A THEORY FOR NATURAL MODELISATION AND IMPLEMENTATION OF FUNCTIONS WITH VARIABLE ARITY

Patrick BELLOT
Centre Scientifique IBM-France
36, Avenue Raymond Poincaré
75116 Paris, France

Véronique JAY
LITP Paris 6, UA 248
2, place Jussieu
75251 Paris Cedex 05, France

Abstract The aim of this article is to provide a new theorical framework based on combinators for the study and implementation of applicative programming languages. This formal theory can be viewed as a Computability theory where functions are defined in a natural and usable way because Curryfication is abolished. This allows short definitions of functions and fast graph reduction machines.

Résumé: Cet article présente un nouveau cadre théorique basé sur les combinateurs pour l'étude des langages de programmation applicative et leurs mises en œuvre. Cette théorie formelle doit être perçue comme une théorie de la Calculabilité où les fonctions sont définies de manière naturelle et utilisable car la Curryfication a été abolie. Cela permet des définitions réduites pour les fonctions et des machines de réduction de graphes efficaces.

0 - Introduction

If we look at functional languages, their implementations and theories supporting them, we must remark that polyadicity is never a primitive concept. It is carried out through Curryfication or structuration. Curryfication is a mathematical method allowing the transformation of a polyadic function into a monadic function **[Curry 58]**. A possible problem is that a single application of a polyadic n-ary function is converted into n applications of unary functions. That implies intermediate terms and reduction steps. Nevertheless, it is a particulary clear and consequently safe method used in numerous functional languages as a basic principle: KRC, ML **[Cousineau 85]**, SUGAR, **[Glaser 84]**, SASL **[Turner 79]**....

The other manner to get polyadicity from monadicity is structuration: one gives a structure to the arguments of a function. Lisp systems **[Chailloux 84]** use linked lists in order to provide tree binding. ML **[ML 84]** provides cartesian product as an alternative to Curryfication. FP systems **[Backus 78]** use the sequence which has an ambiguous status: it can be viewed as an array when accessed through projections or as a linked list when accessed with head and tail functions. The main difficulty with these approaches to polyadicity lies in the fact that structures must be represented in order to be coherent. If the system talks about arguments lists (as in Lisp), they must be present, at least virtually. On the other hand, we know that arrays are inadapted to dynamic management and that lists are rather slow. Moreover, the best effective representation of arguments sequences depends on their use. That is why it seems better to ignore any structure.

Forseeable developments in hardware conceptions **[Arvind 82, Dennis 79]** let us hope for a new deal in functional languages implementations. Nevertheless, it must be taken into account that Von Neumann architectures are there for a long time. Thus, the preceding problematic is still important.

The theory TG was first presented in **[Bellot 87a]**. It allows to deal with polyadic functions in a natural way: functions are not Curryfied. In opposition with classical theories and practices, it is not assumed any structure on arguments. The task of arguments management is rejected on combinators. The theory TG is issued from Combinatory Logic (CL) **[Curry 58]** and the Graal programming language **[Bellot 86b]**. The theory has been shown to satisfy classical properties: CR-property, Completness, Consistency, definability of Partial Recursive Functions and so on.

It has been pointed out that functions with variable arity such as $\pm$:

$\pm$: X1 X2 ... Xn - X1+X2+...+Xn for every $n \geq 1$ and natural numbers X1,X2,...,Xn

are not definable in TG whereas they exist in languages such as Lisp, ML and FP where such functions are programmable thanks to the use of structures on arguments. Therefore the theory is extended (the theory TGE) with two new combinators. Classical properties remain true for TGE and it has been conjectured that TGE is a conservative extension of TG. Some unformal definitions of functions with variable arity are in **[Bellot 87a]**.

This article gives a short presentation of the theories TG and TGE. It proves that we have the following conservative extension: TG $\subset$ TGE. Then it is given a formal definition of functions with variable arity (Fva). The problem of the representation of Fva's in TGE is studied. Firstly, we use classical results of Computability theory. Then we define computable and totally computable Fva's and we prove that they are definable in TGE. The ideas of the proofs are used to give methods for determining representations of totally computable Fva's. A general abstraction algorithm is given.

The notion of Fva means a particular notion of partial recursive function which was treated under the cover of coding the arguments sequences. That is a mathematical well-founded method unfortunately not adapted to computer science purposes. In TGE, Fva's and polyadic functions in general are defined naturally: their definitions are Fva's and polyadic terms. It was stated in **[Bellot 87a]** that such definitions are not possible in classical theories without the use of an heavy machinery (an artefact) such as coding, pairing, lists...

The natural modelisation of functional programming concepts in TGE has interesting consequences. The use of powerful uncurryfied combinators provides abstraction terms (terms given by an abstraction algorithm) shorter that thus provided with classical combinators. Shorter terms are faster programs. Moreover, TGE is given with a fast graph reduction machine running on classical computers. It is that of the Graal language **[Bellot 86b]**. Its execution time is among the best known on Von Neumann computers. Therefore, TGE can be used very efficiently in Turner's like schemes of implementation **[Turner 79]**. Curryfied combinators lead to intermediate terms and reductions which are not needed since they do not appear at human level. This work follows J.W. Backus **[Backus 78]** when he remarks that combinatory terms such as (B f) in CL are needless in practice.

1 - TGE : the theory

This section describes briefly the theory and repeats fundamental theorems given in **[Bellot 87a]**. The theory is a formal system **[Mendelson 64]**.

Alphabet:	K,S,T,L,D	constants
	V0,V1,V2,....	enumerable set of variables
	➡ , =	reduction and equality symbols
	:	application symbol
	(,)	parenthesis

Terms: they are inductively defined using the auxiliary notion of sequence,

- every atom (constant or variable) is a term
- every term is a sequence
- if **a** is a term and **s** is a sequence, **as** is a sequence
- if **a** is a term and **s** is a sequence, **(a : s)** is a term
- closure rule

Remark: A sequence is the concatenation of a finite number of terms. Despite of appearance, it is not structured. As a matter of fact, third formation rule could have been written "if **a** is a term and **s** is a sequence, **sa** is a sequence". Sequence must be viewed as a syntactic notion only. For sake of readability, a sequence which is the concatenation of terms $x_1,...,x_n$ will be denoted $x_1...x_n$ and conversely $x_1...x_n$ denotes a sequence composed with some terms $x_1,...,x_n$

Notations: As usual, application is "associative to the left" so that an application $(f : a_1..a_n) : b_1..b_m$ may be written $f : a_1..a_n : b_1..b_m$. Variables are denoted using small letters x,y,z,... with possible indexes. Capital letters M,N,L,... (possibly indexed) denote terms. If M and N are terms, $M \equiv N$ denotes the syntactic equality of M and N.

Formulas: A formula may be $P \Rightarrow Q$ (P reduces to Q) or $P - Q$ (P is equal to Q) where P and Q are terms.

Axiom-schemes and inference rules:

(K) $K : X_1..X_n : Y_1..Y_m \Rightarrow X_1$

(S) $S : F\ G_1..G_m : X_1..X_n \Rightarrow F : X_1..X_n : (G_1 : X_1..X_n)\ ...\ (G_m : X_1..X_n)$

(T) $T : G_1..G_m : X_1 X_2..X_n \Rightarrow G_1 : X_1 X_2..X_n : X_2..X_n$

(L) $L : F\ G_1..G_m : X_1..X_n \Rightarrow F : X_1..X_n : (G_1 : X_1..X_n)\ X_1..X_n$

(D) $D : G_1 G_2..G_m : X \Rightarrow G_1 : X$

$D : G_1 G_2..G_m : X_1 X_2..X_n \Rightarrow G_2 : X_1 X_2..X_n$

(a) $\dfrac{X_i \Rightarrow Y_i \quad 1 \le i \le n}{F : X_1..X_n \Rightarrow F : Y_1..Y_n}$ (e) $\dfrac{M \Rightarrow N}{M = N}$

(f) $\dfrac{F \Rightarrow G}{F : X_1..X_n \Rightarrow G : X_1..X_n}$ (s) $\dfrac{M = N}{N = M}$

(t) $\dfrac{M \Rightarrow N \quad N \Rightarrow L}{M \Rightarrow L}$ (t') $\dfrac{M = N \ , \ N = L}{M = L}$

(r) $M \Rightarrow M$

Axiom-schemes are sets of axioms which all conform to a given pattern: (S), (K), (T), (L) and (D) are axiom-schemes whereas others are inference rules. Terminology is from **[Curry 58, Hindley 86]**. A deduction of a formula **F** from a set of formulas $\{F_1,..,F_n\}$ is a tree with axioms and formulas from $\{F_1,..,F_n\}$ as leafs and **F** as root. A node must be deduced from its sons using an inference rule. We may write $T_{GE}, \{F_1,..,F_n\} \vdash F$. If the set $\{F_1,..,F_n\}$ is empty, **F** is a provable formula and we write $T_{GE} \vdash F$.

