Compilation of Functional Languages by Program Transformation

PASCAL FRADET
and
DANIEL LE MÉTAYER
IRISA/INRIA

One of the most important issues concerning functional languages is the efficiency and the correctness of their implementation. We focus on sequential implementations for conventional von Neumann computers. The compilation process is described in terms of program transformations in the functional framework. The original functional expression is transformed into a functional term that 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 correctness proofs much simpler. As far as efficiency is concerned, this approach is promising since many optimizations can be described and formally justified in the functional framework.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Functional Programming; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Languages, Verification

Additional Key Words and Phrases: Combinators, continuations, program transformation

1. INTRODUCTION

The design of efficient and correct implementations is a crucial issue concerning functional languages. Functional languages are still less efficient than traditional imperative languages, but much progress has been made within the last few years [6, 7, 12]. However, functional languages possess a definite advantage over imperative languages: their semantic elegance makes program correctness proofs and program transformations easier to establish. In this paper, we show that these properties can be turned into account to build a correct and efficient implementation based on program transformation.

The implementation of functional languages is generally described in terms of an abstract machine [5, 7, 9, 12, 16], reducing either the source functional program or a compiled version of this program. The abstract machine itself is

Authors' address: IRISA, INRIA Campus de Beaulieu, 35042 Rennes Cedex, France.

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 [18, 19, 22],
(3) proof that the implementation of the abstract machine is correct.

Step (1) is generally easy because the compilation produces a functional expression. Step (2), however, involves the operational description of a specific machine that makes proofs much more difficult to tackle. Step (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 involved a nontrivial 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 state of the "machine" is the expression itself, and its evolution is specified by the definition of basic functions (combinators). The functional expressions produced are of the form $F\ G\ S_1 \cdots S_n$, where F is a basic function that behaves like a machine instruction operating on the stack $S_1, \ldots, S_n$ and where G is an expression representing the rest of the code. The code produced can be seen as generic code for a traditional machine and can be specialized for a particular processor by a straightforward macroexpansion.

This approach has interesting payoffs as far as correctness proofs are concerned. We can say that we provide a proof of the whole implementation process. Furthermore, this approach allows the derivation of efficient code because various optimization techniques can be systematically applied at each level of transformation. In our formalism these optimizations (some of them well known by compiler constructors) are expressed as simple transformation rules that can easily be justified.

The execution of a functional program involves two main tasks:

(1) searching for the next expression to be reduced according to a specific computation rule,
(2) 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 using continuations. The resulting expressions can be evaluated from left to right by successively applying the head function. We describe the compilation of call-by-value and call-by-name and give a sketch of the correctness proofs. Many improvements are possible on the expressions produced by this first step; Section 3 introduces the most important ones and applies them to an example. In particular, we

describe a transformation rule corresponding to the well-known tail-recursion optimization. The compilation of environment management, presented in Section 4, is done by an abstraction algorithm in the same spirit as [1], [17], and [30]. This algorithm, independent of the computation rule, is based on a set of combinators acting like traditional machine instructions (move, push, ...). A sketch of its correctness proof is given. We describe in Section 5 various optimizations that allow a drastic improvement of the code; some of them are straightforward translations into the functional framework of well-known compiler optimization techniques such as peephole optimizations [2]. Section 6 describes a representation of closures and the implementation of call-by-need. In conclusion, we provide figures illustrating the efficiency of the produced code, and we compare the method with previous work in two respects: the compiler derivation approach and the implementation choices.

2. COMPILATION OF THE COMPUTATION RULE

Our source language is an extended λ-calculus with the following syntax:

$$E ::= x \mid k \mid \text{opm } E_1 \cdots E_m \mid \text{cond } E_1\ E_2\ E_3 \mid E_1\ E_2 \mid \lambda x.E_1 \mid \text{letrec } f = \lambda x.E_1,$$

where E_i are expressions, x is a variable, k is a basic constant, and opm represents a strict primitive operator of arity m; the primitive cond is the only nonstrict operator. The definition of factorial in this language is

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

The λ-calculus has been chosen because it is a simple language that can easily express any functional program. So if we have an implementation of the λ-calculus, we can implement high-level functional programs just by translating them into λ-expressions.

The evaluation of an expression involves a repeated search for the next redex; for example, in a call-by-value semantics, the first operation to execute in the body of fact is eq, then cond, then either 1 or sub, and so on. The compilation of the computation rule should produce an expression that can be evaluated by systematic rewriting of the head function at each stage. We could have chosen another particular function that can be found without recursive search; our choice results naturally from the analogy with machine code.

We first consider a call-by-value semantics for the language; the compilation of call-by-name is given at the end of the section. Basically, we have to invert the order of subexpressions in a composition: the call-by-value evaluation of a composition $(E_1\ E_2)$ entails the evaluation of E_2, then the evaluation of E_1, and finally the application of the result of E_1 to the result of E_2 (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 into an expression taking a continuation as argument and applying it to the result of its evaluation. In the same way, we define a new operator opm_c for each operator opm such that

$$\text{opm}_c\ C\ E_1 \cdots E_m = C\ (\text{opm } E_1 \cdots E_m)$$

and a new conditional operator cond_c, which takes two continuations (here E_2 and E_3)

$$\text{cond}_c\ E_2\ E_3\ E_1 = \text{cond}\ E_1\ E_2\ E_3.$$

Let us now take an example to illustrate the goal of the transformation described in this section. We describe the call-by-value evaluation of the application $(\lambda x.\text{add}\ x\ (\text{add}\ x\ 1))\ 3$. The operator applied at each step is underlined.

$(\lambda x.\text{add}\ x\ (\text{add}\ x\ 1))\ 3$

$\rightarrow (\underline{\lambda x}.\text{add}\ x\ (\text{add}\ x\ 1))\ 3$	searching for the next expression to reduce,
$\rightarrow \text{add}\ 3\ (\text{add}\ 3\ 1)$	β-reduction,
$\rightarrow \text{add}\ 3\ (\underline{\text{add}}\ 3\ 1)$	searching for the next expression to reduce,
$\rightarrow \underline{\text{add}}\ 3\ 4$	first addition,
$\rightarrow 7$	second addition.

Our transformation (followed by a simple optimization) applied to $\lambda x.\text{add}\ x\ (\text{add}\ x\ 1)$ will produce

$$\lambda x.\text{add}_c\ (\text{add}_c\ \mathbf{id}\ x)\ x\ 1, \qquad \text{where } \mathbf{id} = \lambda c.c.$$

In this expression $(\text{add}_c\ \mathbf{id}\ x)$ is the continuation of the leftmost add_c (it will be applied to its result), and **id** is the continuation of the rightmost add_c. The application of this new function to the value 3 can be evaluated by systematic reduction of the head function:

$(\underline{\lambda x}.\text{add}_c\ (\text{add}_c\ x)\ x\ 1)\ 3$

$\rightarrow \underline{\text{add}_c}\ (\text{add}_c\ \mathbf{id}\ 3)\ 3\ 1,$	β-reduction,
$\rightarrow \underline{\text{add}_c}\ \mathbf{id}\ 3\ 4,$	first addition,
$\rightarrow \underline{\mathbf{id}}\ 7,$	second addition,
$\rightarrow 7,$	application of the continuation **id**.

This example shows in which sense the computation rule has indeed been compiled: there is no dynamic searching for the next redex during the evaluation.

Before giving a formal and comprehensive account of the transformation rules for the compilation of call-by-value, let us introduce them in a more intuitive way. We should keep in mind that the goal of the transformation ($\mathcal{T}'$ for call-by-value) is roughly to rearrange subexpressions according to the order in which they would be evaluated by call-by-value. To this aim, the transformed expressions take an extra argument representing a continuation function to apply to the result of their evaluation (function calls will be of the form: $F\ C\ E_1 \cdots E_n$).

Let us first consider expressions E, which do not require any evaluation (i.e., weak head normal forms): these expressions are constants, variables, and λ-abstractions. Their evaluation consists only in returning their value, so they should be transformed into something like $\lambda c.c\ E'$ which means that the continuation can be applied immediately. If E is a constant or a variable, its value is obviously itself; so we have

$$\mathcal{T}'(x) = \lambda c.c\ x$$
$$\mathcal{T}'(k) = \lambda c.c\ k.$$

If E is a λ-abstraction $\lambda x.F$, this expression itself should be transformed. So E' must take a continuation and is of the form $\lambda c.\lambda x.F'$, where F' represents the

(transformed) expression F applied to the continuation c. Thus we have $F' = \mathscr{T}'(F)\ c$, and the rule for λ-abstraction is

$$\mathscr{T}'(\lambda x.F) = \lambda c.c\ (\lambda c.\lambda x.\mathscr{T}'(F)\ c).$$

The only case where some evaluation is required is application. Let us first consider an expression $E_1\ E_2$, where E_1 is not a primitive operator. We should have as previously $\mathscr{T}'(E_1\ E_2) = \lambda c.E'$. Call-by-value indicates that E_2 should be evaluated first; so $E' = \mathscr{T}'(E_2)\ E''$. After the evaluation of E_2, call-by-value imposes the evaluation of E_1 and the application of the result of E_1 to the result of E_2; E_1 should return a function to be applied to the value of E_2 with continuation c. So we must have $E'' = \mathscr{T}'(E_1)$ **apply** c where **apply** $f\ c = f\ c$. In this context **apply** is equivalent to **id**, and we have

$$\mathscr{T}'(E_1\ E_2) = \lambda c.\mathscr{T}'(E_2)\ (\mathscr{T}'(E_1)\ \mathbf{id}\ c).$$

We can check that the rules for λ-abstraction and application are coherent. According to the previous rules we have

$$\begin{aligned}\mathscr{T}'((\lambda x.F)\ E) &= \lambda c.\mathscr{T}'(E)\ (\mathscr{T}'(\lambda x.F)\ \mathbf{id}\ c)\\ &= \lambda c.\mathscr{T}'(E)\ ((\lambda c.c\ (\lambda c.\lambda x.\mathscr{T}'(F)\ c))\ \mathbf{id}\ c)\\ &= \lambda c.\mathscr{T}'(E)\ ((\lambda c.\lambda x.\mathscr{T}'(F)\ c)\ c)\\ &= \lambda c.\mathscr{T}'(E)\ (\lambda x.\mathscr{T}'(F)\ c).\end{aligned}$$

The application of this expression to the continuation C is evaluated as follows:

$$\begin{aligned}&(\lambda c.\mathscr{T}'(E)\ (\lambda x.\mathscr{T}'(F)\ c))\ C && c \text{ is bound to the global continuation } C,\\ &\quad\rightarrow \mathscr{T}'(E)\ (\lambda x.\mathscr{T}'(F)\ C) && E \text{ is evaluated first,}\\ &\quad\rightarrow (\lambda x.\mathscr{T}'(F)\ C)\ N_E && \text{then } \lambda x.\mathscr{T}'(F)\ C \text{ is applied to its result } N_E,\\ &\quad\rightarrow \mathscr{T}'(F)[N_E/x]\ C && \text{and } \mathscr{T}'(F) \text{ is evaluated with } x \text{ bound to the value of } E.\end{aligned}$$

Let us remark, however, that we must give the general rule for $\mathscr{T}'(E_1\ E_2)$ because E_1 may not be a λ-abstraction: it may be itself an application or a variable.

The transformation of a primitive operator can be explained along the same lines. In the expression opm $E_1 \cdots E_m$, call-by-value indicates the evaluation of $E_m, E_{m-1}, \ldots, E_1$, then the application of opm; so we have

$$\mathscr{T}'(\text{opm}\ E_1 \cdots E_m) = \lambda c.\mathscr{T}'(E_m)\ (\mathscr{T}'(E_{m-1})\ (\cdots(\mathscr{T}'(E_1)\ (\text{opm}_c\ c))\cdots)).$$

$\mathscr{T}'(E_{i-1})\ (\cdots(\mathscr{T}'(E_1)\ (\text{opm}_c\ c))\cdots)$ is the continuation of $\mathscr{T}'(E_i)$ (for all $i > 1$), $\text{opm}_c\ c$ is the continuation of $\mathscr{T}'(E_1)$, and c is the global continuation.

A special rule is attached to the conditional operator because it is strict in its first argument only. Furthermore, cond takes two continuations and chooses among them according to the value of its argument; so we have

$$\mathscr{T}'(\text{cond}\ E_1\ E_2\ E_3) = \lambda c.\mathscr{T}'(E_1)\ (\text{cond}_c\ (\mathscr{T}'(E_2)\ c)\ (\mathscr{T}'(E_3)\ c)).$$

(E_2 and E_3 take the global continuation since they represent the result of the whole expression.)

