Towards a Taxonomy of Functional Language Implementations

Rémi Douence *Pascal Fradet*

INRIA/IRISA

Campus de Beaulieu, 35042 Rennes Cedex, France

`[douence,fradet]@irisa.fr`

Abstract

We express implementations of functional languages as a succession of program transformations in a common framework. At each step, different transformations model fundamental choices or optimizations. A benefit of this approach is to structure and decompose the implementation process. The correctness proofs can be tackled independently for each step and amount to proving program transformations in the functional world. It also paves the way to formal comparisons by estimating the complexity of individual transformations or compositions of them. We focus on call-by-value implementations, describe and compare the diverse alternatives and classify well-known abstract machines. This work also aims to open the design space of functional language implementations and we suggest how distinct choices could be mixed to yield efficient hybrid abstract machines.

1 Introduction

One of the most studied issues concerning functional languages is their implementation. Since the seminal proposal of Landin, 30 years ago [18], a plethora of new abstract machines or compilation techniques have been proposed. The list of existing abstract machines includes (but is surely not limited to) the SECD [18], the FAM [6], the CAM [7], the CMCM [20], the TIM [10], the ZAM [19], the G-machine [15] and the Krivine-machine [8]. Other implementations are not described via an abstract machine but as a collection of transformations or compilation techniques such as CPS-based compilers [1][12][17]. Furthermore, numerous papers present optimizations often adapted to a specific abstract machine or a specific approach [3][4][16]. Looking at this myriad of distinct works, obvious questions spring to mind: what are the fundamental choices? What are the respective benefits of these alternatives? What are precisely the common points and differences between two compilers? Can a particular optimization, designed for machine *A*, be adapted to machine *B*? One finds comparatively very few papers devoted to these questions. There have been studies of the relationship between two individual machines [25][21] but, to the best of our knowledge, no global approach to describe, classify and compare implementations.

This paper presents an advance towards a general taxonomy of functional language implementations. Our approach is to express in a common framework the whole compilation process as a succession of program transformations. The framework considered is a hierarchy of intermediate languages all of which are subsets of the lambda-calculus. Our description of an implementation consists of a series of transformations $\Lambda \xrightarrow{\mathcal{T}1} \Lambda_1 \xrightarrow{\mathcal{T}2} \ldots \xrightarrow{\mathcal{T}n} \Lambda_n$ each one compiling a particular task by mapping an expression from one intermediate language into another. The last language Λ_n consists of functional expressions which can be seen as machine code (essentially, combinators with explicit sequencing and calls). For each step, different transformations are designed to represent fundamental choices or optimizations. A benefit of this approach is to structure and decompose the implementation process. Two seemingly disparate implementations can be found to share some compilation steps. This approach has also interesting payoffs as far as correctness proofs and comparisons are concerned. The correctness of each step can be tackled independently and amounts to proving a program transformation in the functional world. It also paves the way to formal comparisons by estimating the complexity of individual transformations or compositions of them.

The two steps which cause the greatest impact on the compiler structure are the implementation of the reduction strategy (searching for the next redex) and the environment management (compilation of β-reduction). Other steps include implementation of control transfers (calls & returns), representation of components like data stack or environments and various optimizations.

The task is clearly huge and our presentation is by no means complete (partly because of space concerns, partly because some points are still under study). First, we concentrate on pure λ-expressions and our source language Λ is $E ::= x \mid \lambda x.E \mid E_1\ E_2$. Most fundamental choices can be described for this simple language. Second, we focus on the call-by-value reduction strategy and its standard implementations. In section 2 we describe the framework used to model the compilation process. In section 3 (resp. section 4) we present the alternatives and optimizations to compile call-by-value (resp. the environment management). Each section is concluded with a comparison of the main options. Section 5 is devoted to two other simple steps leading to machine code. In section 6, we describe how this work can be easily extended to deal with constants, primitive operators, fix-point and call-by-name strategies. We also mention what remains to be done to model call-by-need and graph reduction. Finally, we indicate how it would be possible to mix different choices within a single compiler (section 8) and conclude by a short review of related works.

2 General Framework

The transformation sequence presented is this paper involves four intermediate languages (very close to each other) and can be described as $\Lambda \to \Lambda_s \to \Lambda_e \to \Lambda_k$. The first one, Λ_s, bans unrestricted applications and makes the reduction strategy explicit using a sequencing combinator. The second one Λ_e excludes unrestricted uses of variables and encodes environment management. The last one Λ_k handles control transfers by using calls and returns. This last language can be seen as a machine code. We focus here on the first intermediate language; the others (and an overview of their use) are briefly described in 2.5.

2.1 The control language Λ_s

Λ_s is defined using the combinators $\circ$, $\mathbf{push}_s$, and $\lambda_s x.\ E$ (this last construct can be seen as a shorthand for a combinator applied to $\lambda x.\ E$). This language is a subset of λ-expressions therefore substitution and the notion of free or bound variables are the same as in λ-calculus.

$$\Lambda_s \qquad E ::= x \mid \mathbf{push}_s\ E \mid \lambda_s x.\ E \mid E_1 \circ E_2 \qquad x \in \mathit{Vars}$$

The most notable syntactic feature of Λ_s is that it rules out unrestricted applications. Its main property is that the choice of the next redex is not relevant anymore (all redexes are needed). This is the key point to compile evaluation strategies which are made explicit using the primitive $\circ$. Intuitively, $\circ$ is a sequencing operator and $E_1 \circ E_2$ can be read “evaluate E_1 then evaluate E_2”, $\mathbf{push}_s$ E returns E as a result and $\lambda_s x.\ E$ binds the previous intermediate result to x before evaluating E.

These combinators can be given different definitions (possible definitions are given at the end of this section (**DEF1**) and in sub-section 5.2). We do not pick a specific one up at this point; we simply impose that their definitions satisfy the equivalent of β-and η-conversions

$$(\beta_s) \qquad (\mathbf{push}_s\ F) \circ (\lambda_s x.\ E) = E[F/x]$$

$$(\eta_s) \qquad \lambda_s x.(\mathbf{push}_s\ x \circ E) = E \quad \textit{if } x \textit{ does not occur free in } E$$

As the usual imperative sequencing operator “;”, it is natural to enforce the associativity of combinator $\circ$. This property will prove especially useful to transform programs.

$$\text{(assoc)} \quad (E_1 \circ E_2) \circ E_3 = E_1 \circ (E_2 \circ E_3)$$

We often omit parentheses and write e.g. $\mathbf{push}_s E \circ \lambda_s x.F \circ G$ for $(\mathbf{push}_s\ E) \circ (\lambda_s x.(F \circ G))$.

2.2 A typed subset

We are not interested in all the expressions of Λ_s. Transformations of source programs will only produce expressions denoting results (i.e. which can be reduced to expressions of the form $\mathbf{push}_s$ F). In order to express laws more easily it is convenient to restrict Λ_s using a type system (Figure 1). This does not impose any restrictions on source programs. For example, we can allow reflexive types ($\alpha=\alpha\to\alpha$) to type any source λ-expression. The restrictions enforced by the type system are on how results and functions are combined. For example, composition $E_1 \circ E_2$ is restricted so that E_1 must denote a result (i.e. has type $\mathsf{R}\sigma$, R being a type constructor) and E_2 must denote a function.

$$\frac{\Gamma \vdash E : \sigma}{\Gamma \vdash \mathbf{push}_s\ E : \mathsf{R}\sigma} \qquad \frac{\Gamma \cup \{x{:}\sigma\} \vdash E : \tau}{\Gamma \vdash \lambda_s x.\ E : \sigma \to_s \tau} \qquad \frac{\Gamma \vdash E_1 : \mathsf{R}\sigma \quad \Gamma \vdash E_2 : \sigma \to_s \tau}{\Gamma \vdash E_1 \circ E_2 : \tau}$$

Figure 1 Λ_s typed subset

2.3 Reduction

We consider only one reduction rule corresponding to the classical β-reduction:

$$\mathbf{push}_s\ F \circ \lambda_s x.\ E \Rightarrow E[F/x]$$

As with all standard implementations, we are only interested in modelling weak reductions. Subexpressions inside $\mathbf{push}_s$’s and λ_s’s are not considered as redexes and from here on we write “redex” (resp. reduction, normal form) for weak redex (resp. weak reduction, weak normal form).

Any two redexes are clearly disjoint and the β_s-reduction is left-linear so the term rewriting system is orthogonal (hence confluent). Furthermore any redex is needed (a rewrite cannot suppress a redex) thus all reduction strategies are normalizing.

Property 1 *If a closed expression E:$\mathsf{R}\sigma$ has a normal form, there exist V such that $E \overset{*}{\Rightarrow} \mathbf{push}_s\ V$*

Due to the lack of space we do not display proofs here and refer the interested reader to a companion paper [9].

The reduction should be done modulo associativity if we allow an unrestricted use of (assoc) which may produce ill-typed programs. The rule (β_s) along with (assoc) specifies a string reduction confluent modulo (assoc).

2.4 Laws

This framework enjoys a number of algebraic laws useful to transform the functional code or to prove the correctness or equivalence of program transformations. We list here only two of them.

