Sequential algorithms (old and new)

Pierre-Louis Curien

(CNRS – Paris 7 – INRIA)

LAP 2015, 21-25/9/2015, Dubrovnik

8/5/2015, Ruanjiansuo, Beijing, 7/5/2015, Jiaoda, Shanghai,

and 11/4/2015, Galop workshop, London

Prologue : denotational and operational semantics

Given a a (piece of) typed program M written in some programming language, we want to understand its meaning.

• The denotational approach associates some mathematical structure to the type of M, and a suitable morphism $[\![M]\!]$ to M. [Typically, continuous functions between complete partial orders (cpo's).]

• The operational approach specifies formal rules of execution (a machine, a rewriting system,. . .) leading to observable results, which one can see as experiments.

• The two approaches induce each a notion of equivalence :

$$M =_{den} N \quad \text{iff} \quad [\![M]\!] = [\![N]\!]$$

$$M =_{op} N \quad \text{iff there is no context } C[] \text{ s.t.} \begin{cases} C[M] \longrightarrow^* v \\ C[N] \longrightarrow^* w \\ \text{and } v \neq w \end{cases}$$

When these two equalities are the same, the (denotational) model is called fully abstract (FA).

Complete partial orders through a key (even the founding) example

Consider the set $\mathbf{N}$ of natural numbers, and the set PF of partial functions from $\mathbf{N}$ to $\mathbf{N}$.

• PF has the structure of a partial order : $f \leq g$ iff whenever f is defined (notation $f\downarrow$), g is also defined and has the same value. [Information order]

• There is a minimum element $\perp$: the nowhere defined partial function. [Diverging computation]

• Every increasing chain has a least upper bound [Useful to give meaning to programs defined by general recursive equations]

A few dates : the old days

• Triggered by the full abstraction problem for PCF (a typed λ-calculus with arithmetical functions, conditionals and recursion), and building on Kahn-Plotkin's notion of sequential functions between (domains generated by) concrete data structures (called information matrices in their original work), Berry and Curien proposed a cartesian closed category of

sequential algorithms (1979) (SA in the sequel)

(first category in denotational semantics with morphisms that were not functions, but programs of some sort).

• This led to the design of the programming language CDS (early 1980's : Berry, Curien, Devin, Ressouche, Montagnac). The development of this language did not survive the 1980's...

• The model SA was shown not to be amenable to a FA model of PCF. Counter-examples to definability were exhibited in my Thèse d'Etat (1983). But...

A few dates : revisitations

• The model SA was shown to be FA for PCF plus a form of control operator (catch) (Curien, Cartwright, Felleisen 1992).

As part of this work, SA's were recovered as observably sequential functions.

• The functions spaces of SA (for its full subcategory of sequential data structures) was shown to be decomposable as $S \to S' = (!S) \multimap S'$ (Lamarche 1992, Curien 1994). [This exponential is to the one of McCusker (1996) for HO games what the set-based exponential of coherence spaces is to its multiset version.]

• The last revisitation was the link with Laird's bistability (Curien 2009).

• Related works : Bucciarelli and Ehrhard's strong stability, Kleene's unimonotone functions, Longley's sequentially realisable functionals,. . .

This talk

Old : I shall recall the sequential algorithms “of the old days” (self-contained).
I shall introduce a category
• whose objects are concrete data structures, which are kits for assembling atoms to build data,
• and whose morphisms will be pairs of an ordinary function + a computation strategy for it.

Old and new : I’ll exhibit an abstract machine describing the composition of sequential algorithms presented as programs. The machine is inspired by the operational semantics underpining the language CDS.

New : As an application, I’ll give a new proof of the utlimate obstinacy theorem (Colson 1989) (my proof follows very much the lines of David’s proof, who had constructed an ad hoc quite “SA”-like setting for this purpose).

Concrete data structures

A concrete data structure (or *cds*) $\mathbf{M} = (C, V, E, \vdash)$ is given by three sets C, V, and $E \subseteq C \times V$ of *cells*, *values*, and *events*, and a relation $\vdash$ between finite parts of E and elements of C, called the enabling relation. We write simply $e_1, \ldots, e_n \vdash c$ for $\{e_1, \ldots, e_n\} \vdash c$. A cell c such that $\vdash c$ is called *initial*.

Proofs of cells c are sets of events defined recursively as follows : If c is initial, then it has an empty proof. If $(c_1, v_1), \ldots, (c_n, v_n) \vdash c$, and if $p_1, \ldots, p_n$ are proofs of $c_1, \ldots, c_n$, then $p_1 \cup \{(c_1, v_1)\} \cup \cdots \cup p_n \cup \{(c_n, v_n)\}$ is a proof of c.

