Compilation of Lambda-Calculus into Functional Machine Code

P. Fradet and *D. Le Métayer*
IRISA / INRIA
Campus de Beaulieu
35042 RENNES CEDEX, FRANCE
{fradet@irisa.fr and lemetayer@irisa.fr}

Abstract

One of the most important issues concerning functional languages today is the efficiency and the correctness of their implementation. In this paper we describe the whole implementation process in the functional framework. The original functional expression is successively transformed into a functional expression which can be seen as a traditional machine code. The two main steps are the compilation of the computation rule by the introduction of continuation functions and the compilation of the environment management using combinators. The advantage of this approach is that we do not have to introduce an abstract machine, which makes the correctness proofs much simpler. As far as efficiency is concerned, this approach is promising since a lot of optimisations can be described and formally justified in the functional framework.

1. Introduction

The implementation of functional languages is generally described in terms of an abstract machine **[1,3,7,10]** reducing either the source functional program or a compiled version of this program. The abstract machine itself is implemented on a traditional von Neumann computer. So a complete correctness proof of the implementation should involve three steps:

(1) proof of the compilation process,

(2) proof that the reduction of a compiled expression by a specific computation rule and its execution on the abstract machine yield the same result,

(3) proof that the implementation of the abstract machine is correct.

Part (1) is generally easy as the compilation produces a functional expression. Part (2) however involves the operational description of a specific machine which is much more difficult to tackle. Part (3) is generally omitted because the abstract machine is supposed to be close to the real one. This step would probably deserve more attention if the implementation of the abstract machine on the real one involves a non trivial translation process.

Since correctness proofs are much easier in the functional framework, we believe that the whole implementation process should be described in a purely functional way. We present a method for transforming λ-expressions into simpler functional expressions whose reduction can be seen as an execution on a traditional machine with two components: the code and the stack. *The important point is that we do not have to introduce a machine with an operational description indicating how the state evolves during the computation.* The functional expressions produced are of the form $f\ g\ s_1 \ldots\ldots s_n$ where f is a basic function which behaves like a machine instruction operating on the stack $(s_1 \ldots\ldots s_n)$ and g is an expression representing the rest of the code.

The execution of a functional program involves two main tasks:

(1) searching for the next expression to reduce according to a specific computation rule,

(2) the management of the environment.

We achieve the compilation of these two tasks in the functional framework. Section 2 describes the compilation of the computation rule. The resulting expressions can be evaluated from left to right by successively reducing the head operator. The compilation of environment management, presented in section 3, is done by an abstraction algorithm in the same spirit as **[11,15]**. This abstraction uses a set of combinators acting like traditional machine instructions (move, push,...). In conclusion we show that the produced code is very efficient and we compare the approach with related works.

2. Compilation of the Computation Rule

Our source language is a λ-calculus with constants described by the following syntax:

$$e ::= x \mid k \mid op_m\ e_1 \ldots e_m \mid \text{cond}\ e_1\ e_2\ e_3 \mid e_1\ e_2 \mid \lambda x.e_1 \mid \text{letrec}\ f = \lambda x.e_1$$

where e_i are expressions, x is a variable, k is a basic constant, and op_m is a strict primitive operator of arity m ; the primitive cond is the only non strict operator.

We consider in this paper that the language is strict, so it can be evaluated by call-by-value. The method has been applied to compile a call-by-name version of the language, but we shall not dwell on this for space considerations. The definition of factorial in this language is:

letrec fact = λx. cond (eq 0 x) 1 (mult x (fact (sub x 1)))

The evaluation of this expression by call-by-value involves a repeated search for the next redex: the first operation to execute is eq, then cond, then either 1 or sub, and so on.... The compilation of the computation rule should produce an expression which can be evaluated by systematic application of the first operator (from left to right). Basically we have to inverse the order of subexpressions in a composition: the evaluation of a composition $(E_1\ E_2)$ by value entails the evaluation of E_2, then the evaluation of E_1, and finally the application of the result of E1 to the result of E2 (we take here the rightmost innermost interpretation of call-by-value: the leftmost innermost strategy could have been chosen as well). But replacing $(E_1\ E_2)$ by $(E_2\ E_1)$ is not correct ; we have to provide a mechanism for putting the result of E_2 back to its place after its evaluation. *This effect is achieved via the use of continuations; we transform each expression e into an expression $\Psi(e)$ taking a continuation as argument and applying it to the result of evaluating e.* In the same way we define a new operator op_{mc} for each operator op_m such that:

$op_{mc}\ c\ e_1 \ldots e_m = c\ (\ op_m\ e_1 \ldots e_m\ )$ and $\text{cond}_c\ e_2\ e_3\ e_1 = \text{cond}\ e_1\ e_2\ e_3$

cond_c is a particular function which takes two possible continuations (e_2 and e_3).