(L1) *if x does not occur free in F* $\qquad (\lambda_s x.E) \circ F = \lambda_s x.\ (E \circ F)$

(L2) $\forall E_1$:$\mathsf{R}\sigma$, $\forall E_2$:$\mathsf{R}\tau$, *if x does not occur free in E_2* $\qquad E_1 \circ (\lambda_s x.\ E_2 \circ E_3) = E_2 \circ E_1 \circ (\lambda_s x.E_3)$

These rules permit code to be moved inside or outside function bodies or to invert the evaluation of two results. Their correctness can be established very simply. For example (L1) is sound since x does not occur free in $(\lambda_s x.E)$ nor, by hypothesis, in F and

$$\begin{aligned}
(\lambda_s x.E) \circ F &= \lambda_s x.\ \mathbf{push}_s\ x \circ ((\lambda_s x.E) \circ F) && (\eta_s)\\
&= \lambda_s x.\ ((\mathbf{push}_s\ x \circ (\lambda_s x.E)) \circ F) && (\text{assoc})\\
&= \lambda_s x.\ (E[x/x] \circ F) && (\beta_s)\\
&= \lambda_s x.\ (E \circ F) && (\text{subst})
\end{aligned}$$

In the rest of the paper, we introduce other laws to express optimizations of specific transformations.

2.5 Overview of the compilation phases

Before describing implementations formally, let us first give an idea of the different phases, choices and the hierarchy of intermediate Λ-languages.

The first phase is the compilation of control which is described by transformations ($\mathcal{V}$) from Λ to Λ_s. The pair $(\mathbf{push}_s, \lambda_s)$ specifies a component storing intermediate results (e.g. a data stack). The main choice is using the eval-apply model ($\mathcal{V}a$) or the push-enter model ($\mathcal{V}m$). For the $\mathcal{V}a$ family we describe other minor options such as avoiding the need for a stack ($\mathcal{V}a_s$, $\mathcal{V}a_f$) or right-to-left ($\mathcal{V}a$) *vs*. left-to-right evaluation ($\mathcal{V}a_L$).

Transformations ($\mathcal{A}$) from Λ_s to Λ_e are used to compile β-reduction. The language Λ_e avoids unrestricted uses of variables and introduces the pair $(\mathbf{push}_e, \lambda_e)$. They behave exactly as $\mathbf{push}_s$ and λ_s and corresponding properties (β_e, η_e) hold. They just act on a (at least conceptually) different component (e.g. a stack of environments). The main choice is using list-like environments ($\mathcal{A}s$) or vector-like environments ($\mathcal{A}c$). For the latter choice, there are several transformations depending on the way environments are copied ($\mathcal{A}c1$, $\mathcal{A}c2$, $\mathcal{A}c3$).

A last transformation ($\mathcal{S}$) from Λ_e to Λ_k is used to compile control transfers (this step can be avoided by using a transformation ($\mathcal{S}\ell$) on Λ_s-expressions). The language Λ_k makes calls and returns explicit. It introduces the pair $(\mathbf{push}_k, \lambda_k)$ which specifies a component storing return addresses.

Control	Λ_s	$(\mathbf{push}_s, \lambda_s)$	$\mathcal{V}a$ $\mathcal{V}a_L$ $\underline{\mathcal{V}a_s}$ $\underline{\mathcal{V}a_f}$ $\mathcal{V}m^{\#}$	$(+\ \underline{\mathcal{S}\ell}^{*})$
Abstraction	Λ_e	$(\mathbf{push}_e, \lambda_e)$	$\mathcal{A}s$ $\mathcal{A}c1$ $\mathcal{A}c2$ $\mathcal{A}c3^{\#}$	
Transfers	Λ_k	$(\mathbf{push}_k, \lambda_k)$	$\mathcal{S}^{*}$	

Figure 2 Summary of the Main Compilation Steps and Options

Figure 2 gathers the different options described in the three following sections. Any two transformations of different phases can be combined except those with the same superscript (# or *). Stack-like components are avoided by underlined transformations.

Combinators, expressed in terms of $\mathbf{push}_x$ and λ_x, are described along with transformations. To simplify the presentation, we also use syntactic sugar such as tuples $(x_1,\dots,x_n)$ and pattern-matching $\lambda_x(x_1,\dots,x_n).E$.

This hierarchy of Λ-languages is a convenient abstraction to express the compilation process. But recall that they are made of combinators and therefore subsets of the λ-calculus. An important point is that we do not have to give a precise definition to pairs $(\mathbf{push}_x, \lambda_x)$. We just enforce that they respect properties (β_x) and (η_x). Definitions do not have to be chosen until the very last step. For example, definitions of $\circ$ and $\mathbf{push}_s$ would be of the form

$$\circ\ E_1\ E_2\ X_1 \ldots X_n \to \ldots \qquad\qquad \mathbf{push}_s\ V\ X_1 \ldots X_n \to \ldots$$

where $X_1,\ldots,X_n$ are components on which the code acts (e.g. control or data stack, registers,…). In other words, $X_1,\ldots,X_n$ along with the Λ_x-code can be seen as the state of an abstract machine. We do not want to commit ourselves to a precise definition of combinators, however we want to ensure that the reduction from left to right using the rules of combinators simulates the reduction ➨. In order to enforce this property, it is possible to state a few conditions that the standard reduction of combinators must verify. We do not expound on this issue, but a possible definition for Λ_s is

(**DEF1**) $\quad \mathbf{push}_s\ E = \lambda c.\ c\ E \qquad \lambda_s x.\ E = \lambda c.\lambda x.\ E\ c \qquad E_1 \circ E_2 = \lambda c.\ E_1\ (E_2\ c) \qquad (c \text{ fresh})$

and we can easily check that β-reduction simulates reduction ➨. Alternative definitions are presented in section 5.2.

3 Compilation of Control

We do not consider left-to-right *vs*. right-to-left as a fundamental choice to implement call-by-value. A more radical dichotomy is *explicit applies* vs. *marks*. The first option is the standard technique (e.g. used in the SECD or CAM) while the second was hinted at in [10] and used in ZINC.

3.1 Compilation of control using apply

Applications $E_1\ E_2$ are compiled by evaluating the argument E_2, the function E_1 and finally applying the result of E_1 to the result of E_2. The compilation of right-to-left call-by-value is described in Figure 3. Normal forms denote results so λ-abstractions and variables (which, in strict languages, are always bound to a normal forms) are transformed into results (i.e. $\mathbf{push}_s\ E$).

$$\mathcal{V}a : \Lambda \to \Lambda_s$$

$$\mathcal{V}a\ \llbracket x \rrbracket\ = \mathbf{push}_s\ x$$

$$\mathcal{V}a\ \llbracket \lambda x.E \rrbracket\ = \mathbf{push}_s\ (\lambda_s x.\ \mathcal{V}a\ \llbracket E \rrbracket)$$

$$\mathcal{V}a\ \llbracket E_1\ E_2 \rrbracket\ = \mathcal{V}a\ \llbracket E_2 \rrbracket \circ \mathcal{V}a\ \llbracket E_1 \rrbracket \circ \mathbf{app} \qquad \text{with } \mathbf{app} = \lambda_s x.\ x$$

Figure 3 Compilation of Right-to-Left CBV with Explicit Applies($\mathcal{V}a$)

The rules can be explained intuitively by reading "return the value" for $\mathbf{push}_s$, "evaluate" for $\mathcal{V}a$, "then" for $\circ$ and "apply" for **app**. $\mathcal{V}a$ produces well-typed expressions. Its correctness is stated by Property 2 which establishes that the reduction of transformed expressions ($\overset{*}{➨}$) simulates the call-by-value reduction (*CBV*) of source λ-expressions.

Property 2 $\forall E \in \Lambda,\ CBV(E) \equiv V \Leftrightarrow \mathcal{V}a\ \llbracket E \rrbracket \overset{*}{➨} \mathcal{V}a\ \llbracket V \rrbracket$

It is clearly useless to store a function to apply it immediately after. This optimization is expressed by the following law

(L3) $\quad \mathbf{push}_s\ E \circ \mathbf{app} = E \qquad\qquad (\mathbf{push}_s\ E \circ \lambda_s x.\ x =_{\beta s} x[E/x] = E)$

Example. Let $E \equiv (\lambda x.x)((\lambda y.y)(\lambda z.z))$ then after simplifications

$\mathcal{V}a\llbracket E \rrbracket \equiv \mathbf{push}_s(\lambda_s z.\ \mathbf{push}_s\ z) \circ (\lambda_s y.\ \mathbf{push}_s\ y) \circ (\lambda_s x.\ \mathbf{push}_s\ x)$

$$➨\ \mathbf{push}_s(\lambda_s z.\ \mathbf{push}_s\ z) \circ (\lambda_s x.\ \mathbf{push}_s\ x)\ ➨\ \mathbf{push}_s(\lambda_s z.\ \mathbf{push}_s\ z) \equiv \mathcal{V}a\llbracket \lambda z.z \rrbracket$$

The choice of redex in Λ_s does not matter anymore. The illicit (in call-by-value) reduction $E \to (\lambda y.y)(\lambda z.z)$ cannot occur within $\mathcal{V}a\llbracket E \rrbracket$. ❒

