A Taxonomy of Functional Language Implementations : Part I : Call by Value

Rémi Douence, Pascal Fradet

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

A Taxonomy of Functional Language Implementations

Part I : Call by Value

Rémi Douence and Pascal Fradet

N° 2783

Janvier 1996

PROGRAMME 2

Calcul symbolique, programmation et
génie logiciel

ISSN 0249-6399

A Taxonomy of Functional Language Implementations*

Part I : Call by Value

Rémi Douence and Pascal Fradet

[douence;fradet]@irisa.fr

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet Lande

Rapport de recherche n°2783— Janvier 1996 — 52 pages

Abstract: We present a unified framework to describe and compare functional language implementations. We express the compilation process as a succession of program transformations in the 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.

Key-words: Compilation, optimizations, program transformations, λ-calculus, combinators

(Résumé : tsvp)

* This research report is the extended version of "*Towards a taxonomy of functional languages implementations*" appeared in *Proc. of 7th Int. Symp. on Programming Languages: Implementations, Logics and Programs* (1995) [10]. It includes a more thorough presentation of the formal framework (e.g. connection with λ-calculus and CPS conversion) and presents several new alternate abstraction algorithms. It also adds an annex gathering proofs of properties.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : (33) 99 84 71 00 - Télécopie : (33) 99 38 38 32

Une Taxonomie des implantations des langages fonctionnels*

Partie I : Appel par valeur

Résumé : Nous proposons un cadre formel pour décrire et comparer les implantations de langages fonctionnels. Nous décrivons le processus de compilation comme une suite de transformations de programmes dans le cadre fonctionnel. Les choix fondamentaux de mise en œuvre ainsi que les optimisations s'expriment naturellement comme des transformations différentes. Les avantages de cette approche sont de décomposer et de structurer la compilation, de simplifier les preuves de correction et de permettre des comparaisons formelles en étudiant chaque transformation ou leur composition. Nous nous concentrons sur les mises en œuvre de l'appel par valeur, décrivons et comparons les différentes options et classifions les compilateurs ou machines abstraites classiques. Ce travail a aussi pour but d'ouvrir de nouvelles perspectives et nous indiquons comment différents choix pourraient cohabiter dans des implantations hybrides plus efficaces.

Mots-clé : Compilation, optimisations, transformation de programmes, λ-calcul, combinateurs

* Ce rapport de recherche est la version étendue de l'article "*Towards a taxonomy of functional languages implementations*" paru dans *Proc. of 7th Int. Symp. on Programming Languages: Implementations, Logics and Programs* (1995) [10]. Il ajoute une présentation plus approfondie du cadre formel (e.g. la connexion avec le λ-calcul et la conversion CPS) et décrit plusieurs autres algorithmes d'abstractions. L'annexe rassemble les preuves de propriétés énoncées dans le texte.

1 Introduction

One of the most studied issues concerning functional languages is their implementation. Since the seminal proposal of Landin, 30 years ago [19], 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 [19], the FAM [6], the CAM [7], the CMCM [21], the TIM [11], the ZAM [20], the G-machine [16] 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][13][18]. Furthermore, numerous papers present optimizations often adapted to a specific abstract machine or a specific approach [3][4][17]. 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 [26][22] 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 $\Lambda \xrightarrow{T1} \Lambda_1 \xrightarrow{T2} \ldots \xrightarrow{Tn} \Lambda_n$ 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. 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 manage-

ment). Each section includes a comparison of the main options. Section 5 is devoted to two other simple steps leading to machine code. In section , 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 $\Lambda \rightarrow \Lambda_s \rightarrow \Lambda_e \rightarrow \Lambda_k$. 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 2.6.

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.

Λ_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 in 2.5 and in 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

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

(η_s) $\qquad \lambda_s x.(\mathbf{push}_s\ x \circ E) = E \qquad$ *if x 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.

(assoc) $\qquad (E_1 \circ E_2) \circ E_3 = E_1 \circ (E_2 \circ E_3)$

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 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. Sub-expressions 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). We note $\blacktriangleright$ the compatible closure of β_s-reduction (i.e. β_s + the natural inductive rules $E \blacktriangleright N \Rightarrow E \circ F \blacktriangleright N \circ F$ and $F \blacktriangleright N \Rightarrow E \circ F \blacktriangleright E \circ N$) and $\overset{*}{\blacktriangleright}$ the reflexive, transitive closure of $\blacktriangleright$.

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 as a consequence :

Property 1 *In Λ_s all reduction strategies are normalizing.*

This property is the key point to view transformations from Λ to Λ_s as compiling the reduction order.

2.3 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).

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

Figure 1 Λ_s **typed subset (**Λ_s^σ **)**

This does not impose any restrictions on source λ-expressions. For example, we can allow reflexive types ($\alpha{=}\alpha{\to}\alpha$) to type any source 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.

Property 2 (***subject reduction property***). *If $E \overset{*}{\blacktriangleright} F$ then $\Gamma \vdash E : \sigma \Rightarrow \Gamma \vdash F : \sigma$*

The type system restricts the set of normal forms (which in general includes expressions such as $\mathbf{push}_s\ E_1 \circ \mathbf{push}_s\ E_2$) and we have the following natural facts.

