Sequential interactive behaviour

of

recursive program schemes

Pierre-Louis Curien

(IRIF, πr^2, CNRS – Univ. Paris 7 – INRIA)

19/12/2017 Tallinn University of Technology

Prologue : denotational and operational semantics

Want to understand the meaning of a (piece of) program M of type B with free identifier of type A.

• Denotational approach : associate some mathematical structures $[\![A]\!]$ and $[\![B]\!]$ with the types A and B, and a suitable function/morphism $[\![M]\!]$ (from $[\![A]\!]$ to $[\![B]\!]$) with M. [Typically, continuous functions between complete partial orders (cpo's).]

• Operational approach : specifiy formal rules of execution (a machine, a rewriting system,. . .) leading to observable results / experiments.

• The two approaches induce each a notion of equivalence :

$$M =_{den} N \quad \text{iff} \quad [\![M]\!] = [\![N]\!] \qquad \text{(denotational)}$$

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

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. [No information yest, or diverging computation]

• This partial order is complete : every increasing chain has a least upper bound [Useful to give meaning to programs defined by general recursive equations]

Scott continuity

The natural notion of morphism between complete partial orders is that of Scott-continuous function, i.e. a monotonic function that preserves the least upper bounds of increasing sequences.

The first appearance of such a function (that certainly influenced **Scott**) was in recursion theory. Recall that ϕ_n is the n-th partial recursive function (Kleene).

Myhill-Shepeherdson (1955) : Let f be a total recursive function that is extensional, i.e., $\phi_{f(m)} = \phi_{f(n)}$ whenever $\phi_m = \phi_n$. Then there is a unique continuous function $F : PF \to PF$ 'extending" f, i.e., such that $F(\phi_n) = \phi_{f(n)}$ for all n.

Full abstraction problem

In the mid 1970's :

— Plotkin showed that Scott continuity failed to provide a fully abstract model for PCF (a Turing complete toy functional language). The model separates programs that are operationally equivalent.

— Milner constructed a fully abstract model of PCF as a quotient of a term model.

This launched the full abstraction problem : trying to find another construction of the fully abstract model of PCF that is 'independent of the syntax".

Intensional behaviour

In the quest for a fully abstract model of PCF, Berry-Curien investigated in the late 1970's a denotational semantics where programs are interpreted as ... "programs", yet retaining key expected features :

• there are only finitely many "programs" of finite types (like $\texttt{bool} \to \texttt{bool}$) ;

• "programs" are not far from being functions : they can be defined as pairs of a function and a computation strategy.

This semantics turned out to be fully abstract, not for PCF, but for an extension of PCF with a control operator (1992, Cartwright-Curien-Felleisen).

In the 1990's, similar ideas came up under the name of game semantics, but failed to satisfy the above criteria. Yet, they turned out to offer much more flexibility to interpret various features and effects in programming.

Full abstraction problem (perspective)

As regards the original full abstraction problem for PCF, the hope was to be able to decide observational equivalence for finite types.

But Loader showed in the early 1990's that this problem is undecidable...

So any attempt to construct a fully abstract model of PCF has to be ineffective at some point. Indeed, the game models of the 1990's achieve full abstraction for PCF only through a quotient by a (semantically formulated) notion of observational equivalence.

Plan of the talk

— I introduce **Kahn-Plotkin**'s **concrete data structures** (1978), which are kits for assembling atoms to build data (the datas then form a complete partial order).

— I introduce **Berry-Curien**'s **sequential algorithms** (1979) as morphisms between concrete data structrues. They feature pairs of an ordinary function + a computation strategy for it. I show how to compose them (interaction).

— I show how to interpret primitive recursive **schemes** (as opposed to functions) as sequential algorithms, exhibiting their interactive behaviour.

— This leads to a new proof of **Colson**'s **utlimate obstinacy** theorem (1989).

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*.

In fact, we shall always assume that $n \leq 1$, i.e., that enablings are only of the form $\vdash c$ or $(d, v) \vdash c$

A sequence $c_n\ v_n \ldots c_1\ v_1\ c_0$ such that $\vdash c_n$ and $(c_i, v_i) \vdash c_{i-1}$ for all $0 < i \leq n$ is called a proof of c_0 (notation $\vdash c_n\ v_n \ldots c_1\ v_1\ c_0$).

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 complete 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.

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 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 $\mathbf{N}_\perp$ of natural numbers.

(2) Terms over a first-order signature : cells are occurrences described by words of natural numbers, values are the symbols of the signature, all events are permitted, $\vdash \epsilon$, and $(u, f) \vdash ui$ for all $1 \leq i \leq arity(f)$.

