λ-definition of Function(al)s by Normal Forms*

Corrado Böhm[1], Adolfo Piperno[1], Stefano Guerrini[2]

[1] Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza",
Via Salaria 113, 00198 Roma, Italy,
e-mail: {boehm,piperno}@dsi-next1.ing.uniroma1.it
[2] Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, I-56100 Pisa, Italy,
e-mail: guerrini@di.unipi.it

Abstract. Lambda-calculus is extended in order to represent a rather large class of recursive equation systems, implicitly characterizing function(al)s or mappings of some algebraic domain into arbitrary sets. Algebraic equality will then be represented by $\lambda\beta\delta$-convertibility (or even reducibility). It is then proved, under very weak assumptions on the structure of the equations, that there always exist solutions in normal form (Interpretation theorem). Some features of the solutions, like the use of parametric representations of the algebraic constructors, higher-order solutions by currification, definability of functions on unions of algebras, etc., have been easily checked by a first implementation of the mentioned theorem, the CuCh machine.

1 Introduction

Combinatory logic [17] and λ-calculus [16] are different logic theories. Since there is still a one to one correspondence between a combinator and a closed λ-term, for the sake of simplicity we will refer to λ-terms most of the time.

A normal form (nf) is a λ-term irreducible respect to any β (η) reduction rule. Term reduction being the theoretic counterpart of computation, Church and its scholar Kleene proved the equivalence between λ-definability and recursive function theory finding out nf's representing any natural number or any recursive function [22].

On the other side Curry and Turing had a more liberal point of view on computability, in that computation shall not imply termination; e.g., Turing wrote of a machine computing the digits of π [30]. They discovered fixed point combinators $\mathbf{Y}_t$ and $\mathbf{Y}_c$, both without nf, to define partial functions introduced, e.g., by the μ-operator. The nf's used by Kleene to represent primitive or even general recursive functions were particularly intricate and not perspicuous.

In the mid 60's a small group of people, Wagner, Strong, and others [32, 28] tried to generalize recursive function theory to any type of data by Uniformly Reflexive Structures (URS), based on entities similar to λ-terms.

In the early 70's the treatment of recursive functions by fixed point combinators appeared more fascinating than the other approach, since they represent

* This work has been partially supported by grants from ESPRIT BRA 7232 working group "Gentzen" and from MURST 40% (Italy).

the idea of universal iterator, a finite object that can iterate functions an infinite number of times. Scott, Wadsworth and others were then able to construct the denotational semantics of programming languages based on fixed point theory and λ-calculus.

Simultaneously Wadsworth, Welch and others developed the notions of "head normal form" and "finite development," namely the $\lambda\Omega$-calculus. Still in the 70' the ADJ group and several other researchers developed an algebraic basis to programming, the "algebraic data types".

Backus [3] proposed to change the programming style a la Von Neumann, using variables and assignment statements, into an applicative style FP or FPP (similar to LISP) that avoided the use of variables.

Böhm [6] proved that FP was embeddable into combinatory logic and then our research group became interested to the embedding of algebraic data types and relative mappings of algebras (into another set) by λ-terms. A method to represent any type of term algebras and functions "iteratively" defined on that algebras, using second order typed nf's was introduced by [11]. This was a first incomplete but meaningful improvement on [22] and on URS. [7] and [8] extended the class of iterative functions, by means of two different methods, to treat primitive recursive and mutual iterative schemes. A remarkable result, preserving Church numeral system, exhibits a normal form for primitive recursive functionals [26].

However, two questions remained still unanswered: *1)* Was the typing really necessary? *2)* Could the same results be achieved for general schemes of recursion, defining any partial functions on any data structures? [9] answered positively both questions. The use of Böhm-tree [14, 15, 4] proved that the combinators representing the constructors of any homogeneous terms algebras are also a basis for the full combinatory logic and indeed for nf's. There is in addition a one to one mapping $*$ transforming the constructors into new constructors, therefore algebras into $*$-algebras, on which a class of equation schemes defining partial recursive functions admits solutions in nf.

[10] illustrated the interdependence between equation schemes and the choice of λ-terms representing zero and the successor function, to obtain nf for the solution of some schemes (generalizing [24]). The schemes examined were iterative, primitive recursive, general recursive and, for the last one, double recursion has been reduced by currification to the single one. The extension of the method of definition to term algebras of arbitrary data structure remains unanswered, as well as the treatment of mutual recursion schemes.

[5] defines rewriting systems, called "canonical and algebraic", and describes a Böhm–Piperno technique to obtain a definition in nf of a self-interpreter and of a reducer of a gödelization of the λ-calculus into itself.

The present paper shows how to expand the class of canonical systems so that our treatment is still valid. In addition, we list some attractive features of our nf solutions, otherwise lacking using fixed point combinators, and of interest for people looking for a concrete application of the theorems here presented.

Without embarking into a deep philosophical treatment, we would like to

convince the reader that the ideas behind our success in finding nf representing recursive functions or functionals on algebraic data can be made at least as popular as those of "structured" or "object oriented" programming. This is the aim of the following introduction. Let us begin quoting J.Shoenfield in connection with formal systems (page 2 of [27]):

> ... if we choose the language for expressing the axioms suitably, then the structure of the sentence will reflect to some extent the meaning of the axiom.