States (or strategies, in the game semantics terminology)

A state is a subset x of E such that :

(1) $(c, v_1), (c, v_2) \in x \Rightarrow v_1 = v_2$.

(2) If $(c, v) \in x$, then x contains a proof of c.

The conditions (1) and (2) are called consistency and safety, respectively.

The set of states of a cds $\mathbf{M}$, ordered by set inclusion, is a partial order denoted by $(D(\mathbf{M}), \leq)$ (or $(D(\mathbf{M}), \subseteq)$). If D is isomorphic to $D(\mathbf{M})$, we say that $\mathbf{M}$ generates D.

[$D(\mathbf{M})$ is a Scott domain with additional properties $\rightarrow$ Kahn-Plotkin's representation theorem.]

Some terminology

Let x be a set of events of a cds. A cell c is called :

- filled (with v) in x iff $(c, v) \in x$,
- enabled in x iff x contains an enabling of c,
- accessible from x iff it is enabled, but not filled in x.

We denote by $F(x)$, $E(x)$, and $A(x)$ the sets of cells which are filled, enabled, and accessible in or from x, respectively. We write :

$$x \prec_c y \quad \text{if} \quad c \in A(x) \text{ and } x \cup \{(c, v)\} = y$$

Some conditions on cds's

Let $\mathbf{M} = (C, V, E, \vdash)$ be a cds. We define three properties defining subclasses of cds's

(A) $\mathbf{M}$ is well-founded : no infinite proofs.

Well-foundedness allows us to reformulate the safety condition as a local condition :
$(2')$ If $(c, v) \in x$, then x contains an enabling $\{e_1, \ldots, e_n\}$ of c.

(B) $\mathbf{M}$ is stable, *i.e.*, for any state x and any cell c, c has at most one enabling in x.

(C) $\mathbf{M}$ is filiform. Every enabling contains at most one event.

We shall always assume that $\mathbf{M}$ is well-founded (for convenience) and *stable* (essential to make sure that our morphisms induce well-defined domain-theoretic function). We shall see that the filiform assumption, while not necessary, allows us to simplify matters greatly.

Some examples of cds's

(1) Flat cpo's : for any set $\mathbf{X}$ we have a cds

$\mathbf{X}_\perp = (\{?\}, \mathbf{X}, \{?\} \times \mathbf{X}, \{\vdash ?\})$ with $D(\mathbf{X}_\perp) = \{\emptyset\} \cup \{(?, x) \mid x \in \mathbf{X}\}$

Typically, we have the flat cpo $\mathrm{N}_\perp$ of natural numbers.

(2) λ-calculus (cells as occurrences) :

$$C = \{0, 1, 2\}^* \quad V = \{\cdot\} \cup \{x, \lambda x \mid x \in Var\} \quad E = C \times V$$
$$\vdash \epsilon \qquad (u, \lambda x) \vdash u0 \qquad (u, \cdot) \vdash u1, u2$$

(3) Pairs of booleans : we have two cells $?.1$ and $?.2$ (both initial) and two values T, F, and all possible events. Then

$$(T, F) = \{(?.1, T), (?.2, F)\} \qquad (F, \perp) = \{(?.1, F)\} \qquad (\perp, \perp) = \emptyset$$

(4) A non-stable cds : $\mathbf{NS} = (\{c_1, c_2, c_3\}, \{1, 2\}, E, |-)$, with $E = \{c_1, c_2, c_3\} \times \{1, 2\}$, $\vdash c_1'$, $\vdash c_2'$, $(c_1', 1) \vdash c_3'$, and $(c_2', 1) \vdash c_3'$.

Key example for this talk : lazy natural numbers

This (filiform) cds has cells $c_0, \ldots, c_n, \ldots$ and values 0 or S, with events $(c_i, 0)$ and (c_i, S), and enablings given by

$$\begin{array}{l} \vdash c_0 \\ (c_i, S) \vdash c_{i+1} \end{array}$$

We have

$$D(\mathbf{N}_L) = \{S^n(\bot) \mid n \in \omega\} \cup \{S^n(0) \mid n \in \omega\} \cup \{S^\omega(\bot)\}$$

which as a partial order is organised as the following tree :

$$c_0 \left\{ \begin{array}{l} 0 \\ S\ c_1 \left\{ \begin{array}{l} 0 \\ S\ c_2 \left\{ \begin{array}{l} 0 \\ \ldots \end{array} \right. \end{array} \right. \end{array} \right. \quad \text{or} \quad \left\{ \begin{array}{l} 0 \\ S(\bot) \left\{ \begin{array}{l} S(0) \\ S(S(\bot)) \left\{ \begin{array}{l} S(S(0)) \\ \ldots \end{array} \right. \end{array} \right. \end{array} \right.$$