Property 3 *A closed expression $E{:}\tau$ is either canonical (i.e. $E \equiv \mathbf{push}_s\ V$ or $\lambda_s x.F$) or reducible.*

We deduce from these two properties that

Property 4 - *If a closed expression $E{:}\mathsf{R}\sigma$ has a normal form then $E \overset{*}{\blacktriangleright} \mathbf{push}_s\ V$*

- *If a closed expression $E{:}\sigma \to_s \tau$ has a normal form then $E \overset{*}{\blacktriangleright} \lambda_s x.F$*

Another consequence of the type system, is that the reduction of typed closed expressions can be specified by the following natural semantics:

$$\frac{E_1 \succ \mathbf{push}_s\ V \qquad E_2 \succ \lambda_s x.F \qquad F[V/x] \succ N}{E_1 \circ E_2 \succ N} \quad \text{(with } N \text{ a normal form)}$$

whereas for general expressions, we should add the inference rule:

$$\frac{E_1 \succ N_1 \qquad E_2 \succ N_2 \qquad N_1 \not\equiv \mathbf{push}_s\ V \text{ or } N_2 \not\equiv \lambda_s x.F}{E_1 \circ E_2 \succ N_1 \circ N_2} \quad \text{(with } N_1, N_2 \text{ normal forms)}$$

Property 5 $\forall E \in \Lambda_s \quad E \overset{*}{\rightarrow} N \Leftrightarrow E \succ N$ *(with N a normal form)*

Note that (assoc) may produce ill-typed programs. We can use (assoc), (η_s) or the laws below, as long as the final expression is well typed, the single rule of reduction is sufficient. If we allow an unrestricted use of (assoc) the reduction should be done modulo associativity. 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 three of them.

if x does not occur free in F	$(\lambda_s x.E) \circ F = \lambda_s x.(E \circ F)$	(L1)
$\forall E_1 : R\sigma$, *if x does not occur free in* E_2	$E_1 \circ (\lambda_s x.E_2 \circ E_3) = E_2 \circ E_1 \circ (\lambda_s x.E_3)$	(L2)
$\forall E_1 : R\sigma$, $E_2 : R\sigma$ *and* $x \not\equiv y$	$E_1 \circ E_2 \circ (\lambda_s x.\lambda_s y.E_3) = E_2 \circ E_1 \circ (\lambda_s y.\lambda_s x.E_3)$	(L3)

These rules permit code to be moved inside or outside function bodies or to invert the evaluation of two results. 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 Connection with the λ-calculus

Λ_s is a convenient abstraction to express reduction strategies. But recall that it is also a subset of the λ-calculus made of combinators. An important point is that we do not have to give a precise definition to combinators. We just assume that they respect properties (β_s), (η_s) and (assoc). Definitions do not have to be chosen until the very last step. However, in order to provide some intuition, we give here one possible definition (alternative definitions are presented in section 5.2).

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

i.e. $\circ = \lambda a.\lambda b.\lambda c.a\;(b\;c)$ $\quad$ $\mathbf{push}_s = \lambda a.\lambda c.c\;a$ $\quad$ $\lambda_s x.E = (\lambda a.\lambda c.\lambda x.a\;x\;c)\;(\lambda x.E)$

$(E_1 \circ E_2)\;C \rightarrow E_1\;(E_2\;C)$ $\quad$ $(\mathbf{push}_s\;E)\;C \rightarrow C\;E$ $\quad$ $(\lambda_s x.E)\;C\;X \rightarrow E[X/x]\;C$

In general, the reduction rules of combinators would be of the form

$$(E_1 \circ E_2)\;X_1 \ldots X_n \rightarrow Y_1 \ldots Y_m \qquad \mathbf{push}_s\;E\;X_1 \ldots X_n \rightarrow Z_1 \ldots Z_p$$

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_s-code can be seen as the state of an abstract machine. We do not want to commit ourselves to a definite definition of combinators, however we want that the reduction from left to right using the rules of combinators simulates the reduction in Λ_s. That is to say :

Property 6 $(\forall E{:}\sigma)\;E \overset{*}{\blacktriangleright} N \Rightarrow (\forall X_1,\ldots,X_n)\;E\;X_1 \ldots X_n \overset{*}{\rightarrow} N\;X_1 \ldots X_n$ *(N normal form)*

In order to enforce this property, it is sufficient to check that there exits n so that, for any closed expressions $E,F,X_1,\ldots,X_n$,

If $E\;X_1 \ldots X_n \overset{*}{\rightarrow} \mathbf{push}_s\;V\;X_1 \ldots X_n$ and $F\;X_1 \ldots X_n \overset{*}{\rightarrow} (\lambda_s x.G)\;X_1 \ldots X_n$ then

$$(E \circ F)\;X_1 \ldots X_n \overset{*}{\rightarrow} G[V/x]\;X_1 \ldots X_n \qquad \text{(C1)}$$

For example, with (**DEF1**), if for any E, F, $C\;E\;C \overset{*}{\rightarrow} \mathbf{push}_s\;V\;C$ and $F\;C \overset{*}{\rightarrow} (\lambda c.\lambda x.G\;c)\;C$

then $(E \circ F)\;C \rightarrow E\;(F\;C) \overset{*}{\rightarrow} \mathbf{push}_s\;V\;(F\;C) \overset{*}{\rightarrow} F\;C\;V \overset{*}{\rightarrow} (\lambda c.\lambda x.G\;c)\;C\;V \rightarrow G[V/x]\;C$

That establishes (C1) and therefore Property 6 holds for (**DEF1**).