Our language is the λ-calculus. We must model algebraic expressions containing data as well as previously or newly defined functions. Our aim is to eliminate recursion from the definition of some function. The only possible way is to move inductive definitions from mappings to be defined into previously defined functions, i.e., the constructors of the domains of mappings. We may then talk of "data driven programming", an idea that extrapolates usual concepts from object oriented programming. Applying Shoenfield's recommendation we may choose for the algebraic language a syntax underlining the mentioned dichotomy between data and functions and simplifying λ-definitions. For the sake of this introduction, constructors will be written in prefix notation, whereas functions needing a recursive definition will be in suffix notation (if unary) and infix notation (if binary or n+2-ary). Let us associate to Booleans ($\mathbf{B}$), natural Integers ($\mathbf{N}$) and Lists of elements of a set A (L_A), the following features of their constructors: {name : arity, ... }. Then we have:

$$\mathbf{B} : \{\mathsf{True} : 0, \mathsf{False} : 0\}, \mathbf{N} : \{0 : 0, 1+ : 1\}, L_A : \{nil : 0, cons : 2\}.$$

We will give three examples of definition of functions: an explicit one for the ternary function if-then-else (ite), an iterative definition for the addition function (+), and a primitive recursive definition for the termial function (?) [23] whose intuitive definition is

$$n\,? = 0 + 1 + ... + (n-1) + n.$$

Using our syntax we can write

$$\mathsf{True}\ \mathsf{ite}\ x\ y = x$$
$$\mathsf{False}\ \mathsf{ite}\ x\ y = y$$

translating into the λ-calculus True and False it appears natural to consider the last two equations as definitions of True and False as combinators (and to treat ite as a variable). This poses the problem of defining a link between a constructor and a function, essentially the need to have a universal λ-definition for a constructor of given arity and simultaneously a systematic way to replace a constructor with a combinator related to the form of the equation. The answer will be found in the paper and, for the moment, we will ignore this problem. Let us write down two properties of the addition:

$$0 + n = n$$
$$(1 + m) + n = 1 + (m + n)$$

This system of equalities can be easily transformed into a recursive definition of the function + by the equality between the functions 1+ and +:

$$1 + x = 1+ x.$$

By replacement we obtain the recursive definitions

$$0 + n = n$$
$$1+ m + n = 1+ (m + n)$$

that become explicit definitions of the combinators 0 and 1+ considering $+, m, n$ as variables. We must notice: a) that positive integers are constructed as follows:

$$1 = 1+ 0, 2 = 1+ 1, 3 = 1+ 2, \ldots$$

b) In the second equation at the rhs $1+ (m+n)$ cannot be reduced since a property of combinator weak reduction is that reduction can take place only if the number of arguments of the combinator is greater or equal to that one appearing in the definition (here 3). Thus, the result of computing $3+n$ is $1+ (1+ (1+ n)))$, the result remaining valid if , before or after the computation, we will replace n by any non negative integer.

A tentative primitive recursive definition of ? like

$$0 ? = 0$$
$$(1+ n) ? = (1+ n) + (n ?)$$

could be translated into an explicit λ-definition only if we possess some delta-rules to define addition or if we can consider + as a predefined combinator. We can obviously form a system of four simultaneous equations and try to solve it, but we would have the same difficulty encountered above. An additional difficulty would arise in mutual simultaneous recursion.

The next section will solve all these difficulties. To spare the efforts of the reader we will return to prefix notation for all kinds of functions. Alert readers will discover factual infix notation hidden during β-reduction of some terms.

2 Recursive Equations and λ-calculus

As usual, we shall consider the set Λ of terms of the λ-calculus to be described by the following BNF, where a and x range over denumerable sets of constants and variables, respectively:

$$L ::= a \mid x \mid (\lambda x.L) \mid (L_1 L_2). \tag{1}$$

Let Σ be a set of function symbols from a given signature. $\Lambda(\Sigma)$ denotes the set of *extended lambda terms* with symbols from the signature Σ. To be precise $\Lambda(\Sigma)$ can be defined by adding the following clause to the clauses (1) for the formation of lambda terms: if $t_1, \ldots, t_n \in \Lambda(\Sigma)$ and $f \in \Sigma$ is an n-ary function symbol, then $f(t_1, \ldots, t_n) \in \Lambda(\Sigma)$. Note that $\mathrm{Ter}(\Sigma) \subseteq \Lambda(\Sigma)$ where $\mathrm{Ter}(\Sigma)$ is the set of first order terms with signature Σ.

Definition 1. Let $\mathcal{E}$ be a set of equations in the extended λ-calculus $\Lambda(\Sigma)$. We say that $\mathcal{E}$ is *canonical* if the function symbols in Σ can be partitioned in two disjoint subsets $\Sigma = \Sigma_0 \cup \Sigma_1$ so that, letting $\Sigma_0 = \{c_1, \ldots, c_r\}$ and $\Sigma_1 = \{f_1, \ldots, f_k\}$, each equation $t = t'$ of $\mathcal{E}$ has the form

$$f_i(c_j(x_1, \ldots, x_m), y_1, \ldots, y_n) = b_{i,j} \tag{2}$$

where $f_i \in \Sigma_1$, $c_j \in \Sigma_0$, $b_{i,j} \in \Lambda(\Sigma)$ is a term depending on i and j, $n, m \geq 0$ and the variables $x_1, \ldots, x_m$, $y_1, \ldots, y_n$ are all distinct (left-linear).

We call the elements of Σ_0 *data constructors* and those of Σ_1 *programs*. We say that $\mathcal{E}$ is *complete* if for all $f_i \in \Sigma_1$ and $c_j \in \Sigma_0$, $\mathcal{E}$ contains exactly one equation of the form *(2)*.

Notice that we allow some lambda abstractions and applications to appear on the right-hand-sides of equations of a canonical system but not on the left-hand-sides.

Important examples of data are natural numbers, with constructors *zero* and *succ* and parametric lists with constructors *cons* and *nil*.

In order not to interdict concrete applications of λ-calculus, we assume constants to be integers (the set of integers will be denoted by $\mathbf{N}_\delta = \{0, 1, 2, \ldots, 10, 11, \ldots\}$) and booleans (notation $\mathbf{B}_\delta = \{\mathsf{True}, \mathsf{False}\}$) together with strict elementary functions on such constants, called *δ-operators*; the notion of reduction associated to them (δ-reduction) will be intended without any special notation. Prefix applicative notation will be used for δ-operators. As an example, conditional expressions will be defined by means of the δ-operator ite : $\mathbf{B}_\delta \to \Lambda$ such that ite $\mathsf{True} = \mathbf{K} \equiv \lambda xy.x$ and ite $\mathsf{False} = \mathbf{O} \equiv \lambda xy.y$.