The following figure describes the transformation rules of the first compilation step.

(Ψ1). $\Psi(x) = \lambda c.\ c\ x$

(Ψ2). $\Psi(k) = \lambda c.\ c\ k$

(Ψ3). $\Psi(op_m\ e_1 \ldots e_m) = \lambda c.\ \Psi(e_m)\ (\Psi(e_{m-1})\ (\ldots(\Psi(e_1)\ (op_{mc}\ c))\ldots))$

(Ψ4). $\Psi(\text{cond}\ e_1\ e_2\ e_3) = \lambda c\ .\ \Psi(e_1)\ (\text{cond}_c\ (\Psi(e_2)\ c)\ (\Psi(e_3)\ c))$

(Ψ5). $\Psi(e_1\ e_2) = \lambda c.\ \Psi(e_2)\ (\Psi(e_1)\ \mathbf{id}\ c)$

(Ψ6). $\Psi(\lambda x.e) = \lambda c.\ c\ (\lambda c.\lambda x.\ \Psi(e)\ c)$

(Ψ7). $\Psi(\text{letrec}\ f = \lambda x.e) = \lambda c.\ c\ (\text{letrec}\ f = \lambda c.\lambda x.\ \Psi(e)\ c)$

id denotes the identity function $\lambda c.c$. Rules **(Ψ1)** and **(Ψ2)** follow from the convention described above that expressions take a continuation and apply it to their result. **(Ψ3)** explicits the call-by-value evaluation of a composition $(\ldots(op_m\ e_1)\ldots e_n)$: e_m is evaluated first, then $e_{m-1},\ldots,e_1$, and op_{mc} can finally be applied with continuation c. Let us note that $\Psi(e_m)$ takes $(\Psi(e_{m-1})\ldots(\Psi(e_1)(op_{mc}c))\ldots)$ as a continuation which means that its result will be put at the right place after its evaluation. **(Ψ4)** can be explained in the same way. Rule **(Ψ5)** applies when e_1 is not a primitive function: the first continuation **id** is necessary to get the functional value of e_1 (look for example at **(Ψ6)** to see how a λ-expression is transformed) and the second continuation c will be the continuation taken by this function. In rules **(Ψ6)** and **(Ψ7)** the continuation is applied to the whole expression because a λ-expression is not evaluated by call-by-value ; it is returned unchanged.

Remark: For call-by-name rules **(Ψ1)** and **(Ψ5)** would become:

(Ψ'1). $\Psi'(x) = x$

(Ψ'7). $\Psi'(e_1\ e_2) = \lambda c.\ \Psi'(e_1)\ \mathbf{id}\ c\ \Psi'(e_2)$

We take the convention that the top-level expression is always applied to the continuation **id**. For example, a top level application of a function f defined by (letrec f = λx.e) to a constant n would be:

$\Psi(f\ n)\ \mathbf{id} = (\lambda c.\ \Psi(n)\ (\Psi(f)\ \mathbf{id}\ c))\ \mathbf{id} = (\lambda c.c\ n)\ (\Psi(f)\ \mathbf{id}\ \mathbf{id})$

$= \Psi(f)\ \mathbf{id}\ \mathbf{id}\ n$

$= (\lambda c.c\ (\text{letrec}\ f = \lambda c.\lambda x.\Psi(e)\ c))\ \mathbf{id}\ \mathbf{id}\ n$

$= (\text{letrec}\ f = \lambda c.\lambda x.\Psi(e)\ c)\ \mathbf{id}\ n$

The property that the top-level continuation is always **id** can sometimes be exploited to achieve drastic improvements of the code: if f does not appear in e we can replace (f **id**) by g where:

letrec g = λx. Ψ(e) **id**

and realize further simplifications when (Ψ(e) **id**) can be reduced.

If f is recursive and (Ψ(e) **id**) can be simplified, by β-reduction, into an expression e' where f occurs only in the context (f **id**) then we can replace (f **id**) by g where:

letrec g = λx. e' [g/(f **id**)] *{E $[e_1/e_2]$ is the expression E where e_2 is replaced by e_1}*

This situation, which can easily be detected, occurs when the continuation of f is always **id**. *This optimisation corresponds to an improvement of the compiler to preserve tail recursion; an important payoff of the functional approach is that most well-known compilator optimisation techniques can be expressed (and formally justified) by simple program transformation rules.*

