A COMPOSITIVE ABSTRACTION ALGORITHM FOR COMBINATORY LOGIC*

Adolfo PIPERNO

Dipartimento di Matematica
Istituto "G.Castelnuovo"
Universita' degli Studi di Roma "La Sapienza"
P.le Aldo Moro 5, I-00185 ROMA, Italy.**

Abstract.

The problem of the translation of λ-terms into combinators (bracket abstraction) is of great importance for the implementation of functional languages. In the literature there exist a lot of algorithms concerning this topic, each of which is based on a particular choice of a combinatory basis, of its cardinality, of an abstraction technique.

The algorithm presented here originated from a modification of the definition of abstraction given by Curry in 1930, and has the following interesting properties:

i) it employs a potentially infinite basis of combinators, each of which depends on at most two parameters and is, therefore, directly implementable;

ii) it gives compact code, introducing a number of basic combinators which is proportional to the size of the expression to be abstracted and invariant for one and multi-sweep abstraction techniques;

iii) it gives the result in the form $\mathbf{R}\,\mathbf{I}\,M_1 \ldots M_n$, where $\mathbf{R}$ is a regular combinator expressed as a composition of basic combinators, $\mathbf{I}$ is the identity combinator, and $M_1, \ldots, M_n$ are the constant terms appearing into the expression subjected to the translation process.

It comes out that a slight modification of the algorithm yields a combinatory equivalent of Hughes' supercombinators.

Keywords: Functional programming, Compiler design, Evaluation techniques.

0. Introduction.

We assume the reader to be familiar with the basic definitions and properties of λ-calculus and theory of combinators not explicitly cited in this paper; for a complete treatment of them see e.g. [Bar] and [CurFe].

In particular:

- λ-terms will be untyped λ-β-η-terms, possibly containing constants;
- the word combinator will indicate a closed λ-term, which will be denoted by a boldface character; we will assume that the correspondence between λ-calculus and combinatory logic has already been defined and adopt λ-notation

*) This research has been supported by grants of the Ministry of Public Instruction, Italy.
**) This work originated as the author's thesis in Mathematics (supervisor Prof. C.Böhm) by such an affiliation.

to indicate the functional behaviour of a combinator;

- the problem of abstraction in combinatory logic (CL), which is the central issue of the process of translation from λ-calculus to CL, is introduced in the following way:

let be given any CL-term U and the variables $x_1,\dots,x_n$ $(n \geq 1)$; the *abstract* of U with respect to $x_1,\dots,x_n$ (denoted $[x_1,\dots,x_n]$ U) is a CL-term F such that:

i) x_i does not occur in F $(1 \leq i \leq n)$;

ii) $Fx_1\dots x_n \geq U$. ($Fx_1\dots x_n$ *reduces to* U).

The abstraction is a correspondence between CL-terms; it is constructively defined by means of an algorithm, which can be viewed as a proof for combinatory completeness; related algorithms can be classified in two branches:

- *multi-sweep* algorithms, which repeatedly operate on one variable at a time, in the following way: $[x_1,\dots,x_n]\ U =_{def} [x_1]([x_2](\dots([x_n]\ U)\dots))$;

- *one-sweep* algorithms, which operate simultaneously on all the variables $x_1,\dots,x_n$; we will use the notation $[\vec{x}_n]$ U to indicate that the operation $[x_1,\dots,x_n]$ U has to be performed by such an algorithm.

The interest in the abstraction algorithms of combinatory logic comes from computer science: in the last few years many so-called functional languages have been proposed, together with special techniques for their implementation. Some of these, starting with the one by Turner [Tur79a,b], make use of an algorithm of translation from λ-calculus to combinatory logic in order to achieve the compilation of a program. Functional languages are in effect enriched versions of λ-calculus, and the combinatory code obtained from the translation process is more easily executable by a computer than performing β-reductions.

Such an approach is of practical interest if the size of the resulting code is not too large; an exhaustive treatment about the complexity of the algorithms existing in the literature can be found in [Mul].

A different implementation technique for functional languages is the one suggested by Hughes [Hug]; the compilation process makes use of the so-called *supercombinators*; these are introduced as a generalisation of Turner's combinators avoiding the growth of the compiled code; however they need to be interpreted somehow, while Turner's ones are directly implementable.

The main inspiration for the present research comes from the comparison between the two schemes described above, the motivation being to find a combinatory equivalent of supercombinators, i.e. a combinatory basis and an abstraction algorithm operating on it in order to give a combinatory interpretation of them.

The resulting algorithm derives from the sources of combinatory logic; in fact the required basis comes from a modification of the one introduced by Curry in 1930-32 [Cur30,Cur32], and the algorithm comes from a modification of the definition of abstraction given by Curry in 1933 [Cur33].

In sections 1,2 we will introduce the set of combinators employed by the algorithm as a compositive basis for regular combinators; in section 3 we will give an intuitive description of the abstraction process; this will be modified and formalized in sections 4 and 5.1,2, with the definitions of two versions of the final algorithm, corresponding to one and multi-sweep techniques; complexity problems will be analysed in section 6.

1. Basic combinators.

We will first recall the definitions of some fundamental combinators:

$\mathbf{I} \equiv \lambda x.x$	the elementary identificator;
$\mathbf{K} \equiv \lambda fx.f$	the elementary cancellator;
$\mathbf{W} \equiv \lambda fx.fxx$	the elementary duplicator;
$\mathbf{C} \equiv \lambda fxy.fyx$	the elementary permutator;
$\mathbf{B} \equiv \lambda fgx.f(gx)$	the elementary compositor (infix: $f \circ g \equiv \mathbf{B}fg$).

We will now give a classification of CL-terms which is useful to introduce the combinatory basis employed in the abstraction algorithm (see [Sta] for an interesting treatment about basis problems in CL):

Definition 1.1:

A CL-term T is said to be *pure* if it is a combination of variables only.

Definition 1.2:

A combinator $\mathbf{T}$ is said to be *proper* if $\mathbf{T} \equiv \lambda x_0 \ldots x_n.T$, where T is a pure combination of the variables $x_0,\ldots,x_n$.

Definition 1.3:

A proper combinator $\mathbf{U}$ is said to be *regular* if $\mathbf{U} \equiv \lambda x_0 \ldots x_n.x_0 U_1 \ldots U_m$, where $U_1,\ldots,U_m$ are pure combinations of $x_1,\ldots,x_n$.