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

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

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

$$\vdash c_0$$
$$(c_i, S) \vdash c_{i+1}$$

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 :

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

Sequential algorithms (preview)

Sequential algorithms 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 $?'$) :

$$addl =$$

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

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, but are different morphisms !

Sequential algorithms between M and M' as forests

$$
\begin{aligned}
F &::= \{T_1, \ldots, T_n\} \\
T &::= \mathit{request}\ c'\ U \\
U &::= \mathit{valof}\ c\ \mathit{is}\ [\ldots v \mapsto U_v \ldots] \mid \mathit{output}\ v'\ F
\end{aligned}
$$

We accept only the typed terms. There are three judgements :

$$(x, r') \vdash F \qquad (x, r') \vdash T \qquad (x, q') \vdash U$$

and the typing rules are as follows :

$$\frac{\ldots\ (x, r') \vdash T_i\ \ldots}{(x, r') \vdash F} \qquad \frac{(x, r'c') \vdash U \quad \vdash r'c'}{(x, r') \vdash \mathit{request}\ c'\ U}$$

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

$$\frac{q' = r'c' \qquad (c', v') \in E_{\mathrm{M}'} \qquad (x, q'v') \vdash F}{(x, q') \vdash \mathit{output}\ v'\ F}$$

Branch by branch :

$$
\begin{aligned}
q^\bullet &::= r^O \; request \; c' \mid r^I \; is \; v \\
r^O &::= \epsilon \mid q^\bullet \; output \; v' \\
r^I &::= q^\bullet \; valof \; c
\end{aligned}
$$

or schematically (using Kleene star)

$$\boxed{(request \; c' \; (valof \; c \; is \; v)^* output \; v')^* \quad \text{(and odd prefixes)}}$$

There are three judgements : $(x, q') \vdash q^\bullet \quad , \quad (x, r') \vdash r^O \qquad (x, c, q') \vdash r^I$,
Typing rules :

$$\frac{(x, r') \vdash r^O \quad \vdash r'c'}{(x, r'c') \vdash r^O \; request \; c'} \qquad \frac{(x, c, q') \vdash r^I \quad (c, v) \in E_{\mathbf{M}}}{(x \cup \{(c, v)\}, q') \vdash r^I \; is \; v}$$

$$\frac{}{(\emptyset, \epsilon) \vdash \epsilon} \qquad \frac{(x, q') \vdash q^\bullet \quad q' = r'c' \quad (c', v') \in E^\circ{}_{\mathbf{M}'}}{(x, q'v') \vdash q' \; output \; v'} \qquad \frac{(x, q') \vdash q^\bullet \quad c \in A(x)}{(x, c, q') \vdash q^\bullet \; valof \; c}$$

Equivalent definitions of sequential algorithms

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

1. as states of some exponent cds $\mathbf{M} \to \mathbf{M}'$
2. as abstract algorithms (or as pairs of a function and a computation strategy for it)
3. (here) as programs

[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 a function from input strategies to output strategies and g is a function from output counter-strategies to input counter-strategies (Curien 1994)]

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 :

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

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

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

Composition of sequential algorithms : a simple example

Consider $(m, n) \in D(\mathbf{N}_\perp \times \mathbf{N}_\perp$ viewed as a morphism

$$\{request\ ?.1\ output\ m, request\ ?.2\ output\ n\}$$

from $\mathbf{1}$ (the empty cds with no cells nor values) to $\mathbf{N}_\perp \times \mathbf{N}_\perp$. then the composition of $addl : \mathbf{N}_\perp \times \mathbf{N}_\perp$ and (m, n) is obtained by exploring the forests of $addl$ and (m, n) as follows :

$$\langle request\ ?', \mathbf{1}\rangle\ valof\ ?.1\ \langle is\ m, \mathbf{3}\rangle\ valof\ ?.2\ \langle is\ n, \mathbf{5}\rangle\ output\ m + n$$

$$\begin{cases} \langle request\ ?.1, \mathbf{2}\rangle\ output\ m \\ \langle request\ ?.2, \mathbf{4}\rangle\ output\ n \end{cases}$$

where the numbers record the progression of the computation. The non-numbered moves are the successive answers of the two stategies that serve to continue the computation in the other strategy.

