PATRICK BELLOT

DJAMIL SARNI

Proposal for a natural formalization of functional programming concepts

Informatique théorique et applications, tome 22, n° 3 (1988), p. 341-360.

<http://www.numdam.org/item?id=ITA_1988__22_3_341_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 22, n° 3, 1988, p. 341 à 360)

PROPOSAL FOR A NATURAL FORMALIZATION OF FUNCTIONAL PROGRAMMING CONCEPTS (*)

by Patrick BELLOT [1] and Djamil SARNI [2]

Communicated by J. E. PIN

Abstract. – *Graal is the name of a variable-free functional programming language based on precepts coming from FP systems and Combinatory Logic. This article proposes a formalization of Graal's concepts using a formal theory TG where the notions of uncurryfied combinator and polyadic application are included. This allows to give a clear semantics of the Graal language because TG becomes its elementary model. TG appears as a new theorical basis for the study of applicative programming languages. TG has been conceived as a theory of intensional functions, that is to say that TG is a new formalization of Computability well suited for Computer Science.*

Résumé. – *Graal désigne un langage de programmation fonctionnelle sans variable basé sur des principes issus des systèmes FP de J. W. Backus et de la théorie des Combinateurs de H. B. Curry. Cet article propose une formalisation des principes du langage à travers une théorie formelle TG incluant la notion de combinateur décurryfié et celle d'application polyadique. Par cette théorie, nous donnons une sémantique précise du langage Graal dont TG est un modèle élémentaire et nous fournissons une nouvelle base théorique pour l'étude des langages de programmation applicatifs. Conçue comme une théorie des fonctions en intension, TG s'inscrit comme une nouvelle formalisation de la théorie de la Calculabilité mieux adaptée à l'informatique théorique.*

0. INTRODUCTION

The language Graal (acronym of General Applicative and Algorithmic Language) is a very efficient functional language which does not compel variables [6, 8]. Despite of the absence of variables, the syntax is not esoteric and numerous programs have been written. Basic tools are generalized functional forms [1] and uncurryfied combinators. The main concept in programming is the combination of functions which allows a clear view on programs and their semantics. Moreover, the intrinsic nature of a program is quite

(*) Received March 1987, revised September 1987.
(1) Compagnie IBM-France, Centre Scientifique, 3 et 5, place Vendôme, 75021 Paris Cedex 01, France.
(2) Université des Sciences et Techniques d'Alger, Bar-El-Beida, Alger.

0296-1598 88/03 341 20/$4.00/© Gauthier-Villars

trivial in contrast with lambda-languages where a function is always a troublesome closure. Definitions of primitive functions and functionals are given using reduction rules. Consequently, Graal is implemented as a virtual graph reduction machine running on Von Neumann architectures [7]. The machine is object-oriented and its execution time is one of the fastest known by the author.

Graal is a safe language in the sense that it does not depend upon its interpretation scheme. Nevertheless, its semantics is only operational via a meta-circular interpreter. The aim of this article is to give a mathematical frame for the description of a denotational semantics which is better suited for the formal study of the language. Unfortunately, the closest theory is Combinatory Logic [9] and seems to be inadapted because of evident incompatibilities described by J. W. Backus for its FP systems [1]. As a matter of fact, CL is simple and can be used for modelling [15] but Curryfication and intermediate results are not features of usable functional languages. The theory which will be presented later avoids them so that it can be used for a natural modelization of functional languages.

The theory T_G can be roughly described as a Combinatory Logic with uncurryfied combinators. That is to say, application is no longer a binary operation as in classical theories such as Lambda-Calculus [12], CL, Uniformly Reflexive Structures [17] and so on. This is done by the introduction of the sequence structure which can be seen as a formalization of the multiple-arguments concept. As an immediate consequence, combinators S and K must be revisited and a new combinator T is added in order to have access to components of a sequence.

T_G is described as a formal system [14, 12]. This article gives a proof of the Consistency of the theory. First of all, we must establish the Diamond Lemma for the reduction relation and the Church-Rosser property for the associated equality. Then Consistency is an easy consequence of uniqueness of normal form for a given term.

After Consistency, we prove the Completeness theorem for T_G using an Abstraction algorithm associating to a variable and a term a function which is the result of the abstraction of the variable in the term. Then a fixed-point combinator is constructed as in CL.

Following the classical presentation, we define a set of Numerals (as iterators) and the notion of Definability for a partial function of natural numbers using the absence of normal form as the definition for the concept of "undefinedness". The Definability theorem establishes that Partial Recursive functions [13] are definable in T_G. Following Church's Thesis, we may admit that T_G can serve as a basis for the study of Computability.

The theory T_G is not handicaped by Curryfication and describes n-ary functions as n-ary terms in a natural way, not as Curryfied terms. This property is useful in order to get definitions which are not cumbersome. For any definition, terms are less complex than within a classical theory and yield fewer elementary reduction steps (contraction) so that computations are faster and simpler. It has been used by the language Graal and could be used by every functional language modelled with T_G. For instance, Turner's scheme of implementation [16] could be repeated for lambda-languages inheriting the efficiency of TG reduction and Abstraction algorithm.

Because of the Completness theorem, T_G allows the definition of every recursive function but we must remark that these functions have a fixed arity, and a function with variable arity such as:

$$f(x_1, \ldots, x_n) = x_1 + \ldots + x_n \qquad \text{for every } n \geqq 0 \text{ and numbers } x_1, \ldots, x_n$$

has no simple description in classical theories, nor in T_G. It is not clear how f can be represented in CL or Lambda-Calculus without a heavy construction. Nevertheless, such functions are programmable in Lisp, in Graal and in any practical functional programming language.