We give now the result of the compilation of the factorial function according to these rules:

letrec fact = λc.λx. Ψ(cond (eq 0 x) 1 (mult x (fact (sub x 1)))) c

= λc.λx. λc.(Ψ (eq 0 x) (cond$_c$ (Ψ(1) c) (Ψ(mult x (fact (sub x 1))) c))) c (Ψ4)

= λc.λx. λc.((λc. Ψ(x) (Ψ(0) (eq$_c$ c)))
  (cond$_c$ ((λc. c 1) c) (Ψ(mult x (fact (sub x 1))) c))) c (Ψ3)&(Ψ2)

= λc.λx. λc.((λc. (λc. c x) ((λc. c 0) (eq$_c$ c)))
  (cond$_c$ ((λc. c 1) c) (Ψ(mult x (fact (sub x 1))) c))) c (Ψ1)&(Ψ2)

= ......

After simplification by β-reduction we get:

letrec fact = λc. λx. eq$_c$ (cond$_c$ (c 1) (sub$_c$ (fact (mult$_c$ c x)) x 1)) 0 x

In order to convince the reader that the function can really be evaluated by reducing systematically the first operator of the current expression, we describe now the evaluation of (fact **id** 1). The operator applied at each step is underlined.

($\underline{\text{letrec}}$ fact = λc. λx. eq$_c$ (cond$_c$ (c 1) (sub$_c$ (fact (mult$_c$ c x)) x 1)) 0 x) **id** 1

= $\underline{\text{eq}_c}$ (cond$_c$ (**id** 1) (sub$_c$ (fact (mult$_c$ **id** 1)) 1 1)) 0 1

= $\underline{\text{cond}_c}$ (**id** 1) (sub$_c$ (fact (mult$_c$ **id** 1)) 1 1) False

= $\underline{\text{sub}_c}$ (fact (mult$_c$ **id** 1)) 1 1

= ($\underline{\text{letrec}}$ fact = λc. λx. eq$_c$ (cond$_c$ (c 1) (sub$_c$(fact (mult$_c$ c x)) x 1)) 0 x) (mult$_c$ **id** 1) 0

= $\underline{\text{eq}_c}$ (cond$_c$ (mult$_c$ **id** 1 1) (sub$_c$ (fact (mult$_c$ (mult$_c$ **id** 1) 0)) 0 1)) 0 0

= $\underline{\text{cond}_c}$ (mult$_c$ **id** 1 1) (sub$_c$ (fact (mult$_c$ (mult$_c$ **id** 1) 0)) 0 1) True

= $\underline{\text{mult}_c}$ **id** 1 1

= $\underline{\textbf{id}}$ 1

= 1

In order to prove the correctness of the transformation Ψ, we have to show that evaluating the transformed expression by systematic reduction of the first symbol amounts to evaluating the original expression by call-by-value. Let CR(e) denote the result of the evaluation of e by the computation rule CR ; CBV denotes call-by-value and FIRST is our new computation rule. We have proved the following properties:

Property 1: if the evaluation of an expression e by CBV does not terminate, then for any c the evaluation of (Ψ(e) c) by FIRST does not terminate.

Property 2: if the evaluation of an expression e by CBV terminates then:

($\forall$c) FIRST (Ψ(e) c) $\equiv$ FIRST (Ψ(CBV(e)) c) *where "$\equiv$" denotes a syntactic equality*

Corollary: ($\forall$e) if CBV(e) $\equiv$ k then FIRST (Ψ(e) **id**) $\equiv$ k

Property 1 is shown by proving that the termination by FIRST implies the termination by CBV. This is done by induction on the length of the reduction sequence by FIRST. Property 2 is shown by induction on the length of the reduction sequence by CBV and structural induction. These proofs are not complicated but involve a tedious inspection of the different cases [5].

3. Compilation of Environment Management

Let us come back to the evaluation of (fact **id** 1) described in the previous section to make a comparison with machine code execution. Throughout the reduction of (fact **id** 1) the expression under evaluation is always of the form:

$$exp_1\ exp_2\ exp_3 \ldots exp_n$$

where exp_1 is the next function to apply and exp_2 its continuation. When exp_1 is evaluated and returns the value e_1, the expression becomes $exp_2\ e_1\ exp_3 \ldots exp_n$. Let us now look at these expressions as machine states. Clearly exp_1 would be the next instruction to execute, exp_2 would be the rest of the code and ($exp_3 \ldots exp_n$) would play the role of a stack. However these expressions are still far from machine code ; the basic reason is the occurence of λ-expressions whose reduction involves some kind of environment management. *We use a well-known technique for the compilation of environment management within the functional framework, which is called abstraction [4,15].* The abstraction process consists in translating a functional expression into an equivalent one which contains no variable via the use of combinators. *We choose here a set of indexed combinators which act on their arguments as machine instructions on a stack:*