The following rules gather the transformations described above. The rule for letrec expressions can be deduced from the other rules and the definition of the

call-by-value recursion combinator $\boldsymbol{Y}_v$ [25]: $\boldsymbol{Y}_v = \lambda f.(\lambda g.f\ (\lambda v.g\,g\,v))\ (\lambda g.f\ (\lambda v.g\,g\,v))$.

$$
\begin{array}{lll}
\mathcal{T}\ (x) & = \lambda c.c\ x & (\mathcal{T}\ 1)\\
\mathcal{T}'(k) & = \lambda c.c\ k & (\mathcal{T}\ 2)\\
\mathcal{T}\ (\text{opm}\ E_1 \cdots E_m) & = \lambda c.\mathcal{T}'(E_m)\ (\mathcal{T}\ (E_{m-1})\ (\cdots(\mathcal{T}'(E_1)\ (\text{opm}_c\ c))\cdots)) & (\mathcal{T}\ 3)\\
\mathcal{T}'(\text{cond}\ E_1\ E_2\ E_3) & = \lambda c.\mathcal{T}'(E_1)\ (\text{cond}_c\ (\mathcal{T}'(E_2)\ c)\ (\mathcal{T}'(E_3)\ c)) & (\mathcal{T}'4)\\
\mathcal{T}'(E_1\ E_2) & = \lambda c.\mathcal{T}'(E_2)\ (\mathcal{T}'(E_1)\ \textbf{id}\ c) & (\mathcal{T}'5)\\
\mathcal{T}'(\lambda x.E) & = \lambda c.c\ (\lambda c.\lambda x.\ \mathcal{T}'(E)\ c) & (\mathcal{T}'6)\\
\mathcal{T}'(\text{letrec}\ f = \lambda x.E) & = \lambda c.c\ (\text{letrec}\ f = \lambda c.\lambda x.\mathcal{T}'(E)\ c). & (\mathcal{T}'7)
\end{array}
$$

Let us return to our introductory example to illustrate the transformation $\mathcal{T}$:

$$
\begin{array}{ll}
\mathcal{T}'(\lambda x.\text{add}\,x\,(\text{add}\,x\,1)) & \\
= \lambda c.(\lambda c.\lambda x.\mathcal{T}'(\text{add}\,x\,(\text{add}\,x\,1))\,c) & (\mathcal{T}'6)\\
= \lambda c.(\lambda c.\lambda x.(\lambda c.\mathcal{T}'(\text{add}\,x\,1)\,(\mathcal{T}\ (x)\,(\text{add}_c\,c)))\,c) & (\mathcal{T}\ 3)\\
= \lambda c.(\lambda c.\lambda x.(\lambda c.(\lambda c.\mathcal{T}\ (1)\,(\mathcal{T}'(x)\,(\text{add}_c\,c)))\,(\mathcal{T}'(x)\,(\text{add}_c\,c)))\,c) & (\mathcal{T}'3)\\
= \lambda c.(\lambda c.\lambda x.(\lambda c.(\lambda c.(\lambda c.c\,1)\,((\lambda c.c\,x)\,(\text{add}_c\,c)))\,((\lambda c.c\,x)\,(\text{add}_c\,c)))\,c) & (\mathcal{T}\ 1),(\mathcal{T}\ 2)
\end{array}
$$

We take the convention that top-level continuations are always **id**; exploiting this property and applying all possible β-reductions we get $\lambda x.\text{add}_c\ (\text{add}_c\ \textbf{id}\ x)\ x\ 1$; we have shown at the beginning of the section how this expression can be computed by systematic reduction of the first function. Section 3 contains a full treatment of optimization rules.

In order to establish the correctness of this first transformation, we have to show that the evaluation of the original expression by call-by-value (CBV) and the evaluation of the resulting expression by a computation rule modeling our intuitive notion of "systematic reduction of the first operator" (FIRST) yield the same result. To this aim, we describe the two computation rules in the form of deductive systems in the style of [21].

Our system is made of (1) a set of axioms $E_1 \rightarrow_{\text{CR}} E_2$, where CR indicates computational rule, which can reduce E_1 into E_2 in one step, and (2) inference rules of the form

$$\frac{E_1 \rightarrow_{\text{CR}} E_2}{\exp(E_1) \rightarrow_{\text{CR}} \exp(E_2)}$$

where $\exp(E_1)$ (respectively, $\exp(E_2)$) denotes an expression (possibly) involving E_1 (respectively, E_2), meaning that if E_1 can be reduced into E_2 in one step by CR, then $\exp(E_1)$ can be reduced into $\exp(E_2)$ in one step by CR. In the following, E_i denotes any expression in our language, whereas N_i denotes a weak head normal form. In the remainder of the paper, the symbol $\equiv$ denotes syntactic equality. We first give our definition of call-by-value.

Axioms

$\text{opm}\ N_1 \cdots N_m \rightarrow_{\text{CBV}} r$

with r the result of applying opm to $N_1 \cdots N_m$

$\text{cond True}\ E_1\ E_2 \rightarrow_{\text{CBV}} E_1$

$\text{cond False}\ E_1\ E_2 \rightarrow_{\text{CBV}} E_2$

$\text{cond}\ NE_1\ E_2 \rightarrow_{\text{CBV}} \perp, \qquad \text{if } N \not\equiv \text{True and } N \not\equiv \text{False}$

$(\lambda x.\ E)\ N \rightarrow_{\text{CBV}} E[N/x]$

where $E[N/x]$ denotes the expression E where free occurrences of x are replaced by N

$$(\text{letrec } f = \lambda x.E)\ N \rightarrow_{\text{CBV}} E[(\text{letrec } f = \lambda x.E)/f][N/x]$$

$$N\ E \rightarrow_{\text{CBV}} \bot, \quad \text{if } N \equiv k, \quad N \equiv x, \quad \text{or } N \equiv \bot.$$

Inference Rules

$$\frac{E_i \rightarrow_{\text{CBV}} E_i'}{\text{opm } E_1 \cdots E_i\ N_{i+1} \cdots N_m \rightarrow_{\text{CBV}} \text{opm } E_1 \cdots E_i'\ N_{i+1} \cdots N_m}$$

$$\frac{E_i \rightarrow_{\text{CBV}} \bot}{\text{opm } E_1 \cdots E_i\ N_{i+1} \cdots N_m \rightarrow_{\text{CBV}} \bot}$$

$$\frac{E_1 \rightarrow_{\text{CBV}} E_1'}{\text{cond } E_1\ E_2\ E_3 \rightarrow_{\text{CBV}} \text{cond } E_1'\ E_2\ E_3} \qquad \frac{E_1 \rightarrow_{\text{CBV}} \bot}{\text{cond } E_1\ E_2\ E_3 \rightarrow_{\text{CBV}} \bot}$$

$$\frac{E_2 \rightarrow_{\text{CBV}} E_2'}{E_1\ E_2 \rightarrow_{\text{CBV}} E_1\ E_2'} \qquad \frac{E_2 \rightarrow_{\text{CBV}} \bot}{E_1\ E_2 \rightarrow_{\text{CBV}} \bot}$$

$$\frac{E_1 \rightarrow_{\text{CBV}} E_1'}{E_1\ N \rightarrow_{\text{CBV}} E_1'\ N}.$$

Here, $\bot$ represents the undefined value; in order to deal with $\bot$, $\mathcal{Y}$ is extended with the rule $\mathcal{Y}(\bot) = \bot$. We must introduce $\bot$ in order to be able to describe precisely the number of reduction steps required by a computation rule.

We describe now the system corresponding to the FIRST computation rule; this deductive system possesses a noteworthy property: it is defined only in terms of axioms. This is a formalization of the fact that FIRST involves no dynamic search of the next redex.

Axioms

$$\text{opm}_c\ C E_1 \cdots E_m \cdots E_n \rightarrow_{\text{FIRST}} C\ r\ E_{m+1} \cdots E_n$$

with r the result of applying opm to $E_1 \cdots E_m$ and $r \not\equiv \bot$

$$\text{opm}_c\ C\ E_1 \cdots E_m \cdots E_n \rightarrow_{\text{FIRST}} \bot, \quad \text{if } r \equiv \bot$$

$$\text{cond}_c\ E_1\ E_2\ \text{True}\ E_4 \cdots E_n \rightarrow_{\text{FIRST}} E_1\ E_4 \cdots E_n$$

$$\text{cond}_c\ E_1\ E_2\ \text{False}\ E_4 \cdots E_n \rightarrow_{\text{FIRST}} E_2\ E_4 \cdots E_n$$

$$\text{cond}_c\ E_1\ E_2\ E_3\ E_4 \cdots E_n \rightarrow_{\text{FIRST}} \bot, \quad \text{if } E_3 \not\equiv \text{True and } E_3 \not\equiv \text{False}$$

$$(\lambda x.E_1)\ E_2\ E_3 \cdots E_n \rightarrow_{\text{FIRST}} E_1[E_2/x]\ E_3 \cdots E_n$$

$$(\text{letrec } f = \lambda x.E_1)\ E_2\ E_3 \cdots E_n \rightarrow_{\text{FIRST}} E_1[(\text{letrec } f = \lambda x.E_1)/f][E_2/x]\ E_3 \cdots E_n$$

$$E_1 \cdots E_n \rightarrow_{\text{FIRST}} \bot, \quad \text{otherwise.}$$

For any computation rule CR (here CR = CBV or CR = FIRST) the result of the evaluation (normal form) of E by CR is denoted by $\text{CR}(E)$; if the evaluation does not terminate, $\text{CR}(E)$ is undefined. $N_{\text{CR}}(E)$ represents the number of reduction steps by CR to reach the normal form of E, and $\text{CR}_i(E)$ is the expression obtained after i reduction steps by CR. We first state two important lemmas relating the evaluation of $\mathcal{Y}(E)$ and $\mathcal{Y}(\text{CBV}_1(E))$ by FIRST. In the following lemmas and properties, E and C represent closed λ-expressions.

LEMMA 1. $(\forall E, C)\ FIRST(\mathcal{Z}'(E)\ C) \equiv FIRST(\mathcal{Z}'(CBV_1(E))\ C)$.

LEMMA 2. $(\forall E, C)$. *If E is not in normal form, then*

$$N_{FIRST}(\mathcal{Z}'(CBV_1(E))C) + 1 \leq N_{FIRST}(\mathcal{Z}'(E)C) \leq N_{FIRST}(\mathcal{Z}'(CBV_1(E))C) + k$$

where k is a constant depending on the arities of primitive functions.

PROOF. These two lemmas are proved by structural induction on the expressions; in fact we prove slightly generalized forms of these lemmas where the continuation C is replaced by a list of expressions $C_1 \cdots C_n$; this generalization is required by the induction itself. The proofs are based on a routine inspection of the different cases [10]. Each case is treated in a mechanical fashion using the following assumptions and properties:

(1) opm is strict and returns a basic value,
(2) $E, C_1, \ldots, C_n$ are closed expressions,
(3) if E is a normal form ($\not\equiv \perp$), then $\mathcal{Z}'(E) \equiv \lambda c.c\ E'$,
(4) if $\mathcal{Z}'(E_2) \equiv \lambda c.c\ E_2'$ (i.e., E_2 is a normal form $\not\equiv \perp$), then $\mathcal{Z}'(E_1[E_2/x]) \equiv \mathcal{Z}'(E_1)[E_2'/x]$.

This last property is easily proved by structural induction on E_1 using only definitions of $\mathcal{Z}'$ and substitution.

Although not tricky, these proofs have to examine too many cases to be presented here in complete detail. To give their flavor, we describe the case $E \equiv E_1\ E_2$, where E_2 is not a normal form:

PROOF OF LEMMA 1

$\text{FIRST}(\mathcal{Z}'(E_1\ E_2)\ C_1 \cdots C_n)$
$\equiv \text{FIRST}((\lambda c.\mathcal{Z}'(E_2)\ (\mathcal{Z}'(E_1)\ \mathbf{id}\ c))\ C_1 \cdots C_n)$ [definition of $\mathcal{Z}'$]
$\equiv \text{FIRST}(\mathcal{Z}'(E_2)\ (\mathcal{Z}'(E_1)\ \mathbf{id}\ C_1)\ C_2 \cdots C_n)$ [definition of FIRST]
$\text{FIRST}(\mathcal{Z}'(\text{CBV}_1(E_1\ E_2))\ C_1 \cdots C_n)$
$\equiv \text{FIRST}(\mathcal{Z}'(E_1\ \text{CBV}_1(E_2))\ C_1 \cdots C_n)$ [definition of CBV_1]
$\equiv \text{FIRST}((\lambda c.\mathcal{Z}'(\text{CBV}_1(E_2))\ (\mathcal{Z}'(E_1)\ \mathbf{id}\ c))\ C_1 \cdots C_n)$ [definition of $\mathcal{Z}'$]
$\equiv \text{FIRST}(\mathcal{Z}'(\text{CBV}_1(E_2))\ (\mathcal{Z}'(E_1)\ \mathbf{id}\ C_1)\ C_2 \cdots C_n)$ [definition of FIRST]
$\equiv \text{FIRST}(\mathcal{Z}'(E_2)\ (\mathcal{Z}'(E_1)\ \mathbf{id}\ C_1)\ C_2 \cdots C_n)$ [induction hypothesis].

Hence Lemma 1. □

PROOF OF LEMMA 2

$$N_{\text{FIRST}}(\mathcal{Z}'(\text{CBV}_1(E_1\ E_2))\ C_1 \cdots C_n) = 1 + N_{\text{FIRST}}(\mathcal{Z}'(\text{CBV}_1(E_2))\ (\mathcal{Z}'(E1)\ \mathbf{id}\ C_1)\ C_2 \cdots C_n)$$

and

$$N_{\text{FIRST}}(\mathcal{Z}'(E_1\ E_2)\ C_1 \cdots C_n) = 1 + N_{\text{FIRST}}(\mathcal{Z}'(E_2)\ (\mathcal{Z}'(E_1)\ \mathbf{id}\ C_1)\ C_2 \cdots C_n)$$

so

$$N_{\text{FIRST}}(\mathcal{Z}'(\text{CBV}_1(E_1\ E_2))\ C) + 1 \leq N_{\text{FIRST}}(\mathcal{Z}'(E_1\ E_2)\ C) \leq N_{\text{FIRST}}(\mathcal{Z}'(\text{CBV}_1(E_1\ E_2))\ C) + k$$

by induction hypothesis on E_2, which is not a normal form. □

Lemma 2 allows us to derive a first correctness property.