Product of two cds's

Let M and M' be two cds's. We define the product $\mathrm{M}\times\mathrm{M}' = (C, V, E, \vdash)$ of M and M' by :

- $C = \{c.1 \mid c \in C_{\mathrm{M}}\} \cup \{c'.2 \mid c' \in C_{\mathrm{M}'}\}$,

- $V = V_{\mathrm{M}} \cup V_{\mathrm{M}'}$,

- $E = \{(c.1, v) \mid (c, v) \in E_{\mathrm{M}}\} \cup \{(c'.2, v') \mid (c', v') \in E_{\mathrm{M}'}\}$,

- $(c_1.1, v_1), \ldots, c_n.1, v_n) \vdash c.1 \Leftrightarrow (c_1, v_1), \ldots (c_n, v_n) \vdash c$ (and similarly for M').

Fact : $\mathrm{M} \times \mathrm{M}'$ generates $D(\mathrm{M}) \times D(\mathrm{M}')$.

Sequential algorithms (preview)

We shall build a **category** whose objects are cds's and whose morphisms are *programs of some sort* (that can also be equivalently described in a number of ways). Here is a prototypical sequential algorithm from $\mathbf{N}_\perp \times \mathbf{N}_\perp$ to $\mathbf{N}_\perp$ (we decorate the output cell as $?'$) :

$$add_L = request\,?'\ (from\{\})\, valof\,?.1\, is \begin{cases} \vdots \\ m \mapsto valof\ ?.2\ is \begin{cases} \vdots \\ n \mapsto m+n \\ \vdots \end{cases} \\ \vdots \end{cases}$$

This program specifies a left-to-right algorithm for addition. By interchanging $?.1$ and $?.2$, we get the right-to-left sequential algorithm for addition. Both compute the same underlying function.

Exponent of two cds's

If $\mathbf{M}$, $\mathbf{M}'$ are two cds's, the cds $\mathbf{M} \to \mathbf{M}'$ is defined as follows :

- If x is a finite state of $\mathbf{M}$ and $c' \in C_{\mathbf{M}'}$, then xc' is a cell of $\mathbf{M} \to \mathbf{M}'$.

- The values and the events are of two types :

– If c is a cell of $\mathbf{M}$, then $valof\ c$ is a value of $\mathbf{M} \to \mathbf{M}'$, and $(xc', valof\ c)$ is an event of $\mathbf{M} \to \mathbf{M}'$ iff c is accessible from x ;
– if v' is a value of $\mathbf{M}'$, then $output\ v'$ is a value of $\mathbf{M} \to \mathbf{M}'$, and $(xc', output\ v')$ is an event of $\mathbf{M} \to \mathbf{M}'$ iff (c', v') is an event of $\mathbf{M}'$.

- The enablings are also of two types :

$(yc', valof\ c) \vdash xc'$	iff	$y \prec_c x$
$\ldots, (x_i c'_i, output\ v'_i), \ldots \vdash xc'$	iff	$x = \bigcup x_i$ and $\ldots, (c'_i, v'_i), \ldots \vdash c'$

The need for the stability condition on cds's

A state of $\mathbf{M} \to \mathbf{M}'$ should define a function from $D(\mathbf{M})$ to $D(\mathbf{M}')$, i.e. from states to *states* :

$$x \mapsto a \bullet x = \{(c', v') \mid \exists\, y \leq x \;\; (yc', output\ v') \in a\}$$

Consider the following state a in $\mathbf{X}_\perp \to \mathbf{NS}$ (with $X = \{\star\}$) :

$$\begin{array}{rcl} a & = & \{(\perp c'_1, output\ 1), (\perp c'_2, valof\ ?), (\{(?, \star)\} c'_2, output\ 1), \\ & & (\perp c'_3, output\ 1), (\{(?, \star)\} c'_3, output\ 2)\} \end{array}$$

Then $a \bullet \{(?, \star)\}$ is not a state of $\mathbf{NS}$, as it contains $(c'_3, 1)$ and $(c'_3, 2)$.

If $\mathbf{M}'$ is stable, then indeed $x \mapsto a \bullet x : D(\mathbf{M}) \to D(\mathbf{M}')$.

[Moreover, $x \mapsto a \bullet x$ is a sequential function, and any sequential function can be computed by at least one such a.]