In the final stage of the compilation process it might be convenient to use (assoc) to reshape the code. For example, we may want the code to be of the form $E_1 \circ (E_2 \circ (\ldots E_n))$, the E_i's being basic instructions. Indeed, this shape along with the left to right reduction makes the reduction of expressions akin to the execution of machine code. If we want the combinators and the classical β-reduction to implement naturally ➨+(assoc), it is sufficient to check that $(E_1 \circ E_2) \circ E_3 =_\beta E_1 \circ (E_2 \circ E_3)$. Contrary to η, the β-rule is valid (i.e. does not change termination properties) in the weak λ-calculus. Since the left-to-right reduction $\overset{*}{\rightarrow}$ is nothing else than (weak) call by name, (assoc) can be applied without restrictions.

Another connection with λ-calculus can be established using an inverse transformation from Λ_s to Λ (Figure 2).

$[\![\ ]\!]^{-1} : \Lambda_s \rightarrow \Lambda$

$[\![E_1 \circ E_2]\!]^{-1} = [\![E_2]\!]^{-1} [\![E_1]\!]^{-1}$

$[\![\mathbf{push}_s\ E]\!]^{-1} = [\![E]\!]^{-1}$

$[\![\lambda_s x.E]\!]^{-1} = \lambda x.\ [\![E]\!]^{-1}$

$[\![x]\!]^{-1} = x$

Figure 2 Back to λ-expressions

If two Λ_s-expressions are $\beta_s\eta_s$-convertible then their corresponding source expressions are equal in the extensional λ-calculus.

Property 7 $E =_{\beta s\, \eta s} F \Rightarrow \lambda\eta \vdash [\![E]\!]^{-1} = [\![F]\!]^{-1}$

The reverse implication is clearly not true: expressions in Λ_s encode a specific reduction order, possibly unsafe like call-by-value, whereas equality in the λ-calculus is bound to normal order. For example, let

$\Omega_s = \mathbf{push}_s\ (\lambda_s x.\mathbf{push}_s x \circ x) \circ (\lambda_s x.\mathbf{push}_s x \circ x)$ $\quad ([\![\Omega_s]\!]^{-1} = (\lambda x.x\ x)\ (\lambda x.x\ x) = \Omega)$

then $[\![\Omega_s \circ \lambda_s x.\lambda_s y.y]\!]^{-1} = (\lambda x.\lambda y.y)\ \Omega = (\lambda y.y)$

whereas $\Omega_s \circ \lambda_s x.\lambda_s y.y$ loops and has no head normal form.

2.6 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 ($\mathbf{push}_s$, λ_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 ($\mathbf{push}_e$, λ_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 (shared) environments ($\mathcal{A}s$) or vector-like (copied) environments ($\mathcal{A}c$). For the latter choice, there are several transformations depending on the way environments are copied

($\mathcal{A}c1$, $\mathcal{A}c2$, $\mathcal{A}c3$). We also present a family of generic transformations modelling other choices related to the management of the environment stack and the representation of closures.

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 ($\mathcal{SL}$) on Λ_s-expressions). The language Λ_k makes calls and returns explicit. It introduces the pair (**push**$_k$, λ_k) which specifies a component storing return addresses.

Control	Λ_s	(**push**$_s$, λ_s)	$\mathcal{V}a$ $\mathcal{V}a_L$ $\underline{\mathcal{V}a_s}$ $\underline{\mathcal{V}a_f}$ $\mathcal{V}m^{\#}$	(+ $\underline{\mathcal{SL}}^*$)
Abstraction	Λ_e	(**push**$_e$, λ_e)	$\mathcal{A}s$ $\mathcal{A}c1$ $\mathcal{A}c2$ $\mathcal{A}c3^{\#}$	(+ $\mathcal{A}g$ *family instantiations*)
Transfers	Λ_k	(**push**$_k$, λ_k)	$\mathcal{S}^*$	

Figure 3 Summary of the Main Compilation Steps and Options

Figure 3 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 **push**$_x$ and λ_x, are described along with transformations. To simplify the presentation, we also use syntactic sugar such as tuples (x_1,…,x_n) and pattern-matching $\lambda_x(x_1,\ldots,x_n).E$.

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 [11] and used in ZINC [20].

3.1 Compilation of control using apply ("eval-apply model")

In this scheme, 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.

3.1.1 Standard transformations

The compilation of right-to-left call-by-value is described in Figure 4. 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. $\mathbf{push}_s\ E$).

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

$\mathcal{V}a\ [\![x]\!] = \mathbf{push}_s\ x$

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

$\mathcal{V}a\ [\![E_1\ E_2]\!] = \mathcal{V}a\ [\![E_2]\!]\ \mathrm{o}\ \mathcal{V}a\ [\![E_1]\!]\ \mathrm{o}\ \mathbf{app}$ with $\mathbf{app} = \lambda_s x.x$

Figure 4 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 o and "apply" for **app**. $\mathcal{V}a$ produces well-typed expressions of result type (Property 8).

Property 8 $\forall E \in \Lambda$, *E closed* $\vdash E: \sigma \Rightarrow \vdash \mathcal{V}a\ [\![E]\!] : \mathsf{R}\overline{\sigma}$ *with* $\overline{\sigma \rightarrow \tau} = \overline{\sigma} \rightarrow_s \mathsf{R}\overline{\tau}$ *and* $\overline{\alpha} = \alpha$ *(*α *type variable)*

Its correctness is stated by Property 9 which establishes that the reduction of transformed expressions ($\overset{*}{\blacktriangleright}$) simulates the call-by-value reduction (*CBV*) of source λ-expressions.