PROPERTY 3. *The evaluation of E by CBV terminates if and only if there exists some C such that the evaluation of* $\mathcal{T}'(E)\ C$ *by FIRST terminates.*

PROOF. If the evaluation of E by CBV terminates, then there exists n such that $\mathrm{CBV}_n(E) \equiv \mathrm{CBV}(E)$. Since $\mathrm{CBV}_n(E)$ is a normal form (i.e., $\perp$, a constant, or a λ-abstraction), $\mathcal{T}'(\mathrm{CBV}_n(E)) \equiv \perp$ or $\mathcal{T}'(\mathrm{CBV}_n(E)) \equiv \lambda c.c\ E'$ with E' a normal form itself; so $N_{\mathrm{FIRST}}(\mathcal{T}'(\mathrm{CBV}_n(E))\ \mathbf{id}) \leq 2$. Applying Lemma 2 n times gives us

$$N_{\mathrm{FIRST}}(\mathcal{T}'(E)\ \mathbf{id}) \leq N_{\mathrm{FIRST}}(\mathcal{T}'(\mathrm{CBV}_1(E))\ \mathbf{id}) + k \leq \cdots \leq N_{\mathrm{FIRST}}(\mathcal{T}'(\mathrm{CBV}_n(E))\ \mathbf{id}) + n \times k \leq 2 + n \times k.$$

So there exists C (= **id**) such that the evaluation of $\mathcal{T}'(E)\ C$ by FIRST terminates since a normal form is reached in less than $2 + n \times k$ steps.

If there exists some C such that $\mathrm{FIRST}(\mathcal{T}'(E)\ C)$ terminates, then there exists n such that $N_{\mathrm{FIRST}}(\mathcal{T}'(E)\ C) = n$. Let us assume that $\mathrm{CBV}(E)$ does not terminate; then $(\forall p \geq 0)\ \mathrm{CBV}_p(E)$ is not a normal form, and we can apply $p > n$ times Lemma 2 to get

$$n = N_{\mathrm{FIRST}}(\mathcal{T}'(E)\ C) \geq N_{\mathrm{FIRST}}(\mathcal{T}'(\mathrm{CBV}_1(E))\ C) + 1 \geq \cdots \geq N_{\mathrm{FIRST}}(\mathcal{T}'(\mathrm{CBV}_p(E))\ C) + p$$

which is a contradiction. So $\mathrm{CBV}(E)$ terminates: it must reach a normal form in less than n steps. □

We can now introduce the general correctness property.

PROPERTY 4. *If the evaluation of E by CBV terminates, then*

$$(\forall E, C)\ FIRST(\mathcal{T}'(E)\ C) \equiv FIRST(\mathcal{T}'(CBV(E))\ C).$$

PROOF. The proof of Property 4 is carried out by induction on $N_{\mathrm{CBV}}(E)$:

(1) $N_{\mathrm{CBV}}(E) = 0 \Rightarrow \mathrm{CBV}(E) \equiv E$, and the property is trivially true.

(2) Let us assume that the property holds for any E' such that $N_{\mathrm{CBV}}(E') < N_{\mathrm{CBV}}(E)$; then

$$\begin{aligned}
&\mathrm{FIRST}(\mathcal{T}'(\mathrm{CBV}(E))\ C)\\
&\quad \equiv \mathrm{FIRST}(\mathcal{T}'(\mathrm{CBV}(\mathrm{CBV}_1(E)))\ C) && \text{[definition of CBV}_1\text{]}\\
&\quad \equiv \mathrm{FIRST}(\mathcal{T}'(\mathrm{CBV}_1(E))\ C) && \text{[induction hypothesis]}\\
&\quad \equiv \mathrm{FIRST}(\mathcal{T}'(E)\ C) && \text{[Lemma 1].}
\end{aligned}$$

This concludes the sketch of the correctness proof of the transformation $\mathcal{T}$. □

Two interesting corollaries follow directly from Property 4.

COROLLARY 5. $(\forall E)\ CBV(E) \equiv k \Rightarrow FIRST(\mathcal{T}'(E)\ \mathbf{id}) \equiv k.$

So when the normal form of the expression by call-by-value is a basic value, the same result is obtained by the application of FIRST to the transformed expression with continuation **id**.

COROLLARY 6. $(\forall E)\ CBV(E) \equiv \lambda x.F \Rightarrow FIRST(\mathcal{T}'(E)\ \mathbf{id}) \equiv \lambda c.\lambda x.\mathcal{T}'(F)\ c.$

This corollary deals with expressions whose normal form is a λ-abstraction: in such cases, the evaluation by FIRST of the transformed expression yields the corresponding transformed λ-abstraction.

For the time being, we have only described the compilation of call-by-value. Other evaluation orders can be compiled by our method, and we now describe the compilation of call-by-name (and therefore call-by-need, since there is no difference at this level). The transformation rules are described as follows:

$$
\begin{array}{lll}
\mathcal{J}'(x) & = x & (\mathcal{J}'1)\\
\mathcal{J}'(k) & = \lambda c.c\ k & (\mathcal{J}'2)\\
\mathcal{J}'(\text{opm}\ E_1 \cdots E_m) & = \lambda c.\ \mathcal{J}'(E_m)\ (\mathcal{J}'(E_{m-1})\ (\cdots(\mathcal{J}'(E_1)\ (\text{opm}_c\ c))\cdots)) & (\mathcal{J}'3)\\
\mathcal{J}'(\text{cond}\ E_1\ E_2\ E_3) & = \lambda c.\ \mathcal{J}'(E_1)\ (\text{cond}_c\ (\mathcal{J}'(E_2)\ c)\ (\mathcal{J}'(E_3)\ c)) & (\mathcal{J}'4)\\
\mathcal{J}'(E_1\ E_2) & = \lambda c.\ \mathcal{J}'(E_1)\ \mathbf{id}\ c\ \mathcal{J}'(E_2) & (\mathcal{J}'5)\\
\mathcal{J}'(\lambda x.E) & = \lambda c.c\ (\lambda c.\lambda x.\ \mathcal{J}'(E)\ c) & (\mathcal{J}'6)\\
\mathcal{J}'(\text{letrec}\ f = \lambda x.E) & = \text{letrec}\ f = \lambda c.c\ (\lambda c.\lambda x.\ \mathcal{J}'(E)\ c). & (\mathcal{J}'7)
\end{array}
$$

The two main departures from transformation $\mathcal{T}'$ appear in rules $(\mathcal{J}'1)$ and $(\mathcal{J}'5)$. By call-by-name, arguments are not evaluated before a function call; this explains rule $(\mathcal{J}'1)$: the argument is evaluated in the body of the function, whereas it is just returned by call-by-value. Rule $(\mathcal{J}'5)$ expresses the fact that the argument E_2 is passed unevaluated to the result of E_1 (since FIRST is intended to be the evaluation rule). As we did for call-by-value, we can check that the rules for λ-abstraction and application are coherent:

$$
\begin{array}{lll}
\mathcal{J}'((\lambda x.F)\ E) & \equiv \lambda c.\mathcal{J}'(\lambda x.F)\ \mathbf{id}\ c\ \mathcal{J}'(E) & (\mathcal{J}'5)\\
& \equiv \lambda c.(\lambda c.c\ (\lambda c.\lambda x.\ \mathcal{J}'(F)\ c))\ \mathbf{id}\ c\ \mathcal{J}'(E) & (\mathcal{J}'6)\\
& = \lambda c.(\lambda c.\lambda x.\mathcal{J}'(F)\ c)\ c\ \mathcal{J}'(E)\\
& = \lambda c.(\lambda x.\mathcal{J}'(F)\ c)\ \mathcal{J}'(E).
\end{array}
$$

The application of this expression to a continuation C is evaluated as follows:

$$
\begin{array}{ll}
(\lambda c.(\lambda x.\mathcal{J}'(F)\ c)\ \mathcal{J}'(E))\ C & c \text{ is bound to the global continuation } C,\\
\quad \rightarrow_{\text{FIRST}}\ (\lambda x.\mathcal{J}'(F)\ C)\ \mathcal{J}'(E) & \lambda x.\ \mathcal{J}'(F)\ C \text{ is applied to its unevaluated argument,}\\
\quad \rightarrow_{\text{FIRST}}\ \mathcal{T}'(F)[\mathcal{J}'(E)/x]\ C & \mathcal{J}'(F) \text{ is evaluated with } x \text{ bound to } \mathcal{J}'(E).
\end{array}
$$

Let us illustrate the transformation $\mathcal{J}'$ by compiling the simple expression previously used with call-by-value:

$$
\begin{array}{lll}
\mathcal{J}'(\lambda x.\ \text{add}\ x\ (\text{add}\ x\ 1)) & & \\
\quad \equiv \lambda c.c\ (\lambda c.\lambda x.\mathcal{J}'(\text{add}\ x\ (\text{add}\ x\ 1))\ c) & & (\mathcal{J}'6)\\
\quad \equiv \lambda c.c\ (\lambda c.\lambda x.(\lambda c.\mathcal{J}'(\text{add}\ x\ 1)\ (\mathcal{J}'(x)\ (\text{add}_c\ c)))\ c) & & (\mathcal{J}'3)\\
\quad \equiv \lambda c.c\ (\lambda c.\lambda x.(\lambda c.(\lambda c.\mathcal{J}'(1)\ (\mathcal{J}'(x)\ (\text{add}_c\ c)))\ (\mathcal{J}'(x)\ (\text{add}_c\ c)))\ c) & & (\mathcal{J}'3)\\
\quad \equiv \lambda c.c\ (\lambda c.\lambda x.(\lambda c.(\lambda c.(\lambda c.c\ 1)\ (x\ (\text{add}_c\ c)))\ (x(\text{add}_c c)))\ c). & & (\mathcal{J}'1), (\mathcal{J}'2)
\end{array}
$$

Applying the same simplifications as above, we get

$$\lambda x.x\ (\text{add}_c\ (x\ (\text{add}_c\ \mathbf{id})))\ 1.$$

So x is bound to the argument and is evaluated first, then the first addition (with 1) can be computed; its continuation is $(x\ (\text{add}_c\ \mathbf{id}))$; x is reevaluated (in the case of call-by-need, it just returns the previously evaluated value (see Section 6)); and the second addition can be performed.

We do not dwell here on the correctness proof of the transformation $\mathcal{M}'$ since it is very similar to the proof of $\mathcal{Y}'$; the interested reader may refer to Fradet [10] for a full description of the proofs.

3. SIMPLIFICATION RULES

As we already noticed, the expressions produced by the application of transformations $\mathcal{Y}'$ and $\mathcal{M}'$ are rather bulky and may be simplified greatly by some trivial transformation rules. We present in this section some methods for improving the result of our transformations, and we relate them to well-known compiler optimization techniques.

3.1 Simplification by β-Reduction

It is clear from the examples above that $\mathcal{Y}'$ and $\mathcal{M}'$ introduce many new applications of λ-expressions that can be reduced by β-reduction. In order to ensure the termination of this reduction, we apply the following strategy: a β-reduction is performed only if it strictly decreases the size of the expression. This restriction does not forbid most of the simplifications since all the compositions introduced by our transformations correspond to applications of an expression to its continuation and, by definition, a continuation is applied only once (at the end of the evaluation of the expression).

We illustrate this simplification with the compilation of the traditional recursive version of factorial by $\mathcal{Y}'$:

$$
\begin{aligned}
&\mathcal{Y}'(\text{letrec fact} = \lambda x.\text{cond (eq 0 } x\text{) 1 (mult } x \text{ (fact (sub } x \text{ 1))))}\\
&\quad \equiv \lambda c.c\ (\text{letrec fact} = \lambda c.\lambda x.\mathcal{Y}'(\text{cond (eq 0 } x\text{) 1 (mult } x \text{ (fact (sub } x \text{ 1)))) } c) \qquad (\mathcal{Y}'7)\\
&\quad \equiv \lambda c.c\ (\text{letrec fact} = \lambda c.\lambda x.\mathcal{Y}'(\text{eq 0 } x\text{) (cond}_c\ (\mathcal{Y}\ (1)\ c)\\
&\qquad\qquad (\mathcal{Y}\ (\text{mult } x \text{ (fact (sub } x \text{ 1))) } c))) \qquad (\mathcal{Y}'4)\\
&\quad \equiv \ldots\\
&\quad \equiv \lambda c.c\ (\text{letrec fact} = \lambda c.\lambda x.(\lambda c.(\lambda c.c\ x)\ ((\lambda c.c\ 0)\ (\text{eq}_c\ c)))\ (\text{cond}_c\ ((\lambda c.c\ 1)\ c)\\
&(((\lambda c.(\lambda c.(\lambda c.c\ x)\ ((\lambda c.c\ 1)\ (\text{sub}_c\ c)))\ ((\lambda c.c\ \text{fact})\ \mathbf{id}\ c))\\
&\ ((\lambda c.c\ x)\ (\text{mult}_c\ c)))\ c))).
\end{aligned}
$$

This expression can be simplified by a succession of β-reductions into

$$\lambda c.c\ (\text{letrec fact} = \lambda c.\lambda x.\text{eq}_c\ (\text{cond}_c\ (c\ 1)\ (\text{sub}_c\ (\text{fact}\ (\text{mult}_c\ c\ x))\ x\ 1))\ 0\ x).$$

We have already mentioned that top-level continuations are assumed to be equal to **id**. This convention is rather natural since top-level expressions are just supposed to return their result. This property can be exploited to achieve new improvements of the code. For example, the result of the compilation of fact above can be transformed into

$$\text{letrec fact} = \lambda c.\lambda x.\text{eq}_c\ (\text{cond}_c\ (c\ 1)\ (\text{sub}_c\ (\text{fact}\ (\text{mult}_c\ c\ x))\ x\ 1))\ 0\ x.$$

The evaluation of this function applied to the continuation **id** and the integer 1 is described in the Appendix.

We should notice that the simplifications by β-reduction described in this section realize incidentally two well-known compiler optimizations called copy propagation and dead-code elimination [2]. In the imperative framework, copy propagation consists in replacing any reference to a variable x after the assignment $x := y$ (and before any further assignment to x or y) by a reference to y.