$\mathbf{id}\ x = x$

$\mathbf{push}\ x\ f = f\ x$

$\mathbf{dupl}_n\ f\ s_1 \ldots s_n = f\ s_n\ s_1 \ldots s_n$

$\mathbf{move}_{m,n}\ f\ s_1 \ldots s_n \ldots s_{m-1}\ s_m = f\ s_1 \ldots s_n \ldots s_{m-1}\ s_n$ *{if $n < m$}*

$\mathbf{move}_{m,n}\ f\ s_1 \ldots s_{m-1}\ s_m\ s_{m+1} \ldots s_n = f\ s_1 \ldots s_{m-1}\ s_n\ s_{m+1} \ldots s_n$ *{if $n > m$}*

$\mathbf{flsh}_n\ f\ s_1 \ldots s_n = f$

$\mathbf{ldcl}_n\ f\ g\ s_1 \ldots s_n = f\ (g\ s_n)\ s_1 \ldots s_n$

These combinators can be seen as machine instructions operating in the following way:

- **id** corresponds to a return instruction: if the stack contains a single element then **id** returns it ; otherwise it provokes a jump to the address given by the top of the stack,
- **push** is the traditional push instruction with an immediate argument,
- $\mathbf{dupl}_n$ pushes the nth element of the stack,
- $\mathbf{move}_{m,n}$ replaces the mth element of the stack by the nth element,
- $\mathbf{flsh}_n$ pops the first n elements of the stack: it amounts to a modification of the stack pointer,
- $\mathbf{ldcl}_n$ is used to build a closure on the top of the stack. In a real machine the top of the stack would contain a pointer to the currently built closure and $\mathbf{ldcl}_n$ would involve the allocation of a new memory cell to the new value. The expression of this operation within the functional framework amounts to adding to the terms new elements representing the memory cells [5].

In order to make the description of the abstraction algorithm clearer, we introduce more powerful combinators which can be defined in terms of the previous ones:

$\mathbf{exed}_{m,n,i}\ s_1 \ldots s_m\ x_1 \ldots x_n = x_i\ s_1 \ldots s_m$

$\mathbf{mkcl}_{m,n}\ f\ g\ s_1 \ldots s_m\ x_1 \ldots x_n = f\ (g\ x_1 \ldots x_n)\ s_1 \ldots s_m\ x_1 \ldots x_n$

$\mathbf{delt}_{m,n}\ f\ s_1 \ldots s_m\ x_1 \ldots x_n = f\ s_1 \ldots s_m$

In operational terms $\mathbf{exed}_{m,n,i}$ is a jump to the address contained in the i+mth element of the stack after having removed from the stack the n elements corrresponding to the current "environment" (i.e. arguments of the function under evaluation). The first m elements correspond to the arguments of the called function. The combinator $\mathbf{mkcl}_{m,n}$ builds in one step a closure on the top of the stack and $\mathbf{delt}_{m,n}$ removes n elements from the stack. The following properties can be easily checked:

$$\mathbf{exed}_{m,n,i} = \mathbf{dupl}_{m+i}\,(\mathbf{move}_{m+n+1,m+1}\,(\ldots(\mathbf{move}_{n+1,1}\,(\mathbf{flsh}_n\,\mathbf{id}))\ldots))$$
$$\mathbf{mkcl}_{m,n}\,f\,g = \mathbf{ldcl}_{m+1}\,(\ldots(\mathbf{ldcl}_{m+n}\,f)\ldots)\,g$$
$$\mathbf{delt}_{m,n}\,f = \mathbf{move}_{m+n,m}\,(\ldots(\mathbf{move}_{n+1,1}\,(\mathbf{flsh}_n\,f))\ldots)$$

We can now present our abstraction algorithm ; the abstraction of variables $x_1,\ldots,x_n$ from M is denoted by $[x_1,\ldots,x_n]_0\,M$. In other words $[x_1,\ldots,x_n]_0\,M$ is an expression containing no variable such that $([x_1,\ldots,x_n]_0\,M)\,x_1\ldots x_n = M$. Actually we give a more general definition of the abstraction $[x_1,\ldots,x_n]_p\,M$ such that:

$$(\,[x_1,\ldots,x_n]_p\,M)\,s_1\ldots s_p\,x_1\ldots x_n = M\,s_1\ldots s_p$$

We take the convention that when a function is called its arguments are on the top of the stack. The function execution may involve the installation of new elements on the top of the stack: index p in the abstraction algorithm denotes the number of values pushed on the stack over the arguments of the function at a particular execution step. So the ith argument of a function can always be found at the p+ith position in the stack.