Another example of execution : composition + pairing

We now consider the composition of $addl$ and $\langle id, id\rangle : \mathbf{N}_\perp \to \mathbf{N}_\perp \times \mathbf{N}_\perp$ (which yields an algorithm for computing $\lambda x.x + x$). One constructs the branch

$$\langle request\ ?', \mathbf{1}\rangle\ valof\ ?\ \langle is\ n, \mathbf{3}\rangle\ output\ 2n$$

of this composition as follows :

$addl$: $\quad \langle request\ ?', \mathbf{1}\rangle\ valof\ ?.1\ \langle is\ n, \mathbf{4}\rangle\ valof\ ?.2\ \langle is\ n, \mathbf{6}\rangle\ output\ 2n$

$\langle id, id\rangle$: $\begin{cases} \langle request\ ?.1, \mathbf{2}\rangle\ valof\ ?\ \langle is\ n, \mathbf{3}\rangle\ output\ n \\ \langle request\ ?.2, \mathbf{5}\rangle\ valof\ ?\ is\ n\ output\ n \end{cases}$

The fact we can move directly from move 5 to move 6' $output\ n$ on the second branch of $\langle id, id\rangle$ displayed above follows from the fact that the branch $\langle request\ ?', \mathbf{1}\rangle\ valof\ ?\ \langle is\ n, \mathbf{3}\rangle$ under construction remembers that $?$ has been already visited and has value n.

Composing abstract algorithms

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

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

Composing sequential algorithms as programs : preparations

We store information about branches of F in a "fattened" branch of F', where each $is\ v'$ is now replaced by a pair $\langle is\ v', r^O \rangle$:

$$\begin{array}{l} \underline{q^\bullet} ::= \underline{r^O}\ request\ c' \mid \underline{r^I}\ \langle is\ v', r^O \rangle \\ \underline{r^O} ::= \epsilon \mid \underline{q^\bullet}\ output\ v' \\ \underline{r^I} ::= \underline{q^\bullet}\ valof\ c \end{array}$$

The stripping of the F information in a fattened branch $\underline{q^{O\prime}}$ is denoted by $|\underline{q^{O\prime}}|$. The following notation describes the relevant retrieval of the F information from a fat branch of F' : if $r^{I\prime} = q^{\bullet\prime}\ valof\ c'$, if $(x', c', q'') \vdash r^{I\prime}$, and if (d', w') is the enabling of c' in x', then $r^{I\prime}$ has a unique prefix ending with $valof\ d'\ \langle is\ w', r^O \rangle$, and we write we write $r^{I\prime} \hookrightarrow r^O$. In the case that c' is initial, we write $r^{I\prime} \hookrightarrow \epsilon$.

The states of the machine are either pairs of a branch of F'' and a fattened branch of F', or triples of a branch of F'', a fattened branch of F', and a branch of F. We highlight the component of the machine that has control at any step by boxing it. The initial state is $(\boxed{\epsilon}, \epsilon)$.

Abstract machine for composition

(1) $$\frac{(x, r'') \vdash r^{O''} \quad \vdash r''d''}{(\boxed{r^{O''}}, \underline{r^{0'}}) \longrightarrow (\boxed{r^{O''}\ request\ d''}, \underline{r^{O'}})}$$ Opponent interrogates F''

(2) $$\frac{q^{\bullet''} = r^{O''}\ request\ d''}{(\boxed{q^{\bullet''}}, \underline{r^{0'}}) \longrightarrow (q^{\bullet''}, \boxed{\underline{r^{0'}}\ request\ d''})}$$ F'' forwards the question to F'

(3) $$\frac{|\underline{q^{\bullet'}}|\ u' \in F'}{(q^{\bullet''}, \boxed{\underline{q^{\bullet'}}}) \longrightarrow (q^{\bullet''}, \boxed{\underline{q^{\bullet'}}\ u'})}$$ F' answers the question

(4) $$\frac{\underline{r^{O'}} = \underline{q^{\bullet'}}\ output\ v''}{(q^{\bullet''}, \boxed{\underline{r^{O'}}}) \longrightarrow (\boxed{q^{\bullet''}\ output\ v''}, \underline{r^{O'}})}$$ F' forwards the answer to F''

Abstract machine for composition (continued)

(5) $$\frac{\underline{r^{I\prime}} = \underline{q^{\bullet\prime}}\ valof\ c' \quad \underline{r^{I\prime}} \hookrightarrow r^O}{(q^{\bullet\prime\prime}, \boxed{\underline{r^{I\prime}}}) \longrightarrow [q^{\bullet\prime\prime}, \underline{r^{I\prime}}, \boxed{r^O\ request\ c'}]}$$

F' forwards the answer to the appropriate branch of F

(6) $$\frac{q^\bullet\ u \in F}{[q^{\bullet\prime\prime}, \underline{r^{I\prime}}, \boxed{q^\bullet}] \longrightarrow [q^{\bullet\prime\prime}, \underline{r^{I\prime}}, \boxed{q^\bullet\ u}]}$$

F answers the question

(7) $$\frac{r^O = q^\bullet\ output\ v'}{[q^{\bullet\prime\prime}, \underline{r^{I\prime}}, \boxed{r^O}] \longrightarrow (q^{\bullet\prime\prime}, \boxed{\underline{r^{I\prime}}\ \langle is\ v', r^O\rangle})}$$