Example : left addition as a sequential algorithm in state form

$$\begin{aligned} add_L \;=\; & \{((\bot,\bot)?', valof\ ?.1)\} \cup \\ & \{((m,\bot)?', valof\ ?.2) \mid m \in \mathbf{N}\} \cup \\ & \{((m,n)?', output\ m+n) \mid m,n \in \mathbf{N}\} \end{aligned}$$

But we would like to say that add_L, at $(\bot, n) = \{(?.2, n)\}$, still wants to call $?.1$. Similarly, for

$constant_0 = request\ ?'\ output\ 0 = \{(\bot?', output\ 0\})$ (from $\mathbf{N}_\bot$ to $\mathbf{N}_\bot$)

we woud like to say that $constant_0$, at $\{(?, m)\}$, still wants to output 0.

This leads to a more abstract view of sequential algorithms that is suitable for a crisp "mathematical" definition of composition of sequential algorithms.

Equivalent definitions of sequential algorithms

From the pioneering days, we have 3 equivalent definitions of **sequential algorithms** :

1. as states of $\mathbf{M} \to \mathbf{M}'$
2. (coming next) as abstract algorithms (or as pairs of a function and a computation strategy for it)
3. (cf. preview) as programs (cf. language CDS)

[For the record, other equivalent definitions :

4. as observably sequential functions (idea due to Cartwright and Felleisen : use errors to detect how the algorithm explores the data)
5. as bistable and extensionally monotonic functions (Laird)
6. (in the affine case) as a symmetric pair (f, g), where f is function from input strategies to output strategies and g is a function from output counter-strategies to input counter-strategies (Curien 1994)]

Abstract algorithms

Let $\mathbf{M}$ and $\mathbf{M}'$ be cds's. An abstract algorithm from $\mathbf{M}$ to $\mathbf{M}'$ is a partial function $f : D(\mathbf{M}) \times C_{\mathbf{M}'} \rightharpoonup V_{\mathbf{M}\to\mathbf{M}'}$ satisfying the following axioms :

(A_1) If $f(xc') = u$, then $\begin{cases} \text{if } u = valof\ c \text{ then } c \in A(x) \\ \text{if } u = output\ v' \text{ then } (c', v') \in E_{\mathbf{M}'} \end{cases}$

(A_2) If $f(xc') = u$, $x \le y$ and $(yc', u) \in E_{\mathbf{M}\to\mathbf{M}'}$, then $f(yc') = u$.

(A_3) Let $f{\bullet}y = \{(c', v') \mid f(yc') = output\ v'\}$. Then :

$$f(yc')\downarrow \Rightarrow (c' \in E(f{\bullet}y) \text{ and } (z \le y \text{ and } c' \in E(f{\bullet}z) \Rightarrow f(zc')\downarrow)).$$

Abstract algorithms are ordered by the usual order of extension on partial functions.

Correspondence Sequential algorithms as states
$\leftrightarrow$
abstract algorithms

Easy : by extension / shrinking of the domain of definition.

Let $\mathbf{M}$ and $\mathbf{M}'$ be cds's. The following define inverse order-isomorphisms :

Let a be a state of $\mathbf{M} \to \mathbf{M}'$. Let $a^+ : C_{\mathbf{M}\to\mathbf{M}'} \rightharpoonup V_{\mathbf{M}\to\mathbf{M}'}$ be given by :

$$a^+(xc') = u \text{ iff } \exists\, y \leq x \ \ (yc', u) \in a \text{ and } (xc', u) \in E_{\mathbf{M}\to\mathbf{M}'}.$$

Let f be an abstract algorithm from $\mathbf{M}$ to $\mathbf{M}'$. We set :

$$f^- = \{(xc', u) \mid f(xc') = u \text{ and } (y < x \Rightarrow f(yc') \neq u)\}.$$

Sequential algorithms as programs

A sequential algorithm as program is a forest F whose trees T are declared by the following syntax

$$\begin{array}{l} T ::= \mathit{request}\ c'\ (\mathit{from}\ x)\ U \\ U ::= \mathit{valof}\ c\ \mathit{is}\ [\ldots v \mapsto U_v \ldots] \mid \mathit{output}\ v' \end{array}$$

typed as follows :

$$\frac{c \in A(x) \quad \ldots (x \cup \{(c,v)\}, c') \vdash U_v \ldots}{(x, c') \vdash \mathit{valof}\ c\ \mathit{is}\ [\ldots v \mapsto U_v \ldots]} \qquad \frac{(c', v') \in E_{\mathrm{M}'}}{(x, c') \vdash \mathit{output}\ v'}$$

We require that each tree $\mathit{request}\ c'\ (\mathit{from}\ x)\ U \in F$ is such that $(x, c') \vdash U$, that there is at most one tree beginning with $\mathit{request}\ c'\ (\mathit{from}\ x)$ in F and that

— if $\vdash c'$ then $x = \emptyset$;

— otherwise there exists an enabling $(c'_1, v'_1), \ldots, (c'_n, v'_n)$ of c' and programs $\mathit{request} c'_i (\mathit{from} y_i$ F with for each one a leaf $(x_i, c'_i) \vdash \mathit{output}\ v'_i$ and $x = \bigcup x_i$.