The global expression is first normalized: nested λ-expressions are transformed into combinators in the following way:

$$\lambda x_1.\ \ldots\ \lambda x_n.\ \mathrm{exp}\ \dashrightarrow\ (\lambda y_1.\ \ldots\ \lambda y_k.\ \lambda x_1.\ \ldots\ \lambda x_n.\ \mathrm{exp})\ y_1\ \ldots\ y_k$$

where y_1 , ..., y_k are the free variables of the original lambda expression.

This normalization, which is very much in the spirit of supercombinators [6] (but does not exhibit full laziness), allows us to apply the abstraction algorithm to the innermost λ-expressions in a bottom-up fashion.

Abstraction algorithm

(A1).	$[x_1,\ldots,x_n]_p\,M$	$= \mathbf{delt}_{p,n}\,[]_p M$	if $x_1,\ldots,x_n \notin M$ & $n\neq 0$
(A2).	$[x_1,\ldots,x_n]_p\,M\,x_i$	$= \mathbf{dupl}_{p+i}\,([x_1,\ldots,x_n]_{p+1}\,M)$	if $x_i \in \{x_1,\ldots,x_n\}$
(A3).	$[x_1,\ldots,x_n]_p\,M\,y$	$= ([x_1,\ldots,x_n]_{p+1}\,M)\,y$	if $y \notin \{x_1,\ldots,x_n\}$
(A4).	$[x_1,\ldots,x_n]_p\,M\,k$	$= \mathbf{push}\,k\,([x_1,\ldots,x_n]_{p+1}\,M)$	
(A5).	$[x_1,\ldots,x_n]_p\,op_{mc}\,M$	$= op_{mc}\,([x_1,\ldots,x_n]_{p-m+1}\,M)$	
(A6).	$[x_1,\ldots,x_n]_p\,cond_c\,M\,N$	$= cond_c\,([x_1,\ldots,x_n]_{p-1}\,M)\,([x_1,\ldots,x_n]_{p-1}\,N)$	
(A7).	$[x_1,\ldots,x_n]_p\,x_i$	$= \mathbf{exed}_{p,n,i}$	if $x_i \in \{x_1,\ldots,x_n\}$
(A8)	$[x_1,\ldots,x_n]_p\,y$	$= y$	if $y \notin \{x_1,\ldots,x_n\}$
(A9).	$[x_1,\ldots,x_n]_p\,M\,N$	$= \mathbf{push}\,(\,[x_1,\ldots,x_n]_0\,N)\,(\mathbf{mkcl}_{p,n}\,(\,[x_1,\ldots,x_n]_{p+1}\,M))$	if M is not a basic operator

(A1) means that the arguments of the function can be discarded from the stack as soon as they are no longer referenced in the remaining code.

(A2) indicates that a composition (M x_i) is evaluated by first pushing the value of x_i and then evaluating M with one more element on the stack.

(A4) achieves the same effect with a constant argument k.

(A3) & **(A8)** are applied in the abstraction of a nested expression containing free variables (i.e. function names) which are left unchanged.

(A5) describes the treatment of operators: an operator of arity m consumes m elements and produces one.

(A6) expresses the fact that cond_c consumes one (boolean) element and then tranfers the control to one of its two alternatives.

(A7) is applied when the remaining expression is an argument which means that all other arguments can be discarded. This effect is achieved by the **exed** combinator.

(A9) deals with the evaluation of a non-basic expression M with a non basic continuation N. A representation of the continuation N must be pushed on the stack with its environment so that the expression M can call it after its own evaluation (this is achieved by **push** and **mkcl**).

Let us note that rule **(A9)** can be optimized in the following way:

(A9'). $[x_1,\ldots,x_n]_p\, f_i\, N = \textbf{push}\ ([x_1,\ldots,x_n]_1\, N)\, f_i$

where f_i denotes a user-defined function of p+1 arguments whose definition does not contain $x_1,\ldots,x_n$ as free variables. This implies that the execution of f_i will not destroy the environment, so there is no need to save it. This rule is an optimisation of **(A9)** because it avoids the stack manipulations involved in the construction and execution of a closure.