Example of proof: (t) $\dfrac{(S)\ S : K\,K : x \Rightarrow K : x : (K : x) \qquad (K)\ K : x : (K : x) \Rightarrow x}{S : K\,K : x \Rightarrow x}$

thus: $T_{GE} \vdash S : K\,K : x \Rightarrow x$

Definitions:

a) The notions of subterm and occurrence are classical. It is assumed that they are known (as usual!).

b) A combinator is a term which does not have a variable as a subterm.

c) A lhs of an axiom is a redex, the corresponding rhs is its contractum.

d) A garbage is any term corresponding to one of the following patterns: $(S : F : X_1..X_n)$, $(T : G_1..G_m : X)$, $(L : F : X_1..X_n)$, $(D : G : X_1..X_n)$. Such terms will never become heads of redexes.

e) A term which contains no redex and no garbage as a subterm is a normal form (nf).

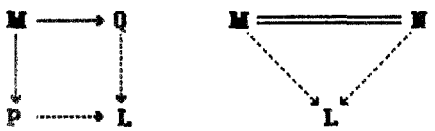
f) A term M equal to a nf N is said normalizable and N is its normal form.

The following theorems are proved in **[Bellot 87a]**.

Diamond Lemma: If $M \rightarrow P$ and $M \rightarrow Q$, there exists L such that $P \rightarrow L$ and $Q \rightarrow L$.
Proof: Adapted from Tait-Löf proof for CL and λ-calculus, 1981. It can be deduced from the results of **[Klop 80]** on Combinatory Reduction Systems.

Church-Rosser (CR): If $M = N$, there exists L such that $M \rightarrow L$ and $N \rightarrow L$.
Proof: Classical: the contraction-expansion path from M to N is reduced until it becomes a two-steps path. The proof uses heavily the Diamond lemma.

Figures:



Corollaries: The following properties are deduced from CR,

a) If $M = N$ and N is a nf., then $M \rightarrow N$

b) The normal form of a term is unique

Consistency: T_{GE} is consistent.
Proof: S and K are non equal normal forms since S cannot reduce to K.

Identity combinator: Let $I \equiv S : K\,K$, then $I : X_1..X_n \rightarrow X_1$.
Proof: $I : X_1..X_n \rightarrow S : K\,K : X_1..X_n \rightarrow K : X_1..X_n : (K\,X_1..X_n) \rightarrow X_1$

Arguments selectors: Let us define the selectors family $(P_k)_{k\geq 1}$ as follows:

$$P_1 \equiv I \qquad P_{k+1} \equiv T : (K : P_k)$$

then we have $P_k : X_1..X_n \rightarrow X_k$ if $1 \leq k \leq n$ and $P_k : X_1..X_n$ has no nf. otherwise.
Proof:

$$\begin{aligned} P_{k+1} : X_1\,X_2..X_n &\rightarrow T : (K : P_k) : X_1\,X_2..X_n \\ &\rightarrow (K : P_k : X_1\,X_2..X_n) : X_2..X_n \\ &\rightarrow P_k : X_2..X_n \\ &\rightarrow X_{n-1-k} \equiv X_{n-(k+1)} \end{aligned}$$

Substitution: If $M, N_1,..,N_k$ are terms and $x_1,..,x_k$ are variables, the parallel substitution of $N_1,..,N_k$ to $x_1,..,x_k$ in M is denoted $[N_1..N_k/x_1..x_k]M$ and is inductively defined as follows:

$[N_1..N_k/x_1..x_k]x_i \equiv N_i$

$[N_1..N_k/x_1..x_k]c \equiv c$ if c is an atom different from the variables $x_1,..,x_k$

$[N_1..N_k/x_1..x_k](F : Y_1..Y_n) \equiv [N_1..N_k/x_1..x_k]F : [N_1..N_k/x_1..x_k]Y_1 .... [N_1..N_k/x_1..x_k]Y_n$

Combinatory Completness: Let M be a term and $x_1,..,x_k$ be variables, there exists a term denoted $(\lambda x_1,..,x_k.M)$ such that none of the variables $x_1,..,x_k$ appears in it and $(\lambda x_1,..,x_k . M) : N_1..N_k \rightarrow [N_1..N_k/x_1..x_k]M$.

Proof: $(\lambda x_1,..,x_k . x_i) = P_i$
$(\lambda x_1,..,x_k . y) = K : y$ if y is an atom different from the x_i
$(\lambda x_1,..,x_k . F : M_1..M_n) = S : (\lambda x_1,..,x_k . F) (\lambda x_1,..,x_k . M_1)..(\lambda x_1,..,x_k . M_n)$

Fixed-Point operator: There exists Y such that $Y : F \rightarrow F : (Y : F)$.
Proof: $Y = \Omega:\Omega$ with $\Omega = S : (K : S) (K : I) (S : I I)$

Normalizing extensional fixed-point family: There exists a family $(Y_n)_{n\geq 1}$ such that for each $n\geq 1$, we have the following properties:

a) Y_n is normalizable
b) $Y_n : F$ is normalizable whenever F is normalizable
c) $Y_n : F : X_1..X_n \rightarrow F : (Y_n : F) : X_1..X_n$

Proof: $Y_n = \Omega_n : \Omega_n$ with $\Omega_n = \lambda a . \lambda f . \lambda x_1..x_n . (f : (a : a : f) : x_1..x_n)$

Numerals: They are defined as iterators following Church's idea for λ-calculus,

$[0] = P_2$ $\quad$ $[n+1] = [s] : [n]$ with $[s] = S : (K : S) (K : I) I$

so that we have $[n] : f\ x \rightarrow f^n x$ with $f^0 x = x$ and $f^{n+1} x = f^n(f:x)$
Proof: Recurrence on n.

Definability: Let f be a n-ary function on natural numbers, f is definable in T_{GE} if there exists a term [f] such that for all numbers $m_1,..,m_n,r$, we have:

$f(m_1,..,m_n) = r$ iff $T_{GE} \vdash [f] : [m_1]..[m_n] = [r]$
$f(m_1,..,m_n)$ undefined iff $[f] : [m_1]..[m_n]$ is not normalizable

Definability theorem: Partial Recursive Functions are definable in T_{GE}.
Proof: **[personal notes]**, facilited by polyadicity of terms.

The theory T_G: Theory T_G is the same as theory T_{GE} except that constants D and L are not present.

It was conjectured in **[Bellot 87a]** that the theory T_{GE} is a conservative extension of T_G. Let us prove this:

Lemma: If P is a T_G-term such that $T_{GE} \vdash P \rightarrow Q$, then $T_G \vdash P \rightarrow Q$.
Proof: A contraction of a T_G-term P in T_{GE} consists of the replacement of a T_{GE}-redex in P by its contractum. Because P is a T_G-term, the redex must be a T_G-redex and so is the contractum. Thus the contraction and the resulting term are in T_G. Now, by an elementary induction, we can prove the result because if $T_{GE} \vdash P \rightarrow Q$, every contraction is done in T_G.

Theorem: T_{GE} is a conservative extension of T_G.
Proof: If $T_{GE} \vdash P = Q$, there exists L such that $T_{GE} \vdash P \rightarrow L$ and $T_{GE} \vdash Q \rightarrow L$. By the lemma, $T_G \vdash P \rightarrow L$ and $T_G \vdash Q \rightarrow L$. Therefore, $T_G \vdash P = Q$.

Conclusions on TGE: TG is a theory as powerful as similar theories. It has been extended with constants L and D because it was unable to provide terms such as B:

$$B : F\ G_1..G_n : X_1..X_m \rightarrow F : (G_1 : X_1..X_m)...(G_n : X_1..X_m)$$

Thanks to Combinatory Completness, we are able to construct a $B_{n,m}$ for given n and m. Indeed, we can construct B_n which does not depend on m but it seems impossible to construct a uniform B without D and L.