Sequential algorithms as programs : the filiform case

When the output cds is filiform, we can directly graft a tree starting with $request\, d'$, where $(c', v') \vdash d'$ at the appropriate leaf $output\, v'$ of the appropriate tree starting with $request\, c'$, and doing this systematically results in a single tree.

An example of a sequential algorithm as forest

From pairs of booleans to $\mathbf{EX}$, which has cells c_0, c_1, c_2, values $0, 1$, and enablings $\vdash c_0$, $\vdash c_1$, $(c_0, 1) \vdash c_2$ and $(c_0, 0), (c_1, 0) \vdash c_2$) :

$$request\ c_0\ (from\ \{\})valof\ ?.1\ is\ \begin{cases} T \mapsto output\ 1 \\ F \mapsto valof\ ?.2\ is\ \{\ F \mapsto output\ 0 \end{cases}$$

$$request\ c_1\ (from\ \{\})\ valof\ ?.2\ is\ \begin{cases} T \mapsto\ output\ 0 \\ F \mapsto\ output\ 0 \end{cases}$$

$$request\ c_2 : (from\ \{(?.1, T)\})\ valof\ ?.2\ is\ \begin{cases} T \mapsto\ output\ 0 \\ F \mapsto\ output\ 0 \end{cases}$$

$$request\ c_2 : (from\ \{(?.1, F), (?.2, F)\})\ output\ 0$$

From state form to program form

This is consequence of (1) in the following

Lemma. The following properties hold ($a \in D(\mathbf{M} \to \mathbf{M}')$, $\mathbf{M}'$ stable) :

(1) If $(xc', u), (zc', w) \in a$ and $x \uparrow z$, then $x \leq z$ or $z \leq x$; if $x < z$, there exists a chain

$$x = y_0 \prec_{c_0} y_1 \cdots y_{n-1} \prec_{c_{n-1}} y_n = z$$

such that $\forall\, i < n \;\; (y_i c', valof\ c_i) \in a$. If u and w are of type 'output', then $x = z$.

(2) The set $a{\bullet}x$ is a state of $\mathbf{M}'$, for all $x \in D(\mathbf{M})$.

(3) For all $xc' \in F(a)$, xc' has only one enabling in a ; hence $\mathbf{M} \to \mathbf{M}'$ is stable.

From program form to state form

This is the easy direction (forgetful).

Formally, we can describe the conversion by following the typing rules.

If U appears as a subtree in the forest, with type $(x, c') \vdash U$, then (xc', u) is an event of the state associated to the forest, where $U = u\ \ldots$.

Where are we ?

We have defined :

• our mathematical structures : concrete data structures

• our morphisms sequential algorithms (presented under three different, equivalent disguises)

We have done little : we need to say how we compose them to make a category ! Concentrate !

Composing sequential algorithms

The format of states is not appropriate for defining composition.

- In my PhD work (1979), I described a (function-like) composition using the presentation as abstract algorithms (next slide).

- I'll present also the composition of sequential algorithms as programs in the form of an abstract machine (inspired by the operational semantics for CDS which I had designed in 1981).

Composing abstract algorithms

Let M, M' and M'' be cds's, and let f and f' be two abstract algorithms from M to M' and from M' to M'', respectively. The function g, defined as follows, is an abstract algorithm from M to M'' :

$$g(xc'') = \begin{cases} \mathit{output}\ v'' & \text{if } f'((f \bullet x)c'') = \mathit{output}\ v'' \\[2ex] \mathit{valof}\ c & \text{if } \begin{cases} f'((f \bullet x)c'') = \mathit{valof}\ c' \text{ and} \\ f(xc') = \mathit{valof}\ c\ . \end{cases} \end{cases}$$

Composing sequential algorithms as programs : preparations

For simplicity, we restrict ourselves to filiform cds's.

Let F and F' be sequential algorithms as programs (and hence in tree form by the filiform assumption) from $\mathbf{M}$ to $\mathbf{M}'$ and from $\mathbf{M}'$ to $\mathbf{M}''$.

The abstract machine builds any branch of the composition $F' \circ F$, by
- exploring a branch of F'
- and interactively interrogating F upon need, through its abstract algorithm version (for which a small abstract machine on the side can be used – see slide 30 for details).