It comes out that we allow an ambiguous representation of natural numbers, e.g. 3 and *succ*(*succ*(*succ zero*)), and *succ* 2, too. The coexistence of such different representations will be clarified in section 3.1.

The following definition imposes some restrictions over right hand sides of canonical systems of equations.

Definition 2. Let $\mathcal{E}$ be a set of equations in the extended λ-calculus $\Lambda(\Sigma)$. We say that $\mathcal{E}$ is *safe* if it is canonical and moreover the following conditions hold for all $t = t' \in \mathcal{E}$:

(i) t' is a $\beta\delta$-normal form;

(ii) $\forall f \in \Sigma_1 . \forall T_1, \ldots, T_n \in \Lambda(\Sigma) . f(T_1, \ldots, T_n)$ occurs in $t' \Rightarrow T_1$ has no initial abstractions;

(iii) $\forall c \in \Sigma_0 . \forall T_1, \ldots, T_n, T_{n+1} \in \Lambda(\Sigma) . c(T_1, \ldots, T_n)T_{n+1}$ never occurs in t';

(iv) $\forall f \in \Sigma_1 . \forall T_1, \ldots, T_n \in \Lambda(\Sigma) . f(T_1, \ldots, T_n)$ occurs in $t' \Rightarrow T_1 \notin \mathbf{N}_\delta$.

Remark. Some of the constraints introduced over canonical systems to make them safe could also be obtained introducing types over the specification language $\Lambda(\Sigma)$. Indeed (ii) and (iii) in definition 2 are implied, in a typed scenario, by the imposition of a basic type on T_1 and $c(T_1, \ldots, T_n)$, respectively. Notice that, since we will interpret $\Lambda(\Sigma)$ in the *pure* λ-calculus (see sect.3), such interpretation will be independent of the choice of a particular type system for the specification language.

As pointed out by one of the referees, the restriction (ii) can be eliminated introducing a new Σ_1 symbol f', a new Σ_0 symbol c', an equation

$$f'(c', x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$$

and replacing all terms of the form $f(T_1, \ldots, T_n)$ in right-hand sides in which T_1 has initial abstractions by $f'(c', T_1, \ldots, T_n)$. This shows that the mentioned restriction does not cause any loss in the expressive power of the language.

Finally, the restriction (iv) is due to technical reasons, only. Also, it does not cause any loss in the expressive power of the language. Indeed, if terms of the form $f(\mathsf{n}, T_2, \ldots, T_n)$ appear in right-hand sides of equations and $\mathsf{n} \in \mathbf{N}_\delta$, we can replace $f(\mathsf{n}, T_2, \ldots, T_n)$ with $f(T, T_2, \ldots, T_n)$, where

$$T \equiv \mathit{zero} \text{ iff } \mathsf{n} = 0, T \equiv \mathit{succ}\ \mathsf{m} \text{ iff } \mathsf{n} = \mathsf{m} + 1.$$

2.1 More on Canonical Systems of Equations

A function definition can be expressed by a canonical set of equations in an extended λ-calculus: let $\Sigma = \Sigma_0 \cup \Sigma_1$ where

$$\Sigma_0 = \{\mathit{zero}, \mathit{succ}, \mathit{nil}, \mathit{cons}\}, \Sigma_1 = \{\mathit{Fac}, \mathit{Map}\};$$

the following declaration is indeed a canonical set of equations in $\Lambda(\Sigma_0 \cup \Sigma_1)$:

$$\begin{array}{rr} \mathit{Fac}(\mathit{zero}) = 1; & \mathit{Fac}(\mathit{succ}(x)) = *(+1x)(\mathit{Fac}(x)); \\ \mathit{Map}(\mathit{nil}, f) = \mathit{nil}; & \mathit{Map}(\mathit{cons}(x, L), f) = \mathit{cons}(fx, \mathit{Map}(L, f)). \end{array}$$

The given set of equations is clearly safe but not complete.

Any pattern of recursion can be manipulated in such a way to be expressed by a canonical set of equations in the extended λ-calculus; as an example, the Ackermann function can be defined by the following set of equations:

$$\begin{aligned} \mathit{Ack}(\mathit{zero}, y) &= +1y; \\ \mathit{Ack}(\mathit{succ}(x), \mathit{zero}) &= \mathit{Ack}(x, 1); \\ \mathit{Ack}(\mathit{succ}(x), \mathit{succ}(y)) &= \mathit{Ack}(x, \mathit{Ack}(+1x, y)). \end{aligned}$$

The above system is not canonical since the last two equations do not have the shape (2), but it is reduced to a canonical system (more precisely, to a safe one) by enlarging the signature with the new function symbol f as follows:

$$\begin{array}{rr} \mathit{Ack}(\mathit{zero}, y) = +1y; & \mathit{Ack}(\mathit{succ}(x), y) = \mathit{Ack}(x, f(y, x)); \\ f(\mathit{zero}, x) = 1 & f(\mathit{succ}(z), x) = \mathit{Ack}(+1x, z). \end{array}$$

To be more general, but still restricting our attention to integer functions let $\Sigma = \Sigma_0 \cup \Sigma_1$, where $\Sigma_0 = \{\mathit{zero}, \mathit{succ}\}$, $\Sigma_1 = \{F\}$; a double integer recursion scheme can be presented by means of the following set of equations, where $h_0, \ldots, h_3 \in \Lambda(\Sigma)$:

$$\begin{array}{rr} F(\mathit{zero}, \mathit{zero}) = h_0; & F(\mathit{zero}, \mathit{succ}(y)) = h_1; \\ F(\mathit{succ}(x), \mathit{zero}) = h_2; & F(\mathit{succ}(x), \mathit{succ}(y)) = h_3. \end{array} \tag{3}$$