Property 9 $\forall E$ *closed* $\in \Lambda$, $E \underset{cbv}{\Rightarrow} V \Leftrightarrow \mathcal{V}a\ [\![E]\!] \overset{*}{\blacktriangleright} \mathcal{V}a\ [\![V]\!]$

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

$$\mathbf{push}_s\ E\ \mathrm{o}\ \mathbf{app} = E \qquad (\mathbf{push}_s\ E\ \mathrm{o}\ \lambda_s x.x =_{\beta s} x[E/x] = E) \qquad \text{(L4)}$$

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

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

$$\blacksquare\ \mathbf{push}_s(\lambda_s z.\mathbf{push}_s\ z) \circ (\lambda_s x.\mathbf{push}_s\ x) \blacksquare\ \mathbf{push}_s(\lambda_s z.\mathbf{push}_s\ z) \equiv \mathcal{V}a\ [\![\lambda z.z]\!]$$

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

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

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

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

$= \mathcal{V}a\ [\![E_n]\!] \circ \ldots \circ \mathcal{V}a\ [\![E_1]\!] \circ (\lambda_s x_1 \ldots (\mathbf{push}_s\ (\lambda_s x_n.\mathcal{V}a\ [\![E_0]\!])\ldots) \circ \mathbf{app} \ldots \circ \mathbf{app}$ (L4)

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

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

$\circ\ \mathbf{app}) \circ \mathbf{app} \ldots \circ \mathbf{app}$ (L1)

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

All the **app** combinators have been statically removed. 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. An important point to note is that, in our framework, $\lambda_s x_1 \ldots \lambda_s x_n.E$ denotes always a function 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 is expressed as before except the rule for composition which becomes

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

It can be derived from $\mathcal{V}a\ [\![E_1\ E_2]\!]$ as follows

$\mathcal{V}a\ [\![E_1\ E_2]\!] = \mathcal{V}a\ [\![E_2]\!] \circ \mathcal{V}a\ [\![E_1]\!] \circ \mathbf{app}$

$= \mathcal{V}a\ [\![E_2]\!] \circ \mathcal{V}a\ [\![E_1]\!] \circ (\lambda_s y.\lambda_s x.\mathbf{push}_s\ x \circ \mathbf{push}_s\ y \circ \mathbf{app})$ (η_s)

$$= \mathcal{V}a\,[\![E_1]\!] \circ \mathcal{V}a\,[\![E_2]\!] \circ (\lambda_s x.\lambda_s y.\mathbf{push}_s\, x \circ y) \qquad \text{(L3), (L4)}$$

Property 9 still holds for $\mathcal{V}a_L$. Decurryfication can also be expressed although it involves slightly more complicated shifts. The equivalent of the rule (L4) is

$$E : R\sigma \qquad \mathbf{push}_s\, F \circ E \circ \mathbf{app}_L = E \circ F \qquad \text{(L5)}$$

3.1.2 Stackless variants

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 and, therefore, disallow several pushes in a row. In this case, the rule for compositions of $\mathcal{V}a$ should 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 $\mathcal{V}a_L$ should 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 ("push-enter model")

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.

3.2.1 Standard transformation

The mark ε is supposed to be a value which can be distinguished from others. Functions are transformed into $\mathbf{grab}_s$ E with the intended reduction rules

$$\mathbf{push}_s\, \varepsilon \circ \mathbf{grab}_s\, E \Rightarrow \mathbf{push}_s\, E$$

and $$\mathbf{push}_s\, V \circ \mathbf{grab}_s\, E \Rightarrow \mathbf{push}_s\, V \circ E \qquad (V \not\equiv \varepsilon)$$

Combinator $\mathbf{grab}_s$ and the mark ε can be defined in Λ_s^*. In practice, $\mathbf{grab}_s$ would 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 5.

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

$\mathcal{V}m \llbracket x \rrbracket = \mathbf{grab}_s\ x$

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

$\mathcal{V}m \llbracket E_1\ E_2 \rrbracket = \mathbf{push}_s\ \varepsilon \circ \mathcal{V}m \llbracket E_2 \rrbracket \circ \mathcal{V}m \llbracket E_1 \rrbracket$

Figure 5 Compilation of Right-to-Left Call-by-Value with Marks ($\mathcal{V}m$)

The correctness of $\mathcal{V}m$ is stated by Property 10 which establishes that the reduction of transformed expressions simulates the call-by-value reduction of source λ-expressions.

Property 10 $\forall E \text{ closed} \in \Lambda, E \xrightarrow[cbv]{} V \Leftrightarrow \mathcal{V}m \llbracket E \rrbracket \overset{*}{\rightarrow} \mathcal{V}m \llbracket V \rrbracket$

There are two new laws corresponding to the reduction rules of $\mathbf{grab}_s$:

$$\mathbf{push}_s\ \varepsilon \circ \mathbf{grab}_s\ E = \mathbf{push}_s\ E \qquad \text{(L6)}$$

$$E : R\sigma \qquad E \circ \mathbf{grab}_s\ F = E \circ F \qquad \text{(L7)}$$

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

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

$\rightarrow \mathbf{push}_s\ \varepsilon \circ \mathbf{grab}_s\ (\lambda_s z.\mathbf{grab}_s\ z) \circ (\lambda_s x.\mathbf{grab}_s\ x)$

$\rightarrow \mathbf{push}_s\ (\lambda_s z.\mathbf{grab}_s\ z) \circ (\lambda_s x.\mathbf{grab}_s\ x)$

$\rightarrow \mathbf{grab}_s\ (\lambda_s z.\mathbf{grab}_s\ z) \equiv \mathcal{V}m \llbracket \lambda z.z \rrbracket$ ❐