Consequently, we provide a conservative extension of the theory T_G which is designed for the formalization of the concept of function with variable arity. This is done by the introduction of some sequence management combinators. Using the extended theory T_{GE}, it is possible to describe the semantics of Lisp lambda-expressions with atomic parameter list or Graal functional forms such as "reduction" whose semantics depends upon the number of arguments.

Thus, T_{GE} is presented as an extension of the mathematical theories of computable functions towards computer science and a better understanding of the practical concepts used in functional programming.

1. T_G: THE THEORY

This section presents concisely the formal system $\mathbf{T_G}$. We must define three sets: *formulas*, *axioms* and inference *rules*. Rules are written with premises above an horizontal line and conclusions below it.

ALPHABET:

K, S, T, constants.

$v_0, v_1, v_2, \ldots$, variables (enumerable set).

$\Rightarrow$, reduction symbol.

$=$, equality symbol.

:, application symbol.

(,), parenthesis.

$\mathbf{T_G}$-TERMS AND $\mathbf{T_G}$-SEQUENCES: the sets of T_G-terms and T_G-sequences are defined inductively:

- every constant is a term;
- every variable is a term;
- every term is a sequence;
- if **a** is a term and **s** is a sequence, **as** is a sequence;
- if **f** is a term and **s** is a sequence, (**f** : **s**) is a term.

Thus a sequence is the concatenation of a finite number of terms. A sequence composed with terms $\mathbf{x_1}$, $-$, $\mathbf{x_n}$ may be denoted $\mathbf{x_1 - x_n}$. The term $(\mathbf{f} : \mathbf{x_1 - x_n})$ is the application of **f** to the sequence of arguments $\mathbf{x_1 - x_n}$.

As usual, application is associative to the left so that we could write $\mathbf{f} : \mathbf{x_1 - x_n} : \mathbf{y_1 - y_m}$ instead of $(\mathbf{f} : \mathbf{x_1 - x_n}) : \mathbf{y_1 - y_m}$.

FORMULAS: A formula of $\mathbf{T_G}$ may be $\mathbf{P \Rightarrow Q}$ or $\mathbf{P = Q}$ where **P** and **Q** are terms.

NOTATIONS: Notations are taken from classical theories:

- variables are denoted with small letters: **x**, **y**, **z**, ... with possible indexes
- **V** is the set of variables
- capital letters **P**, **Q**, ... which are not used for constants denote arbitrary terms, they can be indexed
- $\mathbf{X_1 - X_n}$ denotes the sequence of indexed terms $(\mathbf{X_i})_{1 \leq i \leq n}$.

DEFINITION: a term which does not contain any variable is a *combinator*. Terms S, K and T are *basic combinators*. A term which contains at least one variable is an open term.

THE THEORY: The theory $\mathbf{T_G}$ is defined by the following axiom-schemes and rules:

$(K)\, K : X_1 \ldots X_n : Y_1 \ldots Y_m \;\Rightarrow\; X_1$ (constant)

$(S)\, S : FG_1 \ldots G_m : X_1 \ldots X_n$

$\Rightarrow\; F : X_1 \ldots X_n : (G_1 : X_1 \ldots X_n) \ldots (G_m : X_1 \ldots X_n)$ (substitution)

$(T)\, T : G_1 \ldots G_m : X_1 X_2 \ldots X_n \;\Rightarrow\; G_1 : X_1 X_2 \ldots X_n : X_2 \ldots X_n$ (tail)

$$(a)\ \frac{X_i \Rightarrow Y_i,\quad 1 \leqq i \leqq n}{F : X_1 \ldots X_n \Rightarrow F : Y_1 \ldots Y_n},\qquad (f)\ \frac{F \Rightarrow G}{F : X_1 \ldots X_n \Rightarrow G : X_1 \ldots X_n}$$

$$(t)\ \frac{M \Rightarrow N,\ N \Rightarrow P}{M \Rightarrow P},\qquad (r)\ M \Rightarrow M$$

$$(e)\ \frac{M \Rightarrow N}{M = N},\qquad (s)\ \frac{M = N}{N = M}$$

$$(t')\ \frac{M = N,\ N = P}{M = P}$$

Axioms (e), (s) and (t') establish as usual that equality is the reflexive and transitive closure of reduction.

As it is presented, $\mathbf{T_G}$ is a formal theory [14] where deductions must be viewed as trees. A *deduction* of a formula $\mathbf{F}$ from a set of formulas $\mathbf{S}$ (*assumptions*), is a tree with formulas at the tops of the branches (leaves) being axioms or formulas in $\mathbf{S}$, and $\mathbf{F}$ standing at the bottom (root). If such a deduction exists, we may write:

$$\mathbf{T_G},\ \mathbf{S} \vdash \mathbf{F}$$

If the set $\mathbf{S}$ of assumptions is empty, $\mathbf{F}$ is called a *provable formula* or a *theorem* and the deduction is a *proof.* If $\mathbf{F}$ is a theorem, we may write: $\mathbf{T_G} \vdash \mathbf{F}$.

Example of proof:

$$(t)\ \frac{(\mathbf{S})\, S : K\ K : x \;\Rightarrow\; K : x : (K : x)\quad (\mathbf{K})\, K : x : (K : x) \;\Rightarrow\; x}{S : K\ K : x \;\Rightarrow\; x}$$

Thus: $T_G \vdash S : K\ K : x \Rightarrow x$.