Let us consider the set $\mathbf{R}$ of all regular combinators; it is easy to verify that:

(i) $\mathbf{R}$ constitutes a monoid with respect to the operation ($\circ$) of composition;

(ii) considering the elements of $\mathbf{R}$ as operators acting on variables, $\mathbf{R}_1 \circ \mathbf{R}_2$ is obtained operating first with $\mathbf{R}_1$ and secondly with $\mathbf{R}_2$ ($\mathbf{R}_1$, $\mathbf{R}_2 \in \mathbf{R}$).

We describe some sequences of regular combinators, i.e. parametric subsets of $\mathbf{R}$, in order to specify a subset $\mathbf{B} \subset \mathbf{R}$ such that every $\mathbf{Z} \in \mathbf{R}$ can be expressed as a composition of elements belonging to $\mathbf{B}$.

$\mathbf{B}$ will be called *compositive basis* for regular combinators.

The sequences are given together with their inductive definition:

Identificators: $\mathbf{I}_n \equiv \lambda x_0 x_1 \ldots x_n . x_0 x_1 \ldots x_n$ $(n \geq 0)$;

Def. I1: $\mathbf{I}_0 = \mathbf{I}$; $\mathbf{I}_{t+1} = \mathbf{B}\mathbf{I}_t$;

Cancellators: $\mathbf{K}_n \equiv \lambda x_0 x_1 \ldots x_n . x_0 x_1 \ldots x_{n-1}$ $(n \geq 1)$;

Def. K1: $\mathbf{K}_1 = \mathbf{K}$; $\mathbf{K}_{t+1} = \mathbf{B}\mathbf{K}_t$;

Multiplicators: $\mathbf{W}_{n,r} \equiv \lambda x_0 x_1 \ldots x_n . x_0 x_1 \ldots x_{n-1} x_n \ldots$ (r+1 times) $\ldots x_n$ $(n \geq 1, r \geq 0)$;

Def. W1: $\mathbf{W}_1 = \mathbf{W}$; $\mathbf{W}_{t+1} = \mathbf{B}\mathbf{W}_t$;

Def. W2: $\mathbf{W}_{n,1} = \mathbf{W}_n$; $\mathbf{W}_{n,s+1} = \mathbf{W}_n \circ \mathbf{W}_{n,s}$;

Def. W3: $\mathbf{W}_{n,0} = \mathbf{I}_n$;

Permutators: $\mathbf{C}_{m+n,m} \equiv \lambda x_0 x_1 \ldots x_m \ldots x_{m+n-1} x_{m+n} \,.\, x_0 x_1 \ldots x_{m+n} x_m \ldots x_{m+n-1}$
$(m \geq 1,\ n \geq 0)$;

Def. C1: $\mathbf{C}_1 = \mathbf{C}$; $\mathbf{C}_{t+1} = \mathbf{BC}_t$;

Def. C2: $\mathbf{C}_{n,1} = \mathbf{C}_n$; $\mathbf{C}_{n+k+1,n} = \mathbf{C}_{n+k+1,n+k} \circ \mathbf{C}_{n+k,n}$;

Def. C3: $\mathbf{C}_{n,n} = \mathbf{I}_n$;

Compositors: $\mathbf{B}_n\mathbf{B}_m \equiv \lambda x_0 x_1 \ldots x_m \ldots x_{m+n+1} \,.\, x_0 x_1 \ldots x_m (x_{m+1} \ldots x_{m+n+1})$
$(m \geq 0,\ n > 0)$;

Def. B1: $\mathbf{B}_0 = \mathbf{I}$; $\mathbf{B}_{t+1} = \mathbf{B} \circ \mathbf{B}_t$;

Def. B2: $\mathbf{B}_0\mathbf{B}_t = \mathbf{B}_t$; $\mathbf{B}_{s+1}\mathbf{B}_t = \mathbf{B}(\mathbf{B}_s\mathbf{B}_t)$.

Note that the permutators cited above are different from those introduced by Curry in [Cur30] and [Cur32]; this choice is due to complexity reasons, and will be motivated in section 6.

In the next section we will give a theorem of existence and uniqueness to prove that the sequences of combinators introduced above effectively constitute a compositive basis for **R**; the theorem was proved by Curry in [Cur30], [Cur32]; the complete proof involving the different permutators can be found in [Ptes].

2. Compositive normal form.

Notation: from this point onwards we will use Gothic characters to indicate compositions of basic combinators; in particular:

$\mathfrak{K}$: any composition of cancellators or identificators;
$\mathfrak{W}$: any composition of multiplicators;
$\mathfrak{C}$: any composition of permutators;
$\mathfrak{B}$: any composition of compositors or identificators.

Definition 2.1: (Compositive form of regular combinators)
A regular combinator **R** is said to be in *compositive form* (CF) if it is expressed as a composition of regular combinators (called *atoms* of **R**).

Definition 2.2: (Compositive normal form of regular combinators)
A regular combinator **R** is said to be in *compositive normal form* (CNF) if it is expressed in the form (2.2.0) $\mathfrak{K} \circ \mathfrak{W} \circ \mathfrak{C} \circ \mathfrak{B}$,
where $\mathfrak{K}$, $\mathfrak{W}$, $\mathfrak{C}$, $\mathfrak{B}$ (called *components* of **R**) are respectively:

2.2.1 $\mathfrak{K} \equiv \mathbf{I}_h$ $(h=1,2\ldots)$ or $\mathfrak{K} \equiv \mathbf{K}_{h_r} \circ \mathbf{K}_{h_{r-1}} \circ \ldots \circ \mathbf{K}_{h_1}$, where $h_1 < h_2 < \ldots < h_r$;

2.2.2 $\mathfrak{W} \equiv \mathbf{W}_{h_r,k_r} \circ \mathbf{W}_{h_{r-1},k_{r-1}} \circ \ldots \circ \mathbf{W}_{h_1,k_1}$, where $h_1 < h_2 < \ldots < h_r$;

2.2.3 $\mathfrak{C} \equiv \mathbf{C}_{1+a_1,1} \circ \mathbf{C}_{2+a_2,2} \circ \ldots \circ \mathbf{C}_{n+a_n,n}$, where $a_i \geq 0$, $1 \leq i \leq n$;

2.2.4 $\mathfrak{B} \equiv \mathbf{I}_h$ $(h=1,2\ldots)$ or $\mathfrak{B} \equiv \mathbf{B}_{m_q}\mathbf{B}_{n_q} \circ \mathbf{B}_{m_{q-1}}\mathbf{B}_{n_{q-1}} \circ \ldots \circ \mathbf{B}_{m_1}\mathbf{B}_{n_1}$,
where $m_k \geq 0$, $n_k > 0$, $1 \leq k \leq q$, and $m_q > m_{q-1} > \ldots > m_1$.