The scheme (3) is not a canonical set of equations in $A(\Sigma)$, but it can be easily reduced to a mutual recursion scheme by enlarging the signature Σ with two extra function symbols; the resulting set of equations is a canonical one:

$$\begin{aligned} F(zero, z) &= F'(z); & F(succ(x), z) &= F''(z, x); \\ F'(zero) &= h_0; & F'(succ(y)) &= h_1; \\ F''(zero, x) &= h_2; & F''(succ(y), x) &= h_3. \end{aligned} \tag{4}$$

Such reduction is easily proved to be correct, just considering the four possible cases in (3) and verifying they they are well defined by (4).

Similarly every partial recursive function can be defined by a canonical system (it is enough to verify closure under minimalization, as in [29]). Moreover the correspondence between recursive schemes and canonical systems can be extended to functionals defined over arbitrary algebraic data structures in a straightforward way, as our example of the functional *Map*. Our choice of canonical sets of equations has been made to automatize the execution of *simple pattern matching*, a paradigm used by compilers for functional languages and rewriting systems (see e.g. [25, 2]).

3 Solving equations inside λ-calculus

We have introduced a language which is based on definitions of recursive functions; clearly, a function has an implicitly infinite character.

Historically, solutions of systems of recursive equations are based on the use of fixed point combinators (or similar tools [21, 31]) and yield combinators which make the mentioned infinite character explicit. Such solutions encode all possible unfoldings of functions. Any single datum establishes how many unfoldings of the obtained combinator will be executed, but this, being itself the engine of recursion, is an infinite object from the standpoint of reduction in that it does not have a normal form. To use a slogan, we can say that in this setting, functionals (the fixed point combinator, in particular) are diverging objects which, when applied to data, may "incidentally" converge.

Among the consequences of this, implementations must resort to compute *weak head normal forms* instead of normal forms. Abramsky and Ong introduced the *lazy* λ-calculus [1] to match such implementations.

A different approach is what we propose in this paper, aiming to solve recursive equations without making their infinite character explicit. In our approach, programs are, whenever possible, normal forms; reduction is started only when they receive some input; data become the engine of recursion in that every datum encodes the number of unfoldings it will cause to be executed by any program; hence the infinite characteristics of recursive functions are distributed over an infinity of finite objects. On the other hand, the solution of a set of recursive equations is a combinator encoding exactly the information specified by the equations. To substantiate these ideas we will exhibit an always diverging function represented by a normal form, so that, rephrasing the slogan above, we

can say that in our setting functionals are normal forms which, when applied to data, may "incidentally" diverge.

A *representation* of the signature Σ in the λ-calculus is a function $\phi : \Sigma \to \Lambda$. Any such representation ϕ induces a map $(\cdot)^\phi : \Lambda(\Sigma) \to \Lambda$ in the obvious way, namely

$$\text{for any atomic symbol } a, \quad a^\phi = \begin{cases} \phi(a) & \text{if } a \in \Sigma \\ a & \text{otherwise,} \end{cases}$$

$$(\lambda x.M)^\phi = \lambda x.M^\phi,$$

$$(MN)^\phi = M^\phi N^\phi,$$

$$f(M_1, \ldots, M_n)^\phi = f^\phi M_1^\phi \ldots M_n^\phi.$$

Definition 3. Let $\mathcal{E} = \{a_i = b_i | i \in J\}$ be a set of equations between extended lambda terms $a_i, b_i \in \Lambda(\Sigma)$.

1. We say that a representation ϕ *satisfies* (or *solves*) $\mathcal{E}$ if for each equation $a_i = b_i$ in $\mathcal{E}$ we have $a_i^\phi =_\beta b_i^\phi$. If there exists a representation ϕ which satisfies $\mathcal{E}$ we say that $\mathcal{E}$ can be *interpreted* (or *represented* or *solved*) inside λ-calculus and that ϕ is a *solution* for $\mathcal{E}$.
2. A solution ϕ for $\mathcal{E}$ is called a *normal solution* if, for all h in Σ, $\phi(h)$ is a β-normal form.

Theorem 4 (Interpretation Theorem).
Let $\Lambda(\Sigma)$ be an extended λ-calculus; then every safe set of equations $\mathcal{E}$ has a normal solution $\phi : \Sigma \to \Lambda$ inside λ-calculus. Furthermore we can choose ϕ so that the restriction $\phi|\Sigma_0$ depends only on Σ_0 and not on $\mathcal{E}$, namely there is a fixed representation of the constructors.

Proof. Let $\Sigma_0 = \{c_1, c_2, \ldots, c_r\}$.

For $1 \leq j \leq r$, we define $\vartheta = \phi|\Sigma_0 : \Sigma_0 \to \Lambda$:

$$\vartheta(c_j) = \lambda x_1 \ldots x_m e.e\mathbf{U}_j^r x_1 \ldots x_m, \tag{5}$$

where m is the arity of c_j and $\mathbf{U}_j^r \equiv \lambda x_1 \ldots x_r.x_j$.

It remains to define $\zeta = \phi|\Sigma_1 : \Sigma_1 \to \Lambda$, namely the representation of programs. Without loss of generality we can assume that $\mathcal{E}$ is complete (otherwise adjoin more equations to make it complete).