Copy propagation may give the opportunity to eliminate the assignment $x := y$ if it is unused; this optimization is called dead-code elimination. The counterpart of such a situation in the functional framework is an expression of the form $(\lambda x.E)\ y$, which is simplified by β-reduction into $E[y/x]$: copy propagation is carried out, since y is substituted for x and dead-code is eliminated because the application disappears.

3.2 Improvement of the Transformations

Let us now introduce another straightforward optimization. The transformation of the expression

$$(\lambda x.\lambda y.E)\ X\ Y,$$

by $\mathcal{Z}$, yields after simplification

$$\lambda c.\mathcal{Z}(Y)\ (\mathcal{Z}(X)\ (\lambda x.(\lambda c.\lambda y.\mathcal{Z}(E)\ c))\ c).$$

No β-reduction is possible on this expression; however, it is clear that the continuation of $\mathcal{Z}(E)$ will be the global continuation (i.e., the rightmost occurrence of c). So the result we would like is

$$\lambda c.\mathcal{Z}(Y)\ (\mathcal{Z}(X)\ (\lambda x.(\lambda y.\mathcal{Z}(E)\ c))).$$

This optimization can be obtained by the introduction of a new rule in the definition of $\mathcal{Z}$:

$$\mathcal{Z}\ ((\lambda x_1 \cdots \lambda x_n.F)\ E_1 \cdots E_n) \qquad (\mathcal{Z}6_a)$$
$$= \lambda c.\mathcal{Z}(E_n)\ (\cdots(\mathcal{Z}(E_1)\ (\lambda x_1 \cdots \lambda x_n.\mathcal{Z}(F)\ c))\cdots).$$

This rule is very similar to the rule $(\mathcal{Z}3)$ for the complete application of a primitive operator to its arguments. A top-level function does not have its arguments at compile time, and $(\mathcal{Z}6_a)$ is not directly applicable. If we assume that top-level expressions are applied to all their arguments, then the same kind of optimization is obtained using this rule:

$$\mathcal{Z}(\lambda x_1 \cdots \lambda x_n.F) = \lambda c.\lambda x_1 \cdots \lambda x_n.\mathcal{Z}(F)\ c \qquad (\mathcal{Z}6_b)$$
$$\text{if } \lambda x_1 \cdots \lambda x_n.F \text{ is the top-level expression.}$$

The same technique is used (with certain restrictions) when the function is defined by a letrec-expression. Similar rules are introduced to improve the transformation $\mathcal{M}$.

To illustrate these new definitions, we take the example of the compilation of an iterative version of factorial according to call-by-value:

letrec itfact = $\lambda x.\lambda y.$cond (eq 0 x) y (itfact (sub x 1) (mult x y)).

Applying $\mathcal{Z}$ and the optimizations described above, this function is transformed into

letrec itfact = $\lambda c.\lambda x.\lambda y.\mathrm{eq}_c$ (cond_c (c y) (mult_c (sub_c (itfact c) x 1) x y)) 0 x.

3.3 Tail-Recursion Optimization

Let us continue the example of itfact above to show how the traditional tail-recursion optimization can be expressed in our framework. In operational terms, a tail-recursive function is a function such that no computation has to be carried

out after the recursive calls. In functional terms, this means that the continuation is always the identity function and tail-recursion optimization consists in removing this identity continuation, that is to say, producing a function taking no continuation. Let us define function itfact′ by

letrec itfact′ = itfact **id**.

Using the definition of itfact, we get

letrec itfact′ = $\lambda x.\lambda y.\mathrm{eq_c}$ ($\mathrm{cond_c}$ (**id** y) ($\mathrm{mult_c}$ ($\mathrm{sub_c}$ (itfact **id**) x 1) x y)) 0 x

which can be transformed (folding the definition of itfact′) into

letrec itfact′ = $\lambda x.\lambda y.\mathrm{eq_c}$ ($\mathrm{cond_c}$ (**id** y) ($\mathrm{mult_c}$ ($\mathrm{sub_c}$ itfact′ x 1) x y)) 0 x.

We have obtained a new function that does not take any continuation; this transformation is possible because all the recursive calls to itfact in (itfact **id**) take **id** as a continuation (otherwise the final folding would have been impossible). So, we have expressed in the functional framework the definition of tail recursion and the corresponding optimization.

4. COMPILATION OF ENVIRONMENT MANAGEMENT

Let us compare the evaluation of expressions obtained after the first transformation with machine code execution. Throughout the reduction, the expression under evaluation is always of the form $E_1\ E_2\ E_3\ \cdots\ E_n$, where E_1 is the next function to apply and E_2 its continuation (see, for example, the reduction of fact **id** 1 in the Appendix). When E_1 is evaluated and returns the value N_1, the expression becomes $E_2\ N_1\ E_3\ \cdots\ E_n$. Let us now look at the expression under evaluation as a machine state. Clearly, E_1 would be the next instruction to execute; E_2 would be the rest of the code, and $E_3, \ldots, E_n$ would play the role of a stack. However, these expressions are still far from machine code; the basic reason is the occurrence of λ-expressions whose reduction is not straightforward: it 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 [5]. The abstraction process consists in translating a functional expression into an equivalent one that contains no variables via the use of combinators. Combinator-based compilers generally use combinators such as **S**, **K**, **I** [30] or indexed combinators [1, 17]. These combinators must be then implemented on a real machine; they may be interpreted, or custom hardware may be designed to support them. We take an alternative approach here: we choose a set of indexed combinators that act on their arguments as traditional machine instructions on a stack.

$$
\begin{array}{lll}
\textbf{id}\ E & = E & \\
\textbf{push}\ E\ F & = F\ E & \\
\textbf{dupl}_n\ E\ S_1\ \cdots\ S_n & = E\ S_n\ S_1\ \cdots\ S_n & \\
\textbf{flsh}_n\ E\ S_1\ \cdots\ S_n & = E & \\
\textbf{move}_{m,n}\ E\ S_1\ \cdots\ S_{m-1}\ S_m\ S_{m+1}\ \cdots\ S_n & = E\ S_1\ \cdots\ S_{m-1}\ S_n\ S_{m+1}\ \cdots\ S_n, & \text{if } n \geq m \\
\textbf{move}_{m,n}\ E\ S_1\ \cdots\ S_n\ \cdots\ S_{m-1}\ S_m & = E\ S_1\ \cdots\ S_n\ \cdots\ S_{m-1}\ S_n, & \text{if } n < m \\
\textbf{ldcl}_n\ E\ F\ S_1\ \cdots\ S_n & = E\ (F\ S_n)\ S_1\ \cdots\ S_n. &
\end{array}
$$

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

(1) **id** corresponds to a return instruction: if the stack contains a single element, then **id** returns it; otherwise it causes a jump to the address given by the top of the stack,
(2) **push** is the traditional push instruction with an immediate argument,
(3) $\mathbf{dupl}_n$ pushes the nth element of the stack,
(4) $\mathbf{flsh}_n$ pops the first n elements of the stack: it amounts to a modification of the stack pointer,
(5) $\mathbf{move}_{m,n}$ replaces the mth element of the stack by the nth element,
(6) $\mathbf{ldcl}_n$ 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 hold the new value. Section 6 presents such an implementation of closures in the functional framework.

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:

$$\begin{aligned}
&\mathbf{exed}_{m,n,i}\ S_1 \cdots S_m\ X_1 \cdots X_n &&= X_i\ S_1 \cdots S_m \\
&\mathbf{mkcl}_{m,n}\ E\ F\ S_1 \cdots S_m\ X_1 \cdots X_n &&= E\ (F\ X_1 \cdots X_n)\ S_1 \cdots S_m\ X_1 \cdots X_n \\
&\mathbf{delt}_{m,n}\ E\ S_1 \cdots S_m\ X_1 \cdots X_n &&= E\ S_1 \cdots S_m.
\end{aligned}$$

In operational terms, $\mathbf{exed}_{m,n,i}$ is a jump to the address contained in the $i + m$th element of the stack after removing from the stack the n elements corresponding 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:

$$\begin{aligned}
\mathbf{exed}_{m,n,i} &= \mathbf{dupl}_{m+i}\ (\mathbf{delt}_{m+1,n}\ \mathbf{id}) \\
\mathbf{mkcl}_{m,n}\ E &= \mathbf{ldcl}_{m+1}\ (\cdots(\mathbf{ldcl}_{m+n}\ E)\cdots) \\
\mathbf{delt}_{m,n}\ E &= \mathbf{move}_{n+m,m}\ (\cdots(\mathbf{move}_{n+1,1}\ (\mathbf{flsh}_n\ E))\cdots).
\end{aligned}$$

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 that does not contain $x_1, \ldots, x_n$ such that $([x_1, \ldots, x_n]_0\ M)\ x_1 \cdots x_n = M$. Actually, we give a more general definition of the abstraction $[x_1, \ldots, x_n]_p\ M$ such that

$$(\forall S_1, \ldots, S_p)\ ([x_1, \ldots, x_n]_p\ M)\ S_1 \cdots S_p\ x_1 \cdots x_n = M\ S_1 \cdots S_p.$$

We take the convention that when a function is called, its arguments are on the top of the stack. The execution of a function 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 position $p + i$ in the stack.

The global expression is first normalized: nested λ-abstractions are closed using the following transformation (nested letrec-expressions are treated similarly):

$$\lambda x_1 \cdots \lambda x_n.E \rightarrow (\lambda y_1 \cdots \lambda y_k.\lambda x_1 \cdots \lambda x_n.E)\ y_1 \cdots y_k$$

where $y_1, \ldots, y_k$ are the free variables of the original λ-expression.

This normalization, called λ-lifting [13], is very much in the spirit of supercombinators [11] but does not exhibit full laziness. An expression $[x_1, \ldots, x_n]_0\ E$ will be a combinatory expression without free variables; therefore we do not have to design special rules to abstract letrec-expressions or combinatory expressions. Each subexpression of the form $(\lambda x_1 \cdots \lambda x_n.E)$ (respectively letrec $f = \lambda x_1 \cdots \lambda x_n.E$) is replaced by $[x_1, \ldots, x_n]_0\ E$ (respectively letrec $f = [x_1, \ldots, x_n]_0\ E$), and the abstraction algorithm is applied to the innermost λ-expressions in a bottom-up fashion.

The basic abstraction rules are defined as follows (when the left-hand sides of two rules overlap, the first rule should be applied first):

$$[x_1, \ldots, x_n]_p\ M = \mathbf{delt}_{p,n}\ M, \quad \text{if } x_1, \ldots, x_n \notin M \text{ and } n \neq 0 \tag{A1}$$

$$[x_1, \ldots, x_n]_p\ M\ x_i = \mathbf{dupl}_{p+i}\ ([x_1, \ldots, x_n]_{p+1}\ M), \quad \text{if } x_i \in \{x_1, \ldots, x_n\} \tag{A2}$$

$$[x_1, \ldots, x_n]_p\ M\ y = ([x_1, \ldots, x_n]_{p+1}\ M)\ y, \quad \text{if } y \notin \{x_1, \ldots, x_n\} \tag{A3}$$

$$[x_1, \ldots, x_n]_p\ M\ k = \mathbf{push}\ k\ ([x_1, \ldots, x_n]_{p+1}\ M) \tag{A4}$$

$$[x_1, \ldots, x_n]_p\ \mathrm{opm}_c\ M = \mathrm{opm}_c\ ([x_1, \ldots, x_n]_{p-m+1}\ M), \quad (p \geq m) \tag{A5}$$

$$[x_1, \ldots, x_n]_p\ \mathrm{cond}_c\ M\ N = \mathrm{cond}_c\ ([x_1, \ldots, x_n]_{p-1}\ M)([x_1, \ldots, x_n]_{p-1}\ N), \quad (p \geq 1) \tag{A6}$$

$$[x_1, \ldots, x_n]_p\ x_i = \mathbf{exed}_{p,n,i} \quad \text{if } x_i \in \{x_1, \ldots, x_n\} \tag{A7}$$

$$[x_1, \ldots, x_n]_p\ M\ N = \mathbf{push}\ ([x_1, \ldots, x_n]_0\ N)(\mathbf{mkcl}_{p,n}\ ([x_1, \ldots, x_n]_{p+1}\ M)). \tag{A8}$$

Before tackling the correctness proof, we give an intuitive justification for these rules.

(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 (which can be found at position $p+i$ in the stack) and then evaluating M with one more element on the stack.

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

(A3) is applied when the abstraction of a nested expression contains free variables (i.e., in this context, function names), which are left unchanged.

(A5) describes the treatment of operators. An operator of arity m consumes m elements from the stack and produces one element; so the remainder of the expression is compiled with index $p-m+1$.

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

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

(A8) deals with the evaluation of a nonbasic expression M with a nonbasic 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 now come back to our simple example to illustrate this abstraction algorithm; we have described in Section 2 the transformation of the expression $\lambda x.\text{add } x \text{ (add } x \text{ 1)}$ by $\mathcal{T}$; after simplification the result was:

$$\lambda x.\text{add}_c \text{ (add}_c \textbf{ id } x) \; x \; 1.$$

The application of our abstraction algorithm to this expression gives

$$\begin{aligned}
&[x]_0 \text{ (add}_c \text{ (add}_c \textbf{ id } x) \; x \; 1) \\
&\quad = \textbf{push } 1 \text{ ([}x\text{]}_1 \text{ (add}_c \text{ (add}_c \textbf{ id } x) \; x)) && \text{(A4)} \\
&\quad = \textbf{push } 1 \text{ (}\textbf{dupl}_2 \text{ ([}x\text{]}_2 \text{ (add}_c \text{ (add}_c \textbf{ id } x)))) && \text{(A2)} \\
&\quad = \textbf{push } 1 \text{ (}\textbf{dupl}_2 \text{ (add}_c \text{ ([}x\text{]}_1 \text{ (add}_c \textbf{ id } x)))) && \text{(A5)} \\
&\quad = \textbf{push } 1 \text{ (}\textbf{dupl}_2 \text{ (add}_c \text{ (}\textbf{dupl}_2 \text{ ([}x\text{]}_2 \text{ (add}_c \textbf{ id})))) && \text{(A2)} \\
&\quad = \textbf{push } 1 \text{ (}\textbf{dupl}_2 \text{ (add}_c \text{ (}\textbf{dupl}_2 \text{ (}\textbf{delt}_{2,1} \text{ (add}_c \textbf{ id})))). && \text{(A1)}
\end{aligned}$$

We now state the correctness property of our abstraction algorithm.