To illustrate possible optimizations, let us take the standard case of a function applied to all of its arguments $(\lambda x_1.\ldots\lambda x_n.E_0)\ E_1\ \ldots\ E_n$, then

$$\mathcal{V}a\ [\![(\lambda x_1.\ldots\lambda x_n.E_0)\ E_1\ \ldots\ E_n]\!]$$

$$= \mathcal{V}a\ [\![E_n]\!] \circ \ldots \circ \mathcal{V}a\ [\![E_1]\!] \circ \mathbf{push}_s\ (\lambda_s x_1\ldots(\mathbf{push}_s\ (\lambda_s x_n.\mathcal{V}a\ [\![E_0]\!])\ldots) \circ \mathbf{app} \circ \ldots \circ \mathbf{app}$$

$$= \mathcal{V}a\ [\![E_n]\!] \circ \ldots \circ \mathcal{V}a\ [\![E_1]\!] \circ (\lambda_s x_1.\lambda_s x_2.\ldots\lambda_s x_n.\mathcal{V}a\ [\![E_0]\!])$$

All the **app** combinators have been statically removed using associativity, (L1) and (L3). In doing so, we have avoided the construction of n intermediary closures corresponding to the n unary functions denoted by $\lambda x_1.\ldots\lambda x_n.E_0$. This optimization can be generalized to implement the *decurryfication* phase present in many implementations. In our framework, $\lambda_s x_1.\ldots\lambda_s x_n.E$ denotes a function always applied to at least n arguments (otherwise there would be **push**'s between the λ_s's). More sophisticated optimizations could be designed. For example, if a closure analysis ensures that a set of binary functions are bound to variables always applied to at least two arguments, more **app** and $\mathbf{push}_s$ combinators can be eliminated. Such information requires a potentially costly analysis and still, many functions or application contexts might not satisfy the criteria. Usually, implementations assume that higher order variables are bound to unary functions. That is, functions passed in arguments are considered unary and compiled accordingly.

The transformation $\mathcal{V}a_L$ describing left-to-right call-by-value can be derived from $\mathcal{V}a$. It is expressed as before except the rule for composition which becomes

$$\mathcal{V}a_L\ [\![E_1\ E_2]\!] = \mathcal{V}a_L\ [\![E_1]\!] \circ \mathcal{V}a_L\ [\![E_2]\!] \circ \mathbf{app}_L \qquad \textit{with} \quad \mathbf{app}_L = \lambda_s x.\lambda_s y.\ \mathbf{push}_s\ x \circ y$$

Property 2 still holds for $\mathcal{V}a_L$. Decurryfication can also be expressed although it involves slightly more complicated shifts. The equivalent of the rule (L3) is

(L4) $E : R\sigma \qquad \mathbf{push}_s\ F \circ E \circ \mathbf{app}_L = E \circ F$

Transformations $\mathcal{V}a$ and $\mathcal{V}a_L$ may produce expressions such as $\mathbf{push}_s\ E_1 \circ \mathbf{push}_s\ E_2 \circ\ldots\circ \mathbf{push}_s\ E_n \circ \ldots$. The reduction of such expressions requires a structure (such as a stack) able to store an arbitrary number of intermediate results. Some implementations make the choice of not using a data stack, hence they disallow several pushes in a row. In this case the rule for compositions of $\mathcal{V}a$ must be changed into

$$\mathcal{V}a_s\ [\![E_1\ E_2]\!] = \mathcal{V}a_s\ [\![E_2]\!] \circ (\lambda_s m.\ \mathcal{V}a_s[\![E_1]\!] \circ \lambda_s n.\ \mathbf{push}_s\ m \circ n)$$

This new rule is easily derived from the original. Similarly the rule for compositions of $\mathcal{V}a_L$ can be changed into

$$\mathcal{V}a_f[\![E_1\ E_2]\!] = \mathcal{V}a_f\ [\![E_1]\!] \circ (\lambda_s m.\ \mathcal{V}a_f[\![E_2]\!] \circ m)$$

For these expressions, the component on which $\mathbf{push}_s$ and λ_s act may be a single register. Another possible motivation for these transformations is that the produced expressions now possess a unique redex throughout the reduction. The reduction sequence must be sequential and is unique.

3.2 Compilation of control using marks

Instead of evaluating the function and its argument and then applying the results, another solution is to evaluate the argument and to apply the unevaluated function right away. Actually, this implementation is very natural in call-by-name when a function is evaluated only when applied to an argument. With call-by-value, a function can also be evaluated as an argument and in this case it cannot be immediately applied but must be returned as a result. In order to detect when its evaluation is over, there has to be a way to distinguish if its argument is present or absent: this is the role of marks. After a function is evaluated, a test is performed: if there is a mark, the function is returned as a result (and a closure is built), otherwise the argument is present and the function is applied. This technique avoids building some closures but at the price of dynamic tests.

The mark ε is supposed to be a value which can be distinguished from others. Functions are transformed into **grab** E with the intended reduction rules

$$\textbf{push}_s\ \varepsilon \circ \textbf{grab}\ E \Rightarrow \textbf{push}_s\ E$$

and $$\textbf{push}_s\ V \circ \textbf{grab}\ E \Rightarrow \textbf{push}_s\ V \circ E \quad (V \not\equiv \varepsilon)$$

Combinator **grab** and the mark ε can be defined in Λ_s. In practice, it should be implemented using a conditional which tests the presence of a mark. The transformation of right-to-left call-by-value is described in Figure 4.

$$\mathcal{V}m : \Lambda \rightarrow \Lambda_s$$

$$\mathcal{V}m[\![x]\!] = \textbf{grab}\ x$$

$$\mathcal{V}m[\![\lambda x.E]\!] = \textbf{grab}\ (\lambda_s x.\ \mathcal{V}m[\![E]\!])$$

$$\mathcal{V}m[\![E_1\ E_2]\!] = \textbf{push}_s\ \varepsilon \circ \mathcal{V}m[\![E_2]\!] \circ \mathcal{V}m[\![E_1]\!]$$

Figure 4 Compilation of Right-to-Left Call-by-Value with Marks($\mathcal{V}m$)

The correctness of $\mathcal{V}m$ is stated by Property 3 which establishes that the reduction of transformed expressions simulates the call-by-value reduction of source λ-expressions.

Property 3 $\forall E \in \Lambda,\ CBV(E) \equiv V \Leftrightarrow \mathcal{V}m[\![E]\!] \overset{*}{\Rightarrow} \mathcal{V}m[\![V]\!]$

There are two new laws corresponding to the reduction rules of **grab**:

(L5) $\textbf{push}_s\ \varepsilon \circ \textbf{grab}\ E = \textbf{push}_s\ E$

(L6) $E : R\sigma$ $\quad E \circ \textbf{grab}\ F = E \circ F$

Example. Let $E \equiv (\lambda x.x)((\lambda y.y)(\lambda z.z))$ then after simplifications

$$\mathcal{V}m[\![E]\!] \equiv \textbf{push}_s\varepsilon \circ \textbf{push}_s(\lambda_s z.\textbf{grab}\ z) \circ (\lambda_s y.\textbf{grab}\ y) \circ (\lambda_s x.\textbf{grab}\ x)$$

$$\Rightarrow \textbf{push}_s\ \varepsilon \circ \textbf{grab}\ (\lambda_s z.\ \textbf{grab}\ z) \circ (\lambda_s x.\ \textbf{grab}\ x)$$

$$\Rightarrow \textbf{push}_s\ (\lambda_s z.\ \textbf{grab}\ z) \circ (\lambda_s x.\ \textbf{grab}\ x) \Rightarrow \textbf{grab}\ (\lambda_s z.\ \textbf{grab}\ z) \equiv \mathcal{V}m[\![\lambda z.z]\!]$$ ❐

As before, when a function is known to be applied to *n* arguments, the code can be optimized to save *n* dynamic tests. Actually, it appears that $\mathcal{V}m$ is subject to the same kind of optimizations as $\mathcal{V}a$. Decurryfication and related optimizations can be expressed based on rule (L6).

It would not make much sense to consider a left-to-right strategy here. The whole point of this approach is to prevent building some closures by testing if the argument is present. Therefore the argument must be evaluated before the function.