Functions such as B (or $\pm$ in the introduction) are called Functions with variable arity (Fva) in next sections. Fva's are definable in λ-calculus and CL if we use a powerful artefact such as coding argument sequences into lists. We introduce the unformal notion of natural definability as definability without artefact. It can be said that natural definability is definability without any construction over the theory and its application. Partial Recursive Functions are naturally definable in all theories but they are functions with fixed arity. Fva's are Partial Recursive functions if we admit the coding of finite sequences of natural numbers by natural numbers but yet it is an artefact.

It must be pointed out that natural definability for Fva's has no sense in a Curryfied theory since functions cannot have a variable number of arguments without an artefact: end marquer for arguments, sequence coding, structuring, first argument as a counter and so on.

2 - Computability theory and Functions with variable arity

A function with variable arity (Fva) may have any enumerable domain of definition. For sake of theorical purpose, we will consider only natural numbers in this section. We assume that elementary notions of Computability theory are known, they are to be found in **[Cutland 80]** or **[Rogers 67]** for a very complete presentation. This section enters some results about computability of Fva's which will help us to represent functions in TGE.

Notations: N is the set of natural numbers. For each $k \geq 1$, N^k is the cartesian product of k times N. We define N^∞ as $\bigcup_{k \geq 1} N^k$.

Definition: A set S is effectively enumerable if there exists a bijective function $\partial : S \rightarrow N$ such that ∂ and ∂^{-1} are effectively computable.

Proposition: N^∞ is effectively enumerable.

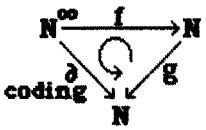
Proof: The bijection is $\partial : N^\infty \rightarrow N$

$$(a_1,..,a_k) \rightarrow 2^{a_1} + 2^{a_1+a_2+1} + ... + 2^{a_1+a_2+...+a_k+k-1} - 1$$

This application is bijective because of unicity of binary representation of natural numbers, and it is known how to compute it and its inverse. For a complete proof, see **[Cutland 80]**.

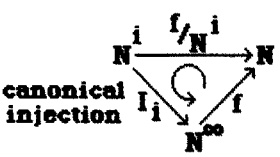
Definition: A function with variable arity (Fva) is a function $f : N^{\infty} \to N$. It can be seen as a function which can be applied to any number of arguments.

Definition: A Fva $f : N^{\infty} \to N$ is an effectively computable (e.c.) Fva if there exists an effectively computable function $g : N \to N$ such that $f \equiv g \circ \partial$ where $\equiv$ denotes extensional equality between functions on any implicit domain.



That is a normal definition for the effective computability of a Fva. We will try some equivalent definitions for effective computability of Fva's.

Definitions: If f is a function from N^{∞} into N, we note $f_{/N^k}$ the restriction of f to N^k. We note I^k the canonical injection from N^k into N^{∞} so that we have:



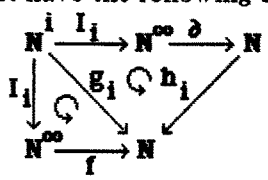
Definition: A Fva $f : N^{\infty} \to N$ is a computable Fva if there exists a family $(g_k)_{k \geq 1}$ of effectively computable functions such that $g_k : N^k \to N$ is the restriction of f to N^k, that is to say: $g_k \equiv f_{/N^k} \equiv f \circ I^k$.

Remark: A computable Fva is not an effectively computable Fva unless we can compute uniformly the g_k from k. Let us choose g_k such that $g_k(x_1,...,x_k)=1$ if the k-th P.R. function is total, 0 otherwise. Each function g_k is effectively computable since it is a constant function. Nevertheless, the Fva given by the family $(g_k)_{k \geq 1}$ is not an e.c. Fva since the total property is not decidable.

The following definition of computability for Fva's is easily shown to be equivalent to the preceding one since ∂ and ∂^{-1} are effectively computable.

Definition: A Fva $f : N^{\infty} \to N$ is a computable Fva if there exists a family $(h_k)_{k \geq 1}$ of effectively computable functions such that $h_k : N \to N$ and such that we have: $h_k \circ \partial \circ I^k \equiv f_{/N^k} \equiv f \circ I^k$.

As a matter of fact, we just have the following commuting diagram:



Definition: - Gödel numbering and Universal Program - **[Gödel 31, Cutland 80]** Every effectively computable function $f : N^k \to N$ has a number denoted (f) associated with it, it is its Gödel number. Moreover, there exists a $(k+1)$-ary function $\mathbf{A}^k$ on natural numbers such that $\mathbf{A}^k((f),n_1,..,n_k) = m$ if and only if $f(n_1,..,n_k)=m$. $\mathbf{A}^k$ is called the Universal Program and is effectively computable.

The following notion is introduced as a substitute to effective computability:

Definition: A Fva $f : N^\infty \to N$ is totally computable if there exists a family $(h_k)_{k\geq 1}$ of effectively computable functions such that $(h_k \circ \partial \circ I^k)$ is the restriction of f to N^k and such that the Gödel numbers $((h_k))_{k\geq 1}$ are given by an effectively computable function h, ie: $h(k) = (h_k)$.

Theorem: An effectively computable Fva is a totally computable Fva.
Proof: There exists an e.c. function $g : N \to N$ such that $f = g \circ \partial$. Thus restriction of f to N^k is $f_k = g \circ \partial \circ I^k$. On the other hand, we search for h_k such that $h_k \circ \partial \circ I^k$ is the same as $g \circ \partial \circ I^k$. We just have to take $h_k = g$ independently of k. Therefore, the Gödel numbers of the $(h_k)_{k\geq 1}$ are given by the constant function $h(k) = (g)$ which is effectively computable.

Theorem: A totally computable Fva is an effectively computable Fva.
Proof: The Fva C such that $C(x_1,..,x_k)=k$ is easily shown to be effectively computable. Thus, there exists an e.c. function $c : N \to N$ such that: $C = c \circ \partial$.
Now: $f(x_1,..,x_k) = h_k(\partial(x_1,..,x_k)) = \mathbf{A}^1((h_k), \partial(x_1,..,x_k)) = \mathbf{A}^1(h(k),\partial(x_1,..,x_k))$.
But k is $C(x_1,..,x_k)$,
thus: $f(x_1,..,x_k) = \mathbf{A}^1(h(C(x_1,..,x_k)),\partial(x_1,..,x_k)) = \mathbf{A}^1(h(c(\partial(x_1,..,x_k))),\partial(x_1,..,x_k))$
The function $g(x) = \mathbf{A}^1(h(c(x)),x)$ is e.c. since its subcomponents are e.c. and it is built using substitution only. Therefore, $f = g \circ \partial$ with g being e.c.

Conclusions: The aim of the introduction of total computability was partly to show that effective computability of Fva must be considering through the coding ∂. Whatever the notion that we choose (effective or total computability), we always need the coding. Another solution would have to require the Gödel numbers $((g_k))_{k\geq 1}$ to be effectively computable in the first definition of computability for Fva's. We would have: $f(x_1,..,x_k) = g_k(x_1,..,x_k) = \mathbf{A}^k((g_k),x_1,..,x_k)$
Thus: $f(x_1,..,x_k) = \mathbf{A}^k(g(k),x_1,..,x_k) = \mathbf{A}^k(g(C(x_1,..,x_k)),x_1,..,x_k)$

The problem is to extract $\mathbf{A}^k$ from $k=C(x_1,..,x_k)$. Therefore, this definition of total computability cannot be proved equivalent to effective computability. An effective computable Fva is a totally computable Fva in the present sense but the converse is not easily provable (if it is really!).

3 - The theory T_{GE} and functions with variable arity

Let $\mathbf{T}$ be the set of T_{GE}-terms, $\mathbf{T}^k$ is the cartesian product $\mathbf{T}\times\mathbf{T}\times..\times\mathbf{T}$ where $\mathbf{T}$ appears k times and $\mathbf{T}^\infty$ is the union $\bigcup_{k\geq 1}\mathbf{T}^k$.

A function with variable arity (Fva) in T_{GE} is a function $f : \mathbf{T}^{\infty} \to \mathbf{T}$.

Definition: A Fva $f : \mathbf{T}^{\infty} \to \mathbf{T}$ is representable in T_{GE} if there exists a term [f] such that $f(t_1,..,t_n) = t$ iff $[f] : t_1...t_n = t$. The term [f] is called a representative of the Fva f.