Let us now come back to the factorial function to illustrate this abstraction algorithm. The function produced by the first transformation Ψ is (section 2):

letrec fact = λc. λx. eq_c (cond_c (c 1) (sub_c (fact (mult_c c x)) x 1)) 0 x

Applying the abstraction rules defined above we get:

letrec fact = $[c,x]_0$ (eq_c (cond_c (c 1) (sub_c (fact (mult_c c x)) x 1)) 0 x)

= $\textbf{dupl}_2$ ($[c,x]_1$ (eq_c (cond_c (c 1) (sub_c (fact (mult_c c x)) x 1)) 0)) **(A2)**

= $\textbf{dupl}_2$ (**push** 0 ($[c,x]_2$ (eq_c (cond_c (c 1) (sub_c (fact (mult_c c x)) x 1)))) **(A4)**

= $\textbf{dupl}_2$ (**push** 0 (eq_c($[c,x]_1$ (cond_c (c 1) (sub_c (fact (mult_c c x)) x 1))))) **(A5)**

= $\textbf{dupl}_2$ (**push** 0 (eq_c (cond_c ($[c,x]_0$ (c 1)) ($[c,x]_0$ (sub_c (fact (mult_c c x)) x 1))))) **(A6)**

=......... *{using (A9') instead of (A9)}*

= $\textbf{dupl}_2$ (**push** 0 (eq_c (cond_c
    (**push** 1 $\textbf{exed}_{1,2,1}$)
    (**push** 1 ($\textbf{dupl}_3$ (sub_c (**push** ($\textbf{dupl}_3$ (mult_c $\textbf{exed}_{1,2,1}$)) fact))))))

$\textbf{exed}_{1,2,1}$ = $\textbf{dupl}_2$ ($\textbf{move}_{4,2}$ ($\textbf{move}_{3,1}$ ($\textbf{flsh}_2$ **id**)))

The following rules allow us to achieve peephole optimisations:

$\textbf{dupl}_i$ ($\textbf{move}_{j,k}$ exp) = $\textbf{move}_{j-1,k-1}$ ($\textbf{dupl}_i$ exp)   with $j, k \neq 1$, $j-1 \neq i$

and $\textbf{dupl}_i$ ($\textbf{flsh}_j$ exp) = $\textbf{flsh}_{j-1}$ exp

We get: $\mathbf{exed}_{1,2,1} = \mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**), and:

letrec fact = $\mathbf{dupl}_2$ (**push** 0 (eq_c (cond_c
(**push** 1 ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**)))
(**push** 1 ($\mathbf{dupl}_3$ (sub_c (**push** ($\mathbf{dupl}_3$ (mult_c ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**))))
fact)))))))

The correctness property of the abstraction algorithm can be stated in the following way:

Property 3: $(\forall s_1 \ldots s_p)$ $([x_1,\ldots,x_n]_p M)s_1\ldots s_p\, x_1\ldots x_n = M\, s_1\ldots s_p$

The proof is a routine inspection of the different cases of the algorithm [5].

The expressions yielded by the abstraction algorithm look very much like machine code. The only remaining difference is the fact that these expressions are still binary trees whereas machine code is made of sequences of instructions.The linearization is achieved by the introduction of names denoting embedded composed expressions. In operational terms these names correspond to code addresses. The last remark concerns function names in recursive definitions. These names remain unchanged by the abstraction process since they are free variables. If we assume that names represent code addresses we must translate the occurences of function names into **jump** instructions. In functional terms (**jump** f) is defined by **jump** f = f. The application of this last transformation to the expression of factorial produced by the previous step, yields:

letrec fact = $\mathbf{dupl}_2$ (**push** 0 (eq_c (cond_c f_0 f_1)))
let f_0 = **push** 1 ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**))
let f_1 = **push** 1 ($\mathbf{dupl}_3$ (sub_c (**push** f_2 (**jump** fact))))
let f_2 = $\mathbf{dupl}_3$ (mult_c ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**)))

We have now several linearized trees which can be written as sequences of code in the following way:

```
fact    dupl    2              f1    push    1
push    0                            dupl    3
        eq_c                         sub_c
        cond_c  f0, f1               push    f2
f0      push    1                    jump    fact
        move    3,1            f2    dupl    3
        flsh    1                    mult_c
        id                           move    3,1
                                     flsh    1
                                     id
```

The appendix describes the evolution of the stack during the execution of (fact **id** 1) and illustrates the duality (functional expression/machine code) of the result of the compilation.

4. Conclusion

We have described a transformation of functions defined in a lambda-calculus with constants into "equivalent" functions defined in terms of combinators acting on their arguments like machine instructions on a stack. *The major originality of this approach as compared to the SECD machine [1,10] and the CAM [3] is that we do not have to introduce a machine and describe it in terms of state transitions. The state of the machine is the expression itself and its evolution is specified by the definition of the combinators.* For example, the evaluation of the result of the compilation of the factorial function can be described as the reduction of a functional expression or as the execution of code on a stack machine (see appendix). This approach has interesting payoffs as far as correctness proofs are concerned. *We do not have to prove that the operational definition of the machine is coherent with the operational semantics of the language as in [3,12,13] since they are identical.* The only operational argument in our proof appears in section 2 where we have to show that reduction of the transformed expression by FIRST amounts to the reduction of the original expression by call-by-value (for instance). However this proof does not involve reasoning on tricky machine states.

