(t_0 \vee t_1) \wedge t_0 \\
S_0[\hat{S}_2/t_2] &= t_1 \to \mathbf{rec}\, t_2.(t_0 \vee t_1)
\end{aligned}$$

$$\begin{aligned}
\hat{S}_1 &= \mathbf{rec}\, t_1.(\mathbf{rec}\, t_2.(t_0 \vee t_1) \wedge t_0) \\
S_0[\hat{S}_2/t_2][\hat{S}_1/t_1] &= \mathbf{rec}\, t_1.(\mathbf{rec}\, t_2.(t_0 \vee t_1) \wedge t_0) \to \\
&\qquad \mathbf{rec}\, t_2.(t_0 \vee \mathbf{rec}\, t_1.(\mathbf{rec}\, t_2.(t_0 \vee t_1) \wedge t_0))
\end{aligned}$$

$$\begin{aligned}
\hat{S}_0 &= \mathbf{rec}\, t_0.(\mathbf{rec}\, t_1.(\mathbf{rec}\, t_2.(t_0 \vee t_1) \wedge t_0) \to \\
&\qquad \mathbf{rec}\, t_2.(t_0 \vee \mathbf{rec}\, t_1.(\mathbf{rec}\, t_2.(t_0 \vee t_1) \wedge t_0)))
\end{aligned}$$

Remark: The order of S_k's may affect the result. For example, in the construction of $\hat{S}_0$, if we start with S_1 instead of S_2, the result will be

$$\mathbf{rec}\, t_0.(\mathbf{rec}\, t_1.(\mathbf{rec}\, t_2.(t_0 \vee \mathbf{rec}\, t_1.(t_2 \wedge t_0)) \wedge t_0) \to \mathbf{rec}\, t_2.(t_0 \vee \mathbf{rec}\, t_1.(t_2 \wedge t_0)))$$

not

$$\mathbf{rec}\, t_0.(\mathbf{rec}\, t_1.(\mathbf{rec}\, t_2.(t_0 \vee t_1) \wedge t_0) \to \mathbf{rec}\, t_2.(t_0 \vee \mathbf{rec}\, t_1.(\mathbf{rec}\, t_2.(t_0 \vee t_1) \wedge t_0)))$$

However, we can ignore the difference for the reason explained later.

Conversely, we define a function $G : \mathbf{TExp}^0 \to T_\Sigma^{\mathrm{fin}}$ that transforms a coformulaic expression into a finite coformula.

Define $* : \mathbf{TExp}^0 \to \Sigma \times \mathbf{TExp}^0 \times \mathbf{TExp}^0$ by:

$$\begin{array}{rcl}(\tau_0 \sigma \tau_1)^* & = & \langle \sigma, \tau_0, \tau_1 \rangle \\ (\mathbf{rec}\, t.\tau)^* & = & (\tau[\mathbf{rec}\, t.\tau/t])^*\end{array}$$

Then by the finality of T_Σ there exists unique arrow $[\![*]\!] : \mathbf{TExp}^0 \to T_\Sigma$ such that the following diagram commutes:

$$\begin{tikzcd}
\Sigma \times T_\Sigma \times T_\Sigma & \Sigma \times \mathbf{TExp}^0 \times \mathbf{TExp}^0 \arrow[l, "\langle id_\Sigma{,}[\![*]\!]{,}[\![*]\!]\rangle"'] \\
T_\Sigma \arrow[u, "\langle \mathbf{head}{,}\mathbf{tail}_0{,}\mathbf{tail}_1 \rangle"] & \mathbf{TExp}^0 \arrow[l, "{[\![*]\!]}"] \arrow[u, "*"']
\end{tikzcd}$$

Call this $[\![*]\!]$ G.

Lemma

Let S_0, S_1, S_2 *be type expressions that consist of type variables* t_0, t_1, t_3 *only and not containing* **rec***. Let* $\overline{\theta}$ *be (possibly empty) successive substitutions each of which is of the form* $[\mathbf{rec}t_i.S_i\overline{\theta'}/t_i]$*. Then* $G(\mathbf{rec}t_0.S_0[\mathbf{rec}t_1.S_1\overline{\theta}_1/t_1][\mathbf{rec}t_2.S_2\overline{\theta}_2/t_2])$ *is bisimilar to* $G(\mathbf{rec}t_0.S_0[\mathbf{rec}t_1.S_1\overline{\theta'_1}/t_1][\mathbf{rec}t_2.S_2\overline{\theta'_2}/t_2])$*.*

Proof. By usual coinduction. ☐

This result justifies the remark above.

Proposition

For each $\mathcal{M} \in T_{\Sigma}^{fin}$, $\mathcal{M}$ is bisimilar to $GF(\mathcal{M})$.

Proof. By usual coinduction, using the above lemma. ☐

Based on above result, we propose that coformulas should be restricted to T_{Σ}^{fin}.

This restriction has some advantage because

- it gets rid of the coformulas that cannot be captured by any finitery method,
- it enables us to focus on coformulas that have relevance in relation to computation and programming language,
- we can avail ourselves of existent semantics and a proof system.