Example of combinators: As usual names are given to particularily useful combinators. Here are two such combinators:

$I \equiv_{\text{def}} S : KK$

$$\begin{aligned} & I : x_1 \dots x_n \\ \equiv \quad & S : KK : x_1 \dots x_n \\ \Rightarrow \quad & K : x_1 \dots x_n : (K : x_1 \dots x_n) \\ \Rightarrow \quad & x_1 \end{aligned}$$

$H \equiv_{\text{def}} S : (K : S) I (K : I)$

$$\begin{aligned} & H : f g_1 \dots g_m : x_1 \dots x_n \\ \equiv \quad & S : (K : S) I (K : I) : f g_1 \dots g_m : x_1 \dots x_n \\ \Rightarrow \quad & K : S : f g_1 \dots g_m : (I : f g_1 \dots g_m)(K : I : f g_1 \dots g_m) : x_1 \dots x_n \\ \Rightarrow \quad & S : f I : x_1 \dots x_n \\ \Rightarrow \quad & f : x_1 \dots x_n : (I : x_1 \dots x_n) \\ \Rightarrow \quad & f : x_1 \dots x_n : x_1. \end{aligned}$$

DEFINITIONS: If $M \Rightarrow N$ is a reduction obtained by the contraction of exactly one redex in M, $M \Rightarrow N$ is called a *contraction* and can be denoted by $M \Rightarrow {}_1 N$. The inverse path from N to M is called an *expansion.* The notions of *subterm* and *occurrence* are defined classically (as in Combinatory Logic, see [12] for instance). It is assumed that they are known, as usual!

DEFINITIONS:

– A *redex* is a term of form

$$\mathbf{K} : \mathbf{x}_1 - \mathbf{x}_n : \mathbf{y}_1 - \mathbf{y}_m$$

or

$$\mathbf{S} : \quad \mathbf{f}\, \mathbf{x}_1 - \mathbf{x}_n : \mathbf{y}_1 - \mathbf{y}_m \text{ or } \mathbf{T} : \mathbf{x}_1 - \mathbf{x}_n : \mathbf{y}_1 \mathbf{y}_2 - \mathbf{y}_m$$

– The term corresponding to a redex in axiom-schemes (K), (S) or (T) is a *contractum.*

– A term which contains no redex is a *normal form* (n.f.).

– Let M be a term and N be a normal form such that $M = N$, then M is said to be *normalizable* and N is its normal form.

DIAMOND LEMMA: *If M, P and Q are terms such that $M \Rightarrow P$ and $M \Rightarrow Q$, there exists a term N such that $P \Rightarrow N$ and $Q \Rightarrow N$.*

Proof [Annexe 1, personal notes]: The proof uses the Tait-Martin-Löf method of 1981 with minimal complete developments adapted to polyadic functions. See [11] for the sketch of the proof.

CHURCH-ROSSER PROPERTY: If M and N are terms such that $M=N$, there exists a term L such that $M \Rightarrow L$ and $N \Rightarrow L$.

Proof: As usual, the contraction-expansion path from M to N is reduced until it becomes a two-step path [2, 11]. This is done by applications of the Diamond Lemma.

Usual figures

$$\begin{tikzcd} H \arrow[r] \arrow[d] & Q \arrow[d] \\ P \arrow[r] & L \end{tikzcd} \qquad \begin{tikzcd} H \arrow[rr, equal] \arrow[dr] & & H \arrow[dl, dash] \\ & L & \end{tikzcd}$$

Diamond Lemma Church-Rosser Property

COROLLARIES: *The following properties are deduced from the Church-Rosser property.*

– *If M is a term and N a normal form such that $M=N$, then $M \Rightarrow N$.*

– *The normal form of a term is unique when it exists.*

CONSISTENCY THEOREM: T_G is consistent.

Proof: S and K are non equal n.f.s since S cannot reduce to K.

NOTATIONS

$\equiv$ denotes the syntactic identity between terms, i. e.: $M \equiv N$ if M and N are written with the same symbols in the same order.

$\equiv_{\text{def}}$ denotes equality by definition. It is just a way to give name to terms in the metalanguage.

PROJECTIONS: We define a family $(P_k)_{k>0}$ of combinators by:

$$P_1 \equiv_{\text{def}} S:K\ K$$

$$P_{k+1} \equiv_{\text{def}} T:(K:P_k)$$

then we have: $P_k: X_1 \ldots X_n \Rightarrow X_k$ if $1 \leqq k \leqq n$.

Proof: Induction on k.

$$
\begin{aligned}
P_1 : X_1 \dots X_n \equiv_{\text{def}} S : K\ K : X_1 \dots X_n &\Rightarrow K : X_1 \dots X_n : (K : X_1 \dots X_n) \\
&\Rightarrow X_1 \\
P_{k+1} : X_1 \dots X_n \equiv_{\text{def}} T : (K : P_k) : X_1 \dots X_n &\Rightarrow K : P_k : X_1 \dots X_n : X_2 \dots X_n \\
&\Rightarrow P_k : X_2 \dots X_n \\
&\Rightarrow X_{k+1}.
\end{aligned}
$$

DEFINITION OF SUBSTITUTION: Let M and N be terms and x be a variable, the result of the substitution of N for all occurrences of x in M is denoted $[N/x]M$. It is defined by induction on M:

- $[N/x]x \equiv N$;
- $[N/x]y \equiv y$ if y is an atom (variable, constant) different from x;
- $[N/x](F : X_1 \dots X_n) \equiv [N/x]F : [N/x]X_1 \dots [N/x]X_n$.