Machine states are triples

$$(q'', q', y) \quad \text{where} \begin{cases} q'' \text{ is the branch of } F' \circ F \text{ being constructed} \\ q' \text{ is the branch induced in } F' \\ y \text{ is the knowledge about the input in } \mathbf{M} \\ \quad \text{acquired as computation proceeds} \end{cases}$$

Abstract machine for composition (filiform case)

$$\frac{q' \; valof \; c' \in F' \quad F^{+}(y, c') = valof \; c \quad (c, v) \in E_{\mathbf{M}}}{(q'', q', y) \stackrel{valof \; c}{\longrightarrow} (q'' \; valof \; c \; is \; v, q', y \cup \{(c, v)\})}$$

$$\frac{q' \; valof \; c' \in F' \qquad F^{+}(y, c') = output \; v'}{(q'', q', y) \longrightarrow (q'', q' \; valof \; c' \; is \; v', y)}$$

$$\frac{q' \; output \; v'' \in F' \quad [d'' \in A(q'' \; output \; v'')]}{(q'', q', y) \stackrel{output \; v''}{\longrightarrow} [(q'' \; output \; v'' \; request \; d'', q' \; output \; v'' \; request \; d'', y)]}$$

- The notation A (accessible), E, F, is easily tailored to be applied to branches.
- In the last rule, $[\ldots]$ means optional : the machine could stop right after outputing v” if there is no more accessible cell d” for which to issue a further request.

The general non filiform case : some preparations

We need to do some book-keeping in order to forward new requests $\mathit{request}\ d''$ to the appropriate tree of the forest F', updating appropriately our knowledge of the input in $\mathbf{M}$.

Machine states are of two forms : (σ'', σ'), and $(q'', \sigma'', q', \sigma', y)$, ($\sigma''$ (resp. σ') is a partial function recording a state of $\mathbf{M}'$ (resp. $\mathbf{M}$) associated to a path of F'' (resp. F').

If q' is an odd-length path of F', then $\mathit{val}\ (q')$ is defined as follows

$$\mathit{val}\ (\mathit{request}\ c''\ (\mathit{from}\ x')) = x' \qquad \mathit{val}\ (q'\ \mathit{valof}\ c'\ \mathit{is}\ v') = \mathit{val}\ (q') \cup \{(c', v')\}$$

The following is an algorithmic analogue of a^+. We set $F^+(x, c') = u$ if the following device outputs u :

$$\frac{\mathit{request}\ c'\ (\mathit{from}\ y)\ U \in F \quad y \leq x}{\longrightarrow U} \qquad \frac{U = \mathit{output}\ v'}{U \stackrel{\mathit{output}\ v'}{\longrightarrow}}$$

$$\frac{U = \mathit{valof}\ c\ \mathit{is}\ [\ldots v \mapsto U_v \ldots] \quad (c, v) \in x}{U \longrightarrow U_v} \qquad \frac{U = \mathit{valof}\ c\ \mathit{is}\ [\ldots v \mapsto U_v \ldots] \quad c \in A(x)}{U \stackrel{\mathit{valof}\ c}{\longrightarrow}}$$

Abstract machine (for general stable cds's)

$$\frac{\begin{array}{l}(c''_1, v''_1), \ldots (c''_n, v''_n) \vdash c'' \\ \ldots (\mathit{request}\ c''_i\ (\mathit{from}\ y_i)\ \ldots\ \mathit{output}\ v''_i, z'_i) \in \sigma'' \quad \text{(with } (x_i, c'') \vdash \mathit{output}\ v''_i) \ldots \\ \ldots (\mathit{request}\ c''_i\ (\mathit{from}\ y'_i)\ \ldots\ \mathit{output}\ v''_i, y_i) \in \sigma' \quad \text{(with } y'_i \leq \bigcup z'_j,\ (x'_i, c'') \vdash \mathit{output}\ v''_i) \ldots\end{array}}{(\sigma'', \sigma') \longrightarrow (\mathit{request}\ c''\ (\mathit{from}\ \bigcup x_i), \sigma'', \mathit{request}\ c''\ (\mathit{from}\ \bigcup x'_i), \sigma', \bigcup y_i)}$$

$$\frac{q'\ \mathit{output}\ v'' \in F'}{(q'', \sigma'', q', \sigma', y) \stackrel{q''\ \mathit{output}\ v''}{\longrightarrow} (\sigma'' \cup \{(q''\ \mathit{output}\ v'', \mathit{val}\ (q')\}, \sigma' \cup \{(q'\mathit{output}\ v'', y)\})}$$

$$\frac{q'\ \mathit{valof}\ c' \in F' \quad F^+(y, c') = \mathit{valof}\ c}{(q'', \sigma'', q', \sigma', y) \stackrel{q''\mathit{valof}\ c}{\longrightarrow} (q''\ \mathit{valof}\ c\ \mathit{is}\ v, \sigma'', q', \sigma', y \cup \{(c, v)\})}$$

$$\frac{q'\ \mathit{valof}\ c' \in F' \quad F^+(y, c') = \mathit{output}\ v'}{(q'', \sigma'', q', \sigma', y) \longrightarrow (q'', \sigma'', q'\ \mathit{valof}\ c'\ \mathit{is}\ v', \sigma', y)}$$

Primitive recursive program schemes (p.r.s.)