F forwards the answer and its branch to F'

Abstract machine for composition (end)

(8) $$\frac{r^I = q^\bullet \; valof \; c \quad (x, q'') \vdash q^{\bullet\prime\prime} \quad (c, v) \in x}{[q^{\bullet\prime\prime}, \underline{r^{I\prime}}, \boxed{r^I}\,] \longrightarrow [q^{\bullet\prime\prime}, \underline{r^{I\prime}}, \boxed{r^I \; is \; v}\,]}$$

The branch of F'' already knows the value of c (call-by-need)

(9) $$\frac{r^I = q^\bullet \; valof \; c \quad (x, q'') \vdash q^{\bullet\prime\prime} \quad c \in A(x)}{[q^{\bullet\prime\prime}, \underline{r^{I\prime}}, \boxed{r^I}\,] \longrightarrow [\,\boxed{q^{\bullet\prime\prime} \; valof \; c}, \underline{r^{I\prime}}, r^I\,]}$$

F forwards the answer to F''

(10) $$\frac{r^{I\prime\prime} = q^{\bullet\prime\prime} \; valof \; c \quad (c, v) \in E_{\mathrm{M}}}{[\,\boxed{r^{I\prime\prime}}, \underline{r^{I\prime}}, r^I\,] \longrightarrow [\,\boxed{r^{I\prime\prime} \; is \; v}, \underline{r^{I\prime}}, r^I\,]}$$

Opponent interrogates F''

(11) $$\frac{q^{\bullet\prime\prime} = r^{I\prime\prime} \; is \; v}{[\,\boxed{q^{\bullet\prime\prime}}, \underline{r^{I\prime}}, r^I] \longrightarrow [q^{\bullet\prime\prime}, \underline{r^{I\prime}}, \boxed{r^I \; is \; v}\,]}$$

F'' forwards the question to F

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 primitive recursive scheme

Every primitive recursive scheme. f of arity m defines a function $[f]$ from $\mathbf{N}^m$ to $\mathbf{N}$.

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

$$rec(g,h)(0,\vec{y}) = g(\vec{y})$$
$$rec(g,h)(Sx,\vec{y}) = h(x, rec(g,h)(x,\vec{y}),\vec{y})$$

Sequential algorithm associated with a primitive recursive scheme

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. 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 $[\![rec(f,g)]\!]$ through an abstract machine (as we have done already for composition + tupling, cf. previous slides).

The sequential algorithms for constant $\mathsf{0}$, successor, and projections

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

$$[\![S]\!] \ = \ \mathit{request}\ c'_{\mathsf{0}}\ \mathit{output}\ S\ \mathit{request}\ c'_{\mathsf{1}}\ \mathit{valof}\ c_{\mathsf{0}}\ \mathit{is} \begin{cases} \mathsf{0} \mapsto \mathit{output}\ \mathsf{0} \\ S \mapsto \mathit{output}\ S\ \mathit{request}\ c'_{\mathsf{2}}\ \mathit{valof}\ c_{\mathsf{1}}\ \mathit{is} \begin{cases} \mathsf{0} \mapsto \mathit{output}\ \mathsf{0} \\ \ldots \end{cases} \end{cases}$$

$$[\![\pi_i]\!] = \mathit{request}\ c'_{\mathsf{0}}\ \mathit{valof}\ c_{\mathsf{0}}.i\ \mathit{is} \begin{cases} \mathsf{0} \mapsto \mathit{output}\ \mathsf{0} \\ S \mapsto \mathit{output}\ S\ \mathit{request}\ c'_{\mathsf{1}}\ \mathit{valof}\ c_{\mathsf{1}}.i\ \ldots \end{cases}$$

Abstract algorithm for primitive recursion

$$\frac{}{[\![f]\!](\bot, \vec{y}) = valof\ c_0.1}$$

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

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

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

Abstract machine for primitive recursion : preparations

We set $f = rec(g, h)$. We (pre)tag the different copies of $\mathbf{N}_L$ as follows :

$$g : \mathbf{N}_L.1 \times \ldots \times \mathbf{N}_L.n \to \mathbf{N}'_L$$

$$h : \mathbf{N}_L^{\diamond} \times \mathbf{N}_L^{\star} \times \mathbf{N}_L.1 \times \ldots \times \mathbf{N}_L.n \to \mathbf{N}'_L$$

$$f : \mathbf{N}_L^{\diamond} \times \mathbf{N}_L.1 \times \ldots \times \mathbf{N}_L.n \to \mathbf{N}'_L$$

We shall write simply $request$, instead of $request\ c'_i$ (since i is always uniquely determined).

