1 Non-indexed co-algebras

Formation

$$\frac{A : Set}{F^\infty(A) : Set}$$

Morphism rule

$$\frac{A : Set \quad B : Set \quad f : A \to B}{\overline{F^\infty}(A, B, f) : F^\infty(A) \to F^\infty(B)}$$

Introduction

$$\frac{A : Set \quad B : Set \quad \gamma : B \to A + F(F^\infty(A) + B) \quad b : B}{intro(B, \gamma, b) : F^\infty(A)}$$

Elimination

$$\frac{A : Set \quad p : F^\infty(A)}{elim(A, p) : A + F(F^\infty(A))}$$

$$elim \circ intro(\gamma) \quad = \quad (1 + \overline{F}[1, intro(\gamma)]) \circ \gamma$$

2 Computational phenomena

The following is a concrete example of why co−iteration gives rise to a computationally expensive successor operation.

Let $S(X)$ be the functor $1 + X$, with constructors 0 and $s(x)$. So the set N^∞ of co−naturals equals $\nu(S)$. I also use $\gamma \mapsto S(\gamma)$ for the action of S on morphisms.

The eliminator $unS = elim : N^\infty \to S(N^\infty)$ decides whether its argument is 'zero' or a 'successor' (and of what). By finality, it has an inverse:

$$intro(S(N^\infty), S(unS)) : S(N^\infty) \to N^\infty$$

The successor on N^∞ is

$$intro(S(N^\infty), S(unS)) \circ s : N^\infty \to N^\infty$$

. . . where

$$\begin{array}{rcl} g & : & (S(N^\infty)) \to S^2(N^\infty) \\ g & = & [\lambda x \bullet \mathsf{in_R}(\mathsf{in_R}(n_0)), \lambda x \bullet \mathsf{in_L}(elim(x))] \end{array}$$

Now

$$elim(intro(1 + N^\infty, g, \mathsf{in_L})) = \mathsf{in_R}(intro(1 + N^\infty, g, \mathsf{in_R}(n_0)))$$

which is bisimular to $\mathsf{in_R}(n_0)$.

If $n_0 = \mathsf{s}(n_1)$ and we apply $elim$ once more to the above i.e.

$$elim(intro(1 + N^\infty, g, \mathsf{in_R}(n_0)) = \mathsf{in_L}(intro(1 + N^\infty, g, \mathsf{in_R}(n_1'))$$

where n_1' is the predecessor of n_0, (*i.e.* $n_1' = \mathsf{in_R}(intro(1 + N^\infty, g', \mathsf{in_R}(n_1)))$ for some g'), then we obtain

$$\mathsf{in_L}(intro(1 + N^\infty, g, \mathsf{in_R}(\mathsf{in_R}(intro(1 + N^\infty, g', \mathsf{in_R}(n_1))))))$$

instead of n_1.

So $pred^n(\mathsf{s}^n(a))$ gives rise to a term which has length at least $length(a) + n$

3 State-indexed co-algebras

Here is now a new version of the rules for co−algebras with which I am quite satisfied. There will probably be some mistakes (my concentration is not at the top level at the moment).

Assume $S : Set$, and that α is a code (of one kind or another) for a functor F_α with monotonicity map $\overline{F_\alpha}$

$$\begin{array}{rcl} F_\alpha & : & Pred(S) \to Pred(S) \\ \overline{F_\alpha} & : & (A, B : Pred(S), f : A \subseteq B) : F_\alpha(A) \subseteq F_\alpha(B) \end{array}$$

We suppress α.

3.1 Rules

Formation

$$\frac{s : S}{Nu(s) : Set}$$

or $Nu : Pred(S)$

Introduction (Corecursion: dual of iteration with "side−argument".)

$$\frac{s : S \quad B : Pred(S) \quad \gamma : B \subseteq F(Nu \cup B) \quad b : B(s)}{intro(B, \gamma, b) : Nu(s)}$$

($intro$ is "polymorphic" in s) or

$$\frac{B : Pred(S) \quad \gamma : B \subseteq F(Nu \cup B)}{intro(B, \gamma) : B \subseteq Nu}$$

Elimination (Final co−algebra.)

$$\frac{s : S \quad a : Nu(s)}{elim(s, a) : F(Nu, s)}$$

or $elim : Nu \subseteq F(Nu)$

Computation

$$elim(s, intro(B, \gamma, b)) \quad = \quad \overline{F}(Nu \cup B, Nu, [id, intro(B, \gamma)], s, \gamma(s, b))$$

or $elim \circ intro(B, \gamma) = \overline{F}(Nu \cup B, Nu, [id, intro(B, \gamma)]) \circ \gamma$.