3.3 Comparison

We compare the efficiency of codes produced by both transformations. We saw before that both transformations are subject to identical optimizations and we examined unoptimized codes only. A code produced by $\mathcal{V}m$ builds less closures than the corresponding $\mathcal{V}a$-code. A mark can be represented by one bit so $\mathcal{V}m$ is likely to be on average less greedy on space resources. Concerning time efficiency, the size of compiled expressions gives a first approximation of the overhead entailed by the encoding of the reduction strategy. It is easy to show that code expansion is linear with respect to the size of the source expression. More precisely

$$\text{If } \mathcal{Size}[\![E]\!] = n \text{ then } \mathcal{Size}(\mathcal{V}[\![E]\!]) < 3n \quad (\text{for } \mathcal{V} = \mathcal{V}a \text{ or } \mathcal{V}m)$$

This upper bound can be reached by taking for example $E \equiv \lambda x.x \ldots x$ (n occurrences of x). A more thorough investigation is possible by associating costs with the different combinators encoding the control: *push* for the cost of "pushing" a variable or a mark, *clos* for the cost of building a closure (i.e. $\mathbf{push}_s\ E$), *app* and *grab* for the cost of the corresponding combinators. If we take n_λ for the number of λ-abstractions and n_v for the number of occurrences of variables in the source expression, we have

$$\mathcal{Cost}(\mathcal{V}a[\![E]\!]) = n_\lambda\, clos + n_v\, push + (n_v\text{-}1)\, app \quad \text{and} \quad \mathcal{Cost}(\mathcal{V}m[\![E]\!]) = (n_\lambda + n_v)\, grab + (n_v\text{-}1)\, push$$

The benefit of $\mathcal{V}m$ over $\mathcal{V}a$ is to sometimes replace a closure construction and an **app** by a test and an **app**. So if *clos* is comparable to a test (for example, when returning a closure amounts to build a pair as in section 4.1) $\mathcal{V}m$ will produce more expensive code than $\mathcal{V}a$.

If closure building is not a constant time operation (as in section 4.2) $\mathcal{V}m$ can be arbitrarily better than $\mathcal{V}a$. Actually, it can change the program complexity in pathological cases. In practice, however, the situation is not so clear. When no mark is present a **grab** is implemented by a test followed by an **app**. If a mark is present the test is followed by a $\mathbf{push}_s$ (for variables) or a closure building (for λ-abstractions). So we have

$$\mathcal{Cost}(\mathcal{V}m[\![E]\!]) = (n_\lambda + n_v)\, test + \bar{p}\,(n_\lambda + n_v)\, app + p\, n_\lambda\, clos + p\, n_v\, push + (n_v\text{-}1)\, push$$

with p (resp. $\bar{p}$) representing the likelihood ($p+\bar{p}=1$) of the presence (resp. absence) of a mark which depends on the program. The best situation for $\mathcal{V}m$ is when no closure has to be built, that is $p=0$ & $\bar{p}=1$. If we take some reasonable hypothesis such as $test=app$ and $n_\lambda < n_v < 2n_\lambda$ we find that the cost of closure construction must be 3 to 4 times more costly than *app* or *test* to make $\mathcal{V}m$ advantageous. With less favorable odds such as $p=\bar{p}=1/2$, *clos* must be worth up to 6 *app*.

We are lead to conclude that $\mathcal{V}m$ should be considered only with a copy scheme for closures. Even so, tests may be too costly in practice compared to the construction of small closures. The best way would probably be to perform an analysis to detect cases when $\mathcal{V}m$ is profitable. Such information could be taken into account to get the best of each approach. We present in section 8.1 how $\mathcal{V}a$ and $\mathcal{V}m$ could be mixed.

4 Compilation of the β-Reduction

This transformation step implements the substitution. Variables are replaced by combinators acting on environments. The value of a variable is fetched from the environment when needed. Because of the lexical scope, paths to values in the environment are static. Compared to Λ_s, Λ_e adds the pair ($\mathbf{push}_e$, λ_e) which is used to define combinators.

4.1 Shared environments

The denotational-like transformation $\mathcal{As}$ is widely used among the functional abstract machines [7][18][19]. The structure of the environment is a tree of closures. A closure is added to the environment in constant time. On the other hand, a chain of links has to be followed when accessing a value. The access time complexity is O(n) where n is the number of λ_s's from the occurrence to its binding λ_s (i.e. its de Bruijn number). The transformation (Figure 5) is done relatively to a compile-time environment ρ made of pairs. The integer i in x_i denotes the rank of the variable in the environment.

$$\mathcal{A}s : \Lambda_s \to env \to \Lambda_e$$

$$\mathcal{A}s[\![E_1 \circ E_2]\!]\, \rho = \mathbf{dupl}_e \circ \mathcal{A}s[\![E_1]\!]\, \rho \circ \mathbf{swap}_{se} \circ \mathcal{A}s[\![E_2]\!]\, \rho$$

$$\mathcal{A}s[\![\mathbf{push}_s\ E]\!]\, \rho = \mathbf{push}_s\ (\mathcal{A}s[\![E]\!]\, \rho) \circ \mathbf{mkclos}$$

$$\mathcal{A}s[\![\lambda_s x.E]\!]\, \rho = \mathbf{bind} \circ \mathcal{A}s[\![E]\!]\, (\rho,x)$$

$$\mathcal{A}s[\![x_i]\!]\, (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{fst}^i \circ \mathbf{snd} \circ \mathbf{appclos}$$

Figure 5 Abstraction with Shared Environments ($\mathcal{A}s$)

$\mathcal{A}s$ needs seven new combinators to express saving and restoring environments (**dupl**$_e$, **swap**$_{se}$), closure building and opening (**mkclos**, **appclos**), access to values (**fst**, **snd**), adding a binding (**bind**). They are defined in Λ_e by

$\mathbf{dupl}_e = \lambda_e e.\ \mathbf{push}_e\ e \circ \mathbf{push}_e\ e$ $\quad$ $\mathbf{swap}_{se} = \lambda_s x.\ \lambda_e e.\ \mathbf{push}_s\ x \circ \mathbf{push}_e\ e$

$\mathbf{mkclos} = \lambda_s x.\ \lambda_e e.\ \mathbf{push}_s\ (x,e)$ $\quad$ $\mathbf{appclos} = \lambda_s (x,e).\ \mathbf{push}_e\ e \circ x$

$\mathbf{fst} = \lambda_e (e,x).\ \mathbf{push}_e\ e$ $\quad$ $\mathbf{snd} = \lambda_e (e,x).\ \mathbf{push}_s\ x$

$\mathbf{bind} = \lambda_e e.\ \lambda_s x.\ \mathbf{push}_e\ (e,x)$

$\mathcal{A}s$ correctness is stated by Property 4.

Property 4 $\forall E$: Rσ *closed*, $E \overset{*}{\Rightarrow} V \Rightarrow \mathcal{A}s[\![E]\!]\ () =_\beta \mathcal{A}s[\![V]\!]\ ()$

Transformation $\mathcal{A}s$ can be optimized by adding the rules

$$\mathcal{A}s[\![\mathbf{app}]\!]\, \rho = \mathcal{A}s[\![\lambda_s x.x]\!]\, \rho = \mathbf{bind} \circ \mathbf{snd} \circ \mathbf{appclos} = \mathbf{appclos'}$$

with $\mathbf{appclos'} = \lambda_e z.\lambda_s (x,e).\ \mathbf{push}_e\ e \circ x$

$\mathcal{A}s[\![\lambda_s x.E]\!]\, \rho = \mathbf{pop}_{se} \circ \mathcal{A}s[\![E]\!]\, \rho$ $\quad$ *with* $\mathbf{pop}_{se} = \lambda_e e.\ \lambda_s x.\ \mathbf{push}_e\ e$ *and x is not free in E*

$$\mathcal{A}s[\![\mathbf{push}_s\ x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{fst}^i \circ \mathbf{snd}$$

Variables are bound to closures stored in the environment. With the original rules, $\mathcal{A}s[\![\mathbf{push}_s x_i]\!]$ would build yet another closure. This useless “boxing” is avoided by the above rule.

Example. $\mathcal{A}s[\![\lambda_s x_1.\lambda_s x_0.\ \mathbf{push}_s\ E \circ x_1]\!]\ \rho = \mathbf{bind} \circ \mathbf{bind} \circ \mathbf{dupl}_e \circ \mathbf{push}_s\ (\mathcal{A}s[\![E]\!]\, ((\rho,x_1),x_0))$

$\circ\ \mathbf{mkclos} \circ \mathbf{swap}_{se} \circ \mathbf{fst} \circ \mathbf{snd} \circ \mathbf{appclos}$

Two bindings are added (**bind** o **bind**) to the current environment and the x_1 access is now coded by **fst** o **snd**. ❒

In our framework, $\lambda_s x_1.\ldots\lambda_s x_n.E$ denotes a function always applied to at least n arguments. So the corresponding links in the environment can be collapsed without any loss of sharing [8]. The list-like environment can become a vector locally and variable accesses have to be modified consequently.

Also, the combinator **mkclos** can be avoided by an abstraction which unfolds the pair (code,env) in the environment itself, as in TIM [10].

4.2 Copied environments

Another choice is to provide a constant access time [1][12]. In this case, the structure of the environment must be a vector of closures. Code which copies the environment (a O(*length* ρ) operation) has to be inserted in $\mathcal{A}s$ in order to avoid links.

The macro-combinator **Copy** ρ produces code that copies an environment according to ρ's structure.

$$\textbf{Copy}\ (\ldots((),x_n),\ldots,x_0) = \lambda_e e.\ \textbf{push}_e\ ()\ \circ\ (\textbf{push}_e\ e\ \circ\ \textbf{get}_n\ \circ\ \textbf{bind})\ \circ\ \ldots\ \circ\ (\textbf{push}_e\ e\ \circ\ \textbf{get}_0\ \circ\ \textbf{bind})$$

Combinators $\textbf{get}_i$ are a shorthand for $\textbf{fst}^i \circ \textbf{snd}$. However, if environments are represented by vectors, $\textbf{get}_i$ can be considered as a constant time operation and **bind** can be seen as adding a binding in a vector.