As usual, $[N_1/x_1 \dots N_k/x_k]M$ denotes $[N_1/x_1] \dots [N_k/x_k]M$ where substitutions are done in parallel. It is equivalent to admit a Variable Convention [12] (i. e.: automatic renaming of variables).

COMPLETENESS THEOREM: Let M be a term and $x_1, \dots, x_n$ be variables, there exists a term denoted $(\lambda x_1 \dots x_n . M)$ such that none of the variables $x_1, \dots, x_n$ appears in it and:

$$(\lambda x_1 \dots x_n . M) : N_1 \dots N_n \quad \Rightarrow \quad [N_1/x_1 \dots N_n/x_n]M.$$

Proof: $(\lambda x_1 \dots x_n . M)$ is constructed inductively

- $\lambda x_1 \dots x_n . x_i \equiv P_i$ since: $P_i : N_1 \dots N_n \Rightarrow N_i \equiv [N_1/x_1] \dots [N_n/x_n] x_i$
- $\lambda x_1 \dots x_n . y \equiv K : y$ if y is an atom different from $x_1, \dots, x_n$

since: $K : y : N_1 \dots N_n \Rightarrow y \equiv [N_1/x_1 \dots N_n/x_n] y$

- $\lambda x_1 \dots x_n . (F : M_1 \dots M_m)$

$$\equiv S : (\lambda x_1 \dots x_n . F)(\lambda x_1 \dots x_n . M_1) \dots\dots (\lambda x_1 \dots x_n . M_m)$$

since:

$$S:(\lambda x_1 \ldots x_n . F)(\lambda x^1 \ldots x_n . M_1) \ldots\ldots (\lambda x_1 \ldots x_n . M_m): N_1 \ldots N_n$$

$$\Rightarrow \quad (\lambda x_1 \ldots x_n . F): N_1 \ldots N_n : ((\lambda x_1 \ldots x_n . M_1):$$

$$N_1 \ldots N_n) \ldots ((\lambda x_1 \ldots x_n . M_m): N_1 \ldots N_n)$$

$$\Rightarrow \quad ([N_1/x_1 \ldots N_n/x_n] F):([N_1/x_1 \ldots N_n/x_n] M_1) \ldots\ldots ([N_1/x_1 \ldots N_n/x_n] M_m)$$

$$\equiv \quad [N_1/x_1 \ldots N_n/x_n](F: M_1 \ldots M_m).$$

PROPERTY: Let M be a term, $x_1, \ldots, x_n$ be variables, $(\lambda x_1 \ldots x_n . M)$ is a normal form.

Proof: Easy induction.

LEMMA 1: *Let N be a normal form, M be a term, $x_1, \ldots, x_n$, y be variables, $[N/y](\lambda x_1 \ldots x_n . M)$ is a normal form.*

Proof: Trivial induction since variable y may only appear in subterms $(K: y)$ which are not the prefix of a redex.

LEMMA 2: *Let N be normalizable, M be a term, $x_1, \ldots, x_n$, y be variables, $[N/y](\lambda x_1 \ldots x_n . M)$ is normalizable.*

Proof: Remark that if L is the normal form of N (i. e.: $N \Rightarrow L$), we have $[N/y](\lambda x_1 \ldots x_n . M) \Rightarrow [L/y](\lambda x_1 \ldots x_n . M)$. Then apply lemma 1.

PROPERTY: Let M be a term, $X_1, \ldots, X_n$ be normalizable terms, $x_1, \ldots, x_n, y_1, \ldots, y_m$ be variables, then:

$$(\lambda x_1 \ldots x_n . (\lambda y_1 \ldots y_m . M)): X_1 \ldots X_n$$

is normalizable.

Proof: We have:

$$(\lambda x_1 \ldots x_n . (\lambda y_1 \ldots y_m . M)): X_1 \ldots X_n$$
$$\Rightarrow \quad [X_1/x_1 \ldots X_n/x_n](\lambda y_1 \ldots y_m . M)$$

and we apply previous lemma with the variable convention.

FIXED-POINT THEOREM: *Let $Y \equiv_{\text{def}} \Omega : \Omega$ where $\Omega \equiv_{\text{def}} S:(K: S)\,(K: I)(S: I\, I)$, we have: $Y: F \Rightarrow F:(Y: F)$.*

Proof:

$$\begin{aligned}\Omega : a : b \equiv S : (K : S)(K : I)(S : I\ I) : a : b \\ &\Rightarrow K : S : a : (K : I : a)(S : I\ I : a) : b \\ &\Rightarrow S : I(a : a) : b \\ &\Rightarrow I : b : (a : a : b) \\ &\Rightarrow b : (a : a : b)\end{aligned}$$

Now: $Y : f \equiv \Omega : \Omega : f \Rightarrow f : (\Omega : \Omega : f) \equiv f : (Y : f)$.

NORMALIZING EXTENSIONAL FIXED-POINT FAMILY: There exists a family $(Y_n)_{n \geqq 1}$ of combinators such that for each $n \geqq 1$, we have the following:

(*a*) Y_n is normalizable

(*b*) $Y_n : F$ is normalizable whenever F is normalizable

(*c*) $Y_n : F : X_1 \ldots X_n \Rightarrow F : (Y_n : F) : X_1 \ldots X_n$.