3.2 Alternative rule

There is an alternative introduction rule:

$$\begin{array}{lll} \mathit{intro}'(& s & : & S, \\ & B & : & \mathit{Pred}(S), \\ & \gamma & : & B \subseteq \mathit{Nu} \cup F(B), \\ & b & : & B(s)) \\ \to & \mathit{Nu} \end{array}$$

But this is reducible to the rule above, since intro' can be defined by

$$\begin{array}{lllll} \mathit{intro}'(s, B, \gamma, b) & = & \mathsf{case} & \gamma(b) & \mathsf{of} \\ & & & \mathsf{in_L}(p) & \to & p \\ & & & \mathsf{in_R}(p) & \to & \mathit{intro}(s, \\ & & & & & \quad F(B), \\ & & & & & \quad \overline{F}(B, \\ & & & & & \qquad \mathit{Nu} \cup F(B), \\ & & & & & \qquad \gamma), \\ & & & & & \quad p) \end{array}$$

i.e.

$$\mathit{intro}'(B, \gamma) = [\mathit{id}, \mathit{intro}(F(B), \overline{F}(B, \mathit{Nu} \cup F(B), \gamma))] \circ \gamma$$

3.3 Modifying the functor

If $A : \mathit{Pred}(S)$, let α_A be a code for the functor

$$\begin{array}{l} F_{\alpha_A}(X) = A \cup F_\alpha(X) \\ \overline{F_{\alpha_A}}(X, Y, f) = \mathit{id} + \overline{F_\alpha}(X, Y, f) \end{array}$$

Let $F^\infty_\alpha(A) = \mathit{Nu}_{\alpha_A}$.

So we have the rules:

$$\mathit{elim}_{\alpha_A} : F^\infty_\alpha(A) \subseteq A \cup F_\alpha(F^\infty_\alpha(A))$$

$$\frac{\xi : X \subseteq A \cup F_\alpha(F^\infty_\alpha(A) \cup X)}{\mathit{intro}_{\alpha_A} : X \subseteq F^\infty_\alpha(A)}$$

We expect $F^\infty(\cdot)$ to be monadic.

- Leaf. We have $\mathsf{in_L} : A \subseteq A \cup F_\alpha(\ldots)$.

  Therefore $\mathit{intro}_{\alpha_A}(A, \mathsf{in_L}) : A \subseteq F^\infty_\alpha(A)$.

- Do.

3.4 Repeat

We can now define repeat:

Assume

$$\begin{array}{rcl} B & : & Pred(S) \\ \gamma & : & B \subseteq F_\alpha(F_\alpha^\infty(A) + F_\alpha^\infty(B)) \\ s & : & S \\ b & : & B(s) \end{array}$$

Define

(following is jumbled up...)

$$\begin{array}{l} repeat(B, \gamma, s, b) : F_\alpha^\infty(A) \\ = intro_{\alpha_A}(F_\alpha^\infty(B), \lambda s \bullet [\mathsf{in_L} \circ \gamma, \mathsf{in_L} \circ \overline{F_\alpha}(\ldots, \mathsf{in_R})] \circ elim_{\alpha_B}, s)], leaf(s, b)) \end{array}$$

where

$$leaf(s, b) \quad = \quad intro_\alpha(1, \lambda x \bullet \mathsf{in_L}(b), 0)$$

3.5 While

(Jumbled up.)

$$\begin{array}{l} while(B : Pred(S), \gamma : B \subseteq F_\alpha^\infty(A) \cup F_\alpha(F_\alpha^\infty(B)), s : S, b : B(s)) \\ = intro'_\alpha(F_\alpha^\infty(B), \lambda s \bullet [\gamma, \mathsf{in_R}] \circ elim_{\alpha_B}(s), leaf(b)) \end{array}$$

3.6 Do

...

3.7 Redirect

...

3.8 Discussion

So I have the feeling that the definitions of while and repeat are essentially as complex as direct computation rules would be, therefore I think the above rules (modulo some typing errors) are the right ones.

4 Some Haskell explorations

This is a so-called 'literate' Haskell script. What this means is that the code lines are the ones that start with `>`; all the other lines are comments.