Primitive recursive program schemes are defined as formal terms generated as follows :

$(i\ )$ $\lambda\vec{x}.0$ is a p.r.s. of arity n (where n is the length of $\vec{x}$) ;

(ii) S is a p.r.s. of arity 1 ;

(iii) π_i^n is a p.r.s. of arity n (for all i, n s.t. $1 \leq i \leq n$) ;

(iv) if f is a p.r.s. of arity n and if $g_1, \ldots, g_n$ are p.r.s.'s of arity m then $h = f \circ \langle \vec{g} \rangle$ is a p.r.s. of arity m ;

(v) if g, h are p.r.s.'s of arities $n, n+2$, respectively, then $rec(g, h)$ is a p.r.s. of arity $n+1$.

Function associated with a p.r.s.

Every p.r.s. f of arity m defines a function $[f]$ from $\mathbf{N}^m$ to $\mathbf{N}$.

All cases but rec are pretty obvious (constant $\mathsf{0}$, successor, projection, tupling and composition). The meaning of $rec(g, h)$ is given as follows (primitive recursion !) :

$$\begin{aligned} rec(g, h)(\mathsf{0}, \vec{y}) &= g(\vec{y}) \\ rec(g, h)(Sx, \vec{y}) &= h(x, rec(g, h)(x, \vec{y}), \vec{y}) \end{aligned}$$

Sequential algorithm associated with a p.r.s.

Proposition. Every p.r.s. f of arity m gives rise to a sequential algorithm $[\![f]\!]$ from $(\mathbf{N}_L)^m$ to $\mathbf{N}_L$, in such a way that we always have

$$[\![f]\!]\bullet(S^{n_1}(0),\ldots,S^{n_m}(0)) = [f](n_1,\ldots,n_m)$$

As before, we label the output (resp. i-th input) cells as c'_n (resp. $c_n.i$).

We define $[\![f]\!]$ by induction. For the case (iv), we use composition of sequential algorithms, and tupling (easy, omitted). We detail all other cases in the next two slides :

- We define $[\![\lambda\vec{x}.0]\!]$, $[\![S]\!]$ and $[\![\pi_i]\!]$ as programs.

- We give the definition of $[\![rec(f,g)]\!]$, using abstract algorithms. [Extending the abstract machine to cover primitive recursion is work in progress.]

The s.a.'s for constant 0, successor, and projections

$$[\![\lambda\vec{x}.0]\!] = \mathit{request}\ c'_0\ \mathit{output}\ 0$$

$$[\![S]\!] \;=\; \mathit{request}\ c'_0\ \mathit{output}\ S\ \mathit{request}\ c'_1\ \mathit{valof}\ c_0\ \mathit{is} \begin{cases} 0 \mapsto \mathit{output}\ 0 \\ S \mapsto \mathit{output}\ S\ \mathit{request}\ c'_2\ \mathit{valof}\ c_1\ \mathit{is} \begin{cases} 0 \mapsto \mathit{output}\ 0 \\ \ldots \end{cases} \end{cases}$$

$$[\![\pi_i]\!] = \mathit{request}\ c'_0\ \mathit{valof}\ c_0.i\ \mathit{is} \begin{cases} 0 \mapsto \mathit{output}\ 0 \\ S \mapsto \mathit{output}\ S\ \mathit{request}\ c'_1\ \mathit{valof}\ c_1.i\ \ldots \end{cases}$$

Preparation for the primitive recursion :

- Since in a finite state x of $D(\mathrm{N}_L)$ at most one cell is enabled, we can dispense with the c' component in $[\![rec(f,g)]\!](xc')$.
- We shall write $f(x,\vec{y})$ for $[\![f]\!]\bullet(x,\vec{y})$.