The formalization of the implementation process has also been studied by Reynolds [14], followed by Wand [16]; they proceed by successive transformations of a semantics of the source language to derive an interpreter or a compiler and an abstract machine. [16] presents some heuristics for analysing the compilation process. This method also involves continuations and combinators, but in a quite different way: it takes a continuation-based semantics in input whereas in our work continuations appear in the compilation process as a formalization of the computation rule. *Furthermore Wand translates the semantics of the program into a sequence representing the code and a program (or "machine") to execute it; in our approach semantics or machines do not appear explicitly.* We believe that staying in the functional framework and proceeding exclusively by program transformation (instead of interpreter transformation) makes formal proofs easier. Let us remark however that Wand's goal is a bit different since he deals in the same way with any language (imperative or functional) which can be described by a continuation semantics.

The benefits of the use of continuations to compiler design have already been illustrated by previous work on ORBIT [9]. They integrate continuation conversion as a preliminary "standardisation" step but the compilation (code generation in particular) is not entirely described by program transformation.

Even if the performance considerations are not the main topic of this paper we have to say a few words about the produced code. *First we should mention that the first transformation does not depend on the chosen implementation and could as well be applied in the context of graph reduction (it would lead to a simpler graph evaluator reducing systematically the first term of the expression).* We have chosen the environment-based approach rather than graph reduction [7,8,15] because it is closer to traditional Von Neumann machines. The code produced by our transformation rules is rather close to the code of [1] for the SECD machine and [3] for the CAM. The main difference with the SECD machine is the place where environment savings are achieved: like in the CAM we achieve the savings when encountering intermediate subexpressions rather than at function call. We depart from the CAM as far as environment representation is concerned ; in the CAM, environments are represented as trees which entails less expensive closure building but costly access time. Another possible choice is to keep a pointer to the global environment and to distinguish access to local identifiers and to global identifiers [2]; this makes function calls more efficient because context switching amounts to a pointer movement.

A prototype compiler based on this approach has been implemented on a SUN workstation. Let us point out that the implementation of a compiler based on our transformation rules in a language with pattern matching such as ML is quite straightforward. The following table shows the execution times of the code produced by our compiler for the traditional fib 20 with call-by-value, call-by-name and call-by-need.

	c. b. value	c. b. name	c. b. need
time	80 ms	2.2 s	0.55 s
call/sec	274 000	10 000	40 000

These results show that the code produced by our compiler is realistic (the efficiency of the code produced by the C compiler for the same function is 263,000 call/sec).*We believe that these performances are made possible by the program transformation approach which allows the systematic application of optimisation rules.* Actually most well-known compiler optimisation techniques can be described in a functional way; each technique should be applied at the appropriate transformation level; for example common subexpression elimination, and tail recursion should be achieved after the first transformation step whereas peephole optimisations must be applied on the resulting code. However we should mention that only the simple language described in section 1 has been implemented so far and these performances have to be confirmed for a more realistic language including lists, user-defined data types and pattern matching.

Acknowledgements

We would like to thank Simon B. Jones for his careful proofreading of this paper. We are grateful to Jean-Pierre Banâtre, Michel Lemaître, David Schmidt and Philip Wadler for commenting on an earlier draft of this paper.

References

[1] W.H. Burge, "Recursive Programming Techniques", Addison-Wesley, 1975.

[2] L. Cardelli, "Compiling a Functional Language", Proc. of ACM Symp. on Lisp and Functional Prog., pp. 208-226, 1984.

[3] G. Cousineau, P.-L. Curien, M. Mauny, "The Categorical Abstract Machine", Science of Computer Programming, Vol.8, pp.173-202, 1987.

[4] H.B. Curry, R. Feys, W. Craig, "Combinatory Logic", Vol. I North-Holland, 1958, Second printing 1968.

[5] P. Fradet, "Compilation des langages fonctionnels par transformation de programmes", Thèse, Université de Rennes, Nov. 1988.

[6] R.J.M. Hughes, "Supercombinators: a new implementation method for applicative langages", Proc. of ACM Symp. on Lisp and Functional Prog., pp. 208-226, 1982.

[7] T. Johnsson, "Efficient Compilation of Lazy Evaluation", Proc. of the ACM SIGPLAN Symp. on Compiler Construction, SIGLAN Notices, Vol. 19, 6, pp. 58-69, 1984.

[8] T. Johnsson, "Target Code Generation from G-Machine Code", Proc. of the Workshop on Graph Reduction, LNCS Vol.279, pp.119-159, 1986.