The dot means composition. As in Agda, `` `foo` `` is an infix form of `foo`. The function `fmap` is like `F_mor` – there is some general magic for figuring out the morphism part of functors. The definition is worked out by the type checker; I think the occurrences of `fmap` are all for `Maybe`.

```
> import Prelude hiding ((*)) -- ((*),Maybe,maybe)

It is useful to define an operator for "backwards" composition

> (*)           = flip (.)

The type operator Maybe is defined by

  data Maybe a  = Nothing | Just a
  maybe         :: a -> (b -> a) -> Maybe b -> a
  maybe a f     = \ m -> case m of Nothing -> a ; Just b -> f b

We introduce the "naturals".  In Haskell we just have
fixed points, so they can serve both as naturals (iter, rec)
and co-naturals (coiter, corec).

> data N        = N (Maybe N) deriving (Show ,Eq)

The constructor N is the initial algebra for Maybe.
The following is the final co-algebra.

> unN (N n)     = n

  zeroN         = N Nothing
  succN         = N . Just

> iter f        = f . fmap (iter f) . unN
> coit f        = f * fmap (coit f) * N

> recur f       = f . fmap (id `conj` (recur f)) . unN
> corec f       = f * fmap (id `disj` (corec f)) * N

We can also define recursors and co-recursors using
iteration and co-iteration.

> recur' f      = p1 . iter ((N   . fmap p0) `conj` f)
> corec' f      = i1 * coit ((unN * fmap i0) `disj` f)
```

The types of the functions above are as follows.

```
> iter          :: (Maybe a -> a) -> N -> a
> recur         :: (Maybe (Conj N a) -> a) -> N -> a
> recur'        :: (Maybe (Conj N a) -> a) -> N -> a

> coit          :: (a -> Maybe a) -> a -> N
> corec         :: (a -> Maybe (Disj N a)) -> a -> N
> corec'        :: (a -> Maybe (Disj N a)) -> a -> N
```

The following are versions of + and *.

```
> data Conj a b = Conj a b
> conj a b      = \c-> Conj (a c) (b c)
> p0 (Conj a b) = a
> p1 (Conj a b) = b

> data Disj a b = I0 a | I1 b
> disj a b xy   = case xy of I0 x -> a x ; I1 y -> b y
> i0            = I0
> i1            = I1
```

For testing

```
> zeroN         = N Nothing
> succN         = N . Just
> num n         = it succN zeroN !! n where it f a = a : it f (f a)

> add  m n      = iter  (maybe m succN)  n
> add' m n      = recur (maybe m (succN . p1))  n
```

Inverse of initial algebra

```
> predSlow      :: N -> Maybe N
> predSlow      = iter  (fmap N)
> predFast      = recur (fmap p0)
```

Inverse of final coalgebra (made into successor)

```
> succSlow      :: N -> N
> succSlow      = coit  (fmap unN) . Just
> succFast      = corec (fmap i0)  . Just
```

5 Codes

5.1 famfam

Given a (global) set S of states.

A *predicate* is a function from states to sets.

$$P : \mathit{Pred}(S) = S \to \mathit{Set}$$

A *family* is a ....

$$\mathit{Fam}(S) = \sum I : \mathit{Set} \bullet I \to S$$

A *quantifier* is a predicate of predicates:

$$Q : \mathit{Pred}(\mathit{Pred}(S))$$

A *family of families*

$$\mathit{ff} = \langle C, \lambda c \bullet \langle R(c), \lambda r \bullet n(c,r) \rangle \rangle : \mathit{Fam}(\mathit{Fam}(S))$$

Each family of families ff determines a quantifier (d.n.f.)

$$Q_{\mathit{ff}}(P) = \sum c : C \bullet \prod r : R(c) \bullet P(n(c,r))$$

A *predicate transformer* is a function from predicates to predicates:

$$F : \mathit{Pred}(S) \to \mathit{Pred}(S)$$

Each $\phi : S \to \mathit{Pred}(\mathit{Pred}(S))$ determines a predicate transformer:

$$\widetilde{\phi}(P) = \lambda s \bullet \phi(s, P)$$

We can use $\alpha : S \to \mathit{Fam}(\mathit{Fam}(S))$ as codes.
If α represents F, then we can find α^* representing $\mu X . A \cup F(X)$.
Can we find α^∞ representing $\nu X . A \cup F(X)$?