As before, when a function $\lambda x_1 \ldots \lambda x_n.E$ 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 rules (L7) and (L2). Let us take again the expression $(\lambda x_1 \ldots \lambda x_n.E_0)\ E_1 \ldots E_n$, then

$\mathcal{V}m \llbracket (\lambda x_1 \ldots \lambda x_n.E_0)\ E_1 \ldots E_n \rrbracket$

$\equiv (\mathbf{push}_s\ \varepsilon \circ \mathcal{V}m \llbracket E_n \rrbracket) \circ \ldots \circ (\mathbf{push}_s\ \varepsilon \circ \mathcal{V}m \llbracket E_1 \rrbracket) \circ \mathbf{grab}_s(\lambda_s x_1 \ldots \mathbf{grab}_s(\lambda_s x_n.\mathcal{V}m \llbracket E_0 \rrbracket) \ldots)$

* For example : $\mathbf{grab}_s\ E \equiv \mathbf{push}_s\ E \circ \lambda_s x.\lambda_s(m,v).\mathbf{push}_s\ (\mathbf{push}_s(\mu,x)) \circ \mathbf{push}_s\ (\mathbf{push}_s\ v \circ x)) \circ m$
Each argument is associated with a mark in a pair. The mark $\mu \equiv \lambda_s x.\lambda_s y.x$ selects the first alternative (apply the function E) whereas $\varepsilon \equiv (\lambda_s x.\lambda_s y.y,id)$ is a mark (associated with a dummy function id) selecting the second alternative (yield E as result).

$= (\mathbf{push}_s\ \varepsilon \circ \mathcal{V}m\ [\![E_n]\!]) \circ \ldots \circ (\mathbf{push}_s\ \varepsilon \circ \mathcal{V}m\ [\![E_1]\!]) \circ (\lambda_s x_1 \ldots \mathbf{grab}_s\ (\lambda_s x_n.\mathcal{V}m\ [\![E_0]\!])\ldots)$ (L7)

$= (\mathbf{push}_s\ \varepsilon \circ \mathcal{V}m\ [\![E_n]\!]) \circ \ldots \circ (\mathbf{push}_s\ \varepsilon \circ \mathcal{V}m\ [\![E_1]\!]) \circ (\lambda_s x_1.(\mathbf{push}_s\ \varepsilon \circ \mathcal{V}m\ [\![E_2]\!])$

$\circ(\ \mathbf{grab}_s(\ \lambda_s x_2 \ldots)\ldots)$ (L2)

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

All the $\mathbf{grab}_s$ have been statically removed and we have avoided *n* dynamic tests.

3.2.2 Variants

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.However, there are other, closely related, transformations using marks. A generic transformation can be described as follows :

$\mathcal{V}m_g\ [\![x]\!] = x\ x$

$\mathcal{V}m_g\ [\![\lambda x.E]\!] = \gamma\ (\lambda_s x.\mathcal{V}m_g\ [\![E]\!])$

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

$x\gamma$ and z being combinators such that $\gamma = x \circ z$, $\mathbf{push}_s\ \varepsilon \circ \gamma\ (E) \Rightarrow \mathbf{push}_s\ z(E)$, and $\mathbf{push}_s\ V \circ \gamma\ (E) \Rightarrow \mathbf{push}_s\ V \circ E$

Figure 6 Generic Compilation of Right-to-Left Call-by-Value with Marks ($\mathcal{V}m_g$)

We get back $\mathcal{V}m$ by taking γ=x=$\mathbf{grab}_s$ and z=**id**. The second "canonical" transformation (see [20] page 27) is $\mathcal{V}m$' with γ=z=$\mathbf{grab}_{sL}$ and x=**id** (i.e. the reduction rule of $\mathbf{grab}_{sL}$ is recursive). By making all the $\mathbf{grab}_s$ explicit in the code, $\mathcal{V}m$ permits more simplifications than the alternative. For example,

$$\mathcal{V}m\ [\![(\lambda x.x\ x)\ (\lambda y.E)]\!] = \mathbf{push}_s\ (\lambda_s y.\mathcal{V}m\ [\![E]\!])\ \circ (\lambda_s x.\mathbf{push}_s\ x \circ x)$$

(one mark&$\mathbf{grab}_s$ has been simplified), whereas the other transformation $\mathcal{V}m$' yields $\mathbf{push}_s\ (\mathbf{grab}_{sL}(\lambda_s y.\mathcal{V}m'\ [\![E]\!]))\ \circ (\lambda_s x.\mathbf{push}_s\ \varepsilon \circ x \circ x)$ and $\mathbf{grab}_{sL}$ would be executed twice.

3.3 Comparison

We compare the efficiency of codes produced by transformations $\mathcal{V}a$ and $\mathcal{V}m$. Let us first emphasize that the point of this section is just to illustrate one advantage of an unified framework: making formal comparisons possible (such as finding complexity upper bounds or pathological examples). Of course, this style of comparisons does not take the place of benchmarks which remain needed in order to take into account complex implementation as-

pects (e.g. interactions with memory cache or the GC) or compare different reduction strategies (e.g. call-by-value *vs.* call-by-need).

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. Since a mark can be represented by one bit (in a bit stack parallel to the data stack for example), $\mathcal{V}_m$ 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 (assuming $\mathbf{push}_s$, $\mathbf{grab}_s$ and **app** have a constant time implementation). It is easy to show that code expansion is linear with respect to the size of the source expression. More precisely, for $\mathcal{V} = \mathcal{V}_a$ or $\mathcal{V}_m$, we have : If $\mathcal{S}ize\,(E) = n$ then $\mathcal{S}ize\,(\mathcal{V}[\![E]\!]) < 3n$.