Convention: we will omit the combinators $\mathbf{I}_h$ $(h=1,2\ldots)$ in a compositive form different from an identificator.

Definition 2.3: (Principal CNF of regular combinators)

A regular combinator $\mathbf{R}$ is said to be in *principal compositive normal form* (PCNF) if it is in CNF and $\mathfrak{C}$ corresponds to a permutation which does not interchange variables having the same name.

Example: We consider the combinators:

$\mathfrak{X} \equiv \mathbf{W}_{2,1} \circ \mathbf{C}_{2,1} \circ \mathbf{B}_1\mathbf{B}_1$ and $\mathfrak{Z} \equiv \mathbf{W}_{2,1} \circ \mathbf{C}_{3,1} \circ \mathbf{B}_1\mathbf{B}_1$; it is easy to verify that both $\mathfrak{X}$ and $\mathfrak{Z}$ represent the combinator $\mathbf{S}$ ($\equiv \lambda xyz.xz(yz)$), but only $\mathfrak{X}$ is in PCNF.

Theorem 2.4: (Existence and uniqueness theorem)

Let $\mathbf{R}$ be a regular combinator; there exists one and only one combinator $\mathfrak{R}$ in PCNF such that $\mathfrak{R}$ is extensionally equal to $\mathbf{R}$; we will call $\mathfrak{R}$ the PNCF of $\mathbf{R}$.

Corollary 2.5: (PCNF of proper combinators)

Let $\mathbf{P}$ be a proper combinator; there exists one and only one regular combinator $\mathfrak{R}$ in PCNF such that $\mathfrak{R}\,\mathbf{I}$ is extensionally equal to $\mathbf{P}$.

The difference between the known concept of normal form of a combinator (i.e. a term constituted only by constants and not containing any redex as a subterm) and the one described above will become more evident in the next section, where we will define the notion of normal representation of a CL-term. In the world of regular combinators however the concept of PCNF shares some similarity with that of strong normal form, and it is possible to define an algorithm which, given a regular combinator $\mathfrak{R}$ expressed as a composition of basic combinators, yields the PCNF of $\mathfrak{R}$, repeatedly applying some schemes of tranformation rules [Ptes]; by corollary 2.5 this algorithm can be extended to the set of proper combinators.

3. Normal representation of a CL-term.

In this section we will extend the notion of PCNF to the whole set of CL-terms, in order to express any term as a function of a predetermined sequence of variables $x_1,\ldots,x_n$ (even not occurring in the considered term). A *normal representation* will be defined also for terms not possessing normal form in the usual sense.(1)

The definition of *normal representation* of a CL-term X is given in three steps [CurFe]:

3.1) Reduction to the case where X is a pure term: let X be a combination of the variables $x_1,\ldots,x_n$ (possibly not occurring in X) and the atomic constants $a_1,\ldots,a_p$, appearing in X exactly in the specified order (free variables, i.e.

(1) Obviously in such a case uniqueness is lost; note that the uniqueness of the normal representation is lost also for not proper terms possessing normal form.

variables different from $x_1,\dots,x_n$, are treated as constants); let X' be the term obtained replacing $a_1,\dots,a_p$ with the variables $y_1,\dots,y_p$, not occurring in X; if the combinator **H'** represents X' as a function of $y_1,\dots,y_p,x_1,\dots,x_n$, then $\mathbf{H}'a_1\dots a_p$ represents X as a function of $x_1,\dots,x_n$.

3.2) Reduction to a regular term: let Y be a pure combination of the variables $x_1,\dots,x_n$ (possibly not occurring in Y) and let y be a variable not occurring in Y; it follows that $Y' \equiv yY$ is a regular combination of $y,x_1,\dots,x_n$; if the combinator **U'** represents Y' as a function of $y,x_1,\dots,x_n$, then $\mathbf{U} \equiv \mathbf{U'I}$ represents Y as a function of $x_1,\dots,x_n$.

3.3) Analysis of a regular term: see definitions 2.2, 2.3 and theorem 2.4.

The notion of normal representation of a CL-term actually defines an abstraction algorithm, which is constituted by:

a) a preliminary phasis, corresponding to step 3.1 of the above given definition;

b) a properly abstraction phasis, which yields the PCNF of the proper combinator corresponding to the pure combination resulting from the preliminary phasis. This is done by a Markov algorithm in which the rewriting rules generate the components of the PCNF of a regular combinator; the intuitive meaning of the abstraction process is the following:

let M be a pure combination of the variables $x_1,\dots,x_n$ (possibly not occurring in M); the combinator $\mathfrak{R}$ such that $\mathfrak{R}\mathbf{I}x_1\dots x_n \geq M$ is obtained, starting from M, in four steps:

(i) elimination of parentheses appearing in M, in the order specified in 2.2.4;

(ii) reordering of variables (2.2.3);

(iii) elimination of multiplicities of variables (2.2.2);

(iv) elimination of cancellations of variables (2.2.1).

4. Introducing the compositive abstraction algorithm.

The sketch of algorithm described in the previous section, given by Curry in [Cur33], is complicated by the substitution process attending the preliminary phasis. However we observe that the new variables introduced in 3.1 are not involved in the steps (ii), (iii) and (iv) of the abstraction algorithm (note that this was not true with Curry's choice of permutators); in addition to this, we will modify the substitution rule of 3.1 in order to 'neutralize' the new variables also with respect to step (i); after this, we will define the final version of the abstraction algorithm.