Abstract machine for primitive recursion : initialisation

$$\xrightarrow{request} \ ?_{\vec{\emptyset}}()$$

$$?_{\vec{\emptyset}}() \xrightarrow{valof\ c^{\diamond}_{\mathsf{0}}} !_{\vec{\emptyset}}()$$

$$!_{\vec{\emptyset}}() \xrightarrow{is\ \mathsf{0}} ?_{\vec{\emptyset}}(\{request\ \})$$

$$!_{\vec{\emptyset}}() \xrightarrow{is\ S} ?_{\vec{\emptyset}}([S(\bot), request\ ])$$

Copycat with g

$$?_{\vec{y}}(\{q^b\}.\mathtt{S}) \longrightarrow ?_{\vec{y}}(\{q^b\ u\}.\mathtt{S}) \qquad (q^b\ u \in [\![g]\!])$$

$$?_{\vec{y}}(\{q^b\ output\ v\}) \xrightarrow{output\ v} !_{\vec{y}}(\{q^b\ output\ v\})$$

$$!_{\vec{y}}(\{q^b\ output\ v\}) \xrightarrow{request} ?_{\vec{y}}(\{q^b\ output\ v\ request\ \})$$

$$?_{\vec{y}}(\{q^b\ valof\ c_i.j\}.\mathtt{S}) \longrightarrow ?_{\vec{y}}(\{q^b\ valof\ c_i.j\ is\ v\}.\mathtt{S}) \qquad ((c_i.j, v) \in \vec{y})$$

$$?_{\vec{y}}(\{q^b\ valof\ c_i.j\}.\mathtt{S}) \xrightarrow{valof\ c_i.j} !_{\vec{y}}(\{q^b\ valof\ c_i.j\}.\mathtt{S}) \qquad (c_i.j \in A(\vec{y}))$$

$$!_{\vec{y}}(\{q^b\ valof\ c_i.j\}.\mathtt{S}) \xrightarrow{is\ v} ?_{\vec{y}\cup\{(c_i.j,v)\}}(\{q^b\ valof\ c_i.j\ is\ v\}.\mathtt{S})$$

Notation for stack update

$\mathtt{S}[\leftarrow v]$ is defined as follows :

$$([x, q^r].k.\mathtt{S})[\leftarrow v] = [x[\leftarrow v], q^r].k.\mathtt{S}[\leftarrow v]$$

with

$$\begin{cases} S^n(\bot)[\leftarrow 0] = S^n(0) \\ S^n(\bot)[\leftarrow S] = S^{n+1}(\bot) \end{cases}$$

Copycat with h

$$?_{\vec{y}}([x, q^r].\mathtt{S}) \longrightarrow ?_{\vec{y}}([x, q^r\ u].\mathtt{S}) \qquad (q^r\ u \in [\![h]\!])$$

$$?_{\vec{y}}([x, q^r\ valof\ c_i.j].\mathtt{S}) \longrightarrow ?_{\vec{y}}([x, q^r\ valof\ c_i.j\ is\ v].\mathtt{S}) \qquad ((c_i.j, v) \in \vec{y})$$

$$?_{\vec{y}}([x, q^r\ valof\ c_i.j].\mathtt{S}) \xrightarrow{valof\ c_i.j} !_{\vec{y}}([x, q^r\ valof\ c_i.j].\mathtt{S}) \qquad (c_i.j \in A(\vec{y}))$$

$$!_{\vec{y}}([x, q^r\ valof\ c_i.j].\mathtt{S}) \xrightarrow{is\ v} ?_{\vec{y}\cup\{(c_i.j,v)\}}([x, q^r\ valof\ c_i.j\ is\ v].\mathtt{S})$$

$$?_{\vec{y}}([x, q^r\ valof\ c_i^\diamond].\mathtt{S}) \longrightarrow ?_{\vec{y}}([x, q^r\ valof\ c_i^\diamond\ is\ v].\mathtt{S}) \qquad (c_i^\diamond, v) \in x)$$

$$?_{\vec{y}}([x, q^r\ valof\ c_i^\diamond].\mathtt{S}) \xrightarrow{valof\ c_{i+|S|}^\diamond} !_{\vec{y}}([x, q^r\ valof\ c_i^\diamond].\mathtt{S}) \qquad (c_i^\diamond \in A(x))$$

$$!_{\vec{y}}([x, q^r\ valof\ c_i^\diamond].\mathtt{S}) \xrightarrow{is\ v} ?_{\vec{y}}([x \cup \{(c_i^\diamond, v)\}, q^r\ valof\ c_i^\diamond is\ v].\mathtt{S}[\leftarrow v])$$

$$?_{\vec{y}}([x, q^r\ output\ v]) \xrightarrow{output\ v} !_{\vec{y}}([x, q^r\ output\ v])$$

$$!_{\vec{y}}([x, q^r\ output\ v]) \xrightarrow{request} ?_{\vec{y}}([x, q^r\ output\ v\ request\ ])$$