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

$$\mathcal{C}ost\ (\mathcal{V}_a[\![E]\!]) = n_\lambda\ clos + n_v\ push + (n_v\text{-}1)\ app$$

and $$\mathcal{C}ost\ (\mathcal{V}_m[\![E]\!]) = (n_\lambda + n_v)\ grab + (n_v\text{-}1)\ push$$

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 4.3) $\mathcal{V}_m$ 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 $\mathbf{grab}_s$ 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

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

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.

3.4 Connection with CPS conversion

Transformations $\mathcal{V}_{\mathcal{X}}$ share the goal of compiling control with CPS transformations. Since CPS expressions have only one redex throughout the reduction, the closest transformations are the stackless ones. Indeed if we take the definitions (**DEF1**) (section 2.5) for the combinators, $\mathcal{V}a_f$ is Fischer's CPS transformation [12]. Using definitions (**DEF1**) we can rewrite $\mathcal{V}a_f$ as follows :

$$\mathcal{V}a_f[\![x]\!] = \mathbf{push}_s\, x = \lambda c.c\; x \qquad (\mathbf{DEF1})$$

$$\mathcal{V}a_f[\![\lambda x.E]\!] = \mathbf{push}_s\, (\lambda_s x.\mathcal{V}a_f[\![E]\!]) = \lambda c.c\; (\lambda c.\lambda x.\mathcal{V}a_f[\![E]\!]\; c) \qquad (\mathbf{DEF1})$$

$$\begin{aligned}\mathcal{V}a_f[\![E_1\, E_2]\!] &= \mathcal{V}a_f\,[\![E_1]\!]\; \mathrm{o}\; (\lambda_s m_1.\mathcal{V}a_f[\![E_2]\!]\; \mathrm{o}\; m_1)\\ &= \lambda c.\mathcal{V}a_f\,[\![E_1]\!]\,(\lambda m_1.\mathcal{V}a_f\,[\![E_2]\!]\,(m_1\; c)) \qquad (\mathbf{DEF1})\\ &= \lambda c.\mathcal{V}a_f\,[\![E_1]\!]\,(\lambda m_1.\mathcal{V}a_f\,[\![E_2]\!]\,(\lambda m_2.m_1\; c\; m_2)) \qquad (\eta)\end{aligned}$$

which is exactly Fischer's CPS.

As far as types are concerned we saw that if $E : \sigma$ then $\mathcal{V}a_f[\![E]\!] : \mathsf{R}\overline{\sigma}$ with $\overline{\sigma \to \tau} = \overline{\sigma} \to_s \mathsf{R}\overline{\tau}$ and $\alpha = \overline{\alpha}$. We recognize CPS types by giving to R and $\to_s$ the meanings:

$$\mathsf{R}\sigma = (\sigma \to \mathit{Ans}) \to \mathit{Ans} \qquad \text{and} \qquad \sigma \to_s \mathsf{R}\tau = (\tau \to \mathit{Ans}) \to \sigma \to \mathit{Ans}$$

Ans being the distinguished type of answers. Note that if n-ary functions are allowed we should add the rule $\sigma \to_s (\tau \to \mathit{Ans}) \to \upsilon = (\tau \to \mathit{Ans}) \to \sigma \to \upsilon$

As for CPS-expressions, it is also possible to design an inverse transformation [9]. Actually, the transformation $[\![]\!]^{-1}$ presented in section 2.5 (Figure 2) can be seen as a generic direct style transformation. It is easy to show that

Property 11 $\forall E \in \Lambda,\ [\![\mathcal{C}\ [\![E]\!]\ ]\!]^{-1} = E$ *(for* $\mathcal{C} = \mathcal{V}a, \mathcal{V}a_L, \mathcal{V}a_s, \mathcal{V}a_f, \mathcal{N}a, \mathcal{N}m$*)*

Proof. Structural induction. For example, the proof for $\mathcal{V}a$ is

- $E \equiv x$ $\qquad [\![\mathcal{V}a\ [\![E]\!]\ ]\!]^{-1} = [\![\mathbf{push}_s\, x]\!]^{-1} = [\![x]\!]^{-1} = x = E$
- $E \equiv \lambda x.F$ $\qquad [\![\mathcal{V}a\ [\![E]\!]\ ]\!]^{-1} = [\![\mathbf{push}_s\, (\lambda_s x.\mathcal{V}a\ [\![F]\!]\,)]\!]^{-1} = [\![\lambda_s x.\mathcal{V}a\ [\![F]\!]\ ]\!]^{-1}$

  $= \lambda x.\,[\![\mathcal{V}a\ [\![F]\!]\ ]\!]^{-1}$

  by induction hypothesis $\qquad = \lambda x.F = E$
- $E \equiv E_1\, E_2$ $\qquad [\![\mathcal{V}a\ [\![E]\!]\ ]\!]^{-1} = [\![\mathcal{V}a\ [\![E_2]\!]\ \mathrm{o}\ (\mathcal{V}a\ [\![E_1]\!]\ \mathrm{o}\ \lambda_s x.x)]\!]^{-1}$

  $= (\,[\![\lambda_s x.\,x]\!]^{-1}\ [\![\mathcal{V}a\ [\![E_1]\!]\ ]\!]^{-1})\ [\![\mathcal{V}a\ [\![E_2]\!]\ ]\!]^{-1}$

  by induction hypothesis $\qquad = (\lambda x.x)\, E_1\, E_2 \qquad = E$ ❐

4 Compilation of the β-Reduction

This compilation step implements the substitution using transformations from Λ_s to Λ_e. These transformations are akin to abstraction algorithms and consist in replacing variables 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) and uses only a fixed number of variables (in order to define combinators).

