Sequential algorithms
“from the source”

Pierre-Louis Curien

(CNRS – Université Paris Cité – INRIA)

9/6/2022, Sequential algorithms and friends, Marseille

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) (Scott).]

• 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 :

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

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

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

A few dates

• 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 (presented as) 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).

From *por* to *Gustave* ... and Condorcet

The path to sequentiality went through "narrowing down" steps :

• Scott continuous functions f (any single piece of the output $f(x)$ may be computed using a finite part of the input x, which one can take be minimal). But Scott pointed out the problematic parallel disjunction satisfying

$$por(\bot, T) = T \qquad por(T, \bot) = T$$

which is not definable in PCF. But adding it to the syntax, Plotkin showed that Scott model “becomes" fullly abstract.

• Gérard Berry “killed” *por* by introducing stable functions (for a fixed x and a fixed piece of $f(x)$, such a minimal input is unique, and thus minimum). But he noticed the problematic character of the function $Gustave$ satisfying

$$Gustave(T, F, \bot) = T \quad Gustave(F, \bot, T) = T \quad Gustave(\bot, T, F) = T$$

(Coquand suggested that Gustave function had to do with the Condorcet voting paradox, and indeed, Huet exhibited the connection in an hilarious talk of 2011 to be found at `http://gallium.inria.fr/~huet/PUBLIC/GGJJ.pdf`, where he also explains the name Gustave...)

Another path : strong stability

• The problem with $Gustave$ is that it is not sequential. For $\mathbf{B}_\perp = \{\perp, T, F\}$, a function $f : \mathbf{B}_\perp^n \to \mathbf{B}_\perp$ is called sequential (at $(\perp, \ldots, \perp)$) if

$$f(\perp, \ldots, \perp) = \perp \text{ and}$$
$$\exists (x_1, \ldots, x_n)\ f(x_1, \ldots, x_n) \neq \perp \text{ and}$$
$$\exists i \in \{1, \ldots n\}\ \forall (x_1, \ldots, x_n)\ (f(x_1, \ldots, x_n) \neq \perp \Rightarrow x_i \neq \perp)$$

The i in the definition is called a sequentiality index.

• The problem with going to a sequential model is that this original (style of) definition by Vuillemin does not carry to higher types. Berry's idea to overcome this was to move from functions to algorithms = pairs of a function and (successive) choices of sequentiality indices.

• In the turn of year 1990, Bucciarelli and Ehrhard came up with a reformulation of Vuillemin's definition, as strongly stable function, which does carry over to higher types. Their model remains a model of functions and turned out to be an extensional collapse of the Berry-Curien model. [While stability = preservation of compatible binary meets, strongly stability = preservation of a judicious collection of finite meets.]

A few dates : revisitations of sequential algorithms

• 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 last revisitation was the link with Laird's bistability (Curien 2009) (not covered here).
• In the mean time, the function 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 so-called Lamarche-Curien exponential is to the one of McCusker (1996) for HO games what the set-based exponential of coherence spaces is to its multiset version.]
• Related works : strong stability, Kleene's unimonotone functions, Longley's sequentially realisable functionals,. . .
• Nice application : sequential algorithms provide the right framework to give a proof of the utlimate obstinacy theorem (Colson 1989), following 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*.

We ask that "every cell can be filled" : $\forall c \in C \exists v \in V \; (c, v) \in E$.

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 a partial order 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 $(c, v) \in x$,
- accessible from x if it is not filled in x, but x contains an enabling of c,
- enabled in x if it is either filled or accessible.

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 functions). 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 $D(\mathbf{X}_\perp) = \{\emptyset\} \cup \{(?, x) \mid x \in \mathbf{X}\}$ (the usual flat cpo). We have in particular $\mathbf{N}_\perp$ (Scott natural numbers) and $Bool = \{T, F\}_\perp$.

(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 : $\mathbf{NS} = (\{c_1, c_2, c_3\}, \{1,2\}, E, \vdash)$, with $E = \{c_1, c_2, c_3\} \times \{1,2\}$, $\vdash c_1'$, $\vdash c_2'$, $(c_1', 1) \vdash c_3'$, and $(c_2', 1) \vdash c_3'$.

Another example : 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

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

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 :

$$c_0 \begin{cases} 0 \\ S\ c_1 \begin{cases} 0 \\ S\ c_2 \begin{cases} 0 \\ \ldots \end{cases} \end{cases} \end{cases} \quad \text{or} \quad \begin{cases} 0 \\ S(\bot) \begin{cases} S(0) \\ S(S(\bot)) \begin{cases} S(S(0)) \\ \ldots \end{cases} \end{cases} \end{cases}$$

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

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 :

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

The function induced by a sequential algorithm

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

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

Indeed, $a \bullet x$ is always a state, provided x is a state and $\mathbf{M}'$ is *stable*.