There are several abstractions according to the time of the copies. We present only the rules differing from $\mathcal{A}s$ scheme. A first solution (Figure 6) is to copy the environment just before adding a new binding (as in [10]). From the first step we know that n-ary functions $(\lambda_s x_1 \ldots \lambda_s x_n.E)$ are never partially applied and cannot be shared: they need only one copy of the environment. The overhead is placed on function entry and closure building remains a constant time operation. This transformation produces environments which can be shared by several closures but only as a whole. So, there must be an indirection when accessing the environment.

$$\mathcal{A}c1[\![\lambda_s x_i \ldots x_0.E]\!]\ \rho = \textbf{Copy}\ \overline{\rho}\ \circ\ \textbf{bind}^{i+1}\ \circ\ \mathcal{A}c1[\![E]\!]\ (\ldots(\overline{\rho},x_i)\ldots,x_0)$$

$$\mathcal{A}c1[\![x_i]\!]\ \ (\ldots(\rho,x_i),\ldots,x_0) = \textbf{get}_i\ \circ\ \textbf{app}$$

Figure 6 Copy at Function Entry ($\mathcal{A}c1$ Abstraction)

The environment $\overline{\rho}$ represents ρ restricted to variables occurring free in the subexpression *E*.

Example. $\mathcal{A}c1[\![\lambda_s x_1.\lambda_s x_0.\ \textbf{push}_s\ E_1 \circ x_1]\!]\ \rho$

$$= \textbf{Copy}\ \overline{\rho}\ \circ\ \textbf{bind}^2\ \circ\ \textbf{dupl}_e\ \circ\ \textbf{push}_s\ (\mathcal{A}c1[\![E]\!]\ ((\overline{\rho},x_1),x_0)))\ \circ\ \textbf{mkclos}\ \circ\ \textbf{swap}_{se}\ \circ\ \textbf{get}_1\ \circ\ \textbf{appclos}$$

The code builds a vector environment made of a specialized copy of the previous environment and two new bindings ($\textbf{bind}^2$) ; the x_1 access is now coded by $\textbf{get}_1$. ❒

A second solution (Figure 7) is to copy the environment when building and opening closures (as in [12]). The copy at opening time is necessary in order to be able to add new bindings in contiguous memory (the environment has to remain a vector). This transformation produces environments which cannot be shared but may be accessed directly (they can be packed with a code pointer to form a closure).

$$\mathcal{A}c2[\![\textbf{push}_s\ E]\!]\ \rho = \textbf{Copy}\ \overline{\rho}\ \circ\ \textbf{push}_s(\textbf{Copy}\ \overline{\rho}\ \circ\ \mathcal{A}c2[\![E]\!]\ \overline{\rho})\ \circ\ \textbf{mkclos}$$

$$\mathcal{A}c2[\![x_i]\!]\ \ (\ldots((\rho,x_i),x_{i\text{-}1})\ldots,x_0) = \textbf{get}_i\ \circ\ \textbf{appclos}$$

Figure 7 Copy at Closure Building and Opening ($\mathcal{A}c2$ Abstraction)

A third solution is to copy the environment only when building closures (as in [6]). In order to be able to add new bindings after closure opening, a local environment ρ_L is needed. When a closure is built, the concatenation of the two environments (ρ_G++ρ_L) is copied. The code for variables now has to specify which environment is accessed. Although the transformation scheme remains similar, every rule must be redefined to take into account the two environments. We list here only two of them.

$$\mathcal{A}c3[\![\textbf{push}_s\ E]\!]\ (\rho_G,\rho_L) = \textbf{Copy}\ (\overline{\rho}_G{+}{+}\overline{\rho}_L)\ \circ\ \textbf{push}_s\ (\mathcal{A}c3[\![E]\!]\ (\overline{\rho}_G{+}{+}\overline{\rho}_L,()))\ \circ\ \textbf{mkclos}$$

$$\mathcal{A}c3[\![\lambda_s x.E]\!]\ (\rho_G,\rho_L) = \textbf{bind3}\ \circ\ \mathcal{A}c3[\![E]\!]\ \ (\rho_G,(\rho_L,x))\quad \textit{with}\ \textbf{bind3} = \lambda_e(e_g,e_l).\ \lambda_s x.\ \textbf{push}_e\ (e_g,(e_l,x))$$

Figure 8 Abstraction with Local Environments ($\mathcal{A}c3$ Abstraction)

Local environments are not compatible with $\mathcal{V}m$: $\mathcal{A}c3[\![\textbf{grab}\ E]\!]$ would generate two different versions of $\mathcal{A}c3[\![E]\!]$ since *E* may appear in a closure or may be applied. This code duplication is obviously not realistic.

4.3 Refinements

The sequencing can be exploited by the abstraction process. Instead of saving and restoring the environment (as in $\mathcal{A}s[\![E_1 \circ E_2]\!]$), we can pass it to E_1 which may add new bindings (**bind**) but has to remove them (using **fst**) before passing the environment to E_2. For example the rules for sequences and λ_s-abstractions might be

$$\mathcal{A}seq[\![E_1 \circ E_2]\!]\ \rho = \mathcal{A}seq[\![E_1]\!]\ \rho \circ \mathcal{A}seq[\![E_2]\!]\ \rho$$

and $$\mathcal{A}seq[\![\lambda_s x.E]\!]\ \rho = \mathbf{bind} \circ \mathcal{A}seq[\![E]\!]\ (\rho,x) \circ \mathbf{fst}$$

Many other refinements are possible. For example, environments can be unfolded so that the environment stack becomes a closure stack [12]. This avoids an indirection and provides a direct access to values.

4.4 Comparison

The size of the abstracted expressions gives a first approximation of the overhead entailed by the encoding of the β-reduction. It is easy to show that code expansion is quadratic with respect to the size of the source expression. More precisely

$$\text{if } \mathcal{S}ize[\![E]\!] = \text{n then } \mathcal{S}ize(\mathcal{A}s(\mathcal{V}a[\![E]\!])) \leq n_\lambda n_v - n_v + 6n + 6$$

with n_λ the number of λ-abstractions and n_v the number of variable occurrences ($n=n_\lambda+n_v$) of the source expression. This expression reaches a maximum with $n_v=(n-1)/2$. This upper bound can be approached with, for example, $\lambda x_1....\lambda x_{n\lambda}.\ x_1...\ x_{n\lambda}$. The product $n_\lambda n_v$ indicates that the efficiency of $\mathcal{A}s$ depends equally on the number of accesses (n_v) and their length (n_λ). For $\mathcal{A}c1$ we have

$$\text{if } \mathcal{S}ize[\![E]\!] = \text{n then } \mathcal{S}ize(\mathcal{A}c1(\mathcal{V}a[\![E]\!])) \leq 6n_\lambda^{2} - 6n_\lambda + 7n + 6$$

which makes clear that the efficiency of $\mathcal{A}c1$ is not dependent of accesses. The abstractions have the same complexity order, nevertheless one may be more adapted than the other to individual source expressions. These complexities highlight the main difference between shared environments that favors building, and copied environments that favors access. Let us point out that these bounds are related to the quadratic growth implied by Turner's abstraction algorithm [29]. Balancing expressions reduces this upper bound to O(*n*log*n*) [16]. It is very likely that this technique could also be applied to λ-expressions to get a O(*n*log*n*) complexity for environment management.

The abstractions can be compared according to their memory usage too. $\mathcal{A}c2$ copies the environment for every closure, where $\mathcal{A}c1$ may share a bigger copy. So, the code generated by $\mathcal{A}c2$ consumes more memory and implies frequent garbage collections whereas the code generated by $\mathcal{A}c1$ may create space leaks and needs special tricks to plug them (see [25] section 4.2.6).

5 Compilation To Machine Code

In this section, we make explicit control transfers and propose combinator definitions. After these steps the functional expressions can be seen as realistic machine code.

5.1 Control transfers

A conventional machine executes linear code where each instruction is basic. We have to make explicit calls and returns. In our framework reducing expressions of the form **appclos** $\circ$ *E* involves evaluating a closure and returning to *E*. There are two solutions to save the return address.