4.1 A generic abstraction

The denotational-like transformation $\mathcal{Ag}$ is a generic abstraction which will be specialized to model several choices in the following subsections. The transformation (Figure 7) is done relatively to a compile-time environment ρ (initially empty for a closed expression). The integer i in x_i denotes the rank of the variable in the environment.

$$\mathcal{Ag} : \Lambda_s \to env \to \Lambda_e$$

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

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

$$\mathcal{Ag}\ [\![\lambda_s x.E]\!]\ \rho = \mathbf{mkbind} \circ \mathcal{Ag}\ [\![E]\!]\ (\rho,x)$$

$$\mathcal{Ag}\ [\![x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{access}_i \circ \mathbf{appclos}$$

Figure 7 Generic Abstraction ($\mathcal{Ag}$)

$\mathcal{Ag}$ needs six new combinators to express saving and restoring environments ($\mathbf{dupl}_e$, $\mathbf{swap}_{se}$), closure building and opening (**mkclos**, **appclos**), access to values ($\mathbf{access}_i$) and adding a binding (**mkbind**). The first combinator pair is defined in Λ_e by:

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

The closure combinators (**mkclos**, **appclos**) can have different definitions in Λ_e as long as they verify the property:

$$(\mathbf{push}_e\ E \circ \mathbf{push}_s\ X \circ \mathbf{mkclos}) \circ \mathbf{appclos} \overset{+}{\rightarrow} \mathbf{push}_e\ E \circ X$$

For example, two possible definitions are

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

or $$\mathbf{mkclos} = \lambda_s x.\lambda_e e.\mathbf{push}_s(\mathbf{push}_e\ e \circ x) \qquad \mathbf{appclos} = \mathbf{app} = \lambda_s x.x$$

The first option uses pairs and is, in a way, more concrete than the other one. The second option abstracts from representation considerations. It simplifies the expression of correctness properties and will be used in the rest of the paper.

In the same way, the environment combinators (**mkbind**, **access**$_i$) can have several instantiations in Λ_e. Different definitions will be detailed in the next subsections, we only state here their common property:

$$(\mathbf{push}_s\ X_0 \circ \ldots \circ \mathbf{push}_s\ X_i \circ \mathbf{push}_e\ E \circ \mathbf{mkbind}^{i+1}) \circ \mathbf{access}_i \overset{+}{\Rightarrow} \mathbf{push}_s\ X_i$$

The transformation $\mathcal{A}_g$ can be optimized by adding the rules

$\mathcal{A}_g\ [\![E \circ \mathbf{app}]\!]\ \rho = \mathcal{A}_g\ [\![E]\!]\ \rho \circ \mathbf{appclos}$

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

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

$\mathcal{A}_g\ [\![\mathbf{push}_s\ x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{access}_i$

The abstraction of a naive definition of $\mathbf{grab}_s$ would result in an inefficient combinator. We introduce a new combinator $\mathbf{grab}_e$ and we add the following rule to $\mathcal{A}_g$:

$\mathcal{A}_g\ [\![\mathbf{grab}_s\ E]\!]\ \rho = \mathbf{grab}_e\ (\mathcal{A}_g\ [\![E]\!]\ \rho)$

with $\mathbf{push}_s\ \varepsilon \circ \mathbf{push}_e\ e \circ \mathbf{grab}_e\ F \Rightarrow \mathbf{push}_e\ e \circ \mathbf{push}_s\ F \circ \mathbf{mkclos}$

and $E : R\sigma$ $\quad E \circ \mathbf{push}_e\ e \circ \mathbf{grab}_e\ F \Rightarrow E \circ \mathbf{push}_e\ e \circ F$

The variables are bound to the closures stored in the environment. With the previous rules, $\mathcal{A}_g\ [\![\mathbf{grab}_s\ x]\!]$ builds yet another closure. This boxing can be avoided with a new combinator $\mathbf{grab}_e$**'** and the rule :

$\mathcal{A}_g\ [\![\mathbf{grab}_s\ x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{grab}_e\text{'}(\mathbf{access}_i)$

with $\mathbf{push}_s\ \varepsilon \circ \mathbf{push}_e\ e \circ \mathbf{grab}_e\text{'}(\mathbf{access}_i) \Rightarrow \mathbf{push}_e\ e \circ \mathbf{access}_i$

and $E : R\sigma$ $\quad E \circ \mathbf{push}_e\ e \circ \mathbf{grab}_e\text{'}(\mathbf{access}_i) \Rightarrow E \circ \mathbf{push}_e\ e \circ \mathbf{access}_i \circ \mathbf{appclos}$

A mark is a constant and no closure is necessary in this case, so :

$\mathcal{A}_g\ [\![\mathbf{push}_s\ \varepsilon \circ E]\!]\ \rho = \mathbf{push}_s\ \varepsilon \circ \mathbf{swap}_{se} \circ \mathcal{A}_g\ [\![E]\!]\ \rho$

4.2 Shared environments

A first choice is to instantiate $\mathcal{A}_g$ with linked environments. This specialization, noted $\mathcal{A}_s$, is widely used among the functional abstract machines [7][19][20]. The structure of the environment is a tree of closures and 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).