PROPERTY 7. $(\forall p \geq 0)\ (\forall M, S_1, \ldots, S_p)\ ([x_1, \ldots, x_n]_p\ M)\ S_1 \cdots S_p\ x_1 \cdots x_n = M\ S_1 \cdots S_p$. *This property includes (for* $p = 0$*) the classic property of abstraction algorithms:* $([x_1, \ldots, x_n]\ M)\ x_1 \cdots x_n = M$; *the generalization is needed to show the property by induction on the structure of the expressions.*

PROOF. The proof is a routine inspection of the different cases [10]. Each case is treated in a mechanical fashion, using only the definitions of the combinators, the rules of the abstraction algorithm, and the induction hypothesis. We describe the cases $E \equiv M\ x_i$ and $E \equiv M\ N$:

(1) $E \equiv M\ x_i$

$$\begin{aligned}
&([x_1, \ldots, x_n]_p\ M\ x_i)\ S_1 \cdots S_p\ x_1 \cdots x_n \\
&\quad = \textbf{dupl}_{p+i}\ ([x_1, \ldots, x_n]_{p+1}\ M)\ S_1 \cdots S_p\ x_1 \cdots x_n && \text{(A2)} \\
&\quad = ([x_1, \ldots, x_n]_{p+1}\ M)\ x_i\ S_1 \cdots S_p\ x_1 \cdots x_n && \text{[definition of } \textbf{dupl}\text{]} \\
&\quad = M\ x_i\ S_1 \cdots S_p. && \text{[induction hypothesis].}
\end{aligned}$$

(2) $E \equiv M\ N$

$$\begin{aligned}
&([x_1, \ldots, x_n]_p\ M\ N)\ S_1 \cdots S_p\ x_1 \cdots x_n \\
&\quad = \textbf{push}([x_1, \ldots, x_n]_0\ N)\ (\textbf{mkcl}_{p,n}\ ([x_1, \ldots, x_n]_{p+1}\ M))\ S_1 \cdots S_p\ x_1 \cdots x_n && \text{(A8)} \\
&\quad = \textbf{mkcl}_{p,n}\ ([x_1, \ldots, x_n]_{p+1}\ M)([x_1, \ldots, x_n]_0\ N)\ S_1 \cdots S_p\ x_1 \cdots x_n \\
&\qquad\qquad && \text{[definition of } \textbf{push}\text{]} \\
&\quad = ([x_1, \ldots, x_n]_{p+1}\ M)\ (([x_1, \ldots, x_n]_0\ N)\ x_1 \cdots x_n)\ S_1 \cdots S_p\ x_1 \cdots x_n \\
&\qquad\qquad && \text{[definition of } \textbf{mkcl}\text{]} \quad \square \\
&\quad = M\ (([x_1, \ldots, x_n]_0\ N)x_1 \cdots x_n)\ S_1 \cdots S_p \\
&\qquad\qquad && \text{[induction hypothesis]} \\
&\quad = \text{M N S}_1 \cdots \text{S}_p \\
&\qquad\qquad && \text{[induction hypothesis].}
\end{aligned}$$

5. OPTIMIZATION TECHNIQUES

The abstraction algorithm given in Section 4 is straightforward, and the code produced can be improved in several ways. We first describe some optimizations of the algorithm itself; then we present a simplification process, which can be applied to the result of the abstraction algorithm.

5.1 Reducing the Size of Closures

According to rule (A8) of our abstraction algorithm, a closure is made of an expression and the current environment. However, saving the whole environment may be unnecessarily expensive if some identifiers are not referenced at all in the expression contained in the closure. Rule (A8) can be improved in order to gather in the closure only the values required to evaluate the expression:

$$[x_1, \ldots, x_n]_p\ M\ N = \mathbf{push}\ ([x_{i1}, \ldots, x_{ir}]_0\ N)\ (\mathbf{ldcl}_{p+i1} \cdots (\mathbf{ldcl}_{p+ir}\ ([x_1, \ldots, x_n]_{p+1}\ M))\cdots) \quad \text{(A8)}$$

where $x_{i1}, \ldots, x_{ir}$ are the free variables of N.

This new version of (A8) is more efficient than the original in our implementation because we do not have an explicit environment (the stack is used to store the environment as well as the intermediate values); so the construction and the installation on the stack of a closure take time proportional to the size of the environment.

5.2 Avoiding the Creation of Closures

It is well known that the manipulation of closures is an important source of inefficiency in the implementation of functional languages [15]. Closures involve environments, and their construction is time and space consuming. It is necessary to build a closure when the execution of an expression is deferred (because the expression is a function or due to the normal-order evaluation rule) in order to save the right environment for its subsequent evaluation. This strategy is the most general and must be used when the compiler cannot know when the expression whose evaluation has been deferred must be evaluated. However, there are expressions whose evaluation time can be predicted. The typical case is expressions $M\ N$, where N is the continuation of M; in such a case, N will be evaluated when the evaluation of M is finished with the value of M on the top of the stack. The straightforward compilation indicated by rule (A8) would involve saving the environment with N in a closure, executing M, which provokes the destruction of the current environment, and reinstalling this environment (or a part of it) for the evaluation of N. It would, obviously, be more efficient to avoid constructing a closure for N and to compile M in such a way that its evaluation keeps the current environment in place. So, in order to carry out this optimization, one must be able to

(1) detect expressions $M\ N$ such that N is a continuation of M,

(2) compile an expression in such a way that it does not destroy its environment.

Let us consider item (1) first. The following definition allows us to make more precise what we mean by "N is a continuation of M."

Definition 8. Any expression M such that for all $S_1, \ldots, S_m$ there exists some M' such that $(\forall N)\ M\ N\ S_1 \cdots S_m = N\ M'$ is called a *stable expression of order m.*

In other words, if M is a stable expression of order m then in the expression $M\ N\ E_1 \cdots E_n$ (with $n \geq m$), N is a continuation of M. The order of an expression produced by the first transformation step can be obtained from the transformation itself. For example, correctness properties of transformations $\mathcal{T}$ (Property 4, Section 2) and $\mathcal{T}'$ allow us to derive the following property.

PROPERTY 9. $(\forall E)\ \mathcal{T}'(E)$ *and* $\mathcal{T}'(E)$ *are stable expressions of order* 0; $(\forall E)\ \mathcal{T}'(\lambda x.E)\ \mathbf{id}$ *and* $\mathcal{T}'(\lambda x.E)\ \mathbf{id}$ *are stable expressions of order* 1.

PROOF. Let us just justify that $\mathcal{T}'(E)$ is a stable expression of order 0. By Property 4, we have $\mathcal{T}(E)\ C = \mathcal{T}(\mathrm{CBV}(E))\ C$. By definition of $\mathcal{T}$, if N is a normal form, then $\mathcal{T}(N)$ is of the form $\lambda c.c\ N'$. Thus $\mathcal{T}'(E)\ C = \mathcal{T}(\mathrm{CBV}(E))\ C \equiv (\lambda c.c\ E')\ C = C\ E'$. □

Let us now consider item (2). What we need is an abstraction algorithm $[\ ]'$ such that

$$([x_1, \ldots, x_n]'_p\ M)\ S_1 \cdots S_p\ x_1 \cdots x_n = M\ S_1 \cdots S_p\ x_1 \cdots x_n.$$

In fact, this algorithm can be obtained from the previous one in the following way.

Definition 10. We consider the abstraction algorithm $[x_1, \ldots, x_n]'_p\ M$ similar to the previous one (replacing $([x_1, \ldots, x_n]_p$ by $[x_1, \ldots, x_n]'_p$) except that (A1), (A7), (A8) become

$$[x_1, \ldots, x_n]'_p\ k = k \qquad \text{(A1')}$$
$$[x_1, \ldots, x_n]'_p\ x_i = \mathbf{dupl}_{p+i}\mathbf{id} \qquad \text{(A7')}$$
$$[x_1, \ldots, x_n]'_p\ M\ N = \mathbf{push}\ ([x_1, \ldots, x_n]_0\ N)\ (\mathbf{mkcl}_{p,n}([x_1, \ldots, x_n]'_{p+1}\ M)). \qquad \text{(A8')}$$

One can show that the following property is true.

PROPERTY 11. $(\forall p \geq 0)\ (\forall M, S_1, \ldots, S_p)\ ([x_1, \ldots, x_n]'_p\ M)\ S_1 \cdots S_p\ x_1 \cdots x_n = M\ S_1 \cdots S_p\ x_1 \cdots x_n$.

PROOF. Similar to the proof of Property 7.

We can gather these two pieces of information to improve the abstraction algorithm by adding a new rule (between (A7) and (A8)):

$$[x_1, \ldots, x_n]_p\ M\ N = \mathbf{push}\ ([x_1, \ldots, x_n]_{p-m+1}\ N)\ ([x_1, \ldots, x_n]'_{p+1}\ M), \qquad \text{(A8}_a\text{)}$$

if M is a stable expression of order $m \leq p$.

PROOF. The proof of the algorithm remains the same; we just have to prove one more induction case corresponding to rule (A8$_a$):

$$M\ N\ S_1 \cdots S_p = N\ M'\ S_{m+1} \cdots S_p$$
[Definition 8, M is a stable expression of order m]

and

$$\begin{aligned}
&\mathbf{push}([x_1, \ldots, x_n]_{p-m+1}\ N)([x_1, \ldots, x_n]'_{p+1}\ M)\ S_1 \cdots S_p\ x_1 \cdots x_n \\
&\quad = ([x_1, \ldots, x_n]'_{p+1}\ M)\ ([x_1, \ldots, x_n]_{p-m+1}\ N)\ S_1 \cdots S_p\ x_1 \cdots x_n \\
&\qquad\qquad \text{[definition of } \mathbf{push}\text{]} \\
&\quad = M\ ([x_1, \ldots, x_n]_{p-m+1}\ N)\ S_1 \cdots S_p\ x_1 \cdots x_n \\
&\qquad\qquad \text{[Property 11]} \qquad \square \\
&\quad = ([x_1, \ldots, x_n]_{p-m+1}\ N)\ M'\ S_{m+1} \cdots S_p\ x_1 \cdots x_n \\
&\qquad \text{[Definition 8, } M \text{ is a stable expression of order } m\text{]} \\
&\quad = N\ M'\ S_{m+1} \cdots S_p \\
&\qquad\qquad \text{[induction hypothesis].}
\end{aligned}$$

This optimization may be applied to expressions produced by $\mathcal{T}'$ or $\mathcal{A}$, but its most useful application is to subexpressions $(x\ E)$ produced by $\mathcal{A}$. In this context, every variable x (not introduced by $\mathcal{A}$) is to be bound to an expression of the form $\mathcal{A}(E)$, which is a complete expression of order 0. So $(A8_a)$ can be applied, and $(x\ E)$ is abstracted as follows:

$$\begin{aligned}
[x_1, \ldots, x_n]_p\ x_i\ E &= \mathbf{push}\ ([x_1, \ldots, x_n]_{p+1}\ E)([x_1, \ldots, x_n]'_{p+1}\ x_i) && (A8_a) \\
&= \mathbf{push}\ ([x_1, \ldots, x_n]_{p+1}\ E)(\mathbf{dupl}_{p+1+i}\ \mathbf{id}). && (A7')
\end{aligned}$$

This case is so common that it is convenient to introduce a new combinator

$$\mathbf{call}_n\ E\ P_1 \cdots P_n = P_n\ E\ P_1 \cdots P_n \quad [\mathbf{call}_n\ E = \mathbf{push}\ E\ (\mathbf{dupl}_{n+1}\ \mathbf{id})].$$

This combinator can be seen as the machine instruction "subprogram call", which pushes the return address before jumping to the subprogram. Using this combinator, we can add a new rule used only for expressions previously compiled by $\mathcal{A}'$:

$$[x_1, \ldots, x_n]_p\ x_i\ E = \mathbf{call}_{i+p}\ ([x_1, \ldots, x_n]_{p+1}\ E) \qquad (A8_b)$$

[if x is not a continuation variable].

The detection of continuation expressions has already been used by Kranz et al. [15] to improve the code produced by the ORBIT compiler. The optimization consists of allocating on the stack rather than on the heap closures whose lifetime can be predicted.