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

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

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

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

Example : left addition

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

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

Similarly, for

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

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)

[Other equivalent definitions (the first two already mentioned) :

4. as observably sequential functions
5. as bistable and extensionally monotonic functions
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 M and M' be cds's. An abstract algorithm from M to M' is a partial function $f : D(\mathrm{M}) \times C_{\mathrm{M}'} \rightharpoonup V_{\mathrm{M}\to\mathrm{M}'}$ satisfying the following axioms :

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

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

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

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

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

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

$$\mathit{request}\ c'_i\ (\mathit{from}\ y_i)\ U_i \in F$$

with for each one a leaf $(x_i, c'_i) \vdash \mathit{output}\ v'_i$ and $x = \bigcup x_i$.

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 $(c_0, 0), (c_1, 0) \vdash c_2$:

$$\mathit{request}\ c_0\ (\mathit{from}\ \{\})\mathit{valof}\ ?.1\ \mathit{is} \left\{ \begin{array}{l} T \mapsto \mathit{output}\ 1 \\ F \mapsto \mathit{valof}\ ?.2\ \mathit{is}\ \left\{\ F \mapsto \mathit{output}\ 0 \right. \end{array} \right.$$

$$\mathit{request}\ c_1\ (\mathit{from}\ \{\})\ \mathit{valof}\ ?.2\ \mathit{is} \left\{ \begin{array}{l} T \mapsto \mathit{output}\ 0 \\ F \mapsto \mathit{output}\ 0 \end{array} \right.$$

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

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

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.

Here is for example the left-addition in tree form :

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

Algorithms as states $\leftrightarrow$ Algorithms as programs

• State to program : 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.

• Program to state : easy (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 $U = u \ \ldots$.

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'' :

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

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).

Machine states are triples

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

Initial states are $(request\ c'', request\ c'', \emptyset)$.

Abstract machine for composition (filiform case)

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

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

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

In the last rule, $[d'']$ is a shorthand for $request \; d''$ and is optional : the machine could stop right after outputting v” if there is no more accessible cell d” for which to issue a further request.

Sequential algorithms are not functions, you said ?

• We assume that there exists a reserved value T, not belonging to V for any cds $\mathbf{M} = (C, V, E, \vdash)$.
Given a cds $\mathbf{M} = (C, V, E, \vdash)$, we call an observable state of **M** a set x of pairs (c, w), where either $(c, w) \in E$ or $w = \top$, satisfying the conditions that define a state of a cds. The set of observable states of **M** is denoted by $D^\top(\mathbf{M})$.
Notice that enablings are not allowed to contain error values, because the enabling relation is part of the structure of a cds, which we did not change. Thus, in the tree representation of an observable state, error values can occur only at the leaves.

• Every sequential algorithm a gives rise to a function $D^\top(\mathbf{M}) \to D^\top(\mathbf{M}')$ extending the one defined before.

$$\begin{array}{rcl} a{\bullet}x & = & \{(c', \mathit{output}\ v') \mid \exists\, y \leq x\ \ (yc', \mathit{output}\ v') \in a\} \cup \\ & & \{(c', \top) \mid \exists\, y \leq x\ \ (yc', \mathit{valof}\ c) \in a \text{ and } (c, \top) \in x\}\ . \end{array}$$

Sequential algorithms $\leftrightarrow$ Observably sequential functions

The above map $a \mapsto (x \mapsto a{\bullet}x)$ is actually a one-to-one correspondence with observably sequential functions :

- A monotone function $f : D^{\top}(\mathbf{M}) \to D^{\top}(\mathbf{M}')$ is called sequential at x, c' if for any $c' \in A(f(x))$ either $\forall\, y \geq x \;\; c' \notin F(f(y))$ or

$$\exists\, c \in A(x) \;\; \forall\, y > x \;\; c' \in F(f(y)) \;\Rightarrow c \in F(y)$$

- If moreover

$$(c', \top) \in f(x \cup \{(c, \top)\}$$

then we say that f is observably sequential at x, c' (note that there can then be no other sequentiality index).

- For example, we have $add_L(\bot, \top) = \bot$ and $add_R(\bot, \top) = \top$.

Primitive recursive schemes as sequential algorithms

• Primitive recursive schemes (p.r.s.) 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 $n+1$.

• Every p.r.s. f of arity m gives rise to a sequential algorithm $[\![f]\!]$ from $(\mathbf{N}_L)^m$ to $\mathbf{N}_L$ (the lazy natural numbers).