Let $\Sigma_1 = \{f_1, \ldots, f_k\}$. Consider $k \times r$ lambda terms $t_{i,j}$, $1 \leq i \leq k, 1 \leq j \leq r$ to be defined later. Recall the definition of *Church n-tuple*:

$$\langle M_1, \ldots, M_n \rangle \equiv \lambda x.xM_1 \ldots M_n.$$

For $1 \leq i \leq k$, let $t_i \equiv \langle t_{i,1}, \ldots, t_{i,r} \rangle$ and define

$$\zeta(f_i) \equiv \langle t_i, t_1, t_2, \ldots, t_k \rangle.$$

Thus $\zeta(f_i)$ is a Church $k+1$-tuple of Church r-tuples of terms. The lambda terms $t_{i,j}$ are chosen in the only natural way which makes ζ a solution of the canonical system of equations $\mathcal{E}$. More precisely consider the equation

$$f_i(c_j(x_1, \ldots, x_m), y_1, \ldots, y_n) = b_{i,j}$$

belonging to $\mathcal{E}$ ($b_{i,j} \in \Lambda(\Sigma)$). After applying $\phi = \vartheta \circ \zeta$ the equation becomes

$$\langle t_i, t_1, \ldots, t_k \rangle (c_j^\phi x_1 \ldots x_m) y_1 \ldots y_n = b_{i,j}^\phi.$$

By definition of Church tuple, this simplifies to

$$c_j^\phi x_1 \ldots x_m t_i t_1 \ldots t_k y_1 \ldots y_n = b_{i,j}^\phi.$$

Recalling the definition of c_j^ϕ we have

$$c_j^\phi x_1 \ldots x_m t_i = t_i \mathbf{U}_j^r x_1 \ldots x_m = t_{i,j} x_1 \ldots x_m.$$

Hence the equation becomes

$$t_{i,j} x_1 \ldots x_m t_1 \ldots t_k y_1 \ldots y_n = b_{i,j}^\phi.$$

We can now solve this equation for $t_{i,j}$ by replacing on both sides all the occurrences of $t_1, \ldots, t_k$ by fresh variables $v_1, \ldots, v_k$ and abstracting with respect to all variables present in left-hand-side. More precisely define:

$$t_{i,j} \equiv \lambda x_1 \ldots x_m v_1 \ldots v_k y_1 \ldots y_n . (b_{i,j}^\vartheta)^\psi$$

where $\psi : \Sigma_1 \to \Lambda$ is defined by

$$\psi(f_i) = \langle v_i, v_1, \ldots, v_k \rangle. \tag{6}$$

Note that, for any $V \in \Lambda(\Sigma)$, $V^\zeta = V^\psi[t_h/v_h]_{1 \le h \le k}$.

With this definition

$$\begin{aligned} t_{i,j} x_1 \ldots x_m t_1 \ldots t_k y_1 \ldots y_n &\to (b_{i,j}^\vartheta)^\psi [t_h/v_h]_{1 \le h \le k} \\ &= b_{i,j}^{\vartheta \circ \zeta} = b_{i,j}^\phi \end{aligned}$$

and all the equations will be satisfied.

We now prove that the given technique yields normal solutions for safe systems of equations.

Let $\mathcal{E}$ be safe and let $\vartheta : \Sigma_0 \to \Lambda$ and $\psi : \Sigma_1 \to \Lambda$ be as in (5) and (6), respectively.

For any equation $a = b \in \mathcal{E}$, we first prove by induction on the number of occurrences of constructors in b that b^ϑ is a normal form. Indeed, if constructors do not appear in b, then $b^\vartheta = b$, a normal form (by definition 2.i); if $c_j(X_1, \ldots, X_m)$ occurs in b, for some $X_1, \ldots, X_m \in \Lambda(\Sigma)$, then

$$c_j(X_1, \ldots, X_m)^\vartheta = \lambda e . e \mathbf{U}_j^r X_1^\vartheta \ldots X_m^\vartheta$$

which is, by inductive hypothesis, a normal form and, by 2.iii, does not create any new redex.

It comes out from 2.ii that for any equation $a = b \in \mathcal{E}$, b^ϑ is such that $\forall f \in \Sigma_1 . \forall T_1, \ldots, T_n \in \Lambda(\Sigma)$ if $f(T_1 \ldots T_n)$ occurs in b^ϑ then T_1 either has no initial abstractions or it has the shape $\lambda e . e \mathbf{U}_j^r X_1 \ldots X_n$, for some $X_1, \ldots, X_n \in$

$\Lambda(\Sigma)$. Now, if programs do not appear in b^ϑ, then $(b^\vartheta)^\psi = b^\vartheta$, a normal form; if $f_i(T_1, \ldots, T_n)$ occurs in b^ϑ, for some $T_1, \ldots, T_n \in \Lambda(\Sigma)$, then

$$f_i(T_1, \ldots, T_n)^\psi = T_1^\psi v_i v_1 \ldots v_k T_2^\psi \ldots T_n^\psi$$

which reduces in at most one step to a head normal form without any initial abstraction. It follows by induction that $(b^\vartheta)^\psi$ reduces to a normal form, so that, applying to such normal form the construction in the first part of the proof of the theorem, we obtain a normal solution for $\mathcal{E}$.

Example 1. Given $\Sigma = \Sigma_0 \cup \Sigma_1$ where

$$\Sigma_0 = \{zero, succ, nil, cons\} \text{ and } \Sigma_1 = \{Fac, Map\},$$

let $\mathcal{E}$ be the following set of equations:

$$\begin{aligned} Fac(zero) &= 1; \\ Fac(succ(x)) &= *(+1x)(Fac(x)); \\ Map(nil, f) &= nil; \\ Map(cons(x, L), f) &= cons(fx, Map(L, f)). \end{aligned}$$

We can complete $\mathcal{E}$ adding the equations

$$\begin{aligned} Fac(nil) &= Type\_err_1; \\ Fac(cons(x, L)) &= Type\_err_1; \\ Map(zero, f) &= Type\_err_2; \\ Map(succ(x), f) &= Type\_err_2, \end{aligned}$$

where $Type\_err_1$ and $Type\_err_2$ are two variables.