Proof: Let us define $\Omega_n \equiv_{\text{def}} \lambda a . (\lambda f . (\lambda x_1 \ldots x_n . f : (a : a : f) : x_1 \ldots x_n)$ and $Y_n \equiv_{\text{def}} \Omega_n : \Omega_n$. We have:

(*a*)

$$\begin{aligned}Y_n \Rightarrow\ & (\lambda a . (\lambda f . (\lambda x_1 \ldots x_n . f : (a : a : f) : x_1 \ldots x_n)) : \Omega_n \\ \Rightarrow\ & [\Omega_n / a] (\lambda f . (\lambda x_1 \ldots x_n . f : (a : a : f) : x_1 \ldots x_n)).\end{aligned}$$

Thus, Y_n is normalizable because of previous properties of the Abstraction algorithm: Ω_n is a normal form and the substitution of a normal form in an abstraction is a normal form

(*b*) $Y_n : F \Rightarrow [F/f][\Omega_n/a]\ (\lambda x_1 \ldots x_n . f : (a : a : f) : x_1 \ldots x_n)$. Because f does not occur in Ω_n and we may suppose that a does not occur in F, $Y_n : F$ is equal to: $[F/f,\ \Omega_n/a]\ (\lambda x_1 \ldots x_n . f : (a : a : f) : x_1 \ldots x_n)$, therefore it is normalizable.

(*c*)

$$\begin{aligned}Y_n : F : X_1 \ldots X_n \ &\Rightarrow\ [X_1/x_1] \ldots [X_n/x_n][F/f][\Omega_n/a](f : (a : a : f) : x_1 \ldots x_n) \\ &\Rightarrow\ F : (Y_n : F) : X_1 \ldots X_n.\end{aligned}$$

NUMERALS: Numerals are defined following Church's numerals of Lambda-Calculus [2]. The numeral $[n]$ representing the natural number n reduces to an iterator $\lambda f, x . f^n : x$ where $f^0 : x \equiv x$ and $f^{k+1} : x \equiv f : (f^k : x)$. A particular definition of numerals is the following:

$$[0] \equiv_{\text{def}} P_2$$

$$[n+1] \equiv_{\text{def}} [s]:[n] \qquad \text{where:} \quad [s] \equiv_{\text{def}} S:(K:S)(K:I)\,I.$$

Proof:

$$\left.\begin{aligned} &[0]:f\,x \equiv_{\text{def}} P_2:f\,x \Rightarrow x \text{ by definition of } P_2\\ &[n+1]:f\,x \equiv_{\text{def}} [s]:[n]:f\,x\\ &\qquad \equiv_{\text{def}} S:(K:S)(K:I)\,I:[n]:f\,x\\ &\qquad \Rightarrow \quad K:S:[n]:(K:I:[n])(I:[n]):f\,x\\ &\qquad \Rightarrow \quad S:I[n]:f\,x\\ &\qquad \Rightarrow \quad I:f\,x:([n]:f\,x)\\ &\qquad \Rightarrow \quad f:([n]:f\,x) \end{aligned}\right\}$$

DEFINABILITY: Let f be a n-ary partial function of natural numbers, f is definable (in T_G) if there exists a T_G-function F such that for all natural numbers $m_1, \ldots, m_n$ and r, we have:

$$f(m_1, \ldots, m_n) = r \Leftrightarrow T_G \vdash F:[m_1] \ldots [m_n] = [r]$$

$f(m_1, \ldots, m_n)$ undefined $\Leftrightarrow F:[m_1] \ldots [m_n]$ has no normal form.

DEFINABILITY THEOREM: Partial Recursive Functions are definable in the theory T_G.

Proof: Very long. Facilited by polyadicity of terms. *See* [2] or [12] for a sketch of the proof.

CONCLUSION

As described, T_G is a formal theory similar to Combinatory Logic [9] except that functions are uncurryfied. It seems to have the same power than CL since CL-combinators are easily defined in T_G and conversely, T_G-combinators can be modelized by families of CL-combinators. As a matter of fact, T_G can be used for the study of functional languages through models in the same way as CL [5, 15] but without the inconvenience mentioned in [1].

T_G could be substituted for CL in Turner's like implementations of lambda-languages [16]. The gain would be efficiency since T_G-combinators abolish curryfication and consequently diminish the number of reduction steps needed for computations. The practical counterpart of T_G, that is the Graal language, is certainly one of the most efficient functional language running on classical computers.

For the author, the principal interest of T_G is that it provides an elementary model for the language Graal. We could say that T_G is to Graal what Lambda-Calculus is to pure lambda-languages such as KRC, ML, ... [10]. Therefore, the theory T_G provides a clean semantics for Graal where proof for programs equivalence can be formally done in the extensional theory $T_G+(\text{ext})$ using axioms similar to laws of the Algebra of Programs [1, 5], and semantics of recursive functions may be established as in [6].

2. T_{GE}: THE EXTENDED THEORY

The theory T_G is adapted for the description of functions with fixed arity as are Partial Recursive functions. Nevertheless, functions with variable arity are programmable in functional languages such as Lisp or Graal. This feature is Hidden by Curryfication in classical theories. We could imagine a function $[+]$ such that for numbers $x_1, \ldots, x_n$, we have:

$$[+]:[x_1]\ldots[x_n]=[x_1+\ldots+x_n]$$

whatever the length n of the argument sequence is.

In the same vein, we could need a combinator B such that:

$$(B)\quad B: f g_1 \ldots g_m : x_1 \ldots x_n \Rightarrow f:(g_1 : x_1 \ldots x_n)\ldots(g_m : x_1 \ldots x_n)$$

It is a fact that we can define a B_m for each length m of the sequence $f g_1 \ldots g_m$ but we are unable to construct a uniform B but there is no proof of this negative result at the present time.