Definition 4.1:

Let M be a CL-term and $\mathbf{A}$ the set of the *atoms* (variables and atomic constants) of M; the binary tree bT(M) associated to M is inductively defined as follows:

$\mathrm{bT}(a) = a \quad \text{if} \quad M \equiv a \in \mathbf{A};$

$\mathrm{bT}(AB) = \mathrm{bT}(A) {/\backslash} \mathrm{bT}(B) \quad \text{if} \quad M \equiv AB.$

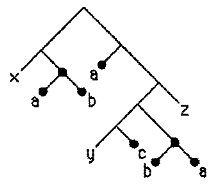
Definition 4.2: (Constant components of a CL-term)

Let bT(M) be the binary tree associated to the term M containing in this order the (not necessarily distinct) atomic constants $a_1,\dots,a_p$; we will call *constant components* of M the subterms of M associated to those subtrees of bT(M) whose leaves are labelled by constants ($\in \{a_1,\dots,a_p\}$) only.

Definition 4.3: (Maximal constant components of a CL-term)

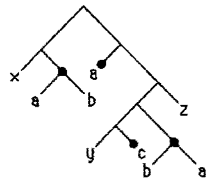
Let $\mathbf{C_C}(M)$ be the multiset of the constant component of the term M; a term $A \in \mathbf{C_C}(M)$ is said to be *maximal* if there does not exist a term $B \in \mathbf{C_C}(M)$ such that bT(A) is a son of bT(B) in bT(M).

We will write $\mathbf{C_{MC}}(M)$ to indicate the set of maximal costant components of M.



(*a*)

Fig. 4.4



(*b*)

Fig.4.4 shows (*a*) the constant components and (*b*) the maximal constant components of the term $M \equiv x(ab)(a(yc(ba)z))$.

Hence $\mathbf{C_{MC}}(M) \equiv \{(ab), a, c, (ba)\}$

5.1. Compositive abstraction algorithm (one-sweep).

Initial position: Let be given the problem $[\vec{x}_n]\,T$.

Let $\mathbf{C_{MC}}(T) \equiv \{M_1,\dots,M_q\}$ be the set of maximal constant components of T, and $\mathbf{V} \equiv \{x_1,\dots,x_n\}$.

If $T \equiv h_1T_2\dots T_m$, where $h_1 \in \mathbf{C_{MC}}(T) \cup \mathbf{V}$, we make the following position:

$$[\vec{x}_n]\,T \;\rightarrow\; [\vec{x}_n]\,\mathbf{I}\,h_1T_2\dots T_m.$$

Abstraction:

(Notation: $\mathbf{H}$ will indicate a combinator: at the beginning $\mathbf{H} \equiv \mathbf{I}$ by initial position).

Termination:

$$[\vec{x}_n]\,\mathbf{H}\,M_1\dots M_q x_1\dots x_n \;\rightarrow\; \mathbf{H}\,M_1\dots M_q,$$ where x_i does not occur in M_j, for any $x_i \in \mathbf{V}$ and $1 \le j \le q$.

Elimination of parentheses:

$$[\vec{x}_n]\,\mathbf{H}\,h_{0,1}\dots h_{0,j_0}(h_{1,1}M_{1,2}\dots M_{1,j_1})\dots(h_{p,1}M_{p,2}\dots M_{p,j_p}) \;\rightarrow$$

$$\rightarrow\; [\vec{x}_n]\,\mathbf{B}_{j_0}\mathbf{B}_{j_1-1}\,\mathbf{H}\,h_{0,1}\dots h_{0,j_0}h_{1,1}M_{1,2}\dots M_{1,j_1}\dots(h_{p,1}M_{p,2}\dots M_{p,j_p}),$$

where $h_{i,j} \in \mathbf{C_{MC}}(T) \cup \mathbf{V}$ ($0 \le i \le p$; $1 \le j \le j_i$), $M_{h,k}$ is any term, and for all t ($1 \le t \le p$) there exists an r_t ($1 \le r_t \le n$) such that in $M_{t,2}$ there is at least an occurrence of x_{r_t}.

Permutation:

$$[\vec{x}_n]\,\mathbf{H}\; h_1 \ldots h_{r-1} h_r h_{r+1} \ldots h_j h_{j+1} \ldots h_p \rightarrow$$
$$\rightarrow [\vec{x}_n]\,\mathbf{C}_{j,r}\,\mathbf{H}\; h_1 \ldots h_{r-1} h_{r+1} \ldots h_j h_r h_{j+1} \ldots h_p,$$

where $h_k \in \mathbf{C}_{MC}(T) \cup \mathbf{V}$ $(1 \leq k \leq p)$, $h_r \equiv x_s \in \mathbf{V}$, and:

if there exists a z $(r+1 \leq z \leq p-1)$ such that $h_z \equiv x_v \in \mathbf{V}$,

then $h_{z+1} \equiv x_u \in \mathbf{V}$ and $v \leq u$;

if $h_j \equiv x_t$ $(x_t \in \mathbf{V})$ then $t < s$;

if $h_{j+1} \equiv x_t$ $(x_t \in \mathbf{V})$ then $t \geq s$.

Elimination of multiplications:

$$[\vec{x}_n]\,\mathbf{H}\; M_1 \ldots M_q x_{j_1} \ldots x_{j_k} x_{j_{k+1}} \ldots x_{j_{k+r}} \ldots x_{j_s} \rightarrow$$
$$\rightarrow [\vec{x}_n]\,\mathbf{W}_{q+k,r}\,\mathbf{H}\; M_1 \ldots M_q x_{j_1} \ldots x_{j_k} x_{j_{k+r+1}} \ldots x_{j_s},$$

where $j_i \in \mathbf{V}$ $(1 \leq i \leq s)$, and:

$j_{k+t} = j_k$ $(1 \leq t \leq r)$;

$j_i < j_{i+1}$ $(1 \leq i \leq k-1)$;

$j_i \leq j_{i+1}$ $(k+r+1 \leq i \leq s-1)$.

Elimination of cancellations:

$$[\vec{x}_n]\,\mathbf{H}\; M_1 \ldots M_q x_1 x_2 \ldots x_{i-1} x_{j_1} \ldots x_{j_s} \rightarrow$$
$$\rightarrow [\vec{x}_n]\,\mathbf{K}_{i+q}\,\mathbf{H}\; M_1 \ldots M_q x_1 x_2 \ldots x_{i-1} x_i x_{j_1} \ldots x_{j_s},$$

where $j_s \leq n$; $j_1 > i$; $j_r < j_{r+1}$ $(1 \leq r \leq s-1)$.