If we assume

$$\begin{aligned} &\phi(zero) = \lambda e.e\mathbf{U}_1^4 \ , \ \phi(succ) = \lambda xe.e\mathbf{U}_2^4 x, \\ &\phi(nil) = \lambda e.e\mathbf{U}_3^4 \ , \ \phi(cons) = \lambda xLe.e\mathbf{U}_4^4 xL, \\ &\phi(Fac) = \langle t_1, t_1, t_2 \rangle, \text{ where } t_1 = \langle t_{1,1}, \ldots, t_{1,4} \rangle \\ &\phi(Map) = \langle t_2, t_1, t_2 \rangle, \text{ where } t_2 = \langle t_{2,1}, \ldots, t_{2,4} \rangle \end{aligned}$$

and we consider the derived set of equations, we obtain

$$\begin{aligned} t_1 \mathbf{U}_1^4 \, t_1 \, t_2 &= 1; \\ t_1 \mathbf{U}_2^4 \, x \, t_1 \, t_2 &= *(+1x)(x \, t_1 \, t_1 \, t_2); \\ t_1 \mathbf{U}_3^4 \, t_1 \, t_2 = t_1 \mathbf{U}_4^4 \, x \, L \, t_1 \, t_2 &= Type\_err_1; \\ t_2 \mathbf{U}_1^4 \, t_1 \, t_2 f = t_2 \mathbf{U}_2^4 \, x \, t_1 \, t_2 f &= Type\_err_2; \\ t_2 \mathbf{U}_3^4 \, t_1 \, t_2 f &= \phi(nil); \\ t_2 \mathbf{U}_4^4 \, x \, L \, t_1 \, t_2 f &= \phi(cons) \, (fx) \, (L \, t_2 \, t_1 \, t_2 f); \end{aligned}$$

hence, we have

$$\begin{aligned} t_{1,1}\, t_1\, t_2 &= 1; \\ t_{1,2}\, x\, t_1\, t_2 &= *(+1x)(x\, t_1\, t_1\, t_2); \\ t_{1,3}\, t_1\, t_2 = t_{1,4}\, x\, L\, t_1\, t_2 &= Type\_err_1; \\ t_{2,1}\, t_1\, t_2 f = t_{2,2}\, x\, t_1\, t_2 f &= Type\_err_2; \\ t_{2,3}\, t_1\, t_2 f &= \phi(nil); \\ t_{2,4}\, x L\, t_1\, t_2 f &= \phi(cons)\,(fx)(L\, t_2\, t_1\, t_2 f); \end{aligned}$$

which is solved taking

$$\begin{aligned} &t_{1,1} \equiv \lambda v_1 v_2.1 \ , \ t_{1,2} \equiv \lambda x v_1 v_2.*(+1x)(x\, v_1\, v_1\, v_2), \\ &t_{1,3} \equiv \lambda v_1 v_2.Type\_err_1 \ , \ t_{1,4} \equiv \lambda x_1 x_2 v_1 v_2.Type\_err_1, \\ &t_{2,1} \equiv \lambda v_1 v_2 f.Type\_err_2 \ , \ t_{2,2} \equiv \lambda x v_1 v_2 f.Type\_err_2, \\ &t_{2,3} \equiv \lambda v_1 v_2 f.\phi(nil), \\ &t_{2,4} \equiv \lambda x_1 x_2 v_1 v_2 f.\phi(cons)\,(fx_1)(x_2\, v_2\, v_1\, v_2 f). \end{aligned}$$

It follows that the representation for Fac and Map is a Church 3-tuple of Church 4-tuples of normal forms, hence a normal form.

Remark. It has to be noted that we obtain normal solutions also for definitions of intrinsically diverging programs. Given $\Sigma = \Sigma_0 \cup \Sigma_1$ where

$$\Sigma_0 = \{zero, succ\} \text{ and } \Sigma_1 = \{F\},$$

let $\mathcal{E}$ be the following set of equations:

$$\begin{aligned} F(zero) &= F(zero); \\ F(succ(x)) &= F(succ(x)), \end{aligned} \tag{7}$$

First observe that such system is safe, in spite of the fact that it defines an always diverging function. If we assume

$$\begin{aligned} &\phi(zero) = \lambda e.e\mathbf{U}_1^2 \ , \ \phi(succ) = \lambda x e.e\mathbf{U}_2^2 x, \\ &\phi(F) = \langle t_1, t_1 \rangle, \ \text{ where } t_1 = \langle t_{1,1}, t_{1,2} \rangle \end{aligned}$$

and we consider the derived set of equations, we obtain

$$\begin{aligned} t_1 \mathbf{U}_1^2\, t_1 &= t_1 \mathbf{U}_1^2\, t_1; \\ t_1 \mathbf{U}_2^2\, x\, t_1 &= t_1 \mathbf{U}_2^2\, x\, t_1; \end{aligned}$$

hence, we have

$$\begin{aligned} t_{1,1}\, t_1 &= t_1 \mathbf{U}_1^2\, t_1; \\ t_{1,2}\, x\, t_1 &= t_1 \mathbf{U}_2^2\, x\, t_1; \end{aligned}$$

which is solved taking

$$t_{1,1} \equiv \lambda v_1.v_1 \mathbf{U}_1^2\, v_1 \ , \ t_{1,2} \equiv \lambda x v_1.v_1 \mathbf{U}_2^2\, x\, v_1.$$

It follows that the representation for F is a Church 2-tuple of Church 2-tuples of normal forms, hence a normal form.

3.1 A double citizenship for data

The Interpretation Theorem 4 enables to obtain normal solutions for safe systems of equations.