The rest of this section is devoted to the search for representatives of Fva's in the theory T_{GE}. We know that ordinary functions (with fixed arity) are definable and therefore representable in T_{GE} (cf. §1). Thus, we restrict our attention and do not consider this particular kind of Fva.

3.a) Using fixed-points

Given a Fva $f : \mathbf{T}^{\infty} \to \mathbf{T}$ to represent, it is sometimes possible to define it recursively. The recurrence must be done on the length of the arguments sequence and will be handled with the D combinator which allows a discrimination on arguments count.

For instance, let us consider the Fva: $d : \mathbf{T}^{\infty} \longrightarrow \mathbf{T}$, $(t_1,..,t_k) \mapsto [k]$

d is an argument counter. It could be expressed with a recursive scheme:

$d(t_1) = [1]$

$d(t_1,t_2,..,t_k) = 1 + d(t_2,..,t_k)$

Therefore, we must have:

$[d] : t_1 = K : [1] : t_1$

$[d] : t_1\ t_2\ ..\ t_k = [+] : [1]\ ([d] : t_2\ ..\ t_k)$

$= S : (K:[+])\ (K:[1])\ [d] : t_2\ ..\ t_k$

$= T : (K : (S : (K:[+])\ (K:[1])\ [d])) : t_1\ t_2\ ..\ t_k$

Using D, we obtain the following recursive equation:

$[d] = D : (K : [1])\ (T : (K : (S : (K:[+])\ (K:[1])\ [d])))$

which is solved using the extensional fixed-point operator of section 1 and the abstraction algorithm (this could be done manually too):

$[d] = Y : (\lambda d\ .\ D : (K : [1])\ (T : (K : (S : (K:[+])\ (K:[1])\ d))))$

Remark: It can be proved that combinator D could be replaced by [d]. We can construct [d] from D as above. The converse is quite easy to show.

3.b) Using Church's iterators

The scheme of this method is quite similar to the fixed-point method but conclusion is different. It uses basically the representation given for total computability in section 2. Let $f : \mathbf{T}^{\infty} \to \mathbf{T}$ be a Fva to represent, we try to determine the $[f_k]$ which represents f restricted to $\mathbf{T}^k$ and to express $[f_k]$ from $[f_{k-1}]$. In this sense, it is very close to recursivity in 3.a.

For instance, let us search for a such that:
$a([n_1],...,[n_k]) = [n_1+..+n_k]$ where $n_1,..,n_k$ are natural numbers

We have: $a_1([n_1]) = [n_1]$ therefore we can take: $[a_1] = I$

And: $a_k([n_1],...,[n_k]) = [+] : [n_1]\ a_{k-1}([n_2],...,[n_k])$

Thus:
$$\begin{aligned}&[a_k] : [n_1] .. [n_k]\\ &= [+] : [n_1]\ ([a_{k-1}] : [n_2] .. [n_k])\\ &= [+] : (I : [n_1] .. [n_k])\ (T : (K : [a_{k-1}]) : [n_1] .. [n_k])\\ &= S\ :\ (K : [+])\ \ I\ \ (T : (K : [a_{k-1}]))\ :\ [n_1] .. [n_k]\end{aligned}$$

Then:
$$[a_1] = I$$
$$[a_k] = A : [a_{k-1}] \quad \text{with} \quad A = (\lambda a\ .\ S\ :\ (K : [+])\ \ I\ \ (T : (K : [a])))$$

Such a recurring family of combinators is in canonical form **[Robinet 82]** and can be easily computed with Church's iterators, ie: those taking their places in the theory TGE. The result is:

$$[a_k] = [k-1] : A\ I$$

But we know how to compute [k] with the combinator [d] of section 3.a. We have:
$$\begin{aligned}[k-1] &= [-] : ([d] : [n_1]..[n_k])\ [1]\\ &= [-] : ([d] : I\ [n_1]..[n_k])\ \ [2]\\ &= S : (K : [-])\ [d]\ (K : [2])\ :\ I\ [n_1]..[n_k]\end{aligned}$$

Let us define: $[d'] = S : (K : [-])\ [d]\ (K : [2])$, we have: $[k-1] = [d'] : I\ [n_1]..[n_k]$

Thus:
$$\begin{aligned}&[a] :\ [n_1] .. [n_k]\\ &= [a_k] : [n_1] .. [n_k]\\ &= [k-1] : A\ I : [n_1] .. [n_k]\\ &= [d'] : I\ [n_1]..[n_k] : A\ I : [n_1] .. [n_k]\\ &= S\ :\ [d']\ \ (K:A)\ (K:I)\ :\ I\ [n_1]..[n_k]\ :[n_1] .. [n_k]\\ &= T : (S\ :\ [d']\ \ (K:A)\ (K:I)) : I\ [n_1]..[n_k]\\ &= L\ :\ (K : (T : (S\ :\ [d']\ \ (K:A)\ (K:I))))\ \ (K : I)\ :\ [n_1] .. [n_k]\end{aligned}$$

Therefore: $[a] = L : (K : (T : (S\ :\ (\ S : (K : [-])\ [d]\ (K : [2]))\ \ (K:A)\ (K:I))))\ \ (K : I)$

Property: If we know $[f_1]$ and if $[f_{k+1}] = F : [f_k]$, a representative of the Fva f is: $[f] = L : (K : (T : (S\ :\ (\ S :(K : [-])\ [d]\ (K : [2]))\ \ (K:F)\ (K: [f_1]))))\ \ (K : I)$
Proof: As above.

3.c) Direct intuition

The following example is that of a uniform composition operator. It emphasizes on the fact that general abstractions are not feasible without difficulty. The definition of B is: $B : F\ G_1..G_m : X_1..X_n = F : (G_1 : X_1..X_n) .. (G_m : X_1..X_n)$

B could be called a functional with variable arity. The difficulty is that we have two variable arguments sequences. We have:

$$\begin{aligned}&F : (G_1 : X_1..X_n) .. (G_m : X_1..X_n)\\ &= K : F : X_1..X_n : (G_1 : X_1..X_n) .. (G_m : X_1..X_n)\\ &= S : (K : F)\ G_1..G_m : X_1..X_n\end{aligned}$$

and: $S : (K : F)\ G_1..G_m$
$= K : S : G_1..G_m : (K : (K : F) : G_1..G_m)\ G_1..G_m$
$= L : (K : S)\ (K : (K : F)) : G_1..G_m$
$= K : L : F\ G_1..G_m : (K : (K:S) : F\ G_1..G_m)\ (S : (K:K)\ (S : (K:K)\ I) : F\ G_1..G_m) : G_1..G_m$
$= S : (K:L)\ (K:(K:S))\ (S : (K:K)\ (S : (K:K)\ I))\ : F\ G_1..G_m : G_1..G_m$
$= T : (S : (K:L)\ (K:(K:S))\ (S : (K:K)\ (S : (K:K)\ I))) : F\ G_1..G_m$

Therefore, we can choose: $B = T : (S : (K:L)\ (K:(K:S))\ (S : (K:K)\ (S : (K:K)\ I)))$

This example sets the problem of an abstraction algorithm which embodies unspecified arguments sequences and returns a result which does not depend on the length of the arguments sequences. Such an algorithm would have been useful to find B such that: $B : F\ G_1..G_m = S : (K : F)\ G_1..G_m$

3.d) General Abstraction algorithm

We call general abstraction the fact of abstracting an unspecified arguments sequence in a term which can contain part of this argument sequence. It is strongly different from the notion of abstraction in the Completness theorem. This later abstraction deals only with finite and known arguments sequences and terms such as $(\lambda x,y.\ x : x\ y)$.