5.2. Compositive abstraction algorithm (multi-sweep).

Initial position:

Let be given the problem $[x_1,\ldots,x_n]\;T$.

Let $\mathbf{C}_{MC}(T) \equiv \{M_1,\ldots,M_q\}$ be the set of maximal constant components of T,

and $\mathbf{V} \equiv \{x_1,\ldots,x_n\}$.

If $T \equiv h_1 T_2 \ldots T_m$, where $h_1 \in \mathbf{C}_{MC}(T) \cup \mathbf{V}$, we make the following position:

$$[x_1,\ldots,x_n]\;T \rightarrow [x_1]([x_2](\ldots([x_n]\;h_1 T_2 \ldots T_m)\ldots)).$$

Abstraction:

For every $x \in \mathbf{V}$:

Termination:

$$[x]\,\mathbf{H}\; V_1 \ldots V_s x \rightarrow \mathbf{H}\; V_1 \ldots V_s, \quad \text{if } x \text{ does not occur in } V_j, \text{ for } 1 \leq j \leq s.$$

Elimination of parentheses:

$$[x]\,\mathbf{H}\; h_{0,1} \ldots h_{0,j_0} (h_{1,1} M_{1,2} \ldots M_{1,j_1}) \ldots (h_{p,1} M_{p,2} \ldots M_{p,j_p}) \rightarrow$$
$$\rightarrow [x]\,\mathbf{B}_{j_0} \mathbf{B}_{j_1-1}\,\mathbf{H}\; h_{0,1} \ldots h_{0,j_0} h_{1,1} M_{1,2} \ldots M_{1,j_1} \ldots (h_{p,1} M_{p,2} \ldots M_{p,j_p}),$$

where $h_{i,1} \in \mathbf{C}_{MC}(T) \cup \mathbf{V}$ $(0 \leq i \leq p)$, and

- for $2 \leq j \leq j_0$, $h_{0,j} \equiv x$ or x does not occur in $h_{0,j}$
- there is at least an occurrence of x in $(h_{1,1} M_{1,2} \ldots M_{1,j_1})$.

Permutation:

$[x]\ \mathbf{H}\ h_1 \ldots h_{r-1} h_r h_{r+1} \ldots h_j h_{j+1} \ldots h_p \rightarrow$

$\rightarrow [x]\ \mathbf{C}_{j,r}\ \mathbf{H}\ h_1 \ldots h_{r-1} h_{r+1} \ldots h_j h_r h_{j+1} \ldots h_p$, where:

- for $1 \le k \le r-1$, $h_k \equiv x$ or x does not occur in h_k ;
- $h_r \equiv x$;
- for $r+1 \le k \le j$, x does not occur in h_k ;
- for $j+1 \le k \le p$, $h_k \equiv x$.

Elimination of multiplications:

$[x]\ \mathbf{H}\ V_1 \ldots V_s x \ldots (\text{r times}) \ldots x \rightarrow$

$\rightarrow [x]\ \mathbf{W}_{s+1,r}\ \mathbf{H}\ V_1 \ldots V_s x$,

where x does not occur in V_k $(1 \le k \le s)$.

Elimination of cancellations:

$[x]\ \mathbf{H}\ V_1 \ldots V_s \rightarrow$

$\rightarrow [x]\ \mathbf{K}_{s+1}\ \mathbf{H}\ V_1 \ldots V_s x$,

where x does not occur in V_k $(1 \le k \le s)$.

We now give an example of the application of the compositive abstraction algorithm, for both one and multi-sweep techniques. Note that the result of the one-sweep version of the algorithm is a regular combinator in PCNF, while the result of the multi-sweep version is a composition of regular combinators in PCNF, not on its whole in PCNF.

Let us consider the problem $[\vec{x}_4]\ x(ab)(a(ydz(baz)))(c(bx))$,
with $x_1 \equiv x$, $x_2 \equiv y$, $x_3 \equiv z$, $x_4 \equiv t$:

Initial position:

$[\vec{x}_4]\ x(ab)(a(ydz(baz)))(c(bx)) \rightarrow [\vec{x}_4]\ \mathbf{I}x(ab)(a(ydz(baz)))(c(bx)) \rightarrow$

Elimination of parentheses:

$\rightarrow [\vec{x}_4]\ \mathbf{B}_2\mathbf{B}_1\mathbf{I}x(ab)a(ydz(baz))(c(bx)) \rightarrow$

$\rightarrow [\vec{x}_4]\ \mathbf{B}_3\mathbf{B}_3(\mathbf{B}_2\mathbf{B}_1\mathbf{I})x(ab)aydz(baz)(c(bx)) \rightarrow$

$\rightarrow [\vec{x}_4]\ \mathbf{B}_6\mathbf{B}_1(\mathbf{B}_3\mathbf{B}_3(\mathbf{B}_2\mathbf{B}_1\mathbf{I}))x(ab)aydz(ba)z(c(bx)) \rightarrow$

$\rightarrow [\vec{x}_4]\ \mathbf{B}_8\mathbf{B}_1(\mathbf{B}_6\mathbf{B}_1(\mathbf{B}_3\mathbf{B}_3(\mathbf{B}_2\mathbf{B}_1\mathbf{I})))x(ab)aydz(ba)zc(bx) \rightarrow$

$\rightarrow [\vec{x}_4]\ \mathbf{B}_9\mathbf{B}_1(\mathbf{B}_8\mathbf{B}_1(\mathbf{B}_6\mathbf{B}_1(\mathbf{B}_3\mathbf{B}_3(\mathbf{B}_2\mathbf{B}_1\mathbf{I}))))x(ab)aydz(ba)zcbx \rightarrow$

Permutation: (We put $\mathfrak{B} \equiv \mathbf{B}_9\mathbf{B}_1 \circ \mathbf{B}_8\mathbf{B}_1 \circ \mathbf{B}_6\mathbf{B}_1 \circ \mathbf{B}_3\mathbf{B}_3 \circ \mathbf{B}_2\mathbf{B}_1$)

$\rightarrow [\vec{x}_4]\ \mathbf{C}_{11,8}(\mathfrak{B}\mathbf{I})x(ab)aydz(ba)cbxz \rightarrow$

$\rightarrow [\vec{x}_4]\ \mathbf{C}_{10,6}(\mathbf{C}_{11,8}(\mathfrak{B}\mathbf{I}))x(ab)ayd(ba)cbxzz \rightarrow$