[9] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, N. Adams "Orbit: an Optimizing compiler for Scheme", Proc. of the ACM SIGPLAN Symp. on Compiler Construction, pp. 219-233, 1986.

[10] P.J. Landin, "The Mechanical Evaluation of Expressions", Computer Journal, Vol.6, pp. 308-320, 1964.

[11] M. Lemaître, M. Castan, M.-H. Durand, G. Durrieu, B. Lecussan, "Mechanisms for Efficient Multiprocessor Combinator Reduction", Proc. of ACM Symp. on Lisp and Functional Prog., pp. 113-121, 1986.

[12] D. Lester, "The G-Machine as a Representation of Stack Semantics", Proc. of Functional Prog. Lang. and Comp. Arch., LNCS Vol. 274, pp. 46-59, 1987.

[13] G.D. Plotkin, "Call-by-name, Call-by-value and the λ-Calculus", Theoretical Computer Science, Vol. 1, pp. 125-159, 1975.

[14] J. C. Reynolds, "Definitional interpreters for higher-order programming languages", Proc. of ACM annual conference, Vol. 2, pp. 717-740, 1972.

[15] D.A. Turner, "A New Implementation Technique for Applicative Languages", Software-Practice and Experience, Vol. 9, pp. 31-49, 1979.

[16] M. Wand, "Deriving target code as a representation of continuation semantics", ACM Trans. on Prog. Lang. and Systems Vol 4, 3, pp. 496-517, 1982.

Appendix

We describe the evaluation of the result of the compilation of fact given at the end of section 3. The function is applied to **id** 1 since the new definition of fact takes two arguments: a continuation which is equal to **id** and an integer value. We show in parallel the evaluation as the execution of code on a stack machine and as the call-by-name reduction of a functional expression. In the left part of the figure we represent the state of the stack (the top being the leftmost element) before the execution of the corresponding instruction.

	Machine Code			**Functional Expression**
fact	dupl	2	id:1	**dupl$_2$ (push** 0 (eq$_c$ (cond$_c$ f0 f1))) **id** 1
	push	0	1: id:1	**push** 0 (eq$_c$ (cond$_c$ f0 f1)) 1 **id** 1
	eq$_c$		0:1: id:1	eq$_c$ (cond$_c$ f0 f1) 0 1 **id** 1
	cond$_c$	f0, f1	False: id:1	cond$_c$ f0 f1 False **id** 1
f1	push	1	id:1	**push** 1 (**dupl$_3$** (sub$_c$(**push** f2 (**jump** fact)))) **id** 1
	dupl	3	1:id:1	**dupl$_3$** (sub$_c$ (**push** f2 (**jump** fact))) 1 **id** 1
	sub$_c$		1:1:id:1	sub$_c$ (**push** f2 (**jump** fact)) 1 1 **id** 1
	push	f2	0:id:1	**push** f2 (**jump** fact) 0 **id** 1
	jump	fact	f2:0:id:1	**jump** fact f2 0 **id** 1
fact	dupl	2	f2:0:id:1	**dupl$_2$ (push** 0 (eq$_c$ (cond$_c$ f0 f1))) f2 0 **id** 1
	push	0	0:f2:0:id:1	**push** 0 (eq$_c$ (cond$_c$ f0 f1)) 0 f2 0 **id** 1
	eq$_c$		0:0:f2:0:id:1	eq$_c$ (cond$_c$ f0 f1) 0 0 f2 0 **id** 1
	cond$_c$	f0, f1	True:f2:0:id:1	cond$_c$ f0 f1 True f2 0 **id** 1
f0	push	1	f2:0:id:1	**push** 1 (**move$_{3,1}$** (**flsh$_1$ id**)) f2 0 **id** 1
	move	3,1	1:f2:0:id:1	**move$_{3,1}$** (**flsh$_1$ id**) 1 f2 0 **id** 1
	flsh	1	1: f2:1:id:1	**flsh$_1$ id** 1 f2 1 **id** 1
	id		f2:1:id:1	**id** f2 1 **id** 1
f2	dupl	3	1:id:1	**dupl$_3$** (mult$_c$ (**move$_{3,1}$** (**flsh$_1$ id**))) 1 **id** 1
	mult$_c$		1: 1:id:1	mult$_c$ (**move$_{3,1}$** (**flsh$_1$ id**)) 1 1 **id** 1
	move	3,1	1:id:1	**move$_{3,1}$** (**flsh$_1$ id**) 1 **id** 1
	flsh	1	1: id:1	**flsh$_1$ id** 1 **id** 1
	id		id:1	**id id** 1
	id		1	**id** 1
	result = 1			1