In the following, we use $x_1..x_n$ as the denotation for the unspecified arguments sequence. We just know that $n \geq 1$ and that the sequence is composed with variables x with numerical indexes. The purpose of the general abstraction algorithm is to compute $(\lambda x_1..x_n.M)$ for a certain class of A-terms so that the result does not depend on n (which always remains unknown). A complete algorithm is possible. First at all, we must define the set of A-terms on which the algorithm applies. We must admit that variables x_i are no more ordinary variables since they appear in the arguments sequence. Moreover, it is intuitively clear that a A-term may contain subsequences like $x_2..x_{n-3}$ which implies that $n \geq 5$. Therefore, for each A-term we have a least bound for the value of the lengths of admissible arguments sequences. This minimal value for n is included in the definition of A-terms as an indexe to A as follows:

A_k-terms and A_k-sequences: they are inductively defined,
- every atom (constant or variable) is an A_1-term
- if i is a natural number and $i \geq 1$, then x_i is an A_i-term
- every A_k-term is an A_k-sequence
- if k_1,k_2 are natural numbers such that $k_1 \geq 1$, $x_{k_1}..x_{n-k_2}$ is an $A_{k_1+k_2}$-sequence
- if **s** is an A_p-sequence and **t** is an A_q-term, then **st** is an A_k-sequence with $k = max(p,q)$.
- if **s** is an A_p-sequence and k_1,k_2 are natural numbers such that $k_1 \geq 1$, **s** $x_{k_1}..x_{n-k_2}$ is an A_q-sequence with $q = max(p,k_1+k_2)$
- if **f** is an A_p-term and **s** is an A_q-term, then **(f : s)** is an A_k-sequence with $k = max(p,q)$
- closure rule

Examples: $(S : K\ S)$ is an A_1-term ,
$(S : x_1 : x_{2..n-3}\ K\ x_3\ x_4\ x_{1..n})$ is an A_5-term

Remark: There is no need to introduce formally subsequences like $x_{p..q}$ with p and q being natural numbers since such a sequence can be written extensionally. For instance, $x_{3..7}$ is exactly the same as $x_3\ x_4\ x_5\ x_6\ x_7$.

Property: Subterms and subsequences of an A_p-term are respectively A_q-subterms and A_q-subsequences with some $q \leq p$.

Substitution in A_k-terms and A_k-sequences: Let $X_{1..N}$ be an ordinary sequence and M be an A_k-term with $k \leq N$, we note $\pi(X_{1..N})[M]$ the result of the substitution of the sequence $X_{1..N}$ to the unspecified sequence $x_{1..n}$ in M. This substitution is inductively defined:

- $\pi(X_{1..N})[a] = a$ if a is an atom (constant or variable)
- $\pi(X_{1..N})[x_i] = X_i$ if i is a natural number
- $\pi(X_{1..N})[x_{k1..n-k2}] = X_{k1..N-k2}$ if k1 and k2 are natural numbers such that $k1 \geq 1$
- $\pi(X_{1..N})[\mathbf{st}] = \pi(X_{1..N})[\mathbf{s}]\ \pi(X_{1..N})[\mathbf{t}]$ if **s** is an A-sequence and **t** an A-term
- $\pi(X_{1..N})[\mathbf{s}\ x_{k1..n-k2}] = \pi(X_{1..N})[\mathbf{s}]\ X_{k1..N-k2}$ if k1 and k2 are natural numbers such that $k1 \geq 1$ and $k2 \geq 0$
- $\pi(X_{1..N})[\mathbf{f} : \mathbf{s}] = \pi(X_{1..N})[\mathbf{f}] : \pi(X_{1..N})[\mathbf{s}]$ if **f** is a term and **s** is a sequence

Examples:
$\pi(a\ b\ c\ d\ e\ f)[S : x_1 : x_{2..n-3}\ K\ x_3\ x_4\ x_{1..n}] = S : a : b\ c\ K\ c\ d\ a\ b\ c\ d\ e\ f$
$\pi(a\ b\ c\ d\ e)[S : x_1 : x_{2..n-3}\ K\ x_3\ x_4\ x_{1..n}] = S : a : b\ K\ c\ d\ a\ b\ c\ d\ e$

Now, we have the tools for expressing the main theorem of this section:

General Completness theorem: Let M be an A_k-term for some natural number $k \geq 1$, there exists a term denoted $\Lambda[M]$ such that for every sequence $X_{1..N}$ with $k \leq N$, we have: $\Lambda[M] : X_{1..N} \rightarrow \pi(X_{1..N})[M]$. Moreover, $\Lambda[M]$ is given by an abstraction algorithm.

Using a cumbersome syntax, this theorem establishes that we are able to uniformly abstract an unspecified arguments sequence in any A-term. For instance we are able to find B in 3.c as $B = \Lambda[S : (K : x_1)\ x_{2..n}]$. The term between brackets is an A_2-term. The proof of the theorem is done constructively: we describe an algorithm which computes $\Lambda[M]$. In order to do this, we need some technical lemmas (U,V and W) proved in the appendix. They set the existence of particular combinators used in the description of the General Abstraction algorithm.

Lemma A: There exists a combinator A such that for every term F and every sequence $X_{1..N}$, we have:

$$A : F : X_{1..N} \rightarrow F : X_{1..N} : X_{1..N}$$

Proof: we have:

$F : X_1..X_N : X_1..X_N$
$\Leftarrow (K : F : I\ X_1..X_N) : X_1..X_N : X_1..X_N$
$\Leftarrow T : (K : F) : I\ X_1..X_N : X_1..X_N$
$\Leftarrow T : (T : (K : F)) : I\ X_1..X_N$
$\Leftarrow K : (T : (T : (K : F))) : X_1..X_N : (K : I : X_1..X_N)\ X_1..X_N$
$\Leftarrow L : (K : (T : (T : (K : F)))) (K : I) : X_1..X_N$

Thus, we just have to set: $A = (\lambda f\ .\ L : (K : (T : (T : (K : f)))) (K : I))$

Lemma U: There exists a combinator U such that for all terms F and M and every sequence $X_1..X_N$, we have:

$$U : F\ M : X_1..X_N \rightarrow F : X_1..X_N\ M$$

Lemma V: There exists a family $(V_{p,q})_{p\geq 1, q\geq 0}$ of combinators such that for all natural numbers $p\geq 1$ and $q\geq 0$, every term F and every sequences $X_1..X_N$ and $Y_1..Y_M$ with $M \geq p+q$, we have:

$$V_{p,q} : F : X_1..X_N : Y_1..Y_M \rightarrow F : X_1..X_N\ Y_p..Y_{M-q}$$

Lemma W: There exists a family $(W_{p,q})_{p\geq 1, q\geq 0}$ of combinators such that for all natural numbers $p\geq 1$ and $q\geq 0$, every term F and every sequence $X_1..X_N$ of length $N \geq p+q$, we have:

$$W_{p,q} : F : X_1..X_N \rightarrow F : X_p..X_{N-q}$$

Generalized Abstraction Algorithm: In the computation of $\Lambda[M]$, we must take into account cases where M is an atom or a single x_i and cases when M is an application (f : s). In these last cases, we make an induction on the sequence s.

a) If $M = a$ and a is an atom, then $\Lambda[M] = K : a$, since we have:

$\Lambda[M] : X_1..X_N$
$= K : a : X_1..X_N$
$\rightarrow a$
$= \pi(X_1..X_N)a$
$= \pi(X_1..X_N)M$

b) If $M = x_i$ and i is a natural number, then $\Lambda[M] = P_i$, since we have:

$\Lambda[M] : X_1..X_N$
$= P_i : X_1..X_N$
$\rightarrow X_i$
$= \pi(X_1..X_N)x_i$
$= \pi(X_1..X_N)M$

c) If $M = F : x_p..x_{n-q}$ with p and q being natural numbers ($p\geq 1, q\geq 0$) and F being an A-term, then $\Lambda[M] = A : \Lambda[W_{p,q} : F]$, since we have:

$\Delta[M] : X_1..X_N$
$=$ $A : \Delta[W_{p,q} : F] : X_1..X_N$
$\Rightarrow$ $\Delta[W_{p,q} : F] : X_1..X_N : X_1..X_N$
$\Rightarrow$ $\Pi(X_1..X_N)[W_{p,q} : F] : X_1..X_N$
$\Rightarrow$ $W_{p,q} : \Pi(X_1..X_N)[F] : X_1..X_N$
$\Rightarrow$ $\Pi(X_1..X_N)[F] : X_p..X_{N-q}$
$=$ $\Pi(X_1..X_N)[F : x_p..x_{n-q}]$
$=$ $\Pi(X_1..X_N)M$

d) If $M = F : M_1..M_k$ with k being a natural number and F, $M_1,..,M_k$ being A-terms, then $\Delta[M] = S : \Delta[F]\ \Delta[M_1].. \Delta[M_k]$, since we have:

$\Delta[M] : X_1..X_N$
$=$ $S : \Delta[F]\ \Delta[M_1].. \Delta[M_k] : X_1..X_N$
$\Rightarrow$ $(\Delta[F] : X_1..X_N) : (\Delta[M_1] : X_1..X_N) ... (\Delta[M_k] : X_1..X_N)$
$\Rightarrow$ $\Pi(X_1..X_N)[F] : \Pi(X_1..X_N)[M_1] ... \Pi(X_1..X_N)[M_k]$
$=$ $\Pi(X_1..X_N)[F] : \Pi(X_1..X_N)[M_1..M_k]$
$=$ $\Pi(X_1..X_N)[F : M_1..M_k]$
$=$ $\Pi(X_1..X_N)[M]$

e) If $M = F : s\ t$ with s being an A-sequence and F,t being A-terms, then $\Delta[M] = \Delta[U : F\ t : s]$, since we have:

$\Delta[M] : X_1..X_N$
$=$ $\Delta[U : F\ t : s] : X_1..X_N$
$\Rightarrow$ $\Pi(X_1..X_N)[U : F\ t : s]$
$\Rightarrow$ $\Pi(X_1..X_N)[U] : \Pi(X_1..X_N)[F]\ \Pi(X_1..X_N)[t] : \Pi(X_1..X_N)[s]$
$\Rightarrow$ $U : \Pi(X_1..X_N)[F]\ \Pi(X_1..X_N)[t] : \Pi(X_1..X_N)[s]$
$\Rightarrow$ $\Pi(X_1..X_N)[F] : \Pi(X_1..X_N)[s]\ \Pi(X_1..X_N)[t]$
$=$ $\Pi(X_1..X_N)[F] : \Pi(X_1..X_N)[s\ t]$
$=$ $\Pi(X_1..X_N)[F : s\ t]$
$=$ $\Pi(X_1..X_N)[M]$

f) If $M = F : s\ x_p..x_{n-q}$ with $p \geq 1, q \geq 0$ being natural numbers, s being an A-sequence and F being an A-term, then $\Delta[M] = A : \Delta[V_{p,q} : F : s]$, since:

$\Delta[M] : X_1..X_N$
$=$ $A : \Delta[V_{p,q} : F : s] : X_1..X_N$
$\Rightarrow$ $\Delta[V_{p,q} : F : s] : X_1..X_N : X_1..X_N$
$\Rightarrow$ $\Pi(X_1..X_N)[V_{p,q} : F : s] : X_1..X_N$
$\Rightarrow$ $\Pi(X_1..X_N)[V_{p,q}] : \Pi(X_1..X_N)[F] : \Pi(X_1..X_N)[s] : X_1..X_N$
$\Rightarrow$ $V_{p,q} : \Pi(X_1..X_N)[F] : \Pi(X_1..X_N)[s] : X_1..X_N$
$\Rightarrow$ $\Pi(X_1..X_N)[F] : \Pi(X_1..X_N)[s]\ X_p..X_{N-q}$
$=$ $\Pi(X_1..X_N)[F] : \Pi(X_1..X_N)[s\ x_p..x_{n-q}]$
$=$ $\Pi(X_1..X_N)[F : s\ x_p..x_{n-q}]$
$=$ $\Pi(X_1..X_N)[M]$

The definition of A[M] follows inductive definition of A-terms and A-sequences, it contains the sketch of its proof which can be formally done by a recurrence on ß(M) = ß1(M)+ß2(M) where ß1(M) is the number of subsequences $x_p..x_{n-q}$ (for some p,q) in M and ß2(M) the maximal length of an Ak-sequence in M (trivial inductive definition).

Example: Let us compute B = A[S : (K : x_1) $x_2..x_n$]

B = A[S : (K : x_1) $x_2..x_n$]
= A : A[$V_{2,0}$: S : (K : x_1)]
= A : (S : A[$V_{2,0}$: S] A[K : x_1])
= A : (S : (S : A[$V_{2,0}$] A[S]) (S : A[K] A[x_1]))
= A : (S : (S : (K : $V_{2,0}$) (K : S)) (S : (K : K) P_1))

Thus: B = A : (S : (S : (K : $V_{2,0}$) (K : S)) (S : (K : K) P_1))

Optimisations: As usual, an abstraction algorithm is given in a simply provable but unrefined version. There exists almost every time a better technic. For instance, we could remark that if M does not contain any occurrence of an x_i or an x_{n-i} (even in a sequence), then we have: A[M] = K : M. This allows a simpler version for B since:

B = A : A[$V_{2,0}$: S : (K : x_1)]
= A : (S : A[$V_{2,0}$: S] A[K : x_1])
= A : (S : (K : ($V_{2,0}$: S)) (S : (K : K) P_1))

Once B is known, it could be used: if F does not contain any occurrence of an x_i or an x_{n-i}, then we have: A[F : $M_1..M_k$] = B : F A[M_1]... A[M_k]. The fields of exploration for a better algorithm are quite unlimited as it is today for ordinary combinators. As a matter of fact, combinators U, $(V_{p,q})_{p\geq 1,q\geq 0}$ and $(W_{p,q})_{p\geq 1,q\geq 0}$ are a little bit too intricate to be used efficiently in practice. Their main virtues are theorical since they made a proved opening in a new field.

4 - Conclusions

This article presents a powerful theory of combinators named TCE whose main differences with Combinatory Logic **[Curry 58]**, λ-calculus **[Barendregt 81]**, Category theory **[Cousineau 85]**, URS **[Strong 68]** and other functional theories, are that combinators are uncurryfied and that functions have naturally a variable arity **[Bellot 87a]**.

The main consequences of these choices are that ordinary abstractions give very short and efficient terms and that functions with variable arity are naturally representable, that is to say representable without any artefact over the theory such as coding, structuration, end markers and so on. The task of arguments manipulation is rejected on combinators.

It has been shown that natural representation of functions with variable arity was not possible in usual languages (Lisp, ML,..) and theories. The theory TCE

has been proved very complete since we have constructed a General Abstraction algorithm (the A-algorithm) which computes representations of functions with variable arity which are specified under a very general form (A-terms).

The theory is given with a very fast reduction machine which is that of the Graal programming language **[Bellot 86b]**. This virtual machine is one of the more efficient way of implementation on Von Neumann architectures. Graal has been designed using uncurryfied combinators for efficiency and generalized functional forms of FP systems for clarity. The result is a new, powerful and pleasant functional language where programming is very different from that of lambda-languages. Nevertheless, uncurryfied combinators can be used for compiled versions of lambda-languages (such as Lisp or ML) in a Turner's like approach. That is to say compilation of generalized lambda-expressions with the A-algorithm and execution or compilation on the Graal reduction machine. The result would be an efficient lambda-language allowing functions with variable arity without the previously used notion of list or cartesian product. For instance, we could write in some general syntax the polyadic addition function:

```
#define plus(x1..xn)  n=1 ------> x1 ; x1 + plus(x2..xn) ;;
```

This function would be compiled in:

```
plus = IF : A[eq : ([d] : x1..xn) 1] A[x1] A[+ : x1 (plus : x2..xn)]
```

with: IF = λp,f,g. A[p : x1..xn : f g : x1..xn] = λp,f,g.(A : (S : p (K : f) (K : g)))

and is more naturally expressed that the usual Lisp version below which assumes that arguments are structured into a list. The reason for this is syntactic at first sight but, more deeply, a lambda-expression cannot express arguments managing without giving them a structure or introducing special combinators:

```
(de plus lx
   (if (null lx)
       0
       (+ (car lx) (apply 'plus (cdr lx)))
) )
```

Therefore, TGE-theory, Graal language and reduction machine are a lot for a possible new point of view on functional programming style and implementation on conventional architectures.

5 - Acknowledgments

The authors are thankfull to A. Belkhir, R. Legrand, and D. Sarni for interesting discussions on the subject and for their help in the realisation of the Graal system and its theory. This work has been partly supported by the Gréco de Programmation (Bordeaux) under project PACTE.

6 - References

[Arvind 82] Arvind, K.P. Gostelow
The U-interpreter, IEEE Computer, vol. 15, pp. 42-49, Feb. 1982.

[Backus 78] J.W. Backus
Can Programming be liberated from the Von Neumann style? A functional style and its Algebra of Programs, CACM Vol. 1, n°8, pp 613-641, 1978.

[Barendregt 81] H.P. Barendregt
The Lambda-Calculus, its syntax and semantics, Studies in Logic and the Foundations of Mathematics, Vol. 103, North Holland, 1981.