$\rightarrow [\vec{x}_4]\ \mathbf{C}_{9,4}(\mathbf{C}_{10,6}(\mathbf{C}_{11,8}(\mathfrak{B}\mathbf{I})))x(ab)ad(ba)cbxyzz \rightarrow$

$\rightarrow [\vec{x}_4]\ \mathbf{C}_{7,1}(\mathbf{C}_{9,4}(\mathbf{C}_{10,6}(\mathbf{C}_{11,8}(\mathfrak{B}\mathbf{I}))))(ab)ad(ba)cbxxyzz \rightarrow$

Elimination of multiplications: (We put $\mathfrak{C} \equiv \mathbf{C}_{7,1} \circ \mathbf{C}_{9,4} \circ \mathbf{C}_{10,6} \circ \mathbf{C}_{11,8}$)

$\rightarrow [\vec{x}_4]\ \mathbf{W}_{7,1}(\mathfrak{C}(\mathfrak{B}\mathbf{I}))(ab)ad(ba)cbxyzz \rightarrow$

$\rightarrow [\vec{x}_4]\ \mathbf{W}_{9,1}(\mathbf{W}_{7,1}(\mathfrak{C}(\mathfrak{B}\mathbf{I})))(ab)ad(ba)cbxyz \rightarrow$

Elimination of cancellations: (We put $\mathfrak{W} \equiv \mathbf{W}_{9,1} \circ \mathbf{W}_{7,1}$)

$\rightarrow [\vec{x}_4]\ \mathbf{K}_{10}(\mathfrak{W}(\mathfrak{C}(\mathfrak{B}\mathbf{I})))(ab)ad(ba)cbxyzt \rightarrow$

Termination: (We put $\mathfrak{K} \equiv \mathbf{K}_{10}$) $\rightarrow$ $\mathfrak{K}(\mathfrak{W}(\mathfrak{C}(\mathfrak{B}\mathbf{I})))$ (ab)ad(ba)cb.

Thus $[\vec{x}_4]$ x(ab)(a(ydz(baz)))(c(bx)) $\rightarrow$ $(\mathfrak{K} \circ \mathfrak{W} \circ \mathfrak{C} \circ \mathfrak{B})\,\mathbf{I}$ (ab) a d (ba) c b

Let us now consider the problem [x,y,z,t] x(ab)(a(ydz(baz)))(c(bx)):

Initial position: [x,y,z,t]x(ab)(a(ydz(baz)))(c(bx)) $\rightarrow$

[x]([y]([z]([t] $\mathbf{I}$ x(ab)(a(ydz(baz)))(c(bx))))) $\rightarrow$

Abstraction with respect to the variable t:

$\rightarrow$ [x]([y]([z]([t] $\mathbf{K}_5\mathbf{I}$ x(ab)(a(ydz(baz)))(c(bx))t))) $\rightarrow$ (We put $\mathfrak{R}_4 \equiv \mathbf{K}_5$)

$\rightarrow$ [x]([y]([z] $\mathfrak{R}_4\mathbf{I}$ x(ab)(a(ydz(baz)))(c(bx)))) $\rightarrow$

Abstraction with respect to the variable z:

$\rightarrow$ [x]([y]([z] $\mathbf{B}_2\mathbf{B}_1(\mathfrak{R}_4\mathbf{I})$ x(ab)a(ydz(baz))(c(bx)))) $\rightarrow$

$\rightarrow$ [x]([y]([z] $\mathbf{B}_3\mathbf{B}_3(\mathbf{B}_2\mathbf{B}_1(\mathfrak{R}_4\mathbf{I}))$ x(ab)aydz(baz)(c(bx)))) $\rightarrow$

$\rightarrow$ [x]([y]([z] $\mathbf{B}_6\mathbf{B}_1(\mathbf{B}_3\mathbf{B}_3(\mathbf{B}_2\mathbf{B}_1(\mathfrak{R}_4\mathbf{I})))$ x(ab)aydz(ba)z(c(bx)))) $\rightarrow$

$\rightarrow$ [x]([y]([z] $\mathbf{C}_{9,8}(\mathbf{B}_6\mathbf{B}_1(\mathbf{B}_3\mathbf{B}_3(\mathbf{B}_2\mathbf{B}_1(\mathfrak{R}_4\mathbf{I}))))$ x(ab)aydz(ba)(c(bx))z)) $\rightarrow$

$\rightarrow$ [x]([y]([z] $\mathbf{C}_{8,6}(\mathbf{C}_{9,8}(\mathbf{B}_6\mathbf{B}_1(\mathbf{B}_3\mathbf{B}_3(\mathbf{B}_2\mathbf{B}_1(\mathfrak{R}_4\mathbf{I})))))$ x(ab)ayd(ba)(c(bx))zz)) $\rightarrow$

$\rightarrow$ [x]([y]([z]$\mathbf{W}_{8,1}(\mathbf{C}_{8,6}(\mathbf{C}_{9,8}(\mathbf{B}_6\mathbf{B}_1(\mathbf{B}_3\mathbf{B}_3(\mathbf{B}_2\mathbf{B}_1(\mathfrak{R}_4\mathbf{I}))))))$x(ab)ayd(ba)(c(bx))z)) $\rightarrow$

(We put $\mathfrak{R}_3 \equiv \mathbf{W}_{8,1} \circ \mathbf{C}_{8,6} \circ \mathbf{C}_{9,8} \circ \mathbf{B}_6\mathbf{B}_1 \circ \mathbf{B}_3\mathbf{B}_3 \circ \mathbf{B}_2\mathbf{B}_1$)

$\rightarrow$ [x]([y] $\mathfrak{R}_3(\mathfrak{R}_4\mathbf{I})$ x(ab)ayd(ba)(c(bx))) $\rightarrow$

Abstraction with respect to the variable y:

$\rightarrow$ [x]([y] $\mathbf{C}_{7,4}(\mathfrak{R}_3(\mathfrak{R}_4\mathbf{I}))$ x(ab)ad(ba)(c(bx))y) $\rightarrow$ (We put $\mathfrak{R}_2 \equiv \mathbf{C}_{7,4}$)

$\rightarrow$ [x] $\mathfrak{R}_2(\mathfrak{R}_3(\mathfrak{R}_4\mathbf{I}))$ x(ab)ad(ba)(c(bx)) $\rightarrow$