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

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

Affine sequential algorithms

• Lamarche (1992, unpublished) showed that the function space of sequential algorithms (in the filiform case) has an affine decomposition

$$\mathbf{S} \to \mathbf{S}' = (!\mathbf{S}) \multimap \mathbf{S}'$$

• An affine sequential algorithm is a sequential algorithm which (in program form) is such that along any branch, paying attention only to the successive queries

$$valof\ c_1\ is\ v_1\ \ldots\ valof\ c_i\ is\ v_i\ \ldots\ valof\ c_{i=1}\ is\ v_{i+1}\ \ldots$$

we have $(c_i, v_i] \vdash c_{i+1}$.

• A typical non-affine algorithm is left (or right) addition.

• One can reformulate Colson's ultimate obstinacy by saying that "all infinite behaviours of primitive recursive schemes are eventually affine".

On the symmetry of sequentiality

In an invited talk at MFPS (1993, New Orleans), I gave a formulation of affine algorithms from S to S' as pairs of two functions (f, g), where (using a game semantics vocabulary : states as strategies) :

f maps strategies of S to strategies of S',
g maps counter-strategies of S' to counter-strategies of S,

in such a way that

• playing strategies x against counter strategies $g(\alpha')$ provides a choice of sequentiality indexes for f,

• and the same for $f(x)$, α', g.

The Lamarche-Curien exponential

• Girard’s original exponential for coherence spaces consists in implementing multiple use of tokens by a single use of “coherent multitokens" in the form of a finite clique.

• The transposition of this in the setting of sequential algorithms consists in implementing the use of several threads of computation by a use of a single thread providing and enumeration of a finite state.

In order to make this precise, we need to move from stable filiform cds to the equivalent formalism of sequential data structures.

Sequential data structures

• A sequential data structure $\mathbf{S} = (C, V, P)$ is given by two sets C and V of *cells* and *values* and by a collection P of words p of the form :

$$c_1 v_1 \cdots c_n v_n \text{ or } c_1 v_1 \cdots c_{n-1} v_{n-1} c_n \quad (n \geq 1)$$

where $c_i \in C$ and $v_i \in V$ for all i. Thus any $p \in P$ is alternating and starts with a cell. Moreover, it is assumed that P is closed under non-empty prefixes. We call the elements of P *positions* of $\mathbf{S}$. A position ending with a value (resp. cell) is called a *response* (resp. *query*). We denote by Q and R the sets of queries and responses, respectively.

• Let $\mathbf{S} = (C, V, P)$ be an sds. We set

$$!\mathbf{S} = (Q, R, P_!)$$

where Q and R are the sets of queries and of responses of $\mathbf{S}$, respectively, and where $P_!$ consists of all prefix respecting enumerations of finite strategies of $\mathbf{S}$, i.e. $Q = q_1 r_1 \ldots q_n r_n$ is a response of $!\mathbf{S}$ if and only $\{r_1, \ldots, r_n\}$ is a strategy of $\mathbf{S}$ and, for all i, all prefixes of r_i appear in Q before r_i.

Strategies and counter-strategies

A strategy of **S** is a subset x of R that is closed under response prefixes and binary non-empty glb's :

$$r_1, r_2 \in x \ \ , \ \ r_1 \wedge r_2 \neq \epsilon \ \ \Rightarrow \ \ r_1 \wedge r_2 \in x$$

where ϵ denotes the empty word. A counter-strategy is a non-empty subset of Q that is closed under query prefixes and under binary glb's. We use $x, y, \ldots$ and $\alpha, \beta, \ldots$ to range over strategies and counter-strategies, respectively.

Sequential data structures $\leftrightarrow$ Stable filiform concrete data structures

• Let $\mathbf{S} = (C, V, P)$ be an sds, and let Q and R be the associated sets of queries and responses. We define $\mathbf{cds}(\mathbf{S}) = (Q, R, E, \vdash)$, with

$$E = \{(q, qv) \mid qv \in P\} \qquad \vdash c \text{ if } c \in C \cap P \qquad (q, qv) \vdash qvc \text{ if } qvc \in P.$$

• Let $\mathbf{M} = (C, V, E, \vdash)$ be a well-founded, stable, and filiform cds. We define $\mathbf{sds}(\mathbf{M}) = (C, V, P)$, where

$$\begin{array}{rcl} P & = & \{c_1 v_1 \cdots c_n v_n c \mid (c_1, v_1), \ldots, (c_n, v_n) \text{ is a proof of } c\} \cup \\ & & \{rcv \mid rc \in P \text{ and } (c, v) \in E\} \end{array}$$

Under the correspondence, states of $\mathbf{M}$ become strategies of $\mathbf{sds}(\mathbf{M})$ (defined in the next slide).