[Belkhir 86] A. Belkhir
Programmation Fonctionnelle et Parallèlisme, Rapport Gréco de Programmation 1986.

[Bellot 86a] P. Bellot
Graal : a functional programming system with uncurryfied combinators and its reduction machine, ESOP 86, LNCS 213, B. Robinet ed., pp. 82-98, Saarbrücken, 1986.

[Bellot 86b] P. Bellot
Sur les sentiers du Graal, étude, conception et réalisation d'un langage de programmation sans variable, Thèse d'Etat, UPMC Paris 6, Rapport LITP 86-62, 1986.

[Bellot 87a] P. Bellot
Proposal for a natural formalization of functional programming concepts, submitted to publication at RAIRO, 1987.

[Cardelli 85] L. Cardelli
Compiling a Functional Language, 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, 1984.

[Chailloux 84] J. Chailloux, M. Devin, J-M. Hullot
Lelisp, a portable and efficient Lisp system, 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, 1984.

[Cousineau 85] G. Cousineau, P-L. Curien, M. Mauny, A. Suàrez
Combinateurs Catégoriques et Implémentation des langages fonctionnels, 13th Spring School of the LITP, LNCS. 242. G. Cousineau, P-L. Curien, B. Robinet ed., pp. 85-103, Val d'Ajol, France, 1985.

[Curry 58] H.B. Curry , R. Feys
Combinatory Logic Vol I, North Holland, 1958.

[Cutland 80] N.J. Cutland
Computability, an introduction to recursive function theory, Cambridge University Press, 1980.

[Dennis 74] J.B. Dennis
The varieties of DataFlow computers, IEEE Int. Conf. on Distributed Systems, 1979.

[Glaser 84] H. Glaser , C. Hankin , D. Till
Principles of Functional Programming, Prentice/Hall International, 1984.

[Godel 31] K. Godel
Uber formal unentscheidbare Satze des Principia Mathematica und verwandter System I, english translation in M. Davis, The undecidable, Raven N.Y. 1965.

[Hindley 86] J.R. Hindley , J.P. Seldin
Introduction to Combinators and Lambda-Calculus, London Mathematical Society, Student Texts 1, Cambridge University Press, 1986.

[Klop 80] J.W. Klop
Combinatory Reduction Systems, Ph.D. Thesis, University of Amsterdam, 1980.

[Mendelson 64] E. Mendelson
Introduction to Mathematical Logic, Van Nostrand, 2nd ed., 1979.

[ML 84]
The ML handbook, Rapport Inria, 1984.

[Robinet 82] B. Robinet
Combinateurs récurrents et itérateurs de Church, Rapport LITP 82-1, 1982.

[Rogers 67] H. Rogers
Theory of Recursive Functions and Effective Computability, Mc Graw Hill, N.Y., 1967.

[Strong 68] H.R. Strong
Algebraically generalized recursive function theory, IBM Journal for Research and Development, nov. 1968.

[Turner 79] D.A. Turner
A new implementation technique for applicative language, Software-Practice and Experience, Vol. 9, pp. 31-49, 1979.

7 - Technical Appendix

This appendix is devoted to the proof of the three lemmas U, V and W. Combinators E^1 and E^2 of the following lemmas are strongly used in the proofs.

Notation: The notation $M \Leftarrow N$ stands for $N \Rightarrow M$.

Lemma E^1: There exists a combinator E^1 such that:

$E^1 : F : X_1 : X_2..X_N \Rightarrow F : X_1\, X_2..X_N$

Proof: $E^1 : F : X_1 : X_2..X_N$
$\Rightarrow$ $F : X_1\, X_2..X_N$
$\Leftarrow$ $(K : F : X_2..X_N) : (K : X_1 : X_2..X_N)\, X_2..X_N$
$\Leftarrow$ $L : (K : F)\, (K : X_1) : X_2..X_N$

Thus, it suffices to take: $E^1 = \lambda f.(\lambda x.\ (L : (K : f)\ (K : x)))$

Lemma E^2: There exists a combinator E^2 such that:

$E^2 : F : X_1\, X_2..X_N \Rightarrow F : X_1 : X_2..X_N$

Proof: $E^2 : F : X_1\, X_2..X_N$
$\Rightarrow$ $F : X_1 : X_2..X_N$
$\Leftarrow$ $F : (I : X_1\, X_2..X_N) : X_2..X_N$
$\Leftarrow$ $B : F\ I : X_1\, X_2..X_N : X_2..X_N$
$\Leftarrow$ $T : (B : F\ I) : X_1\, X_2..X_N$

Thus, it suffices to take: $E^2 = \lambda f.(T : (B : f\ I))$

Lemma U: There exists a combinator U such that:

$U : F\ M : X_1..X_N \Rightarrow F : X_1..X_N\, M$

Proof: We distinguish cases N=1 and N>1 in order to do a recursion using combinator D as in section 3.a.

N=1: $U : F\ M : X_1$
$\Rightarrow$ $F : X_1\, M$
$\Leftarrow$ $F : (I : X_1)\ (K : M : X_1)$
$\Leftarrow$ $B : F\ I\ (K : M) : X_1$

In case N=1, it suffices to have: $U : F\ M \Rightarrow B : F\ I\ (K : M)$

N>1: $U : F\ M : X_1\, X_2..X_N$
$\Rightarrow$ $F : X_1\, X_2..X_N\, M$
$\Leftarrow$ $E^1 : F : X_1 : X_2..X_N\, M$ (-> lemma E^1)
$\Leftarrow$ $U : (E^1 : F : X_1)\ M : X_2..X_N$ (-> U used at level N-1)
$\Leftarrow$ $U : (E^1 : F : X_1)\ (K : M : X_1) : X_2..X_N$
$\Leftarrow$ $B : U\ (E^1 : F)\ (K : M) : X_1 : X_2..X_N$
$\Leftarrow$ $E^2 : (B : U\ (E^1 : F)\ (K : M)) : X_1\, X_2..X_N$ (-> lemma E^2)

In case N>1, it suffices to have: $U : F\ M \Rightarrow E^2 : (B : U\ (E^1 : F)\ (K : M))$

Therefore, using the D combinator, we obtain a proper recursion if:

$U : F\,M \Rightarrow D : (B : F\,I\,(K : M))\ (E^2 : (B : U\,(E^1 : F)\,(K : M)))$

it suffices:

$U = \lambda f,m.(D : (B : f\,I\,(K : m))\ (E^2 : (B : U\,(E^1 : f)\,(K : m))))$

Therefore:

$U = Y : (\lambda u . (\lambda f,m. (D : (B : f\,I\,(K : m))\ (E^2 : (B : u\,(E^1 : f)\,(K : m))))))$

Lemma D': There exists a combinator D' such that:

$D' : F\,G : X_1 \Rightarrow F$ and $D' : F\,G : X_1\,X_2..X_n \Rightarrow G$

Proof: $D' = S : (K : D)\,(S : (K : K)\,P_1)\,(S : (K : K)\,P_2)$

we have: $D' : F\,G \Rightarrow D : (K : F)\,(K : G)$

Lemma E: There exists a combinator E such that:

$E : F : X_1..X_N \Rightarrow F : X_1..X_{N-1}$ when $N > 1$

Proof: We distinguish cases N=2 and N>2 in order to do a recursion using combinator D as in section 3.a.