The key idea allowing such result is that data are considered as functionals, interpreting them in λ-calculus. Now, a natural question arises: are we willing to fully represent data structures in λ-calculus? The answer is surely negative, for we want to preserve the structure of data during computation. Roughly speaking, we would like to give data a double citizenship: they should act as functionals during function application, otherwise preserving their original status.

The Interpretation Theorem itself hints to a satisfactory solution to this problem: being the representation of data constructors fixed, we will consider them as predefined combinators, i.e. extra constants to be included in Λ; furthermore, we will consider their functional behaviour to be defined by *weak reduction rules*, in such a way that a constructor with arity m needs (by the weak version of (5)) at least $m+1$ arguments to be reduced. Assuming then $\{c_1, \ldots, c_r\} \in \Lambda$ with

$$c_j X_1 \ldots X_m E \rhd E\mathbf{U}_j^r X_1 \ldots X_m,$$

where '$\rhd$' denotes weak reduction, it turns out that all definitions and results obtained in the previous subsections still hold taking now $\Sigma \equiv \Sigma_1$, since $\Sigma_0 = \emptyset$, being the constructors considered constants in Λ.

Finally, the coexistence of different representations for natural numbers is ruled by the following notion of reduction (χ-reduction) which allows rewriting δ-integers when these appear in functional position in an application:

$$\frac{0 \in \mathbf{N}_\delta \quad T \in \Lambda}{0\,T \longrightarrow_\chi zero^\phi\, T} \quad , \quad \frac{\mathsf{n} = \mathsf{m}{+}1 \in \mathbf{N}_\delta \quad T \in \Lambda}{\mathsf{n}\,T \longrightarrow_\chi succ^\phi\, \mathsf{m}\,T} \; .$$

Example 2. (Ex.1 continued)

To give the example of a computation, let $\phi(Map)$ be as in example 1. We have e.g. (superscript ϕ is sometimes omitted below)

$$\begin{aligned}
& (Map(cons(1, cons(2, nil)), f))^\phi \\
= \;& cons\,1(cons\,2\;nil)t_2\,t_1\,t_2\,f \\
\rhd \;& t_2\mathbf{U}_4^4\,1(cons\,2\;nil)t_1\,t_2\,f \\
\longrightarrow_\beta \;& t_{2,4}\,1(cons\,2\;nil)t_1\,t_2\,f \\
\longrightarrow_\beta \;& cons(f1)(cons\,2\;nil\,t_2\,t_1\,t_2 f) \\
\rhd \;& \cdots \longrightarrow_\beta cons(f1)(cons(f2)(nil\,t_2\,t_1\,t_2 f)) \\
\rhd \;& cons(f1)(cons(f2)(t_2\mathbf{U}_3^4\,t_1\,t_2 f)) \\
\longrightarrow_\beta \;& cons(f1)(cons(f2)nil), \text{ a } \rhd\text{-normal form.}
\end{aligned}$$

On the other hand, we have:

$$\begin{aligned}
& (Map(7, f))^\phi \\
= \;& 7\,t_2\,t_1\,t_2\,f \longrightarrow_\chi succ\,6\,t_2\,t_1\,t_2\,f \\
\rhd \;& t_2\mathbf{U}_2^4\,6\,t_1\,t_2\,f \longrightarrow_\beta t_{2,2}\,6\,t_1\,t_2\,f \\
\longrightarrow_\beta \;& Type_err_2.
\end{aligned}$$

Summarizing, $\triangleright$-normal forms are important tools to recognize algebraic objects as results of computations. This solves a problem mentioned in [5], in the context of self-interpretation of λ-calculus.

4 Further Properties: some examples

This section is devoted to the illustration of possible applications in functional programming of the theoretical issues just presented.

The following examples refer to recursive definitions *(D)* of function(al)s, execution commands *(E)* and results *(R)* of computations based on the implementation of the methods described in this paper, called *CuCh-machine*. This is an acronym for Curry and Church, first introduced in [12, 13] to describe a machine simultaneously accepting combinators and λ-terms and reducing them to normal form. Further properties not examplified below are the allowance of free variables, and the use of lazy data structures, in the style of [18], implemented by normal order reduction (to be compared with [20]).

4.1 Currification

(D) `ACK zero f := f 1;`
(D) `ACK (succ m)f := f (ACK m f);`
(D) `ack zero x := + 1 x;`
(D) `ack (succ n) x := ACK x (ack n);`

Currified `ack`
(E) `ack3 :=ack 3;`
(R) `λx0 . ACK x0 (λx1 . ACK x1 (λx2 . ACK x2 (λx3 . + 1 x3)))` `(3 beta)`
(E) `ack34:= ack3 4;`
(R) `125 (15520 beta)`
(E) `ack35:= ack3 5;`
(R) `253 (63780 beta)`

Non currified `ack`
(E) `r:= ack 3 4;`
(R) `125 (15640 beta)`
(E) `s:= ack 3 5;`
(R) `253 (64027 beta)`

4.2 Iterative Functions

(see [19])
(D) `map := λx0 x1 x2 x3 . x1 (λx4 x5 . x2 (x0 x4) x5) x3;`
(E) `mapmap := λx . map f(map g x);`
(R) `λx0 x1 x2 . x0 (λx3 x4 . x1 (f (g x3)) x4) x2`
(E) `comp := λx . map (B f g) x;`
(R) `λx0 x1 x2 . x0 (λx3 x4 . x1 (f (g x3)) x4) x2`
(D) `foldr nil a b := b;`
(D) `foldr (cons x L) a b := a x(foldr L a b);`
(D) `list := [1,4,5,6,7];`

```
(E) flist := foldr list;
(R) λx0 x1 . x0 1 (x0 4 (x0 5 (x0 6 (x0 7 x1))))
(E) bb := mapmap flist cons nil;
(R) [ f (g 1), f (g 4), f (g 5), f (g 6), f (g 7) ]
```

4.3 Complete Systems

```
(D) mbf zero x :=λy.mbf y x;
(D) mbf (succ n) x:=λy.mbf y (+ x(+ 1 n));
(D) mbf nil x:= x;
(D) mbf(cons t l)x:= mbf l (+ t x);
(E) ee:= mbf [1, 3] 0;
(R) 4
(E) xy:= mbf 7 6 4 nil;
(R) 17
```

Acknowledgments

We would like to thank Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro for helpful discussions and suggestions about the topics of this paper. We are grateful to the referees of the preliminary version of the paper for their criticism and suggestions for improving the presentation.