We model the first one with a transformation $\mathcal{S}$ on Λ_e-expressions. It shifts the code following the function call using $\mathbf{push}_k$, and returns to it with $\mathbf{rts}_s$ ($= \lambda_s x.\lambda_k f.\ \mathbf{push}_s\ x \circ f$) when the function ends (as in [12][18][19]). Intuitively these combinators can be seen as implementing a control stack. Compared to Λ_e, Λ_k-expressions do not have **appclos** $\circ$ E code sequences.

$$\mathcal{S} : \Lambda_e \to \Lambda_k$$

$$\mathcal{S}[\![\mathbf{dupl}_e \circ E_1 \circ \mathbf{swap}_{se} \circ E_2]\!] = \mathbf{dupl}_e \circ \mathbf{push}_k\ (\mathbf{swap}_{se} \circ \mathcal{S}[\![E_2]\!]) \circ \mathbf{swap}_{ke} \circ \mathcal{S}[\![E_1]\!]$$

$$\mathcal{S}[\![\mathbf{push}_s\ E \circ \mathbf{mkclos}]\!] = \mathbf{push}_s\ \mathcal{S}[\![E]\!] \circ \mathbf{mkclos} \circ \mathbf{rts}_s$$

$$\mathcal{S}[\![\mathbf{bind} \circ E]\!] = \mathbf{bind} \circ \mathcal{S}[\![E]\!]$$

$$\mathcal{S}[\![E \circ \mathbf{appclos}]\!] = \mathbf{push}_k\ (\mathbf{appclos}) \circ \mathbf{swap}_{ke} \circ \mathcal{S}[\![E]\!]$$

$$\mathcal{S}[\![\mathbf{fst}^i \circ \mathbf{snd}]\!] = \mathbf{fst}^i \circ \mathbf{snd} \circ \mathbf{rts}_s$$

Figure 9 Compilation of Control Transfers ($\mathcal{S}$)

The combinator $\mathbf{swap}_{ke} = \lambda_k x.\ \lambda_e e.\ \mathbf{push}_k\ x \circ \mathbf{push}_e\ e$ is necessary in order to mix the new component k with the other ones. The resulting code can be simplified to avoid useless sequence breaks. To get a real machine code a further step would be to introduce labels to name sequences of code (such as E in $\mathbf{push}_x\ E$).

A second solution uses a transformation $\mathcal{S}\ell$ between the control and the abstraction phases. It transforms the expression into continuation passing style. The continuation encodes return addresses and will be then abstracted in the environment as any variable. This solution, known as stackless, is chosen in Appel's ML compiler [1]. It prevents the use of a control stack but relies heavily on the garbage collector. Appel claims that it is simple, not inefficient and well suited to implement *callcc*.

$$\mathcal{S}\ell : \Lambda_s \to \Lambda_s$$

$$\mathcal{S}\ell[\![E_1 \circ E_2]\!] = \lambda_s k.\ \mathbf{push}_s\ (\mathbf{push}_s\ k \circ \mathcal{S}\ell[\![E_2]\!]) \circ \mathcal{S}\ell[\![E_1]\!]$$

$$\mathcal{S}\ell[\![\mathbf{push}_s\ E]\!] = \lambda_s k.\ \mathbf{push}_s\ (\mathcal{S}\ell[\![E]\!]) \circ k$$

$$\mathcal{S}\ell[\![\lambda_s x.E]\!] = \lambda_s k.\lambda_s x.\ \mathbf{push}_s\ k \circ \mathcal{S}\ell[\![E]\!]$$

$$\mathcal{S}\ell[\![x]\!] = x$$

$$\mathcal{S}\ell[\![\mathbf{app}]\!] = \mathcal{S}\ell[\![\lambda_s x.\ x]\!] = \lambda_s k.\lambda_s x.\ \mathbf{push}_s\ k \circ x = \mathbf{app}_k$$

Figure 10 Compilation of the Control as a Standard Argument ($\mathcal{S}\ell$)

The following optimization removes unnecessary manipulations of the continuation k :

$$\mathbf{push}_s\ E_1 \circ (\lambda_s k.\ \mathbf{push}_s\ E_2 \circ k) = \mathbf{push}_s\ E_2 \circ E_1$$

5.2 Separate vs. merged components

The pairs of combinators (λ_s, $\mathbf{push}_s$), (λ_e, $\mathbf{push}_e$), and (λ_k, $\mathbf{push}_k$) do not have definitions yet. Each pair can be seen as encoding a component of an underlying abstract machine and their definitions specify the state transitions. We can now choose to keep the components separate or merge (some of) them. Both options share the same definition $\circ = \lambda xyz.\ x\ (y\ z)$.

Keeping the components separate brings new properties, allowing code motion and simplifications. The sequencing of two combinators on different components is commutative and administrative combinators such as $\mathbf{swap}_{se}$ are useless. Possible definitions (c, s, e being fresh variables) follow

$\lambda_s x.X = \lambda c.\lambda(s,x).\ X\ c\ s$ $\qquad$ $\mathbf{push}_s\ N = \lambda c.\lambda s.c\ (s,N)$

$\lambda_e x.X = \lambda c.\lambda s.\lambda(e,x).\ X\ c\ s\ e$ $\qquad$ $\mathbf{push}_e\ N = \lambda c.\lambda s.\lambda e.c\ s\ (e,N)$

and similarly for (λ_k, $\mathbf{push}_k$). The reduction of our expressions can be seen as state transitions of an abstract machine, e.g. :

$\mathbf{push}_s\ N\ C\ S\ E\ K \to C\ (S,N)\ E\ K$ $\qquad$ $\mathbf{push}_e\ N\ C\ S\ E\ K \to C\ S\ (E,N)\ K$

A second option is to merge all components. Here, administrative combinators remain necessary.

$\lambda_a x.X = \lambda c.\lambda(z,x).X\ c\ z$ $\qquad$ $\mathbf{push}_a\ N = \lambda c.\lambda z.\ c\ (z,N)$ $\qquad$ *with* (a ≡s,e *or* k)

6 Extensions

We describe here several extensions needed in order to handle realistic languages and to describe a wider class of implementations.

6.1 Constants, primitive operators & data structures

We have only considered pure λ-expressions because most fundamental choices can be described for this simple language. Realistic implementations also deal with constants, primitive operators and data structures. Concerning basic constants, a question is whether base-typed results are of the form $\mathbf{push}_s\ n$ or another component is introduced (e.g. $\mathbf{push}_b$, λ_b). Both options can be chosen. The latter has the advantage of marking a difference between pointers and values which can be exploited by the garbage collector. But in this case, type information must also be available to transform variables and λ-abstractions correctly. The conditional, the fix-point operator, and primitive operators acting on basic values are introduced in our language in a straightforward way. As far as data structures are concerned we can again choose to treat them as closures or separately. A more interesting choice is whether we represent them using tags or higher-order functions [10].

$$\mathcal{V}[\![rec\ f\ (\lambda x.E)]\!] = \mathbf{push}_s\ (rec_s\ f\ (\lambda_s x.\ \mathcal{V}[\![E]\!]))$$

$$\mathcal{V}[\![\text{if } E_1 \text{ then } E_2\ else\ E_3]\!] = \mathcal{V}[\![E_1]\!] \circ \mathbf{cond}_s\ (\mathcal{V}[\![E_2]\!],\ \mathcal{V}[\![E_3]\!])$$

$$\mathcal{V}[\![E_1 + E_2]\!] = \mathcal{V}[\![E_2]\!] \circ \mathcal{V}[\![E_1]\!] \circ \mathbf{plus}_s \qquad \mathcal{V}[\![n]\!] = \mathbf{push}_s\ n$$

$$\mathcal{V}[\![cons\ E_1\ E_2]\!] = \mathcal{V}[\![E_2]\!] \circ \mathcal{V}[\![E_1]\!] \circ \mathbf{cons}_s \qquad \mathcal{V}[\![head]\!] = \mathbf{head}_s$$

Figure 11 An extension with constants, primitive operators and lists

A possible extension using the component defined by ($\mathbf{push}_s$, λ_s) to store constants and tagged cells of lists is described in Figure 11 with

$$\mathbf{cons}_s = \lambda_s h.\lambda_s t.\ \mathbf{push}_s(\text{tag},h,t) \qquad \mathbf{head}_s = \lambda_s(\text{tag},h,t).\ \mathbf{push}_s\ h$$

$$\mathbf{push}_s\ n_2 \circ \mathbf{push}_s\ n_1 \circ \mathbf{plus}_s \Rightarrow \mathbf{push}_s\ n_1{+}n_2$$

6.2 Call-by-name & mixed evaluation strategies

Many of the choices discussed before remain valid for call-by-name implementations. Only the compilation of the computation rule has to be described. Figure 12 presents two possible transformations. The first one considers λ-abstractions as values and evaluates the function before applying it to the unevaluated argument. The second one (used by the TIM and Krivine machine) directly applies the function to the argument. In this scheme functions are not considered as results.

$\mathcal{Na} : \Lambda \to \Lambda_s$ | $\mathcal{Nm} : \Lambda \to \Lambda_s$
$\mathcal{Na}\ [\![x]\!] = x$ | $\mathcal{Nm}\ [\![x]\!] = x$
$\mathcal{Na}\ [\![\lambda x.E]\!] = \mathbf{push}_s\ (\lambda_s x.\ \mathcal{Na}[\![E]\!])$ | $\mathcal{Nm}\ [\![\lambda x.E]\!] = \lambda_s x.\ \mathcal{Nm}\ [\![E]\!]$
$\mathcal{Na}\ [\![E_1\ E_2]\!] = \mathbf{push}_s\ (\mathcal{Na}\ [\![E_2]\!]) \circ \mathcal{Na}\ [\![E_1]\!] \circ \mathbf{app}$ | $\mathcal{Nm}\ [\![E_1\ E_2]\!] = \mathbf{push}_s(\mathcal{Nm}\ [\![E_2]\!]) \circ \mathcal{Nm}\ [\![E_1]\!]$

Figure 12 Two Transformations for Call-by-Name ($\mathcal{Na}$ & $\mathcal{Nm}$)

The transformation $\mathcal{Nm}$ is simpler and avoids some overhead of $\mathcal{Na}$. On the other hand, making $\mathcal{Nm}$ lazy is problematic: it needs marks to be able to update closures [10][8][27]. This is exactly the same problem as with $\mathcal{Vm}$; without marks we cannot know if a function represents a result or has to be applied. In the first case, we have to return it (cbv, $\mathcal{Vm}$) or update a closure (cbn, $\mathcal{Nm}$).