N=2: $E : F : X_1\,X_2$

$\Rightarrow F : X_1$

$\Leftarrow F : (I : X_1\,X_2)$

$\Leftarrow B : F\,I : X_1\,X_2$

Thus, in case N=2, we must have: $E : F \Rightarrow B : F\,I$

N>2: $E : F : X_1\,X_2..X_N$

$\Rightarrow F : X_1\,X_2..X_{N-1}$

$\Leftarrow E^1 : F : X_1 : X_2..X_{N-1}$ {-> lemma E^1}

$\Leftarrow E : (E^1 : F : X_1) : X_2..X_N$ {-> E used at level N-1}

$\Leftarrow B : E\,(E^1 : F) : X_1 : X_2..X_N$

$\Leftarrow E^2 : (B : E\,(E^1 : F)) : X_1\,X_2..X_N$ {-> lemma E^2}

Thus, in case N>2, we must have: $E : F \Rightarrow E^2 : (B : E\,(E^1 : F))$

The function which must be applied to $X_1..X_N$ depending on N is:

$D' : (B : F\,I)\ (E^2 : (B : E\,(E^1 : F))) : X_2..X_N$

and the result is:

$D' : (B : F\,I)\,(E^2 : (B : E\,(E^1 : F))) : X_2..X_N : X_1..X_N$

$\Leftarrow K : (D' : (B : F\,I)\,(E^2 : (B : E\,(E^1 : F)))) : X_1..X_N : X_2..X_N : X_1..X_N$

$\Leftarrow T : (K : (D' : (B : F\,I)\,(E^2 : (B : E\,(E^1 : F))))) : X_1..X_N : X_1..X_N$

$\Leftarrow A : (T : (K : (D' : (B : F\,I)\,(E^2 : (B : E\,(E^1 : F)))))) : X_1..X_N$

Therefore, we must have:

$E \Rightarrow A : (T : (K : (D' : (B : F\,I)\,(E^2 : (B : E\,(E^1 : F))))))$

Then:

$E = Y : (\lambda e. (\lambda f. A : (T : (K : (D' : (B : f\,I)\,(E^2 : (B : e\,(E^1 : f)))))))))$

Lemma Z: There exists a family $(Z_q)_{q\geq 1}$ of combinators such that:

$$Z_q : F\ X_1..X_N \rightarrow F : X_1..X_{N-q} \quad \text{if} \quad N > q$$

Proof: We distinguish cases q=N-1 and q<N-1 in order to do a recursion as in the proof of lemma E.

q=N-1: $Z_q : F\ X_1..X_N$
$\rightarrow$ $F : X_1$
$\leftarrow$ $F : (I : X_1..X_N)$
$\leftarrow$ $B : F\ I : X_1..X_N$
In case q=N-1, we must have: $Z_q \rightarrow B : F\ I$

q<N-1: $Z_q : F\ X_1..X_N$
$\rightarrow$ $F : X_1..X_{N-q}$
$\leftarrow$ $E^1 : F : X_1 : X_2..X_{N-q}$ (-> lemma E^1)
$\leftarrow$ $Z_q : (E^1 : F : X_1) : X_2..X_N$ (-> Z_q used at level N-1)
$\leftarrow$ $B : Z_q\ (E^1 : F) : X_1 : X_2..X_N$
$\leftarrow$ $E^2 : (B : Z_q\ (E^1 : F)) : X_1\ X_2..X_N$ (-> lemma E^2)
In case q<N-1, we must have: $Z_q \rightarrow E^2 : (B : Z_q\ (E^1 : F))$

We have q=N-1 if the sequence $X_2..X_{N-(q-1)}$ has length 1. Therefore, the term which must be applied to $X_1..X_N$ is:

$$D' : (B : F\ I)\ (E^2 : (B : Z_q\ (E^1 : F))) : X_2..X_{n-(q-1)}$$

And the result is:

$D' : (B : F\ I)\ (E^2 : (B : Z_q\ (E^1 : F))) : X_2..X_{n-(q-1)} : X_1\ X_2..X_N$
$\Leftarrow$ $Z_{q-1} : (\ D' : (B : F\ I)\ (E^2 : (B : Z_q\ (E^1 : F)))) : X_2..X_n : X_1\ X_2..X_N$
$\Leftarrow$ $K : (Z_{q-1} : (\ D' : (B : F\ I)\ (E^2 : (B : Z_q\ (E^1 : F))))) : X_1\ X_2..X_N : X_2..X_n : X_1\ X_2..X_N$
$\Leftarrow$ $T : (K : (Z_{q-1} : (\ D' : (B : F\ I)\ (E^2 : (B : Z_q\ (E^1 : F)))))) : X_1\ X_2..X_N : X_1\ X_2..X_N$
$\Leftarrow$ $A : (T : (K : (Z_{q-1} : (\ D' : (B : F\ I)\ (E^2 : (B : Z_q\ (E^1 : F))))))) : X_1\ X_2..X_N$

Therefore, we may have:

$$Z_q : F \rightarrow A : (T : (K : (Z_{q-1} : (\ D' : (B : F\ I)\ (E^2 : (B : Z_q\ (E^1 : F))))))))$$

And:

$$Z_q = Y : (\lambda z.\ (\lambda f.\ (A : (T : (K : (Z_{q-1} : (\ D' : (B : f\ I)\ (E^2 : (B : z\ (E^1 : f)))))))))))$$

So that:

$$Z_q = Z : Z_{q-1}$$

with: $Z = \lambda r.\ (Y : (\ \lambda z.\ (\ \lambda f.\ (A : (T : (K : (r : (\ D' : (B : f\ I)\ (E^2 : (B : z\ (E^1 : f)))))))))))$

and: $Z_1 = E$

The conclusion is easy using Church's iterators:

$$Z_q = [q-1] : Z\ E$$

Lemma W: There exists a family $(W_{p,q})_{p\geq 1, q\geq 0}$ such that:

$$W_{p,q} : F : X_1..X_N \rightarrow F : X_p..X_{N-q} \quad \text{if} \quad p+q \leq N$$

Proof: First at all, we must remark that $W_{1,q}$ is exactly Z_q of lemma Z. Now, we try to find an iteration on p when p>1 (that implies that N>1).

$W_{p,q} : F : X_1 X_2..X_N$
$\Rightarrow$ $F : X_p..X_{N-q}$
$\Leftarrow$ $W_{p-1,q} : F : X_2..X_N$
$\Leftarrow$ $K : (W_{p-1,q} : F) : X_1 X_2..X_N : X_2..X_N$
$\Leftarrow$ $T : (K : (W_{p-1,q} : F)) : X_1 X_2..X_N$

Therefore, it suffices to have:

$W_{p,q} \equiv \lambda w.\ (\lambda f.\ (T : (K : (w : f)))) : W_{p-1,q}$ if $p>1$
$W_{1,q} \equiv Z_q$

So that we choose: $W_{p,q} \equiv [p-1] : (\lambda w.\ (\lambda f.\ (T : (K : (w : f))))) Z_q$

Lemma V: There exists a family $(V_{p,q})_{p\geq 1, q\geq 0}$ such that:

$V_{p,q} : F : X_1..X_N : Y_1..Y_M \Rightarrow F : X_1..X_N Y_p..Y_{M-q}$ if $p+q \leq M$

Proof: We distinguish cases N=1 and N≥1 in order to use D.

N=1: $V_{p,q} : F : X_1 : Y_1..Y_M$
$\Rightarrow$ $F : X_1 Y_p..Y_{M-q}$
$\Leftarrow$ $E^1 : F : X_1 : Y_p..Y_{M-q}$
$\Leftarrow$ $W_{p,q} : (E^1 : F : X_1) : Y_1..Y_M$

In case N=1, it suffices to have: $V_{p,q} : F \Rightarrow \lambda x\ .\ (W_{p,q} : (E^1 : F : x))$

N>1: $V_{p,q} : F : X_1 X_2..X_N : Y_1..Y_M$
$\Rightarrow$ $F : X_1 X_2..X_N Y_p..Y_{M-q}$
$\Leftarrow$ $E^1 : F : X_1 : X_2..X_N Y_p..Y_{M-q}$ {-> lemma E^1}
$\Leftarrow$ $V_{p,q} : (E^1 : F : X_1) : X_2..X_N Y_1..Y_M$ {-> use of $V_{p,q}$ at level N-1}
$\Leftarrow$ $B : V_{p,q} (E^1 : F) : X_1 : X_2..X_N : Y_1..Y_M$
$\Leftarrow$ $E^2 : (B : V_{p,q} (E^1 : F)) : X_1 X_2..X_N : Y_1..Y_M$ {-> lemma E^2}

In case N>1, it suffices to have: $V_{p,q} : F \Rightarrow E^2 : (B : V_{p,q} (E^1 : F))$

Therefore, we must have:

$V_{p,q} : F \Rightarrow D : (\lambda x\ .\ (W_{p,q} : (E^1 : F : x)))\ (E^2 : (B : V_{p,q} (E^1 : F)))$

Thus:

$V_{p,q} \equiv \lambda f.\ (D : (\lambda x\ .\ (W_{p,q} : (E^1 : f : x)))\ (E^2 : (B : V_{p,q} (E^1 : f))))$

And:

$V_{p,q} \equiv Y : (\lambda v.\ (\lambda f.\ (D : (\lambda x\ .\ (W_{p,q} : (E^1 : f : x)))\ (E^2 : (B : v\ (E^1 : f))))))$