Abstraction with respect to the variable x:

$\rightarrow$ [x] $\mathbf{B}_5\mathbf{B}_1(\mathfrak{R}_2(\mathfrak{R}_3(\mathfrak{R}_4\mathbf{I})))$ x(ab)ad(ba)c(bx) $\rightarrow$

$\rightarrow$ [x] $\mathbf{B}_6\mathbf{B}_1(\mathbf{B}_5\mathbf{B}_1(\mathfrak{R}_2(\mathfrak{R}_3(\mathfrak{R}_4\mathbf{I}))))$ x(ab)ad(ba)cbx $\rightarrow$

$\rightarrow$ [x] $\mathbf{C}_{7,1}(\mathbf{B}_6\mathbf{B}_1(\mathbf{B}_5\mathbf{B}_1(\mathfrak{R}_2(\mathfrak{R}_3(\mathfrak{R}_4\mathbf{I})))))$ (ab)ad(ba)cbxx $\rightarrow$

$\rightarrow$ [x] $\mathbf{W}_{7,1}(\mathbf{C}_{7,1}(\mathbf{B}_6\mathbf{B}_1(\mathbf{B}_5\mathbf{B}_1(\mathfrak{R}_2(\mathfrak{R}_3(\mathfrak{R}_4\mathbf{I}))))))$ (ab)ad(ba)cbx $\rightarrow$

(We put $\mathfrak{R}_1 \equiv \mathbf{W}_{7,1} \circ \mathbf{C}_{7,1} \circ \mathbf{B}_6\mathbf{B}_1 \circ \mathbf{B}_5\mathbf{B}_1$)

$\rightarrow$ [x] $\mathfrak{R}_1(\mathfrak{R}_2(\mathfrak{R}_3(\mathfrak{R}_4\mathbf{I})))$ (ab)ad(ba)cb.

Thus [x,y,z,t]x(ab)(a(ydz(baz)))(c(bx)) $\equiv$ $(\mathfrak{R}_1 \circ \mathfrak{R}_2 \circ \mathfrak{R}_3 \circ \mathfrak{R}_4)\mathbf{I}$ (ab) a d (ba) c b.

6. Complexity.

In the analysis of complexity of the algorithms described in sections 5.1,2 we shall adopt the following conventions:

- the *length* of a CL-term T (denoted by $\mathcal{L}(T)$) is the number of atoms occurring in it, i.e. the number of leaves of bT(T);
- the *length* of a regular combinator $\mathfrak{R}$ expressed in compositive form (denoted by $\ell(\mathfrak{R})$) is the number of basic combinators involved in the representation of $\mathfrak{R}$;
- the *length of an abstraction problem* is the sum of the length of the term to be abstracted and the number of abstracted variables;
- the *complexity* of the algorithm is the order of magnitude of the length of the result as a function of the length of the abstraction problem, in the worst case.

Note that the compositive algorithm makes use of a parametric set of basic combinators; as observed by Mulder [Mul], in such a case we must multiply the complexity of the algorithm by a factor which can be:

(i) 1, if we count each combinator as an item;

(ii) proportional to the size of parameters, if we consider the representation of the introduced combinators into a computer.

We will use the notation $\lfloor h \rfloor$ to indicate the greatest integer $\leq h$ ($\in \mathbb{Q}$).

Complexity measurement will be made for pure CL-terms; in fact, constant subterms occurring in a combination are never handled by the abstraction process of the algorithms.

Theorem 6.1:

Let be given the problem $[\vec{x}_n]$ X, where X is a pure combination of the variables $x_1,\dots,x_n$ (each of them possibly not occurring in X; $n \geq 1$), and let $\mathfrak{R}$ be the regular combinator resulting from the application of the compositive one-sweep algorithm; then, assuming $\mathcal{L}(X) > 1$:

$$(6.1.1) \qquad \ell(\mathfrak{R}) \leq \lfloor {}^5/_2\, \mathcal{L}(X) \rfloor - 3 + t,$$

where t (< n) is the number of variables, between $x_1,\dots,x_n$, not occurring in X.

Proof: $\mathfrak{R} \equiv \mathfrak{K} \circ \mathfrak{W} \circ \mathfrak{C} \circ \mathfrak{B}$; hence $\ell(\mathfrak{R}) = \ell(\mathfrak{K}) + \ell(\mathfrak{W}) + \ell(\mathfrak{C}) + \ell(\mathfrak{B})$, with:

(6.1.2) $\ell(\mathfrak{K}) = t$;

(6.1.3) $\ell(\mathfrak{W}) \leq \lfloor \mathcal{L}(X)/_2 \rfloor$, the maximum number of duplications that may appear in X;

(6.1.4) $\ell(\mathfrak{C}) \leq \mathcal{L}(X) - 1$, the maximum number of permutators needed to represent a permutation of $\mathcal{L}(X)$ items;

(6.1.5) $\ell(\mathfrak{B}) \leq \mathcal{L}(X) - 2$, the maximum number of parentheses that may appear in X.

The 6.1.1 follows immediately from 6.1.2-5.

Let us now consider the problem $[x_1,\dots,x_n]$ X, in order to analyse the complexity of the multi-sweep version of the compositive algorithm with respect to the one-sweep one.

Theorem 6.2:

For every pure combination X of the variables $x_1,\dots,x_n$ (possibly not occurring in X), let $\mathfrak{R}$ be the regular combinator, in PCNF, such that $\mathfrak{R}\,\mathbf{I} \equiv [\vec{x}_n]$ X, and $\mathfrak{R}_1,\dots,\mathfrak{R}_n$ the regular combinators, in PCNF, such that $(\mathfrak{R}_1 \circ\dots\circ \mathfrak{R}_n)\,\mathbf{I} \equiv$
$\equiv [x_1,\dots,x_n]$ X; the following property holds: $\ell(\mathfrak{R}) = \ell(\mathfrak{R}_1 \circ\dots\circ \mathfrak{R}_n)$.

Sketch of the proof: We have:

- $\mathfrak{R}_i \equiv \mathfrak{K}_i \circ \mathfrak{W}_i \circ \mathfrak{C}_i \circ \mathfrak{B}_i$ $(1 \leq i \leq n)$;
- $\mathfrak{R} \equiv \mathfrak{K} \circ \mathfrak{W} \circ \mathfrak{C} \circ \mathfrak{B}$.