Strictness analysis can be taken into account in order to produce mixed evaluation strategies. In fact, the most interesting optimization brought by strictness information is not the change of the evaluation order but avoiding thunks using unboxing [5]. If we assume that a strictness analysis has annotated the code by $\underline{E_1}\ E_2$ if E_1 denotes a strict function and $\underline{x}$ if the variable is defined by a strict λ-abstraction then $\mathcal{Na}$ can be extended as follows

$$\mathcal{Na}\ [\![\underline{x}]\!] = \mathbf{push}_s\ x \qquad \mathcal{Na}\ [\![\underline{E_1}\ E_2]\!] = \mathcal{Na}\ [\![E_2]\!] \circ \mathcal{Na}\ [\![E_1]\!] \circ \mathbf{app}$$

Underlined variables are known to be already evaluated; they are represented as unboxed values. For example, without any strictness information, the expression ($\lambda x.\ x$+1) 2 is compiled into $\mathbf{push}_s\ (\mathbf{push}_s\ 2) \circ (\lambda_s x.\ x \circ \mathbf{push}_s\ 1 \circ \mathbf{plus}_s)$. The code $\mathbf{push}_s\ 2$ will be represented as a closure and evaluated by the call x; it is the boxed representation of 2. With strictness annotations we have $\mathbf{push}_s\ 2 \circ (\lambda_s x.\ \mathbf{push}_s\ x \circ \mathbf{push}_s\ 1 \circ \mathbf{plus}_s)$ and the evaluation is the same as with call-by-value (no closure is built). Actually, more general forms of unboxing and optimizations (as in [26]) could be expressed as well.

6.3 Call-by-need and graph reduction

Call by need brings yet other options. The update mechanism can be implemented by self-updatable closures (as in [24]), by modifying the continuation (as in [12]). Updating is also central in implementations based on graph reduction. Expressing redex sharing and updating is notoriously difficult. In our framework, a straightforward idea is to add a store component along with new combinators. Each expression takes and returns the store; the sequencing ensures that the store is single-threaded. We suspect that adding store and updates in our framework will complicate correctness proofs. On the other hand, this can be done at a very late stage (e.g. after the compilation of call-by-name and β-reduction). All the transformations, correctness proofs, optimizations previously described would remain valid. The complications involved by updating would be confined in a single step. We are currently working on this issue.

7 Classical Functional Implementations

Descriptions of functional compilers often hide their fundamental structure behind implementation tricks and optimizations. Figure 13 states the main design choices which represent the skeleton of several classical implementations.

There are cosmetic differences between our description and the real implementation. Also, some extensions and optimizations are not described here. Let us state precisely the differences for the categorical abstract machine. Let $\mathcal{CAM} = \mathcal{As} \bullet \mathcal{Va}_L$ as stated in Figure 13, by simplifying this composition of transformations we get

$$\mathcal{CAM}\ [\![x_i]\!]\ \rho = \mathbf{fst}^i \circ \mathbf{snd}$$

$$\mathcal{CAM}\ [\![\lambda x.E]\!]\ \rho = \mathbf{push}_s\ (\mathbf{bind} \circ (\mathcal{CAM}[\![E]\!]\ (\rho,x))) \circ \mathbf{mkclos}$$

$$\mathcal{CAM}\ [\![E_1\ E_2]\!]\ \rho = \mathbf{dupl}_e \circ (\mathcal{CAM}[\![E_1]\!]\ \rho) \circ \mathbf{swap}_{se} \circ (\mathcal{CAM}[\![E_2]\!]\ \rho) \circ \mathbf{appclos}$$

The **fst**, **snd, dupl**$_e$ and **swap**$_{se}$ combinators match with CAM's **Fst**, **Snd**, **Push** and **Swap**. The sequence **push**$_s$ (E) o **mkclos** is equivalent to CAM's **Cur**(E). The only difference comes from the place of **bind** (at the beginning of each closure in our case). Shifting this combinator to the place where the closures are evaluated (i.e. merging it with **appclos**), we get $\lambda_s(x,e)$. **push**$_e$ e o **bind** o x, which is exactly CAM's sequence **Cons;App**.

Compiler	Transformations				Components
$\mathcal{SECD}$	$\mathcal{V}a$	$\mathcal{I}d$	$\mathcal{A}s$	$\mathcal{S}$	s (e ≡ k)
$\mathcal{CAM}$	$\mathcal{V}a_L$	$\mathcal{I}d$	$\mathcal{A}s$	$\mathcal{I}d$	s ≡ e
$\mathcal{ZAM}$	$\mathcal{V}m$	$\mathcal{I}d$	$\mathcal{A}s$	$\mathcal{S}$	s (e ≡ k)
$\mathcal{SML\text{-}NJ}$	$\mathcal{V}a_f$	$\mathcal{S}\ell$	$(\mathcal{A}c3+\mathcal{A}s)$	$\mathcal{I}d$	s e (registers)
$\mathcal{TABAC}$ (cbv)	$\mathcal{V}a$	$\mathcal{I}d$	$\mathcal{A}c2$	$\mathcal{S}$	(s ≡ e) k
$\mathcal{TABAC}$ (cbn)	$\mathcal{N}a$	$\mathcal{I}d$	$\mathcal{A}c2$	$\mathcal{S}$	(s ≡ e) k
$\mathcal{TIM}$ (cbn)	$\mathcal{N}m$	$\mathcal{I}d$	$\mathcal{A}c1$	$\mathcal{I}d$	s e

Figure 13 Several Classical Compilation Schemes

Let us quickly review the other differences between Figure 13 and real implementations. The SECD machine [18] saves environments a bit later than in our scheme. Furthermore, the control stack and the environment stack are gathered in a component called dump. The data stack is also (uselessly) saved in the dump. Actually, our replica is closer to the idealized version derived in [13]. The ZAM [19] uses a slightly different compilation of control than $\mathcal{V}m$ and has an accumulator and registers. The SML-NJ compiler [1] uses only the heap which is represented in our framework by a unique environment e. It also includes registers and many optimizations not described here. The TABAC compiler is a by-product of our work in [12] and has greatly inspired this study. It implements strict or non-strict languages by program transformations. Compared to the description above the environments are unfolded in the environment/data stack. The call-by-name TIM [10] unfolds closures in the environment as mentioned in 4.1. The transformation $\mathcal{A}c1$ has the same effect as the preliminary lambda-lifting phase of TIM.

8 Towards Hybrid Implementations

The study of the different options proved that there is no universal best choice. It is natural to strive to get the best of each world. Our framework makes intricate hybridizations and related correctness proofs possible. We first describe how $\mathcal{V}a$ and $\mathcal{V}m$ could be mixed and then how to mix shared and copied environments. In both cases, mixing is a compile time choice and we suppose that a static analysis has produced an annotated code indicating the chosen mode for each subexpression.

8.1 Mixing different control schemes

The annotations are of the form of types $T::=a \mid m \mid T_1 \xrightarrow{a/m} T_2$ with a (resp. m) for apply (resp. marks) mode. Intuitively a function E: $\alpha \xrightarrow{\delta} \beta$ takes an argument which is to be evaluated in the α-mode whereas the body is evaluated in the δ-mode. This style of annotation imposes that each variable is evaluated in a fixed mode.

$$\mathcal{M}ix\mathcal{V}[\![x^{\alpha}]\!] = \mathbf{X}_{\alpha}\, x$$

$$\mathcal{M}ix\mathcal{V}[\![\lambda x.E^{\alpha \xrightarrow{\delta} \beta}]\!] = \mathbf{X}_{\delta}\, (\lambda_s x.\ \mathcal{M}ix\mathcal{V}[\![E]\!])$$

$$\mathcal{M}ix\mathcal{V}[\![E_1{}^{\alpha \xrightarrow{\delta} \beta}\, E_2{}^{\alpha}]\!] = \mathbf{Y}_{\alpha} \circ \mathcal{M}ix\mathcal{V}[\![E_2]\!] \circ \mathcal{M}ix\mathcal{V}[\![E_1]\!] \circ \mathbf{Z}_{\delta}$$

with	$\mathbf{X}_a = \mathbf{push}_s$	$\mathbf{Y}_a = \mathbf{Id}$	$\mathbf{Z}_a = \mathbf{app}$
	$\mathbf{X}_m = \mathbf{grab}$	$\mathbf{Y}_m = \mathbf{push}_s\ \varepsilon$	$\mathbf{Z}_m = \mathbf{Id}$

Figure 14 Hybrid Compilation of Right to Left Call-by-Value

We suppose, as in 3.2, that it is possible to distinguish the special closure ε from the others. The values produced by each mode are of the same form and no coercion is necessary. $\mathcal{M}ix\mathcal{V}$ (Figure 14) just adds $\mathbf{push}_s$ ε before the evaluation of an argument in mode *m* and **app** after the evaluation of a function in mode *a*. Results are returned using $\mathbf{push}_s$ or **grab** according to their associated mode.