Primitive recursion as a sequential algorithm ($f = rec(g, h)$)

$$\frac{[\![g]\!](\vec{y}) = w}{[\![f]\!](0, \vec{y}) = w} \qquad \frac{[\![h]\!](x, f(x, \vec{y}), \vec{y}) = output\ v'}{[\![f]\!](Sx, \vec{y}) = output\ v'}$$

$$\frac{[\![h]\!](x, f(x, \vec{y}), \vec{y}) = valof\ c_i.1}{[\![f]\!](Sx, \vec{y}) = valof\ c_{i+1}.1} \qquad \frac{[\![h]\!](x, f(x, \vec{y}), \vec{y}) = valof\ c_i.n \quad (n \geq 3)}{[\![f]\!](Sx, \vec{y}) = valof\ c_i.(n-1)}$$

$$\frac{[\![h]\!](x, f(x, \vec{y}), \vec{y}) = valof\ c_i.2 \quad [\![f]\!](x, \vec{y}) = output\ v'}{[\![f]\!](Sx, \vec{y}) = output\ v'}$$

$$\frac{[\![h]\!](x, f(x, \vec{y}), \vec{y}) = valof\ c_i.2 \quad [\![f]\!](x, \vec{y}) = valof\ c_j.1}{[\![f]\!](Sx, \vec{y}) = valof\ c_{j+1}.1}$$

$$\frac{[\![h]\!](x, f(x, \vec{y}), \vec{y}) = valof\ c_i.2 \quad [\![f]\!](x, \vec{y}) = valof\ c_j.n\ (n \geq 2)}{[\![f]\!](Sx, \vec{y}) = valof\ c_j.n}$$

An algorithm that is not primitive recursive

Consider the following (total) recursive definition for computing the minimum of two natural numbers :

$$\begin{array}{l} min(Sm, Sn) = min(m, n) + 1 \\ min(0, n) = 0 \\ min(m, 0) = 0 \end{array}$$

Interpreted as a sequential algorithm $[\![min]\!]$ from $\mathbf{N}_L \times \mathbf{N}_L$ to $\mathbf{N}_L$, this program has the following behaviour : it calls each of its two arguments an unbounded number of times. This can be made crisp by considering the infinite branch in $[\![min]\!]$ induced by the computation of

$$min\bullet(S^{\omega}(\bot), S^{\omega}(\bot))$$

This infinite branch contains an infinite numbers of calls to the first argument of min **and** an infinite number of calls to its second argument.

Colson's ultimate obstinacy theorem (next slide) says that such a behaviour cannot be otained with a p.r.s. .

Colson's ultimate obstinacy theorem

We consider $[\![f]\!]$ in program form.

Theorem. Let f be a r.p.s.. of arity n. Than all infinite branches q in $[\![f]\!]$ are such that, for $i \in \{1, \ldots n\}$ fixed, $\{n \mid valof\ c_n.i$ occurs in $q\}$ is finite, except for a unique i_0 (the obstinate sequentiality index !).

In other words, from a certain point on, any infinite branch q is an interleaving of an infinite sequence

$$valof\ c_p.i_0\ is\ v_p\ valof\ c_{p+1}.i_0\ \ldots\ valof\ c_{p+q}.i_0\ is\ v_{p+q}\ \ldots$$

and a finite or infinite sequence

$$request\ c'_r\ output\ v'_r\ \ldots\ request\ c'_{r+s}\ \ldots$$

Sketch of proof of ultimate obstinacy ($f \circ \langle \vec{g} \rangle$)

Let q'' be an infinite branch of $[\![f \circ \langle \vec{g} \rangle]\!]$. Its construction induces the construction of a branch q' of $[\![f]\!]$. There are two cases :

(1) q' is finite, and then must end with a $valof\ c'_p.i$. Then the infiniteness of q'' is fed exclusively by a (thus infinite) branch of $[\![g_i]\!]$, trying to answer the request for c'_p. Obstinacy follows from that of $[\![g_i]\!]$.

(2) q' is infinite. Then the obstinacy of $[\![f]\!]$ induces an infinite branch in $[\![g_{i_0}]\!]$, whose obstinacy in turn yields the obstinacy of q''.

Sketch of proof of ultimate obstinacy ($f = rec(g, h)$)

Let q' be an infinite branch of $[\![f]\!]$. Its construction involves the construction of branches q_h of $[\![h]\!]$ and q_g of $[\![g]\!]$ (both possibly empty). There are two cases :

(1) q_h is finite. Then the infiniteness of q' must be fed ultimately only from $[\![g]\!]$. Hence obstinacy follows from that of $[\![g]\!]$.

(2) q_h is infinite. By induction, there is an obstinate index i_0 in q_h.

- If $i_0 \neq 2$, then no recursive calls are made anymore and the ultimate obstinacy of q' follows from that of $[\![h]\!]$.
- If q_h is infinite and $i_0 = 2$, then there is an infinite cascade of recursive calls, inducing in lock-step

  $valof\ c_r.1\ is\ v_r\ valof\ c_{r+1}.1\ \ldots\ valof\ c_{r+s}.1\ is\ v_{r+s}\ \ldots$ (in q')

  from

  $valof\ c_p.2\ is\ v_p\ valof\ c_{p+1}.2\ \ldots\ valof\ c_{p+q}.2\ is\ v_{p+q}\ \ldots$ (in q_h)