The theorem follows from proving:

(6.2.1) $\ell(\mathfrak{K}) = \Sigma_{i=1,\dots,n}\, \ell(\mathfrak{K}_i)$; (6.2.2) $\ell(\mathfrak{W}) = \Sigma_{i=1,\dots,n}\, \ell(\mathfrak{W}_i)$;

(6.2.3) $\ell(\mathfrak{C}) = \Sigma_{i=1,\dots,n}\, \ell(\mathfrak{C}_i)$; (6.2.4) $\ell(\mathfrak{B}) = \Sigma_{i=1,\dots,n}\, \ell(\mathfrak{B}_i)$.

The complete proof, here omitted, can be found in [Ptes].

Note that theorem 6.2 is not valid with Curry's permutators; this point can be intuitively explained as follows: during the permutation phasis of the abstraction process, variables are moved from left to right; it is easy to verify that this operation can be done in one step by the new permutators, but not by the old ones.

7. Conclusion.

To summarize, we showed a new (old) abstraction algorithm which seems to be interesting for the following reasons:

- it is compositive: the resulting code has a 'structured' look, i.e. it may be viewed as a succession of procedure callings, hence it is quite readable;
- it is efficient: the worst case mentioned above concerns pure combinations; in the general case, the complexity rate is a function of the number of variables occurring in the term to be abstracted, not of its whole length; this fact makes the algorithm suitable for the compilation of a functional program;
- it can be defined in both one and multi-sweep techniques, preserving the length of the resulting code: this 'invariance property' is not valid for any other abstraction algorithm;
- it gives a combinatory equivalent of Hughes' supercombinators: in effect supercombinators are proper combinators; their interpretation can be achieved adding to the abstraction rules of the multi-sweep version of the compositive algorithm a clause of belonging to the set of *maximal free expressions* (see [Hug]). This point will be better explained in the appendix.

Acknowledgement.

The author is grateful to Prof.Corrado Böhm for helpful suggestions and discussions about the subject of this paper.

References.

[Bar] - H.P.Barendregt, The Lambda Calculus, its Syntax and Semantics, Studies in Logic, Vol.103, North-Holland, Amsterdam (1984).

[Cur30] - H.B.Curry, Grundlagen der kombinatorischen Logik, American Journal of Mathematics, Vol.52 (1930).

[Cur32] - H.B.Curry, Some additions to the theory of combinators, American Journal of Mathematics, Vol.54 (1932).

[Cur33] - H.B.Curry, Apparent variables from the standpoint of Combinatory Logic, Annals of Mathematics, Vol.34 (1933).

[CurFe] - H.B.Curry & R.Feys, Combinatory logic, Vol.1, North-Holland, Amsterdam (1958).

[Hug] - R.J.M.Hughes, SuperCombinators: a new implementation method for Applicative Languages, Symp.on LISP and Funct.Progr.,ACM (Aug 1982).

[Mul] - J.C.Mulder, Complexity of combinatory code, University of Utrecht (int.rep., 1985).

[Ptes] - A.Piperno, Metodi di astrazione in logica combinatoria: analisi, proposte, applicazioni, Tesi di laurea, 1986.

[Sta] - R.Statman, On translating lambda terms into combinators: the basis problem, LICS, Boston, 1986.

[Tur79a] - D.A.Turner, Another algorithm for bracket abstraction, The Journal of Symbolic Logic, Vol.44 n.2 (1979).

[Tur79b] - D.A.Turner, A new implementation technique for applicative languages, Software Practice and Experience, n.9 (1979).

APPENDIX: Compositive algorithm and supercombinators.

We suppose the reader to be familiar with the notions of *supercombinators* and *fully lazy evaluation* ; we will show in an intuitive way how the compositive abstraction algorithm can be modified in order to yield a purely combinatory interpretation of supercombinators.

Supercombinators were introduced by Hughes [Hug] with the purpose of giving an efficient implementation technique of full laziness; they are built up, starting from an arbitrary λ-expression E, in the following way:

1) find the innermost λ-expression $\lambda t.H$ appearing in E;

2) let $F_1,\dots,F_n$ be the non-constant maximal free expressions(1) (MFE) of H, ordered as stated by some optimisation rules: replace $F_1,\dots,F_n$ with the variables $x_1,\dots,x_n$, not occurring in E, and let H^* be the term obtained after this substitution;

3) give a name (τ) to $\lambda x_1 \dots x_n t.H^*$;

4) replace $\lambda t.H$ with $\tau F_1 \dots F_n$ in E;

5) repeat steps 1-4 until there are no more λ-expressions.

The method described above can be considered as an algorithm of translation from λ-calculus to CL, working with the infinite basis constituted by the whole set of proper combinators; the resulting supercombinators, however, are not directly implementable, and need an extra level of interpretation.

Let us now consider the multi-sweep version of the compositive abstraction algorithm, where the conditions that rule the steps of the algorithm are enriched in an opportune way with some clauses of belonging to the set of MFE of the expression subjected to the abstraction process.

It comes out that the result of the operation $[x]\,T$, when subjected to the final algorithm, is of the form $\mathfrak{R}|C_1 \dots C_k F_1 \dots F_n$, where $\mathfrak{R}$ is a regular combinator (in PCNF), $C_1,\dots,C_k$ are constant subexpressions of T, and $F_1,\dots,F_n$ are the MFEs of $\lambda x.T$. Thus $\mathfrak{R}|C_1 \dots C_k$ is the combinatory interpretation, via the compositive abstraction algorithm, of the supercombinator resulting from the application of Hughes' method to $\lambda x.T$.

In addition to this, the optimisation rules, introduced by Hughes to improve the efficiency of supercombinators, can be enclosed into the permutation step of the final algorithm.

Summarizing, it is possible to modify the multi-sweep version of the compositive abstraction algorithm, in order to have a purely combinatory equivalent of Hughes' method, which has the following properties:

- it preserves the linearity property of the native algorithm;
- it makes use of a directly implementable set of combinators: no extra level of interpretation is needed.

(1) recall that a *free expression* of $\lambda x.T$ is a subexpression of T which does not depend on the bound variable x, and that a free expression is called *maximal* if it is not a proper subexpression of a free expression.