References

1. S. Abramsky, C.-H.L. Ong, *Full Abstraction in the Lazy Lambda Calculus*, Technical Report 259, Cambridge University Computer Laboratory, 1992, 105 pp. To appear in *Info. and Comp.*
2. L.Augustsson and T.Johnsson, *The Chalmers Lazy-ML Compiler*, The Computer Journal, vol. 32, no. 2, April 1989.
3. J.Backus, *Can programming be liberated from vonNeumann style? A functional style and its algebra of programs*, ACM Comm.,1978, vol.21, no. 8, pp. 613-641.
4. H.P.Barendregt, *The type free lambda-calculus*, in: Handbook of Mathematical Logic, Barwise (ed.), North Holland, 1981, pp.1092-1132.
5. A.Berarducci and C.Böhm, *A self-interpreter of lambda calculus having a normal form*, 6th Workshop CSL '92, San Miniato, Italy, September-October 1992, eds E. Börger et al., Springer Verlag, Berlin (LNCS 702), pp. 85–99.
6. C.Böhm, *Combinatory foundation of functional programming*, in 1982 ACM Symposium on Lisp and functional programming, 1982, Pittsburgh, Pen., pp.29-36.
7. C. Böhm, *Reducing Recursion to Iteration by Algebraic Extension* in: ESOP 86, (LNCS 213), p.111-118, 1986.
8. C. Böhm, *Reducing Recursion to Iteration by means of Pairs and N-tuples*, in: Foundations of Logic and Functional Programming, LNCS 306, p.58-66, 1988.
9. C.Böhm, *Functional Programming and Combinatory Algebras, MFCS*, Carlsbad, August-September 1988, eds M. P. Chytil et al., Springer Verlag, Berlin (LNCS 324), pp. 14–26.
10. C.Böhm, *Subduing Self-Application*, ICALP '89, Stresa, July 11–15 1989, eds G. Ausiello et al., Springer Verlag, Berlin (LNCS 372), pp. 108–122.

11. C.Böhm and A.Berarducci, *Automatic Synthesis of Typed λ-Programs on Term Algebras*, Theoretical Computer Science 39, pp. 135–154, 1985.
12. C.Böhm and M.Dezani-Ciancaglini, *A CUCH-machine: the automatic treatment of bound variables*, International Journal of Computer and Information Sciences, vol. 1, no. 2, pp. 171–191, June 1972.
13. C.Böhm and M.Dezani-Ciancaglini, *Notes on "A CUCH-machine: the automatic treatment of bound variables"*, International Journal of Computer and Information Sciences, vol. 2, no. 2, pp. 157–160, June 1973.
14. C.Böhm and M.Dezani-Ciancaglini, *Combinatorial problems,combinator equations and normal forms*, in: Loeckx (ed.) Automata, Languages and Programming 2th. Colloquium, LNCS 14, 1974, pp.185-199.
15. C.Böhm and M.Dezani-Ciancaglini, *λ-terms as total or partial functions on normal forms*, in: λ-Calculus an computer science theory Böhm (ed.), LNCS 37, Springer, 1975, pp.96-121.
16. A.Church, *The calculi of lambda-conversion*, Princeton Univ.Press, 1941.
17. H.B.Curry, *Combinatory Logic*, Vol I, North Holland, Amsterdam, 1958.
18. D.P.Friedman and D.S.Wise, *Cons should not evaluate its arguments*, Proc.3rd International Colloquium on Automata, Languages and Programming, Edinburgh, 1976, pp.257–284.
19. A.Gill, J.Launchbury and S.L.Peyton-Jones, *A Short Cut to Deforestation*, Functional Programming and Computer Architecture, 1993.
20. J.Hughes, *Why Functional Programming Matters*, The Computer Journal, special issue on Lazy Functional Programming, vol. 32, no. 2, April 1989.
21. J.Hughes, *Supercombinators: a new implementation method for applicative languages*, Symp. on LISP and Functional Programming, ACM, 1982.
22. S.C.Kleene, *λ-definability and recursiveness*, Duke Math.J. 2, pp.340-353.
23. D.E.Knuth, *The Art of Computer Programming*, Vol. 1/Fundamental Algorithms, Addison-Wesley, 1973.
24. M. Parigot. *Programming with proofs: a second order type theory*, ESOP'88, LNCS 300, pp. 145-159.
25. S.L.Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1986.
26. H.Schwichtenberg, *Einige Anwendungen von unendlichen Termen und Wertfunktionalen*, Habilitationsschrift, Münster, 67 pp., 1973.
27. J.R.Shoenfield, *Matematical Logic*, Addison Wesley, 1967.
28. H.R. Strong, *Algebraically Generalized Recursive Function Theory*, IBM J.Res. Develop.12 (1968), pp.465-475.
29. E.Tronci, *Equational programming in lambda-calculus*, Proc.of LICS'91, IEEE Comp.Soc., 1991.
30. A.Turing, *On computable numbers with an application to the Entscheidungsproblem*, Proc.London Math.Soc. 42, pp.230-265.
31. D.A.Turner, *A new implementation technique for applicative languages*, Software practice and experience, no. 9, 1979.
32. E.G.Wagner, *Uniformly Reflexive Structures: An Axiomatic Approach to Computability*, Information Sci.1 (1969), pp.343-362.