Now, let us take an example in order to illustrate the optimized abstraction algorithm. We consider the expression

$$\lambda y.(\lambda x.\text{add } x\ (\text{add } x\ 1))\ (\text{add } y\ 1).$$

After compilation of call-by-name by $\mathcal{A}'$ and simplifications, we get

$$\lambda y.(\lambda x.x\ (\text{add}_c\ (x\ (\text{add}_c\ \mathbf{id})))\ 1)\ (\lambda c.y\ (\text{add}_c\ c)\ 1).$$

The rightmost λ-abstraction is not closed; λ-lifting gives us

$$\lambda y.(\lambda x.x\ (\text{add}_c\ (x\ (\text{add}_c\ \mathbf{id})))\ 1)\ ((\lambda y.\lambda c.y\ (\text{add}_c\ c)\ 1)\ y).$$

The abstraction algorithm can now be applied:

$$\begin{aligned}
&[y]_0\ ([x]_0\ x\ (\text{add}_c\ (x\ (\text{add}_c\ \mathbf{id})))\ 1)\ ((([y,c]_0\ y\ (\text{add}_c\ c)\ 1)\ y) \\
&[x]_0\ x\ (\text{add}_c\ (x\ (\text{add}_c\ \mathbf{id})))\ 1 \\
&\quad = \mathbf{push}\ 1\ (\mathbf{call}_2([x]_2\ \text{add}_c(x(\text{add}_c\mathbf{id})))) && (A4), (A8_b) \\
&\quad = \mathbf{push}\ 1\ (\mathbf{call}_2(\text{add}_c\ (\mathbf{call}_2(\mathbf{delt}_{2,1}(\text{add}_c\mathbf{id}))))) = X && (A5), (A8_b), (A1) \\
&[y,c]_0\ y\ (\text{add}_c)\ 1 = \mathbf{push}\ 1\ (\mathbf{call}_2\ (\text{add}_c\ \mathbf{exed}_{1,2,2})) = Y. && (A4), (A8_b), (A5), (A7)
\end{aligned}$$

So

$$[y]_0\ X\ (Y\ y) = \mathbf{push}\ Y\ (\mathbf{mkcl}_{0,1}\ (\mathbf{delt}_{1,2}\ X)). \qquad \text{(A8), (A1)}$$

Actually, in the abstraction of $[y]_0\ X\ (Y\ y)$, we have implicitly used η-reduction ($[y]_0\ (Y\ y) = Y$). This new rule could be included in the algorithm like our previous optimizations. Another way to perform this optimization (and many others) is to simplify the expressions produced by the abstraction process. This kind of simplification is presented in the next section as rewriting rules on combinatory expressions.

5.3 Peephole Optimizations

Now we introduce simplification rules which may be applied to the result of the abstraction algorithm; most of these rules are the translation in the functional framework of well-known compiler optimization techniques called peephole optimizations [2]. We give here some of the most useful simplification rules:

$$\begin{array}{lll}
\mathbf{dupl}_i\ (\mathbf{move}_{i+1,1}\ \mathbf{E}) = \mathbf{dupl}_i\ \mathbf{E} & & \text{(S1)}\\
\mathbf{dupl}_i\ (\mathbf{move}_{1,j}\ \mathbf{E}) = \mathbf{dupl}_{j-1}\ \mathbf{E} & & \text{(S2)}\\
\mathbf{dupl}_i\ (\mathbf{flsh}_j\ \mathbf{E}) = \mathbf{flsh}_{j-1}\ \mathbf{E}, & \text{if } j > 0 & \text{(S3)}\\
\mathbf{flsh}_i\ (\mathbf{flsh}_j\ \mathbf{E}) = \mathbf{flsh}_{i+j}\ \mathbf{E} & & \text{(S4)}\\
\mathbf{flsh}_0\ \mathbf{E} = \mathbf{E} & & \text{(S5)}\\
\mathbf{move}_{i,j}\ (\mathbf{flsh}_k\ \mathbf{E}) = \mathbf{flsh}_k\ \mathbf{E}, & \text{if } i \leq k & \text{(S6)}\\
\mathbf{move}_{i,j}\ (\mathbf{move}_{i,k}\ \mathbf{E}) = \mathbf{move}_{i,k}\ \mathbf{E} & & \text{(S7)}\\
\mathbf{move}_{i,j}\ (\mathbf{move}_{j,i}\ \mathbf{E}) = \mathbf{move}_{i,j}\ \mathbf{E} & & \text{(S8)}\\
\mathbf{dupl}_i\ (\mathbf{move}_{j,k}\ \mathbf{E}) = \mathbf{move}_{j-1,k-1}\ (\mathbf{dupl}_i\ \mathbf{E}), & \text{if } j \neq 1,\ k \neq 1,\ i \neq j-1 & \text{(S9)}\\
\mathbf{dupl}_i\ (\mathbf{move}_{j,1}\ \mathbf{E}) = \mathbf{move}_{j-1,i}\ (\mathbf{dupl}_i\ \mathbf{E}), & \text{if } j \neq 1 & \text{(S10)}\\
\mathbf{flsh}_i\ (\mathbf{move}_{k,1}\ \mathbf{E}) = \mathbf{move}_{k+i,1+i}\ (\mathbf{flsh}_i\ \mathbf{E}) & & \text{(S11)}\\
\mathbf{flsh}_i\ (\mathbf{dupl}_j\ \mathbf{E}) = \mathbf{move}_{i,i+j}\ (\mathbf{flsh}_{i-1}\ \mathbf{E}). & & \text{(S12)}
\end{array}$$

The general idea behind most of these rules is to avoid redundant stack movements. The correctness of each individual rule is obvious. The termination of the rewriting system can be shown using an ordering involving the size of the expression (rules (S1)–(S8) strictly decrease the size of the expression) weighted by a priority of combinators (rules (S9)–(S12)).

Let us continue our factorial example to show an application of this rewriting system. The function produced by the first transformation $\mathcal{Y}'$ is, after simplification (see Section 3.1):

$$\text{letrec fact} = \lambda c.\lambda x.\mathrm{eq}_c\ (\mathrm{cond}_c\ (c\ 1)\ (\mathrm{sub}_c\ (\mathrm{fact}\ (\mathrm{mult}_c\ c\ x))\ x\ 1))\ 0\ x.$$

Applying the abstraction rules (notably $A8_a$), since this fact is a stable expression of order 1, we get

$$\begin{array}{l}
\text{letrec fact} = \mathbf{dupl}_2\ (\mathbf{push}\ 0\ (\mathrm{eq}_c\ (\mathrm{cond}_c\\
\qquad\qquad (\mathbf{push}\ 1\ \mathbf{exed}_{1,2,1})\\
\qquad\qquad (\mathbf{push}\ 1\ (\mathbf{dupl}_3\ (\mathrm{sub}_c\ (\mathbf{push}\ (\mathbf{dupl}_3\ (\mathrm{mult}_c\ \mathbf{exed}_{1,2,1}))\ \mathrm{fact})))))).
\end{array}$$

The macrocombinator **exed** is unfolded:

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

It is simplified using the rewriting system (S1)–(S12):

$$\begin{aligned}
\mathbf{exed}_{1,2,1} &= \mathbf{move}_{3,1}\ (\mathbf{dupl}_2\ (\mathbf{move}_{3,1}\ (\mathbf{flsh}_2\ \mathbf{id}))) && \text{(S9)}\\
&= \mathbf{move}_{3,1}\ (\mathbf{dupl}_2\ (\mathbf{flsh}_2\ \mathbf{id})) && \text{(S1)}\\
&= \mathbf{move}_{3,1}\ (\mathbf{flsh}_1\ \mathbf{id}) && \text{(S3)}
\end{aligned}$$

and the final version of fact is

$$\begin{aligned}
\text{letrec fact} = \ & \mathbf{dupl}_2\ (\mathbf{push}\ 0\ (\text{eq}_c\ (\text{cond}_c\\
& (\mathbf{push}\ 1\ (\ \mathbf{move}_{3,1}\ (\mathbf{flsh}_1\ \mathbf{id})))\\
& (\mathbf{push}\ 1\ (\mathbf{dupl}_3\ (\text{sub}_c\ (\mathbf{push}\ (\mathbf{dupl}_3\\
& (\text{mult}_c\ (\mathbf{move}_{3,1}\ (\mathbf{flsh}_1 \mathbf{id})))) \text{ fact})))))).
\end{aligned}$$

This example illustrates the interest of the simplification strategy: the **dupl** combinators are moved toward the end of the code where they may be canceled by a **flsh** combinator. In operational terms, we delay the execution of the **dupl** operators until they turn out to be really necessary or useless.

5.4 Linearization

The expressions yielded by the abstraction algorithm look very much like machine code. The only remaining difference is that in general 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, then we must translate the occurrences of function names into **jump** instructions; in functional terms, **jump** f is defined by **jump** $f = f$. This introduction of explicit labels and jumps has also been used in the context of semantics-directed compiling [3, 27]. The application of the linearization process to the previous expression yields:

$$\begin{aligned}
&\text{letrec fact} = \mathbf{dupl}_2\ (\mathbf{push}\ 0\ (\text{eq}_c\ (\text{cond}_c\ f_0\ f_1)))\\
&\text{let } f_0 \quad = \mathbf{push}\ 1\ (\mathbf{move}_{3,1}\ (\mathbf{flsh}_1\ \mathbf{id}))\\
&\text{let } f_1 \quad = \mathbf{push}\ 1\ (\mathbf{dupl}_3\ (\text{sub}_c\ (\mathbf{push}\ f_2\ (\mathbf{jump}\ \text{fact}))))\\
&\text{let } f_2 \quad = \mathbf{dupl}_3\ (\text{mult}_c\ (\mathbf{move}_{3,1}\ (\mathbf{flsh}_1\ \mathbf{id}))).
\end{aligned}$$

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

fact	dupl	2	f_1	push	1
	push	0		dupl	3
	eq_c			sub_c	
	cond_c	f_0, f_1		push	f_2
f_0	push	1		jump	fact
	move	3,1	f_2	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 this code (applied to **id** and 1) and illustrates the duality—functional expression/machine code—of the result of the compilation.

6. IMPLEMENTATION OF CLOSURES AND CALL-BY-NEED

The expressions obtained after the transformations described in the previous sections can be translated into code for a specific machine by a straightforward macroexpansion. For example, $\mathbf{dupl}_i$ can be expanded into the 68020 assembly instruction

movl sp@(4*(*i*−1)), sp@−

where sp is the stack pointer and 4 is the word length. However, one combinateur, **mkcl**, does not have any immediate counterpart in the instruction set of usual computers. It builds a closure by constructing a succession of applications on the top of the stack; this is not realistic since a real stack element cannot contain objects of unbounded size. The usual solution is to build the closure in the heap and to leave a pointer to it in the stack. This representation makes also possible the sharing of closures corresponding to arguments occurring several times in the body of the function. The version of call-by-name that updates a closure after its first evaluation is usually referred to as call-by-need. The transformations seen so far to compile call-by-name remain valid for call-by-need; the difference between those two evaluation rules only appears in the implementation of closures.

In order to express this low-level step in the functional framework, (1) we add a new argument H to expressions (an array representing the heap) and (2) we introduce indexes (a new set of basic constants noted $\mathbb{1}, \mathbb{2}, \ldots$) to access elements of H. Closures are represented as vectors (groups of contiguous elements) in the heap: the first element is a label (or pointer) corresponding to the code of the closure and the remaining elements correspond to the environment of the closure.

We want to obtain expressions of the form $E\ F\ H\ S_1 \cdots S_n$, where $(E\ F)$ is the code (F is a continuation), H the heap, and $S_1, \ldots, S_n$ the stack. To this aim, new versions of combinators and another transformation are introduced. A complete description of this treatment is beyond the scope of this paper; we just present the main ideas by resuming the example in Section 5.3, where the function $\lambda y.(\lambda x.\text{add } x\ (\text{add } x\ 1))\ (\text{add } y\ 1)$ was compiled into

$$\mathbf{push}\ (\mathbf{push}\ 1\ (\mathbf{call}_2\ (\text{add}_c\ \mathbf{exed}_{1,2,2})))\ (\mathbf{mkcl}_{0,1}\ (\mathbf{delt}_{1,2}\ (\mathbf{push}\ 1\ (\mathbf{call}_2\ (\text{add}_c\ (\mathbf{call}_2\ (\mathbf{delt}_{2,1}\ (\text{add}_c\ \mathbf{id}))))))))).$$

This expression is now transformed into

$$\mathbf{Push}\ (\mathbf{Pscl}_1\ (\mathbf{Push}\ 1\ (\mathbf{Call}_2\ (\text{add}_c\ \mathbf{Exed}_{1,2,2}))))\ (\mathbf{Mkcl}_{0,1}\ (\mathbf{Delt}_{1,2}\ (\mathbf{Push}\ 1\ (\mathbf{Call}_2\ (\text{add}_c\ (\mathbf{Call}_2\ (\mathbf{Delt}_{2,1}\ (\text{add}_c\ \mathbf{Fin}))))))))),$$

which has the linearized form

$$\mathbf{Push}\ f_0\ f_1$$

with

$$\text{let } f_0 = \mathbf{Pscl}_1\ (\mathbf{Push}\ 1\ (\mathbf{Call}_2\ \text{ret}_0))\ \text{let ret}_0 = \text{add}_c\ \mathbf{Exed}_{1,2,2}$$
$$\text{let } f_1 = \mathbf{Mkcl}_{0,1}\ (\mathbf{Delt}_{1,2}\ (\mathbf{Push}\ 1\ (\mathbf{Call}_2\ \text{ret}_1)))$$
$$\text{let ret}_1 = \text{add}_c\ (\mathbf{Call}_2\ \text{ret}_2) \qquad \text{let ret}_2 = \mathbf{Delt}_{2,1}\ (\text{add}_c\ \mathbf{Fin}).$$

The combinator **Push** denotes the same operation as **push** but takes an extra argument: **Push** $E\ F\ A = F\ A\ E$. The same trivial transformation is applied to obtain **Dupl**, **Delt**, **Flsh**, and **Move**. **Fin** is the final continuation (previously denoted by **id**) and deletes the store before returning the result: **Fin** $A\ E = E$. In operational terms, $\mathbf{Mkcl}_{m,n}$ builds a closure as a vector in the array; $\mathbf{Exed}_{m,n,i}$ remains the same except in one case: if the $i + n$th element is an index, then $\mathbf{Exed}_{m,n,i}$ is an indexed jump to the function part of a closure; $\mathbf{Call}_n$ is a call to a function in the stack or the function part of a closure according to the type of the nth element (index or not); the new combinator **Pscl** is used to push the arguments of a closure in the stack before its reduction.