8.2 Mixing different abstraction schemes

One solution uses coercion functions which fit the environment into the chosen structure (vector or linked list). The compilation can then switch from one world to another. In particular, switching from $\mathcal{A}s$ to $\mathcal{A}c1$ creates a kind of strict display (by comparison to the lazy display of [22]).

$$\mathcal{A}s[\![E]\!]\ \rho = \mathbf{List2Vect}\ \rho \circ \mathcal{A}c1[\![E]\!]\ \rho$$

Another solution uses environments mixing lists and vectors (as in [28]).

$$\mathcal{M}ix\mathcal{A}[\![\lambda_s x.E^{\ \theta,\oplus}]\!]\ \rho = \mathbf{Mix}\ \rho\ \theta \circ \mathbf{bind}_{\oplus} \circ \mathcal{M}ix\mathcal{A}[\![E]\!]\,(\theta \oplus x)$$

$$\mathcal{M}ix\mathcal{A}[\![x_i]\!]\,(\ldots(\rho,\rho_i),\ldots,\rho_0) = \mathbf{fst}^i \circ \mathbf{snd} \circ \mathcal{M}ix\mathcal{A}[\![x_i]\!]\ \rho_i \qquad \textit{with } x_i \textit{ in } \rho_i$$

$$\mathcal{M}ix\mathcal{A}[\![x_i]\!]\ [\rho{:}\rho_i{:}\ldots{:}\rho_0] = \mathbf{get}_{\mathrm{i}} \circ \mathcal{M}ix\mathcal{A}[\![x_i]\!]\ \rho_i \qquad \textit{with } x_i \textit{ in } \rho_i$$

$$\mathcal{M}ix\mathcal{A}[\![x_i]\!]\ (\ldots(\rho,x_i),\ldots,x_0) = \mathbf{fst}^i \circ \mathbf{snd} \circ \mathbf{app}$$

$$\mathcal{M}ix\mathcal{A}[\![x_i]\!]\ [\rho{:}x_i{:}\ldots{:}x_0] = \mathbf{get}_i \circ \mathbf{app}$$

Figure 15 Hybrid Abstraction

Each λ-abstraction is annotated by a new mixed environment structure θ and ⊕ which indicates how to bind the current value (as a vector ":" or as a link ","). Mixed structures are built by **bind:**, **bind,** and the macro-combinator **Mix** which copies and restructures the environment ρ according to the annotation θ (Figure 15). Paths to values are now expressed by sequences of $\mathbf{fst}^i \circ$ **snd** and $\mathbf{get}_j$. The abstraction algorithm distinguishes vectors from lists in the compile time environment using constructors ":" and ",".

9 Conclusion

In this paper, we have presented a framework to describe, prove and compare functional implementation techniques and optimizations (see Figure 2 in 2.5 for a summary). Our first intermediate language Λ_s bears strong similarities with CPS-expressions. Indeed, if we take combinator definitions (**DEF1**) (section 2.5) we naturally get Fischer's CPS transformation [11] from $\mathcal{V}a_f$ (section 3.1). On the other hand, our combinators are not fully defined (they just have to respect a few properties) and we avoid issues such as administrative reductions. We see Λ_s as a powerful and more abstract framework than CPS to express different reduction strategies. As pointed out by Hatcliff & Danvy [14], Moggi's computational metalanguage [23] is also a more abstract alternative language to CPS. Arising from different roots, Λ_s is surprisingly close to Moggi's. In particular, we may interpret the monadic constructs **[**E**]** as **push** E and (**let** $x \Leftarrow E_1$ **in** E_2) as $E_1 \circ \lambda_s x.E_2$ and get back the monadic laws (let.β), (let.η) and (ass) [23]. On the other hand, we disallow unrestricted applications and Λ_s-expressions are more general than merely combinations of **[ ]** and **let**'s.

Related work also includes the derivation of abstract machines from denotational [30] or operational semantics [13] [27]. Their goal is to provide a methodology to formally derive implementations for a (potentially large) class of programming languages. A few works explore the relationship between two abstract machines such as TIM and the G-Machine [4][25] and CMCM and TIM [21]. The goal is to show the equivalence between seemingly very different implementations. Also, let us mention Asperti [2] who provides a categorical understanding of the Krivine machine and an extended CAM.

Our approach focuses on the description and comparison of fundamental options. The use of program transformations appeared to be suited to model precisely and completely the compilation process. Many standard optimizations (decurryfication, unboxing, hoisting, peephole optimizations) can be expressed as program transformations as well. This unified framework simplifies correctness proofs and makes it possible to reason about the efficiency of the produced code as well as about the complexity of transformations themselves. Our mid-term goal is to provide a general taxonomy of known implementations of functional languages. The last tricky task standing in the way is the expression of destructive updates. This is crucial in order to completely describe call-by-need and graph reduction machines. We hinted in section 6.3 how it could be done and we are currently investigating this issue. Still, as suggested in section 6, many options and optimizations (more than we were able to describe in this paper) are naturally expressed in our framework. Nothing should prevent us from completing our study of call-by-value and call-by-name implementations.

Acknowledgments. Thanks to Charles Consel, Luke Hornof and Daniel Le Métayer for commenting an earlier version of this paper.

References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press. 1992.

[2] A. Asperti. A categorical understanding of environment machines. *Journal of Functional Programming*, 2(1), pp.23-59,1992.

[3] G. Argo. Improving the three instruction machine. In *Proc. of FPCA'89,* pp. 100-115, 1989.

[4] G. Burn, S.L. Peyton Jones and J.D. Robson. The spineless G-machine. In *Proc. of LFP'88*, pp. 244-258, 1988.

[5] G. Burn and D. Le Métayer. Proving the correctness of compiler optimisations based on a global analysis. *Journal of Functional Programming*, 1995. (to appear).

[6] L. Cardelli. Compiling a functional language. In *Proc. of LFP'84*, pp. 208-217, 1984.

[7] G. Cousineau, P.-L. Curien and M. Mauny, The categorical abstract machine. *Science of Computer Programming*, 8(2), pp. 173-202, 1987.

[8] P. Crégut. *Machines à environnement pour la réduction symbolique et l'évaluation partielle*. Thèse de l'université de Paris VII, 1991.

[9] R. Douence and P. Fradet. A taxonomy of functional language implementations. Part I: Call-by-Value, *INRIA Research Report,* 1995. (to appear)

[10] J. Fairbairn and S. Wray. Tim: a simple, lazy abstract machine to execute supercombinators. In *Proc of FPCA'87,* LNCS 274, pp. 34-45, 1987.

[11] M. J. Fischer. Lambda-calculus schemata. In *Proc. of the ACM Conf. on Proving Properties about Programs*, Sigplan Notices, Vol. 7(1), pp. 104-109,1972.

[12] P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. *ACM Trans. on Prog. Lang. and Sys.*, 13(1), pp. 21-51, 1991.

[13] J. Hannan. From operational semantics to abstract machines. *Math. Struct. in Comp. Sci.*, 2(4), pp. 415-459, 1992.

[14] J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Proc. of POPL'94*, pp. 458-471, 1994.

[15] T. Johnsson. *Compiling Lazy Functional Languages.* PhD Thesis, Chalmers University, 1987.

[16] M. S. Joy, V. J. Rayward-Smith and F. W. Burton. Efficient combinator code. *Computer Languages*, 10(3), 1985.

[17] D. Kranz, R. Kesley, J. Rees, P. Hudak, J.Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices, 21(7)*, pp.219-233, 1986.

[18] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4), pp.308-320, 1964.

[19] X. Leroy. The Zinc experiment: an economical implementation of the ML language. *INRIA Technical Report 117*, 1990.

[20] R. D. Lins. Categorical multi-combinators. In *Proc. of FPCA'87*, LNCS 274, pp. 60-79, 1987.

[21] R. Lins, S. Thompson and S.L. Peyton Jones. On the equivalence between CMC and TIM. *Journal of Functional Programming*, 4(1), pp. 47-63, 1992.

[22] E. Meijer and R. Paterson. Down with lambda lifting. copies available at: erik@cs.kun.nl, 1991.

[23] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55-92, 1991.

[24] S.L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127-202, 1992.

[25] S. L. Peyton Jones and D. Lester. *Implementing functional languages, a tutorial*. Prentice Hall, 1992.

[26] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proc. of FPCA'91*, LNCS 523, pp.636-666, 1991.

[27] P. Sestoft. Deriving a lazy abstract machine. *Technical Report 1994-146, Technical University of Denmark*, 1994.

[28] Z. Shao and A. Appel. Space-efficient closure representations. In *Proc. of LFP'94*, pp. 150-161,1994.

[29] D.A. Turner. A new implementation technique for applicative languages. *Soft. Pract. and Exper.*, 9, pp. 31-49, 1979.

[30] M. Wand. Deriving target code as a representation of continuation semantics. *ACM Trans. on Prog. Lang. and Sys.*, 4(3), pp. 496-517, 1982.