6 Possible conventions, notations.

6.1 Monadic closure of a functor

Some notation:

$$\begin{array}{rcl} F^A(X) & = & A + F(X) \\ F_A(X) & = & F(A + X) \end{array}$$

Least closure.

$$\begin{array}{rcccccl} F_1^*(A) & = & \mu Y.F(A+Y) & = & \mu(F_A) & = & F(F^* A) \\ F^*(A) & = & \mu Y.A + F(Y) & = & \mu(F^A) & = & A + F_1^* A \end{array}$$

Greatest closure.

$$\begin{array}{rcccccl} F_1^\infty(A) & = & \nu Y.F(A+Y) & = & \nu(F_A) & = & F(F^\infty A) \\ F^\infty(A) & = & \nu Y.A + F(Y) & = & \nu(F^A) & = & A + F_1^\infty A \end{array}$$

so that

$$\begin{array}{rcl} F^\infty(A) & = & A + F(A + F(A + \ldots)) \\ F_1^\infty(A) & = & F(A + F(A + F(A + \ldots))) \end{array}$$

(That this is ∞ means that we allow always the right branch. That is were $*$ would mean that the left branch gets treated fairly.)

(Notational analogy with $\sum_0^\infty \ldots$, $\int_1^\infty \ldots$, *etc.*.)

Something like $F \circ (A+)$ *vs.* $(A+) \circ F$. So $(A+)$ is the functor which adds A on the left. (For any A it is a monad; a generalisation of $S(X) = 1 + X$; $S = (1+)$.)

6.1.1 Monad laws

$$\eta : A \to M(A) \qquad\qquad \frac{f : A \to M(B)}{f^\sharp : M(A) \to M(B)}$$

$$\begin{array}{lcl} \eta^\sharp & = & id_{M(A)} \\ f^\sharp \circ \eta & = & f \\ (f^\sharp \circ g)^\sharp & = & f^\sharp \circ g^\sharp \end{array}$$

- It is simple to see that $F^*(A) = \mu X.A + F(X)$ is a monad.

  $\eta = \mathsf{in}_{F^A} \circ \mathsf{in}_\mathsf{L}$.

  $f^\sharp = (\![f, \mathsf{in}_{F^B} \circ \mathsf{in}_\mathsf{R}]\!)$

- Less simply, $F^\infty(A) = \nu X.A + F(X)$ is a monad.

  $\eta = (\mathsf{in}_{F^A})^{-1} \circ \mathsf{in}_\mathsf{L}$.

  Given $f : A \to F_0^\infty(B)$, we have

  $$\begin{array}{rcl} f' & = & [(id_B + F(\mathsf{in}_\mathsf{L})) \circ \mathsf{out}_{F^B} \circ f, \mathsf{in}_\mathsf{R} \circ F(\mathsf{in}_\mathsf{R})] \circ \mathsf{out}_{F^A} \\ & : & F_0^\infty(A) \to B + F(F_0^\infty(B) + F_0^\infty(A)) \end{array}$$

  From this we get $f^\sharp$ by corecursion.

6.2 Square bracket notation for coproducts $A + B$

$$\begin{aligned} \mathsf{in_L} &: A \to A + B \\ \mathsf{in_R} &: B \to A + B \end{aligned}$$

$$\frac{a : A \to C \quad b : B \to C}{[a, b] : A + B \to C}$$

$$f = [a, b] \iff f \circ \mathsf{in_L} = a \wedge f \circ \mathsf{in_R} = b$$

This rule can be decomposed:

Cancellation (β) Ordinary computation rule.

$$[a, b] \circ \mathsf{in_L} = a \wedge [a, b] \circ \mathsf{in_R} = b$$

Reflection (η) Somewhat dubious.

$$id = [\mathsf{in_L}, \mathsf{in_R}]$$

Fusion (permutation) Rather dubious, except perhaps in an equality checking algorithm.

$$f \circ [a, b] = [f \circ a, f \circ b]$$

$$\begin{tikzcd}
A \arrow[r, "\mathsf{in_L}"] \arrow[dr, "a"'] \arrow[ddr, "f \circ a"'] & A + B \arrow[d, dashed, "{[a,b]}"] & B \arrow[l, "\mathsf{in_R}"'] \arrow[dl, "b"] \arrow[ddl, "f \circ b"] \\
& C \arrow[d, dashed, "f"] & \\
& D &
\end{tikzcd}$$

The following definition is quite useful.