Recursive calls

$$?_{\vec{y}}([S(0), q^r \ valof \ c_i^\star].\mathtt{S}) \longrightarrow ?_{\vec{y}}(\{request\ \}.0.[S(0), q^r \ valof \ c_i^\star].\mathtt{S})$$

$$?_{\vec{y}}([S(S(z)), q^r \ valof \ c_i^\star].\mathtt{S}) \longrightarrow ?_{\vec{y}}([S(z), request\ ].0.[S(S(z)), q^r \ valof \ c_i^\star].\mathtt{S})$$

$$?_{\vec{y}}([S(\bot), q^r \ valof \ c_i^\star].\mathtt{S}) \overset{valof \ c^\diamond_{|S|}}{\longrightarrow} !_{\vec{y}}([S(\bot), q^r \ valof \ c_i^\star].\mathtt{S})$$

$$!_{\vec{y}}([S(\bot), q^r \ valof \ c_i^\star].\mathtt{S}) \overset{is\ 0}{\longrightarrow} ?_{\vec{y}}(\{request\ \}.0.[S(0), q^r \ valof \ c_i^\star].\mathtt{S}[\leftarrow 0])$$

$$!_{\vec{y}}([S(\bot), q^r \ valof \ c_i^\star].\mathtt{S}) \overset{is\ S}{\longrightarrow} ?_{\vec{y}}([S(\bot), request\ ].0.[S(S(\bot)), q^r \ valof \ c_i^\star].\mathtt{S}[\leftarrow S])$$

Returns to the recursive calls

$$?_{\vec{y}}([x, q_1^r\ output\ S].j.[y, q_2^R\ valof\ c_i^\star].\mathtt{S})$$
$$\longrightarrow ?_{\vec{y}}([x, q_1^r\ output\ S\ request].j+1.[y, q_2^R\ valof\ c_i^\star].\mathtt{S}) \qquad (j < i)$$

$$?_{\vec{y}}([x, q_1^r\ output\ v].i.[y, q_2^R\ valof\ c_i^\star].\mathtt{S}) \longrightarrow ?_{\vec{y}}([y, q_2^R\ valof\ c_i^\star\ is\ v].\mathtt{S})$$

$$?_{\vec{y}}(\{q^b\ output\ S\}.j.[y, q_2^R\ valof\ c_i^\star].\mathtt{S})$$
$$\longrightarrow ?_{\vec{y}}(\{q^b\ output\ S\ request\ \}.j+1.[y, q_2^R\ valof\ c_i^\star].\mathtt{S}) \qquad (j < i)$$

$$?_{\vec{y}}(\{q^b\ output\ v\}.i.[y, q_2^R\ valof\ c_i^\star].\mathtt{S}) \longrightarrow ?_{\vec{y}}([y, q_2^R\ valof\ c_i^\star\ is\ v].\mathtt{S})$$

Illustration : lazy left addition

We define $addl_L = rec(\pi_1^1, S \circ \langle \pi_2^3,) \rangle$. One can compute (coinductively) $[\![addl_L]\!] = B_0$, where

$$B_i = request\ c_i'\ valof\ c_i.1\ is \begin{cases} 0\ valof\ c_i.2\ is \begin{cases} 0\ output\ 0 \\ S\ D_i \end{cases} \\ S\ output\ S\ B_{i+1} \end{cases}$$

where

$$D_i = valof\ c_i.2\ is \begin{cases} 0\ output\ 0 \\ S\ output\ S\ D_{i+1} \end{cases}$$

We have that, say

$$[\![addl_L]\!] \text{ yields } \begin{cases} S^\omega(\bot) & \text{on} & (S^\omega(\bot), \bot) \\ S^2(0) & \text{on} & (S(0), S(0) \\ S^\omega(\bot) & \text{on} & (S(0), S^\omega(\bot) \end{cases}$$

Colson's ultimate obstinacy theorem

Colson's ultimate obstinacy theorem says that such a behaviour cannot be ottained with a primitive recursive scheme.