1. Coformulas
2. Relation between logic and computation
3. Coformulas as recursive types
4. **Proofs**

Abramsky (1991b) formulates “logical interpretations” for the language of domain theory.

This language contains usual type constructors in domain theory, including $\rightarrow$ (function spaces), $\times$ (products), $\oplus$ (coalesced sums), $\mathscr{P}$ (plotkin powerdomains), **rec** (recursive domains), etc.

We can exploit this logic for coformulas.

Proof system

Let $\Sigma = \{\to, \land\}$ for simplicity.

We supply a natural deduction system with additional rules for each coformula $\mathcal{M} = \langle Q, q_0, \delta \rangle$ and $q \in Q$:

$$\frac{M : \delta(q)}{\mathbf{fold}_q^{\mathcal{M}}(M) : q}\ \textbf{rec}\text{-I} \qquad\qquad \frac{M : q}{\mathbf{unfold}_q^{\mathcal{M}}(M) : \delta(q)}\ \textbf{rec}\text{-E}$$

Here we abuse the notation $\delta(q)$ to denote $q' \sigma q''$, where $\delta(q) = \langle \sigma, q', q'' \rangle$.

We say $\mathcal{M}$ is proved if its initial state is proved.

$Q = \{q_0, q_1, q_2\}$
$\delta(q_0) = q_1 \to q_2,\ \delta(q_1) = q_2 \to q_0,\ \delta(q_2) = q_0 \to q_1,$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(x : q_0)}{\mathbf{unfold}_{q_0}(x) : q_1 \to q_2}\,\mathbf{rec}\text{-E} \qquad (y : q_1)}{\mathbf{unfold}_{q_0}(x)y : q_2}}{\mathbf{unfold}_{q_2}(\mathbf{unfold}_{q_0}(x)y) : q_0 \to q_1}\,\mathbf{rec}\text{-E} \qquad (x : q_0)}{\mathbf{unfold}_{q_2}(\mathbf{unfold}_{q_0}(x)y)x : q_1}}{\lambda x^{q_0}.(\mathbf{unfold}_{q_2}(\mathbf{unfold}_{q_0}(x)y)x) : q_0 \to q_1}\,x}{\mathbf{fold}_{q_2}(\lambda x^{q_0}.(\mathbf{unfold}_{q_2}(\mathbf{unfold}_{q_0}(x)y)x)) : q_2}\,\mathbf{rec}\text{-I}}{\lambda y^{q_1}.(\mathbf{fold}_{q_2}(\lambda x^{q_0}.(\mathbf{unfold}_{q_2}(\mathbf{unfold}_{q_0}(x)y)x))) : q_1 \to q_2}\,y}{\mathbf{fold}_{q_0}(\lambda y^{q_1}.(\mathbf{fold}_{q_2}(\lambda x^{q_0}.(\mathbf{unfold}_{q_2}(\mathbf{unfold}_{q_0}(x)y)x)))) : q_0}\,\mathbf{rec}\text{-I}
$$

Curry's paradox

$$Q = \{q_0, q_1\}$$
$$\delta(q_0) = q_0 \wedge q_0,\ \delta(q_1) = q_1 \to q_0$$

$$\dfrac{\dfrac{\dfrac{\dfrac{(x:q_1)}{\mathbf{unfold}_{q_1}(x):q_1\to q_0}\,\mathbf{rec}\text{-E} \qquad (x:q_1)}{\mathbf{unfold}_{q_1}(x)x:q_0}}{\lambda x^{q_1}.(\mathbf{unfold}_{q_1}(x)x):q_1\to q_0}\,x \qquad \dfrac{\dfrac{\dfrac{\dfrac{(x:q_1)}{\mathbf{unfold}_{q_1}(x):q_1\to q_0}\,\mathbf{rec}\text{-E}\qquad (x:q_1)}{\mathbf{unfold}_{q_1}(x)x:q_0}}{\lambda x^{q_1}.(\mathbf{unfold}_{q_1}(x)x):q_1\to q_0}\,x}{\mathbf{unfold}_{q_1}(\lambda x^{q_1}.(\mathbf{unfold}_{q_1}(x)x)):q_1}\,\mathbf{rec}\text{-I}}{(\lambda x^{q_1}.(\mathbf{unfold}_{q_1}(x)x))(\mathbf{unfold}_{q_1}(\lambda x^{q_1}.(\mathbf{unfold}_{q_1}(x)x))):q_0}$$

Remarks

- This system suggest that (finite) coformulas can be understood as self-referential sentences.
- One needs specification of coformulas outside proof. So this system may not seem purely syntactical. However, as far as finite coformulas are concerned, we can write down these specifications in a syntactic fashion.
- Some proofs has no normal form (as expected).

To sum up,

- There is a substantial gap between logic and programming, despite Curry-Howard isomorphism.
- Coformulas are logical analogues to recursive types in programming languages, and hence can fill the gap.
- As such, they may well be restricted to finite ones.
- Then a kind of type theory can be applied to coformulas.