$$a + b \triangleq [\mathsf{in_L} \circ a, \mathsf{in_R} \circ b] : A + B \to A' + B'$$

Write $F[\ldots, \ldots]$, not $F([\ldots, \ldots])$.
As N_0 is a "nullary" $+$, one can also use

$$[] : N_0 \to C$$

6.3 Angle bracket notation for products $A \times B$

$$\begin{array}{lcl} \mathsf{out_L} & : & A \times B \to A \\ \mathsf{out_R} & : & A \times B \to B \end{array}$$

$$\frac{a : C \to A \quad b : C \to B}{\langle a, b \rangle : C \to A \times B}$$

$$f = \langle a, b \rangle \iff \mathsf{out_L} \circ f = a \wedge \mathsf{out_R} \circ f = b$$

This rule can be decomposed:

Cancellation (β) Ordinary computation rule.

$$\mathsf{out_L} \circ \langle a, b \rangle = a \wedge \mathsf{out_R} \circ \langle a, b \rangle = b$$

Reflection (η) Surjective pairing.

$$id = \langle \mathsf{out_L}, \mathsf{out_R} \rangle$$

Fusion (permutation) Rather dubious.

$$\langle a, b \rangle \circ f = \langle a \circ f, b \circ f \rangle$$

$$\begin{tikzcd}
A & A \times B \arrow[l, "\mathsf{out_L}"'] \arrow[r, "\mathsf{out_R}"] & B \\
& C \arrow[ul, "b"] \arrow[u, "\langle a, b \rangle"] \arrow[ur, "a"'] & \\
& D \arrow[u, "f"] \arrow[uul, "b \circ f"] \arrow[uur, "a \circ f"'] &
\end{tikzcd}$$

The following definition is quite useful.

$$a \times b \triangleq \langle a \circ \mathsf{out_L}, b \circ \mathsf{out_R} \rangle : A \times B \to A' \times B'$$

Write $F\langle \ldots, \ldots \rangle$, not $F(\langle \ldots, \ldots \rangle)$.
As N_1 is a "nullary" $\times$, one can also use

$$\langle \rangle : C \to N_1$$

6.4 Convex lens bracket notation for initial algebras μF

$$\mathsf{in}_F \quad : \quad F(A) \to A$$

$$\frac{a : F(A) \to A}{(\![a]\!)_F : \mu F \to F(A)}$$

$$f = (\![a]\!)_F \iff f \circ \mathsf{in}_F = a \circ F(\![a]\!)_F$$

Write $F(\![\ldots,\ldots]\!)$, not $F((\![\ldots,\ldots]\!))$.
The last rule can be decomposed:

Cancellation (β) Ordinary computation rule.

$$(\![a]\!)_F \circ \mathsf{in}_F = a \circ F(\![a]\!)_F$$

Reflection (η) Weird.

$$id = (\![\mathsf{in}_F]\!)_F$$

Fusion No counterpart in proof theory.

$$\frac{f \circ a = b \circ F(f)}{f \circ (\![a]\!)_F = (\![b]\!)_F}$$

$$\begin{tikzcd}
F(\mu F) \arrow[r, "\mathsf{in}_F"] \arrow[d, "F(\![a]\!)_F"'] & \mu F \arrow[d, "(\![a]\!)_F"] \\
F(A) \arrow[r, "a"] \arrow[d, "F(f)"'] & A \arrow[d, "f"] \\
F(B) \arrow[r, "b"] & B
\end{tikzcd}$$

There is a lemma of some kind. (Primitive recursion or "Paramorphisms".)

Suppose $a : F(A) \to A$ is an algebra for F (for example 'the' initial algebra $\mathsf{in}_F : F(\mu F) \to \mu F$). Then for any $f : F(A \times B) \to A$ there is a unique $f' : \mu F \to B$ such that the following diagram commutes.

$$\begin{tikzcd}
F(\mu F) \arrow[r, "\mathsf{in}_F"] \arrow[d, "F\langle (\![a]\!)_F, f' \rangle"'] & \mu F \arrow[d, "f'"] \\
F(A \times B) \arrow[r, "f"'] & B
\end{tikzcd}$$

Write $\mathsf{rec}(f)$ for f' when a is the initial algebra.