Theorem Let f be a primitive recursive scheme 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 (with fixed i_0)

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

and a finite or infinite sequence

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

An algorithm that is not primitive recursive

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

$$min(Sm, Sn) = min(m, n) + 1$$
$$min(0, n) = 0$$
$$min(m, 0) = 0$$

Since sequential algorithms are also a model for general recursive definitions, we can compute its interpretation :

$$B_i = request\ c_i'\ valof\ c_i.1\ is \begin{cases} 0\ output\ 0 \\ S\ valof\ c_i.2\ is \begin{cases} 0\ output\ 0 \\ S\ output\ S\ B_{i+1} \end{cases} \end{cases}$$

the interaction of which with $(S^\omega(\bot), S^\omega(\bot))$ induces the highlighted infinite branch, that contains an infinite number of calls to the first argument of min **and** an infinite number of calls to its second argument.

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''.

Proof of ultimate obstinacy (primitive recursion case)

Consider an infinite branch b of f. Here is the architecture of the proof :

1. b projected to $\mathbf{N}_L$ yields 0.
2. b projected to $\mathbf{N}_L$ yields $S^n(\bot)$ or $S^\omega(\bot)$. Then the computation visits exactly one branch b^h of h (and possibly revisits parts of it).
   (a) The branch b^h is infinite with index j.
      i. b^h contains finitely many $output\ v; request$'s.
      ii. b^h contains infinitely many $output\ v; request$'s.
   (b) The branch b^h is infinite with index $\diamond$.
   (c) The branch b^h is finite.
   (d) The branch b^h is infinite with index $\star$.
3. b projected to $\mathbf{N}_L$ yields $S^n(0)$ $(n > 0)$ (with two subcases (a) and (b)).

Case 1

The computation takes entirely place in g, i.e. the machine states are all the form

$$\{q^g\} \ ,$$

and the obstinacy of b follows immediately from the copycat execution of the machine.

Case 2 (a)

The machine reaches some state

$$[S^m(\perp), q^h].\mathrm{S}$$

such that the rest of b^h consists only of

$$output\ v\ request \text{'s and } valof\ c_i.j\ is\ v\text{'s .}$$

Case 2 (a) i

By taking q^h long enough above, we can assume that the rest of b^h consists of

$$valof\ c_i.j\ is\ S\ valof\ c_{i+1}.j\ is\ S\ \ldots\quad .$$

Then, after exhausting the finite capacities of $\vec{y}$, the machine behaves in copycat regime :

$$?_{\vec{y}}([S^m(\bot), q^h\ valof\ c_k.j].\mathtt{S})$$

$$\xrightarrow{valof\ c_k.j}\ !_{\vec{y}}([S^m(\bot), q^h\ valof\ c_k.j].\mathtt{S})$$

$$\xrightarrow{is\ v_k}\ ?_{\vec{y}\cup\{(c_k.j,v_k)\}}([S^m(\bot), q^h\ valof\ c_k.j]\ is\ v_k.\mathtt{S})$$

$$\vdots$$

which shows that b has index j .

Case 2 (a) ii

The machine has enough fuel to pop the stack entirely. [Note that nothing will be pushed on top of $[S^m(\bot), q^h]$ since the rest of b^h does not have any call to $\star$.] Thus we can assume that the stack S is empty, and that the machine behaves in copycat regime, interleaving infinitely many $output\ S\ request$'s with infinitely many $valof\ c_k.j\ is\ S$'s, which shows that b has index j .

Case 2 (b)

This case is similar to the case 2 (a), with the only difference that when b^h contains finitely many $output\, v$; $request$'s, the induced copycat is shifted by the size of the stack.

Distribution of $output\ S$ and of $valof\ c_i^\star$ in b^h : growing stacks

We say that b^h is i-OK if $valof\ c_i^\star$ occurs in b^h and if the number of occurrences of $output$'s in b^h before $valof\ c_i^\star$ is $> i$. Suppose that b^h is i-OK up to some i. Then it is easily seen by induction on i that the machine successfully reaches a state

$$[S^m(\bot), q^h\ valof\ c_i^\star\ is\ v]$$

after having passed through a state having a stack

$$[S^p(\bot), q_1^h output S].0.[S^{p+1}(\bot), r_0^h valof\, c_0^\star].1.\ldots.i.[S^{p+i+1}(\bot), r_i^h valof\, c_i^\star]$$

of depth $i+1$, where $q_1^h\ output\ S$, $r_0^h valof\ c_0^\star$,..., $r_i^h valof\ c_i^\star$ are growing prefixes of b^h.

Distribution of $output\ S$ and of $valof\ c_i^\star$ in b^h : looping

Suppose moreover that b^h is not $i+1$-OK. Then the machine reaches a state

$$[S^p(\bot), r^h\ valof\ c_i^\star][S^{p+1}(\bot), r^h\ valof\ c_i^\star]$$

without contributing a new $valof\ c_i.j$ to b, since it is the same part of b^h which is revisited, and hence any such calls have thus been already made. From there, the machine produces stacks of unbounded depth in a loop.