The aim of this section is to provide a set of combinators designed for sequence management without any loss in the domain. In particular, the Church-Rosser property will remain true and the extended theory is a conservative extension of T_G.

As can be easily remarked, the purpose of S is composition meanwhile K is an eraser as in CL. T_G needed the additional T combinator for the extraction of individual arguments. As a matter of fact, the projections (argument selectors of previous section) are just compositions of K with some T's.

S, K and T are sufficient for the description of functions with fixed arity and some very particular functions with variable arity such as I. In order to deal with variable arity, we need a new combinator D which can be viewed as an arity discriminator. D is given by the following axiom-scheme:

$$\text{(D)}\quad D:D_1\,G_2\ldots G_m:X \;\Rightarrow\; G_1:X \quad \text{(discriminator)}$$
$$D:G_1\,G_2\ldots G_m:X_1\,X_2\ldots X_n \;\Rightarrow\; G_2:X_1\,X_2\ldots X_n$$

But D is not sufficient since if we can decrease the number of arguments of a function (with combinator T), we are unable to increase it. Therefore, we introduce a new combinator L (for left insertion) given by:

$$\text{(L)}\quad \mathrm{L}:F\,G_1\ldots G_m:X_1\ldots X_n \;\Rightarrow\; F:X_1\ldots X_n:(G_1:X_1\ldots X_n)\,X_1\ldots X_n \quad \text{(left)}$$

We must define the theory T_{GE} as follows:

$\mathbf{T_{GE}}$*-terms and* $\mathbf{T_{GE}}$*-sequences:*

- every constant (S, K, T, D, L) is a term
- every 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.

Notations used for T_G are still valid.

The theory $\mathbf{T_{GE}}$: The set of axioms of T_G is extended with axioms-schemes (D) and (L) given above.

Redex: The definition of a redex is modified. A redex is a term of the form

$$(\mathbf{S}:\mathbf{f}\,\mathbf{g}_1\ldots\mathbf{g}_m:\mathbf{x}_1\ldots\mathbf{x}_n) \quad\text{or}\quad (\mathbf{K}:\mathbf{x}_1\ldots\mathbf{x}_n:\mathbf{y}_1\ldots\mathbf{y}_m)$$

or

$$(\mathbf{T}:\mathbf{f}\,\mathbf{g}_1\ldots\mathbf{g}_m:\mathbf{x}_1\,\mathbf{x}_2\ldots\mathbf{x}_n) \quad\text{or}\quad (\mathbf{L}:\mathbf{f}\,\mathbf{g}_1\ldots\mathbf{g}_m:\mathbf{x}_1\,\mathbf{x}_2\ldots\mathbf{x}_n)$$

or

$$(\mathbf{D}:\mathbf{f}\,\mathbf{g}_1\ldots\mathbf{g}_m:\mathbf{x}_1\,\mathbf{x}_2\ldots\mathbf{x}_n).$$

The definitions of a normal form and a normalizable term are still valid for T_{GE}. Before giving some examples of use of these new combinators, we must establish classical results. The first results about the extended theory are given by [11, chap. 2] for more general CRS (Combinatory Reduction Systems).

Diamond Property: The theory T_{GE} has the Diamond property.

CHURCH-ROSSER PROPERTY: The theory T_{GE} verifies the Church-Rosser property.

Proofs: Reduction rules of T_{GE} are left-linear and non-ambigous. That is to say that there does not exist two axioms $M \Rightarrow N$ and $M' \Rightarrow N'$ with syntactically identical left members (non-ambiguous) and that in any axiom-scheme $M \Rightarrow N$, a metavariable does not have more that one occurrence in M (left-linear). These two properties mean that T_{GE} is a regular CRS in which the preceding results are always provable [11].

All general deductions from the CR-property (such as unicity of *n.f.*, Consistency, and so on) are still valid in T_{GE} and we admit them without repeating proofs for T_{GE}. The following property establishes the Consistency of T_{GE} in another way.

PROPERTY: T_{GE} is a conservative extension of T_G.

Proof: See [11] for a simple proof for more general ARS: every proper extension of a CR-theory is conservative.

3. EXAMPLES OF USE OF T_{GE}

Now, we give some examples of representation of functions with variable arity in T_{GE}. We begin by the paradigmatic example of the generalized addition:

GENERALIZED ADDITION: We search for a combinator P such that:

$$P:[n_1][n_2]\ldots[n_k]=[+]:[n_1]([+]:[n_2]\ldots([+]:[n_{k-1}][n_k])\ldots)$$

where $[+]$ is a representation of the classical binary addition issued from the Representation theorem. We search for combinators P_0 and P_1 which are versions of P respectively to one and more arguments. We want:

$$P_0:N_1=P:N_1=N_1$$

thus it suffices to take: $P_0 \equiv_{\text{def}} \mathrm{I}$. Now we want:

$$\begin{aligned}
&P_1:N_1 N_2\ldots N_k\\
&=[+]:N_1([+]:N_2\ldots([+]:N_{k-1}N_k)\ldots)\\
&=[+]:N_1(P:N_2\ldots N_k)\\
&=[+]:(I:N_1 N_2\ldots N_k)((K:P:N_1 N_2\ldots N_k:N_2\ldots N_k)
\end{aligned}$$

$=[+]:(I:N_1 N_2 \dots N_k)(T:(K:P):N_1 N_2 \dots N_k)$

$=K:[+]:N_1 N_2 \dots N_k:(I:N_1 N_2:N_k)(T:(K:P):N_1 N_2:N_k)$

$=S:(K:[+])I(T:(K:P)):N_1 N_2 \dots N_k.$

Using the D combinator, we just ned: $P=D:P_0 P_1$, that is to say:

$P=D:I(S:(K:[+])I(T:(K:P)))$

$=D:I(K:S:P:(K:(K:[+]):P)(K:I:P)(S:(K:T)K:P)))$

$=D:I(S:(K:S)(K:(K:[+]))(K:I)(S:(K:T)K):P)$

$=K:D:P:(K:I:P)(S:(K:S)(K:(K:(K[+]))(K:I)(S:(K:T)K):P)$

$=S:(K:D)(K:I)(S:(K:S)(K:(K:[+]))(K:I)(S:(K:T)K)):P$

Thus we obtain a fixed-point equation which can be solved and we have:

$$P \equiv_{\text{def}} Y:(S:(K:D)(K:I)(S:(K:S)(K:(K:[+]))(K:I)(S:(K:T)K))).$$

A DISTRIBUTION OPERATOR: We search for a combinator A such that:

$$A:FG:X_1 \dots X_n=F:(G:X_1) \dots (G:X_n)$$

As for P. We distinguish the cases:

$A_0:FG:X$

$=F:(G:X)$

$=K:F:X:(G:X)$

$=S:(K:F)G:X$

$=S:(K:(P_1:FG))(P_2:FG):X$ [P_1 and P_2 are projections]

$=S:(S:(K:K)P_1:FG)(P_2:FG):X$

$=S:(K:S)(S:(K:K)P_1)P_2:FG:X.$

$A_1:FG:X_1 X_2 \dots X_n$

$=F:(G:X_1)(G:X_2) \dots (G:X_n)$

$=K:F:(G:X_2) \dots (G:X_n):(K:(G:X_1):(G:X_2)$
$\dots (G:X_n))(G:X_2) \dots (G:X_n)$

$=L:(K:F)(K:(G:X_1)):(G:X_2) \dots (G:X_n)$

$=K:L:X_1:(K:(K:F):X_1)(S:(K:K)G:X_1):(G:X_2) \dots (G:X_n)$

$=S:(K:L)(K:(K:F))(S:(K:K)G):X_1:(G:X_2) \dots (G:X_n)$

$$=A:(S:(K:L)(K:(K:F))(S:(K:K)G):X_1)G:X_2\ldots X_n$$
$$=(K:A:X_1):(S:(K:L)(K:(K:F))$$
$$(\mathrm{S}:(K:K)G)):X_1)(K:G:X_1):X_2\ldots X_n$$
$$=S:(K:A)(S:(K:K)(S:(K:L)(K:(K:F))$$
$$(\mathrm{S}:(K:K)G)))(K:G):X_1:X_2\ldots X_n.$$

Let us name:

$$\mathrm{U}[\mathrm{F},\mathrm{G},\mathrm{A}]\equiv \mathrm{S}:(\mathrm{K}:\mathrm{D})(\mathrm{S}:(\mathrm{K}:\mathrm{K})(\mathrm{S}:(\mathrm{K}:\mathrm{L})(\mathrm{K}:(\mathrm{K}:\mathrm{F}))$$
$$\times(\mathrm{S}:(\mathrm{K}:\mathrm{K})\mathrm{G})))(\mathrm{K}:\mathrm{G})$$
$$=U[F, G, A]:X_1:X_2\ldots X_n$$
$$=U[F, G, A]:(I:X_1X_2\ldots X_n):X_2\ldots X_n$$
$$=S:(K:U[F, G, A])I:X_1X_2\ldots X_n:X_2\ldots X_n$$
$$=T:(S:(K:U[F, G, A])I):X_1X_2\ldots X_n$$

Thus we may have: $A_1=\lambda f, g.(T:(S:(K:U[f, g, A])I))$.

Thus: $A=D:A_0(\lambda f, g.(T:(S:(K:U[f, g, A])I)))$.

It is a fixed point equation and the solution is:

$$A=Y:(\lambda a.(D:A_0(\lambda f, g.(T:(S:(K:U[f, g, A])I))))).$$

ARGUMENTS COUNT: How to know the number of arguments:

$$C:X_1\ldots X_n \Rightarrow [n].$$

We have: $[n]=P:[1]\ldots[1]$ where $[1]$ occurs n times and P is generalized addition. Thus:

$$[n]=P:(K:[1]:X_1)\ldots(K:[1]:X_n)$$
$$=A:P(K:[1]):X_1\ldots X_n$$

Therefore, we define:

$$C\equiv_{\mathrm{def}} A:P(K:[1]).$$

COMPOSITION: Our last example is the uniform B announced in the introduction of section 2, such that:

$$B:FG_1\ldots G_m:X_1\ldots X_n=F:(G_1:X_1\ldots X_n)\ldots(G_m:X_1\ldots X_n)$$

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

Therefore:

$$B \equiv_{\text{def}} T: (S: (K: L)(K: (K: S))(S: (K: (S: (K: K) K)) I))$$

CONCLUSION

It is a fact that computations in T_{GE} are at least as unreadable as computations in Combinatory Logic. Especially, the kind of abstraction given in previous examples (P.A.B) is not trivial and needs a lot of attention. It could be called sequence-abstraction. Nevertheless, such a language is not intended for human manipulation but for computer science purpose.