6.5 Concave lens bracket notation for final co-algebras νF

$$\mathsf{out}_F \quad : \quad A \to F(A)$$

$$\frac{c : A \to F(A)}{[\!(a)\!]_F : A \to \nu F}$$

$$f = [\!(a)\!]_F \iff \mathsf{out}_F \circ f = F[\!(a)\!]_F \circ c$$

Write $F[\!(\ldots,\ldots)\!]$, not $F([\!(\ldots,\ldots)\!])$.
The last rule can be decomposed:

Cancellation (β) Computation rule.

$$\mathsf{out}_F \circ [\!(a)\!]_F = F[\!(a)\!]_F \circ c$$

Reflection (η) Somewhat weird.

$$id = [\!(\mathsf{out}_F)\!]_F$$

Fusion (permutation) No proof theoretical counterpart.

$$\frac{c \circ f = F(f) \circ b}{[\!(c)\!]_F \circ f = [\!(b)\!]_F}$$

$$\begin{tikzcd}
\nu F \arrow[r, "\mathsf{out}_F"] & F(\nu F) \\
F(C) \arrow[u, "{[\!(c)\!]_F}"] \arrow[r, "c"] & F(C) \arrow[u, "{F[\!(c)\!]_F}"'] \\
B \arrow[u, "f"] \arrow[r, "b"] & F(B) \arrow[u, "F(f)"']
\end{tikzcd}$$

There is a lemma of some kind. ("Apomorphisms".)

Suppose $a : A \to F(A)$ is a coalgebra for F (for example 'the' final coalgebra $\mathsf{out}_F : \nu F \to F(\nu F)$). Then for any $f : B \to F(A + B)$ there is a unique $f' : B \to \nu F$ such that the following diagram commutes.

$$\begin{tikzcd}
\nu F \arrow[r, "\mathsf{out}_F"] & F(\nu F) \\
B \arrow[u, "f'"'] \arrow[r, "f"'] & F(A + B) \arrow[u, "{F[[\!(a)\!]_F, f']}"']
\end{tikzcd}$$

Write $\mathsf{corec}(f)$ for f' when a is the final coalgebra.

7 Some general remarks

7.1 Answers/results

The *simplest* kind of program is one that does not terminate with results – we can of course recover this type by taking an empty set of answers.

The reason for considering answers (*i.e.* for having a monadic operation on types) is to model sequential composition.

7.2 Programs with answers/results.

An interactive program is a value of a coinductive type, more particularly a type $\nu(F)$ for F of the form

$$\lambda X \bullet A + \sum c : C \bullet X^{R(c)}$$

We force there to be a special $\checkmark(a)$ command for each $a : A$, that has no responses. ("Tick" for successful/complete termination.)

$$\begin{array}{lcl} C' & = & A + C \\ R'(\mathsf{in_L}(a)) & = & N_0 \\ R'(\mathsf{in_R}(c)) & = & R(c) \end{array}$$

Then we have a maximal fixed point of $\lambda X \bullet \mathit{fam}_{(C',R')}(X)$.

Such a value represents (can be interpreted, or used as) a program for one agent (the actor, fetch unit, or master) to issue commands (values $c : C$) to another agent (the reactor, execution unit, or slave). The reactor carries out the command somehow, then may (or may not) respond to the command with a value $r : R(c)$. The response is used by the actor to determine which piece of the program to use next. Execution of the program may terminate, yielding an anser/result.

Terminology for agents:		
	actor	reactor
	master	slave
	client	server
	issuer	responder
	user	system
	fetch unit	execution unit
	environment	system

If the language in which the program is written has a theorem of reduction to head normal form, then a command will always be forthcoming.

7.3 How things can turn out

The eventual situations are as follows:

1. At some point, the actor never issues another command. Perhap she dies, becomes bored, breaks, goes into an internal loop, *etc.*. (The case when the set of commands is empty is uninteresting; it is however interesting in the state dependent case.)

2. At some point, the actor issues a command c with no response possible ($R(c)$ is empty).

3. At some point, the actor issues a command command c to which the execution/iterpretation system/mechanism (or environment) returns no response, although a response is possible ($R(c)$ is *not* empty), but the environment dies, becomes bored, breaks, goes into an internal loop, *etc.*.

4. At some point, execution of the program is complete, and terminates with some result $a : A$. This is something like surrendering a game; it is the final outcome of the program, determined by the successive responses of the reactor.

5. The actor keeps issuing commands, and the environment keeps responding. The program is never complete; no result is ever produced.

Stick to terminology: issue a command, get a response, terminate with result (or an answer? Virtue is doesn't begin with 'R').