Let us give an idea of the reduction rules of the new combinators by describing the execution of the above expression applied to the argument 1. The compiled form of this argument is

let arg = **Push** 1 $\mathbf{Exed}_{1,1,1}$ ($\mathcal{A}$ (1) = $\lambda c.c$ 1, and $[c]_0\ c\ 1$ = **push** 1 $\mathbf{exed}_{1,1,1}$)

and the reduction proceeds as follows:

Push $f_0\ f_1$ [ ] arg
→ $\mathbf{Mkcl}_{0,1}$ ($\mathbf{Delt}_{1,2}$ (**Push** 1 ($\mathbf{Call}_2$ ret_1))) [ ] f_0 arg
[the initial array is empty (not initialized)] (1)

$\mathbf{Mkcl}_{0,1}$ builds the closure $\{f_0, \text{arg}\}$ in the array and pushes an index pointing to it.

→ $\mathbf{Delt}_{1,2}$ (**Push** 1 ($\mathbf{Call}_2$ ret_1)) [f_0,arg] 1 f_0 arg (2)

$\mathbf{Delt}_{1,2}$ removes f_0 and arg from the stack.

→ **Push** 1 ($\mathbf{Call}_2$ ret_1) [f_0,arg] 1 f_0 arg (3)
→ $\mathbf{Call}_2$ ret_1 [f_0,arg] 1 1 (4)

$\mathbf{Call}_2$ saves the return address and installs the function part of the closure (f_0) for evaluation.

→ $\mathbf{Pscl}_1$ (**Push** 1 ($\mathbf{Call}_2$ ret_0)) [f_0,arg] 1 ret_1 1 1 (5)

$\mathbf{Pscl}_1$ pushes the argument of the closure, which is now reduced.

→ **Push** 1 ($\mathbf{Call}_2$ ret_0) [f_0,arg] arg ret_1 1 1 (6)
→ $\mathbf{Call}_2$ ret_0 [f_0,arg] 1 arg ret_1 1 1 (7)

The argument to be executed (second stack element) is not a closure, $\mathbf{Call}_2$ is a direct jump to arg, which is reduced.

→ **Push** 1 $\mathbf{Exed}_{1,1,1}$ [f_0,arg] ret_0 1 arg ret_1 1 1 (8)
→ $\mathbf{Exed}_{1,1,1}$ [f_0,arg] 1 ret_0 1 arg ret_1 1 1 (9)

After evaluation, the control returns to ret_0 to complete the reduction of the closure.

→ add_c $\mathbf{Exed}_{1,2,2}$ [f_0,arg] 1 1 arg ret_1 1 1 (10)
→ $\mathbf{Exed}_{1,2,2}$ [f_0,arg] 2 arg ret_1 1 1 (11)

$\mathbf{Exed}_{1,2,2}$ makes a jump to the return address ret_1 after having removed arg from the stack.

→ add_c ($\mathbf{Call}_2$ ret_2 [f_0,arg] 2 1 1 (12)
→ $\mathbf{Call}_2$ ret_2 [f_0,arg] 3 1 (13)

The closure is needed a second time and is reevaluated; the reduction is the same as previously.

$\rightarrow \cdots$ (see steps (5)–(11)) (14)–(20)

The execution of ret_2 takes place when the closure in normal form (2) is on top of the stack.

$\rightarrow$ **Delt**$_{2,1}$ (add$_c$ **Fin**) [f_0,arg] 2 3 ⇓ (21)
$\rightarrow$ add$_c$ **Fin** [f_0,arg] 2 3 (22)
$\rightarrow$ **Fin** [f_0,arg] 5 (23)
$\rightarrow$ 5 (24)

The closure {f_0,arg} (which corresponds to {(add y 1), y=1}) is executed twice because it is bound to two occurrences of x in λx.add x (add x 1); this is why call-by-name is never used in practice. Call-by-need keeps the advantages of normal-order reduction while avoiding recomputation of arguments. The idea consists in replacing the closure by its value after the first evaluation. In order to implement this update operation, we need to introduce two new combinators and a new version of **Call**. The updating operation is performed by the combinator **Updt** which replaces the function part of the closure by the combinator **Pshv** (= **Pscl**$_1$ **Exed**$_{1,3,3}$) and the first argument by the normal form of the closure; **Pshv** is used to push the value of the closure (i.e., the first argument) on the stack. The new version of **Call** still saves the return address before reducing the closure but now installs **Updt** on the top of the stack as an intermediate continuation.

Let us describe the evaluation of the example above by call-by-need. The compiled expression has the same syntax; only the definition of **Call** is changed.

Push f_0 f_1 [ ] arg
$\rightarrow \cdots$ The first steps are not changed. (1)–(3)
$\rightarrow$ **Call**$_2$ ret$_1$ [f_0,arg] 1 ⇓ (4)

Call$_2$ installs **Updt** on the top of the stack as an intermediate continuation. The index pushed between **Updt** and ret$_1$ is used by **Updt** to locate the cells to update.

$\rightarrow$ **Pscl**$_1$ (**Push** 1 (**Call**$_2$ ret$_0$)) [f_0,arg] ⇓ **Updt** ⇓ ret$_1$ 1 ⇓ (5)
$\rightarrow \cdots$ see steps (6)–(10) for call-by-name (6)–(10)
$\rightarrow$ **Exed**$_{1,2,2}$ [f_0,arg] 2 arg **Updt** ⇓ ret$_1$ 1 ⇓ (11)

The closure in normal form is on top of the stack. The intermediate continuation **Updt** is executed.

$\rightarrow$ **Updt** [f_0,arg] 2 ⇓ ret$_1$ 1 ⇓ (12)

The closure is replaced by **Pshv** and its normal form.

$\rightarrow$ add$_c$ (**Call**$_2$ ret$_2$) [**Pshv**,2] 2 1 ⇓ (13)
$\rightarrow$ **Call**$_2$ ret$_2$ [**Pshv**,2] 3 ⇓ (14)

The closure is needed a second time, and **Pshv** is executed.

$\rightarrow$ **Pshv** [**Pshv**,2] ⇓ **Updt** ⇓ ret$_2$ 3 ⇓ (15)

Pshv pushes the value of the closure and deletes the update operation and its associated index. This way, the updating operation is performed at most once per closure (after the first evaluation).

$\rightarrow$ **Delt**$_{2,1}$ (add$_c$ **Fin**) [**Pshv**,2] 2 3 1 (16)
$\rightarrow$ 5 $\cdots$ see steps (22)–(24) with call-by-name. (17)–(19)

Due to its low-level nature, this last step is not so elegantly expressed in the functional framework. We had to introduce an array along with indexes and new combinators so that we could express the classic implementation of closures, sharing, and the updating inherent to call-by-need. For the same reason, the correctness proofs are somewhat less simple. However, we should point out that all the proofs already done by call-by-name remain valid for call-by-need; the correctness proof of the implementation of call-by-need can focus on this last transformation.

The combinator **Mkcl** and the new combinators introduced to implement closures and call-by-need have a straightforward translation into machine instructions; so the compilation of our original functional language into functional machine code is now completely described.

7. CONCLUSIONS

We advocate in this paper that the implementation process of a functional language should be described in a purely functional way. We have presented a method for transforming functions defined in a λ-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 in contrast with the SECD machine [5, 16], the TIM [9, 33], and the CAM [7, 19, 20] 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 the 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 [18] and [22] since they are identical. The only operational argument in our proof appears in Section 2, where it is shown that the reduction of the transformed expression by FIRST amounts to the reduction of the original expression by the computation rule compiled. However, this proof does not involve reasoning on tricky machine states. We should also mention that the first transformation ($\mathcal{Z}$ or $\mathcal{A}$) does not depend on the chosen implementation and could be applied as well in the context of graph reduction (it would lead to a simpler graph evaluator reducing systematically the first term of the expression).

7.1 Related Works

The formalization of the implementation process has also been studied by Reynolds [24], followed by Schmidt [26] and Wand [32]. They proceed by successive transformations of a semantics of the source language to derive an

interpreter or a compiler and an abstract machine. In particular, Wand [32] presents heuristics for analyzing the compilation process. This method also involves continuations and combinators, but in a quite different way: it takes a continuation-based semantics as 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. In some sense, the transformation described in Wand [32] to derive machine code can be seen as the specialization of an abstraction algorithm to a particular program, which is the semantics of the language. 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) that can be described by a continuation semantics. Raoult and Sethi [23] propose a related approach to semantics-directed compiling; they use combinators acting on an explicit tuple (which can also be seen as a stack) and introduce a "pipe" combinator that expresses instruction sequences and plays the role of continuations in our setting.

The benefits of continuations to compiler design have already been illustrated by two-Scheme [29] compilers: Rabbit [28] followed by Orbit [15]. They integrate continuation conversion as a preliminary standardization step, but the compilation is not entirely described in terms of program transformation. Appel and Jim [4] describe a continuation-passing style compiler of ML written in ML. The originality of the paper is the presentation in several phases, within the ML framework, of optimizations that are tangled into one phase in ORBIT. Kelsey and Hudak [14] present a compiler based on program transformations in an intermediate language. The idea is to use a feature-for-feature comparison of the intermediate language and the machine language to guide the transformation process. Continuations are used again to make control explicit, and environments are introduced to manage identifier binding. In contrast with our approach, the source and intermediate language are imperative, and the semantics of the intermediate and target languages are different. These works put deliberate emphasis on efficiency issues rather than on the correctness of the implementation. Let us also mention the presentation of the TIM, which has been rephrased by Wray and Fairbairn [33] as a succession of simple transformations in the spirit of the work presented here.

7.2 Performances

Even if the performance considerations are not the main topic of this paper, we must say a few words about the produced code. We have chosen the environment-based approach rather than graph reduction because it is closer to traditional von Neumann machines. The code produced by our transformation rules is lower level than the code of Burge [5] for the SECD machine and that of [7] and [20] for the CAM. In operational terms, the main difference between our implemen-

tation and the SECD machine is where environment savings are achieved: in our implementation, environments are saved 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.

Most of our transformations can be directly written in a functional language with pattern matching. A prototype compiler based on this approach has been implemented in Miranda [31] on a SUN workstation for three computation rules: call-by-value, call-by-name, and call-by-need. We should mention that only the simple language described in Section 2 has been implemented so far. We illustrate the efficiency of the code produced with some small functions chosen to evaluate specific aspects of the implementation:

```
letrec fact   = λx.cond (eq 0 x) 1 (mult x (fact (sub x 1)))
letrec fib    = λx.cond (leq x 1) 1 (plus (fib (sub x 1)) (fib (sub x 2)))
letrec itfact = λx.λy.cond (eq 0 x) y (itfact (sub x 1) (mult x y))
letrec ack    = λx.λy.cond (eq x 0) (plus y 1)
                    (cond (eq 0 y) (ack (sub x 1) 1)(ack (sub x 1) (ack x (sub y 1))))
letrec zerocl = λx.λy.cond (eq x 0) 0 (zerocl (sub x 1) (λx. plus 1 (y z)))
letrec addcl  = λx.λy.cond (eq x 0)(y 0) (addcl (sub x 1) (λz. plus 1 (y z))).
```

Functions fact and fib are classical benchmarks; itfact is a tail-recursive function; ack involves embedded recursive calls; zerocl is a function designed to illustrate the efficiency of closure creation, and addcl involves closure creation and closure application. Tables I and II show the execution times of the codes produced by several compilers; our compiler is denoted by CPT (Compilation by Program Transformation). All the safe optimizations provided by each compiler have been applied (e.g., using short integers).

The two results for itfact in *C* correspond to the left-recursive definition of itfact and to the iterative version (using a while-loop) of the same function. The relative inefficiency of addcl using our compiler may be explained by the fact that the complexity of closure application is linear in the size of the environment (because environments are installed on the stack. Another possible choice is to keep a pointer in the global environment and to distinguish access to local identifiers and to global identifiers [6]; this makes function calls more efficient because context switching amounts to a pointer movement. This technique could have been implemented in our approach as well.

These results have to be confirmed for a more realistic language including lists, user-defined data types, and pattern matching. Nevertheless, they are promising and show that the code produced by our compiler for simple programs is realistic.

7.3 Further Prospects

We believe that the efficiency of the code produced by our compiler is due to the program transformation approach, which allows the systematic application (and justification) of optimization rules. Each technique is applied at the appropriate

Table I. Execution Times for Functions fact 12, fib 20, and ack 3 3 Produced by Various Compilers

Recursive calls per second	CPT Call by value	CPT Call by need	CAML Call by value	CAML Call by need	Chez scheme (CBV)	Standard ML (CBV)	C (CBV)
fact 12	180,000	40,000	80,000	15,000	15,000	110,000	180,000
fib 20	265,000	40,000	95,000	20,000	30,000	180,000	260,000
ack 3 3	220,000	25,000	60,000	10,000	25,000	100,000	240,000

Table II. Execution Times for Functions itfact 12 1, zerocl 100 **id**, and addcl 100 **id** Produced by Various Compilers

Recursive calls per second	CPT Call by value	CAML Call by value	Chez scheme	Standard ML	C
itfact 12 1	160,000	40,000	10,000	70,000	160,000/ 200,000
zerocl 100 **id**	230,000	40,000	20,000	125,000	—
addcl 100 **id**	50,000	30,000	15,000	80,000	—

transformation level; we have shown that tail-recursion optimization, copy propagation, and dead-code elimination can be carried out after the first transformation step (Section 3), whereas peephole optimizations are applied on combinatory expressions (Section 5).