4. COMPARISON WITH λ-CALCULUS AND *LC*

If we want to deal with functions having variable arity in λ-calculus [L_1 we have two possible choices. The first one is to consider functions applied to a list of their arguments: $F\langle X_1 \ldots X_n \rangle$. The list can be a linked list constructed with a pairing operator or a tuple of values [2]. Lisp systems and FP systems use this method and it is possible to program functions with variable arity. But, if lists are present at the theoretical level, they must be present in the implementation and it is rather expansive to deal explicitely with lists whatever they are (linked lists or tuples) [7]. Another way is to use a discriminable term (named $\square$) as an end marker for arguments and to write application as: $(f\, X_1 \ldots X_n\, \square)$. Then it is possible to recognize the end of the arguments sequence. It is quite a complex solution since each partial application needs a (perhaps non trivial) test on the argument. If we try to play with uncurryfied λ-calculus allowing the lambda-notation with a sequence of variables as first parameter of the lambda operator, the problem is not resolved unless we fall into the Lisp conception which is equivalent to a λ-calculus with lists [10].

5. CONCLUSIONS

This article presents a new theory named T_{GE} based on uncurryfied combinators and issued from the Graal language. First of all, the theory is able to describe functions as are other theories. Because combinators are uncurryfied, it gives rise to simplicity and efficiency. Intermediate results (partial applications) are avoided, so that it decreases the number of elementary steps in a reduction process. Because primitive combinators of T_{GE} are more powerful than in CL, the Abstraction can be very efficient, giving short terms. It is an open problem to know if there exists a linear algorithm.

The theory expresses functions in a natural way. Usual functions from mathematics and computer sciences are not curryfied and can have variable arity. Thus, models of programming languages must be more natural using T_{GE}. Experience has proved (via the Graal reduction machine) that implementation of these uncurrified combinators gives a very good execution time on classical Von Neumann architectures. More than this, [3] establishes a quasi-direct translation between Graal programs and Dataflow programs. Therefore, modelization of languages using T_{GE} must be an efficient technique of implementation either on classical architecture or on new ones.

Linked to Abstraction is the theorical problem of basis. What is the minimal number of primitive combinators needed for Completness in T_G and in T_{GE}? What is the basis which gives best Abstraction algorithms for T_G and for T_{GE}? These problems have evident implementation consequences.

A remarkable point about T_{GE} is its ability to describe functions with variable arity (Fva's for short) without deep constructions (such a lists) which are not efficiently transatable in practice. This has been pointed out with some examples. It is still a problem to describe formally Fva's, this is necessary for the specification and the construction of a Generalized Abstraction algorithm which does not exist till now. Functions such as Fva's are used in practice and it would be useful to give a model for them.

Another prolongation of this work is the study of the extensional theory, that is $T_{GE}+(\mathrm{ext})$. The result may be an Algebra of Programs usable in term rewriting systems as it has been done for FP [4]. It could be helpful for proving programs properties in T_{GE}, Graal and languages compiled in them.

ACKNOWLEDGMENTS

A. Belkhir, R. Legrand, V. Jay, D. Sarni have contributed to this article by interesting discussions. This work has been supported by the Greco de Programmation (Bordeaux) under project PACTE.

REFERENCES

1. J. W. Backus, *Can Programming Be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs*, C.A.C.M., Vol. 1, N° 8, 1978, pp. 613-641.
2. H. P. Barendregt, *The Lambda-Calculus, its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics, Vol. 103, North Holland, 1981.
3. A. Belkhir, *Programmation fonctionnelle et parallélisme*, Rapport Gréco de Programmation, 1986.
4. F. Bellegarde, *Rewriting Systems on FP Expressions that reduce the Number of Sequences they Yields*, A.C.M. Symposium on Lisp and Functional Programming, Austin, 1984.
5. P. Bellot, *Propriétés logico-combinatoires des systèmes de programmation sans variable*, Thèse 3e cycle, U.P.M.C. Paris-VI, Rapport LITP 84-30, 1983.
6. P. Bellot, *High order Programming in Extended FP*, FPCA 85, LNCS 201, J. P. Jouannaud Ed., pp. 65-80, Nancy, 1985.
7. P. Bellot, *Sur les sentiers du Graal, étude, conception et réalisation d'un langage de programmation sans variable*, Thèse d'État, U.P.M.C. Paris-VI, Rapport LITP 86-62, 1986.
8. P. Bellot, *Graal: a Functional Programming System with Uncurryfied Combinators and its Reduction Machine*, ESOP 86, LNCS 213, B. Robinet Ed., pp. 82-98, Saarbrucken, 1986.
9. H. B. Curry and R. Feys, *Combinatory Logic I*, North Holland, 1958.
10. N. Glaser, C. Hankin and D. Till, *Principles of Functional Programming*, Prentice/Hall International, 1984.
11. J. W. Klop, *Combinatory Reduction Systems*, Dissertation, University of Utrecht, 1980.
12. J. R. Hindley and J. P. Seldin, *Introduction to Combinators and Lambda-Calculus*, London Mathematical Society, Student Texts 1, Cambridge University Press, 1986.
13. S. C. Kleene, *Introduction to Metamathematics*, Van Nostrand, 1952.
14. E. Mendelson, *Introduction to Mathematical Logic*, Van Nostrand, 2nd ed., 1979.

15. B. Robinet, *Un modèle logico-combinatoire des systèmes de Backus*, Rapport LITP 80-21, 1980.
16. D. A. Turner, *A New Implementation Technique for Applicative Language*, Software-Practice and Experience, Vol. 9, 1979, pp. 31-49.
17. E. G. Wagner, *URS: in an Axiomatic Approach to Computability*, formation Sciences 1, 1969, pp. 343-362.