The key for these properties is that there is a "race" between the $output$'s and the $valof\ c_i^\star$'s of b^h : enough $output$'s must be issued ahead of the $valof\ c_i^\star$ to prevent such a loop of recursive calls.

Case 2 (c) : analysis of b^h

b^h cannot end with a $valof\ c_i.j$ or a $valof\ c_i^\diamond$, as computation would then proceed and add an $is\ v$ to b^h.

b^h contains $valof\ c_0^\star$. Suppose not : then the stack remains always of depth 1, and the final move cannot be an $output\ 0$ because this would cause b to be finite, not $output\ S$ because the machine would then extend b^h with a $request$.

b^h contains a $valof\ c_i^\star$ for which it is not i-OK. Suppose not, and let i be maximum such that $c_i^\star$ occurs in b^h, and r^h be the prefix of b^h ending with $c_i^\star$. Then the machine reaches $[S^m(\bot), q^h\ valof\ c_i^\star\ is\ v]$, and using the same reasoning as just above, we see that b^h cannot end with an $output$. Since, by the choice of i, no $c_i^\star$ occurs in the rest of b^h, we again reach a contradiction.

Case 2 (c)

Let i be such that b^h is k-OK for all $k < i$ and is not i-OK. Then by the analysis carried out in the analysis of the distribution of $output\ S$ and of $valof\ c_i^\star$ in b^h, the machine reaches stacks

$$[S(\bot), r^h\ valof\ c_i^\star] \ldots . [S^p(\bot), r^h\ valof\ c_i^\star]$$

of unbounded depth. In this case b^h ends with $c_i^\star$ (and this exhausts all the possible cases for a finite b^h), and the unboundedness of the stack results in an infinite sequence of calls to $^\diamond$ in b. Moreover, b ultimately contains only these successive calls (together with the moves $is\ S$ that follow them). Hence b has index $^\diamond$.

Case 2 (d)

The preliminary analysis above shows that b^h must be i-OK for all i, and that, for all i, the machine passes through states of unbounded depth. Hence in this case too b has index $^\diamond$.

[Note that since b^h ultimately does not contain calls to $.j$, b does not either.]

Case 3

Ultimately, all what b contains are either $output\ S\ request$ or $valof\ c.j\ is\ v$, which all must be induced by h or g. Since the execution visits at most n branches of h (each branch being guided by $S^i(0)$ for some $1 \leq i \leq n$) and one branch of g, one of these branches must be infinite.

Case 3 (a)

If there is a branch of h guided by $S^i(0)$ that is infinite and has $\star$ as index, then the branch of h guided by $S^{i-1}(0)$, or in case $i = 1$ the branch of g, must contain infinitely many $output$ and hence must be infinite. If $i > 1$, we consider in turn the index of the branch guided by $S^{i-1}(0)$, and continuing in this way we find either an infinite branch b^g of g or an infinite branch b^h_k of h, in both cases with index j, and there are machine states

$$\{q^g\}.k.[S(0), q^h_1].\mathtt{S} \qquad \text{or} \qquad [S^k(0), q^h_k].k.[S^{k+1}(0), q^h_{k+1}].\mathtt{S}$$

where the rest of b^g (resp. of q^h_k) contains only outputs or calls to $.j$ (together with the next $request$ or is moves), boh occurring infinitely many times. All the outputs are absorbed by the infinite branch q^h_1 (resp. q^h_{k+1}), making the branch b^g (resp. q^h_k) the only contributor to b, in the form of an infinite sequence of $valof\ c_i.j\ is\ S$. [Note that the depth and the form of the stack keep unchanged, since no call to $\star$ is issued.] Therefore b is obstinate with j as index.

Case 3 (b)

Otherwise let k be the largest such that the branch of h guided by $S^k(0)$ is infinite, or if all branches of h are finite let's take the branch of g which then must be infinite. In both cases the branch b^h_k or b^g that we just defined is obstinate with index j. Moreover it can't contain infinitely many outputs unless $k = n$, as there are only finitely many calls $\star$ issued by b^h_{k+1} or b^h_1. So, if $k < n$, one reaches eventually machine states of the form

$$[S^k(0), q^h_k].[S^{k+1}(0), q^h_{k+1}].\mathtt{S} \qquad \text{or} \{q^g\}.[S(0), q^h_1]$$

from which the rest of b^h_k or b^g induces infinite copycat of calls to $.j$, thus exhibiting j as index for b. If $k = n$, the machine reaches a state $[S^n(0), q^h_n]$ from which the rest of b^h_n induces a copycat exhibiting $.j$ as index of b (with possibly outputs interleaved with the calls to $.j$, as dictated by b^h_n).