Other well-known compiler optimization techniques such as common subexpression elimination and code motion [2] can be expressed in the functional framework. For example, let E be an expression in which F occurs several times; the following transformation achieves common subexpression elimination:

$$E \longrightarrow (\lambda x.E[x/F])\ F.$$

Code motion is applied to loops involving a subexpression whose evaluation always yields the same value ("loop-invariant computations"). The optimization consists in taking the loop-invariant expression and placing it before the loop, thus avoiding reevaluation. In the functional framework, loops are represented by recursive functions. A subexpression in the body of a function is constant through the calls ("function-invariant") if it does not involve any argument of the function. We can again resort to β-expansion to name such an expression and extract it from the body. Let F be a subexpression of letrec $f = \lambda x_1 \cdots \lambda x_n.E$, which occurs several times and does not involve any of the arguments $x_1, \ldots, x_n$, and let x be an identifier that does not occur in E. Code motion is carried out in the following way:

$$\text{letrec } f = \lambda x_1 \cdots \lambda x_n.B \longrightarrow (\lambda x.(\text{letrec } f = \lambda x_1 \cdots \lambda x_n.E[x/F]))\ F.$$

These techniques, avoiding reevaluation of expressions, are related to the compilation method proposed in [11] to obtain fully lazy supercombinators.

However, Hughes [11] strives to avoid the duplication of expressions during the compilation phase and is not concerned with the detection of common subexpressions already present in the original function. Finally, let us notice that these techniques do not depend on the compilation rules described in this paper, so they should be applied on the original expression, which is generally simpler.

The results of the strictness analysis can also be exploited very easily in our approach. Let us consider the transformation $\mathcal{A}$ described at the end of Section 2; assuming that function f is strict in its ith argument, we can define an optimization of rule ($\mathcal{A}$5):

$$\mathcal{A}(f\ E_1 \cdots E_i) = \lambda c.\mathcal{A}''(E_i)\ (\mathcal{A}''(f\ E_1 \cdots E_{i-1})\ c). \qquad (\mathcal{A}5)$$

This rule forces the evaluation of E_i before the call of f. Function f itself must be compiled using a composite set of transformation rules ($\mathcal{A}''$) in order to take into account the fact that its ith argument is evaluated before the call (basically rule ($\mathcal{A}$1) is used to compile the occurrences of the $i - 1$ first arguments, and rule ($\mathcal{X}$1) is used for occurrences of the ith argument).

Another promise of our approach could be the expression, within the same framework, of various implementation techniques. This would provide a better understanding of the relations between implementations, which is very difficult to assess, in general.

APPENDIX

We show that the evaluation of the function fact obtained after the first compilation step (Section 2) can be carried out by systematic reduction of the head operator. The function is applied to the continuation **id** and the integer 1.

$$\begin{aligned}
&(\underline{\text{letrec}}\ \text{fact} = \lambda c.\lambda x.\text{eq}_c\ (\text{cond}_c\ (c\ 1)\ (\text{sub}_c\ (\text{fact}\ (\text{mult}_c\ c\ x))\ x\ 1))\ 0\ x)\ \mathbf{id}\ 1\\
&\quad \to_{\text{FIRST}} (\underline{\lambda x}.\text{eq}_c\ (\text{cond}_c\ (\mathbf{id}\ 1)\ (\text{sub}_c\ (\text{fact}\ (\text{mult}_c\ \mathbf{id}\ x))\ 1\ 1))\ 0\ x)\ 1\\
&\quad \to_{\text{FIRST}} \underline{\text{eq}_c}\ (\text{cond}_c\ (\mathbf{id}\ 1)\ (\text{sub}_c\ (\text{fact}\ (\text{mult}_c\ \mathbf{id}\ 1))\ 1\ 1))\ 0\ 1\\
&\quad \to_{\text{FIRST}} \underline{\text{cond}_c}\ (\mathbf{id}\ 1)\ (\text{sub}_c\ (\text{fact}\ (\text{mult}_c\ \mathbf{id}\ 1))\ 11))\ \text{False}\\
&\quad \to_{\text{FIRST}} \underline{\text{sub}_c}\ (\text{fact}\ (\text{mult}_c\ \mathbf{id}\ 1))\ 1\ 1\\
&\quad \to_{\text{FIRST}} (\underline{\text{letrec}}\ \text{fact}\\
&\qquad\quad = \lambda c.\lambda x.\text{eq}_c\ (\text{cond}_c\ (c\ 1)\ (\text{sub}_c\ (\text{fact}\ (\text{mult}_c\ c\ x))\ x\ 1))\ 0\ x)\ (\text{mult}_c\ \mathbf{id}\ 1)\ 0\\
&\quad \to_{\text{FIRST}} (\underline{\lambda x}.\text{eq}_c\ (\text{cond}_c\ (\text{mult}_c\ \mathbf{id}\ 1\ 1)\ (\text{sub}_c\ (\text{fact}\ (\text{mult}_c\\
&\qquad\qquad (\text{mult}_c\ \mathbf{id}\ 1)\ x))\ x\ 1))\ 0\ x)\ 0\\
&\quad \to_{\text{FIRST}} \underline{\text{eq}_c}\ (\text{cond}_c\ (\text{mult}_c\ \mathbf{id}\ 1\ 1)\ (\text{sub}_c\ (\text{fact}\ (\text{mult}_c\ (\text{mult}_c\ \mathbf{id}\ 1)\ 0))\ 0\ 1))\ 0\ 0\\
&\quad \to_{\text{FIRST}} \underline{\text{cond}_c}\ (\text{mult}_c\ \mathbf{id}\ 1\ 1)\ (\text{sub}_c\ (\text{fact}\ (\text{mult}_c\ (\text{mult}_c\ \mathbf{id}\ 1)\ 0))\ 0\ 1))\ \text{True}\\
&\quad \to_{\text{FIRST}} \underline{\text{mult}_c}\ \mathbf{id}\ 1\ 1\\
&\quad \to_{\text{FIRST}} \underline{\mathbf{id}}\ 1\\
&\quad \to_{\text{FIRST}} \underline{1}.
\end{aligned}$$

We now describe the evaluation of the result of the compilation of fact given in Section 5.3. We show in parallel the evaluation as the execution of code on a stack machine and as the reduction of a functional expression (see Table III). In the middle part of the table, we represent the state of the stack (the top being the leftmost element) before the execution of the corresponding instruction.

Table III. Execution of Code on a Stack Machine and as Reduction of a Functional Expression

Machine state			
Code		Stack	Functional expression
fact	dupl 2	id:1	$\mathbf{dupl}_2$ (**push** 0 (eq_c ($cond_c$ f_0 f_1))) **id** 1
	push 0	1:id:1	**push** 0 (eq_c ($cond_c$ f_0 f_1)) 1 **id** 1
	eq_c	0:1:id:1	eq_c ($cond_c$ f_0 f_1) 0 1 **id** 1
	$cond_c$ f_0, f_1	False:id:1	$cond_c$ f_0 f_1 False **id** 1
f_1	push 1	id:1	**push** 1 ($\mathbf{dupl}_3$ (sub_c (**push** f_2 (**jump** fact)))) **id** 1
	dupl 3	1:id:1	$\mathbf{dupl}_3$ (sub_c (**push** f_2 (**jump** fact))) 1 **id** 1
	sub_c	1:1:id:1	sub_c (**push** f_2 (**jump** fact)) 1 1 **id** 1
	push f_2	0:id:1	**push** f_2 (**jump** fact) 0 **id** 1
	jump fact	f_2:0:id:1	**jump** fact f_2 0 **id** 1
fact	dupl 2	f_2:0:id:1	$\mathbf{dupl}_2$ (**push** 0 (eq_c ($cond_c$ f_0 f_1))) f_2 0 **id** 1
	push 0	0:f_2:0:id:1	**push** 0 (eq_c ($cond_c$ f_0 f_1)) 0 f_2 0 **id** 1
	eq_c	0:0:f_2:0:id:1	eq_c ($cond_c$ f_0 f_1) 0 0 f_2 0 **id** 1
	$cond_c$ f_0, f_1	True:f_2:0:id:1	$cond_c$ f_0 f_1 True f_2 0 **id** 1
f_0	push 1	f_2:0:id:1	**push** 1 ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**)) f_2 0 **id** 1
	move 3, 1	1:f_2:0:id:1	$\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**) 1 f_2 0 **id** 1
	flsh 1	1:f_2:1:id:1	$\mathbf{flsh}_1$ **id** 1 f_2 1 **id** 1
	id	f_2:1:id:1	**id** f_2 1 **id** 1
f_2	dupl 3	1:id:1	$\mathbf{dupl}_3$ ($mult_c$ ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**))) 1 **id** 1
	$mult_c$	1:1:id:1	$mult_c$ ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**)) 1 1 **id** 1
	move 3, 1	1:id:1	$\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**) 1 **id** 1
	flsh 1	1:id:1	$\mathbf{flsh}_1$ **id** 1 **id** 1
	id	id:1	**id id** 1
	id	1	**id** 1
	result = 1		

ACKNOWLEDGMENTS

We are grateful to Dave Schmidt for his comments on several versions of this paper. We would like to thank J.-P. Banâtre, S. B. Jones, and P. Wadler for reading an earlier draft of this paper. Thanks are also due to the referees who provided helpful suggestions.

REFERENCES

1. ABDALI, S. K. An abstraction algorithm for combinatory logic. *J Symbolic Logic 41*, 1 (1976), 222–224.
2. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques and Tools* Addison-Wesley, Reading, Mass., 1986.
3. APPEL, A. W. Semantics-directed code generation. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*. ACM, New York, 1985, pp. 315–324.
4. APPEL, A. W., AND JIM, T. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*. ACM, New York, 1989, pp. 293–302.
5. BURGE, W. H. *Recursive Programming Techniques* Addison-Wesley, Reading, Mass., 1975.
6. CARDELLI, L. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*. ACM, New York, 1984, pp. 208–226.
7. COUSINEAU, G., CURIEN, P.-L., AND MAUNY, M. The categorical abstract machine. *Sci. Comput. Program. 8* (1987), 173–202.
8. CURRY, H. B., AND FEYS, R. *Combinatory Logic*. Vol. I, North-Holland, New York, 1958.
9. FAIRBAIRN, J., AND WRAY, S. C. TIM: A simple abstract machine for executing supercombinators. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science 274*, Springer-Verlag, Berlin, 1987, pp. 34–45.

10. FRADET, P. Compilation des langages fonctionnels par transformation de programmes. Thèse de Doctorat de l'Université de Rennes I, 1988.
11. HUGHES, R. J. M. Supercombinators: A new implementation method for applicative languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*. ACM, New York, 1982, pp. 1–10.
12. JOHNSSON, T. Efficient compilation of lazy evaluation. *SIGPLAN Not. 19*, 6 (1984), 58–69.
13. JOHNSSON, T. Lambda-lifting—Transforming programs to recursive equations. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science 201*, Springer-Verlag, Berlin, 1985, pp. 190–203.
14. KELSEY, R., AND HUDAK, P. Realistic compilation by program transformation. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*. ACM, New York, 1989, pp. 281–292.
15. KRANZ, D., KELSEY, R., REES, J., HUDAK, P., PHILBIN, J., AND ADAMS, N. Orbit: An optimizing compiler for scheme. In *Proceedings of the 1986 ACM SIGPLAN Symposium on Compiler Construction*. ACM, New York, 1986, pp. 219–233.
16. LANDIN, P. J. The mechanical evaluation of expressions. *Comput. J. 6* (1964), 308–320.
17. LEMAITRE, M., CASTAN, M., DURAND, M.-H., DURRIEU, G., AND LECUSSAN, B. Mechanisms for efficient multiprocessor combinator reduction. In *Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming*. ACM, New York, 1986, pp. 113–121.
18. LESTER, D. The G-machine as a representation of stack semantics. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science 274*, Springer-Verlag, Berlin, 1987, pp. 46–59.
19. MAUNY, M. Compilation des langages fonctionnels dans les combinateurs catégoriques. Application au langage ML. Thèse de Troisième Cycle de l'Université de Paris VII, 1985.
20. MAUNY, M., AND SUAREZ, A. Implementing functional languages in the categorical abstract machine. In *Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming*. ACM, New York, 1986, pp. 266–278.
21. PLOTKIN, G. D. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Univ. of Aarhus, 1981.
22. PLOTKIN, G. D. Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci. 1* (1975), 125–159.
23. RAOULT, J.-C., AND SETHI, R. Properties of a notation for combining functions. *J. ACM 30*, 3 (1983), 595–611.
24. REYNOLDS, J. C. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference* (1972), pp. 717–740.
25. REYNOLDS, J. C. GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept. *Commun. ACM 13*, 5 (May 1970), 308–319.
26. SCHMIDT, D. A. State transition machines for lambda-calculus expressions. In *Proceedings of the Semantics Directed Compiler Generation Workshop. Lecture Notes in Computer Science 94*, Springer-Verlag, Berlin, 1980, pp. 415–440.
27. SETHI, R. Control flow aspects of semantics directed compiling. *ACM Trans. Program. Lang. Syst. 5*, 4 (1983), 554–595.
28. STEELE, G. L. JR. Rabbit: A compiler for Scheme. AI Tech. Rep. 474, MIT, Cambridge, Mass., 1978.
29. STEELE, G. L., JR., AND SUSSMAN, G. J. The revised report on Scheme. AI Memo 452, MIT, Cambridge, Mass., 1978.
30. TURNER, D. A. A new implementation technique for applicative languages. *Softw. Pract. Exper. 9* (1979), 31–49.
31. TURNER, D. A. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science 201*, Springer-Verlag, Berlin, 1985, pp. 1–16.
32. WAND, M. Deriving target code as a representation of continuation semantics. *ACM Trans. Program. Lang Syst. 4*, 3 (1982), 496–517.
33. WRAY, S. C., AND FAIRBAIRN, J. Non-strict languages—Programming and implementation. *Comput. J. 32*, 2 (1989), 142–151.

Received February 1989; revised October 1989; accepted April 1990