Specializing $\mathcal{A}_g$ into $\mathcal{A}_s$ amounts to define the environment combinators as follows

$$\textbf{mkbind} = \lambda_e e.\lambda_s x.\textbf{push}_e(e,x) \qquad \textbf{access}_i = \textbf{fst}^i \circ \textbf{snd}$$

$$\textit{with} \quad \textbf{fst} = \lambda_e(e,x).\textbf{push}_e\ e \qquad \textbf{snd} = \lambda_e(e,x).\textbf{push}_s\ x$$

Figure 8 Combinators Instantiation for Abstraction with Shared Environments ($\mathcal{A}_s$)

Example. $\mathcal{A}_s\ [\![\lambda_s x_1.\lambda_s x_0.\textbf{push}_s\ E \circ x_1]\!]\ \rho = \textbf{mkbind} \circ \textbf{mkbind} \circ \textbf{dupl}_e \circ$

$$\textbf{push}_s\ (\mathcal{A}_s\ [\![E]\!]\ ((\rho,x_1),x_0)) \circ \textbf{mkclos} \circ \textbf{swap}_{se} \circ \textbf{access}_1 \circ \textbf{appclos}$$

Two bindings are added (**mkbind** ∘ **mkbind**) to the current environment and the x_1 access is coded by $\textbf{access}_1 = \textbf{fst} \circ \textbf{snd}$. ❐

The correctness of $\mathcal{A}_s$ is stated by Property 12.

Property 12 $\forall E \in \Lambda_s$ *closed*, $\textbf{push}_e\ () \circ \mathcal{A}_s\ [\![E]\!]\ () = E$

Example. Let us come back to the example of the previous section $E \equiv (\lambda x.x)((\lambda y.y)(\lambda z.z))$ to illustrate reduction of Λ_e-expressions. After simplifications $\mathcal{V}a\ [\![E]\!] \equiv \textbf{push}_s(\lambda_s z.\textbf{push}_s\ z) \circ (\lambda_s y.\textbf{push}_s\ y) \circ (\lambda_s x.\textbf{push}_s\ x)$ and

$\textbf{push}_e\ () \circ \mathcal{A}_s\ [\![\textbf{push}_s(\lambda_s z.\textbf{push}_s\ z) \circ (\lambda_s y.\textbf{push}_s\ y) \circ (\lambda_s x.\textbf{push}_s\ x)]\!]\ ()$

$\equiv \textbf{push}_e\ () \circ \textbf{dupl}_e \circ \textbf{dupl}_e \circ \textbf{push}_s(\textbf{mkbind} \circ \textbf{access}_0) \circ \textbf{mkclos} \circ \textbf{swap}_{se}$

$\qquad \circ \textbf{mkbind} \circ \textbf{access}_0 \circ \textbf{swap}_{se} \circ \textbf{mkbind} \circ \textbf{access}_0$

$\rightarrow \textbf{push}_e\ () \circ \textbf{push}_e\ () \circ \textbf{dupl}_e \circ \textbf{push}_s(\textbf{mkbind} \circ \textbf{access}_0) \circ \textbf{mkclos} \circ \textbf{swap}_{se}$

$\qquad \circ \textbf{mkbind} \circ \textbf{access}_0 \circ \textbf{swap}_{se} \circ \textbf{mkbind} \circ \textbf{access}_0$

$\overset{+}{\rightarrow} \textbf{push}_s(\textbf{push}_e\ () \circ \textbf{mkbind} \circ \textbf{access}_0) = \textbf{push}_e\ () \circ \mathcal{A}_s\ [\![(\lambda z.z)]\!]\ ()$ ❐

As already noted, 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. The list-like environment can become a vector locally and variable accesses have to be modified consequently. This allows us to formalize the optimization described in [8], and based on a closure analysis result.

4.3 Copied environments

Another choice is to provide a constant access time [1][13]. In this case, the structure of the environment must be a vector of closures. A code copying the environment (a O(*length* ρ) operation) has to be inserted in $\mathcal{A}_g$ in order to avoid links. This scheme is less prone to space leaks since it allows to suppress useless variables during copies.

The macro-combinator **Copy** ρ produces code performing this copy according to ρ's structure.

$$\textbf{Copy}\ (\ldots((),x_n),\ldots,x_0) = (\textbf{dupl}_e \circ \textbf{access}_n \circ \textbf{swap}_{se}) \circ \ldots$$
$$\circ\ (\textbf{dupl}_e \circ \textbf{access}_1 \circ \textbf{swap}_{se}) \circ \textbf{access}_0 \circ \textbf{push}_e\ () \circ \textbf{mkbind}^{n+1}$$

If we still see the structure of the environment as a tree of closures, the effect of **Copy** ρ is to prevent sharing to occur. Environments can thus be represented by vectors (Figure 9): **mkbind** now adds a binding in a vector and $\textbf{access}_i$ becomes a constant time operation.

$$\textbf{mkbind} = \lambda_e e.\lambda_s x.\textbf{push}_e(e[\mathit{next}]{:=}x) \qquad \textbf{access}_i = \lambda_e e.\textbf{push}_s\ (e[i])$$

where $e[\mathit{next}]{:=}x$ *adds the value x in the first empty cell of the vector e*

Figure 9 Combinators Instantiation for Abstraction with Copied Environments ($\mathcal{A}c_1$)

The index *next* designates the first free cell in the vector. It can be statically computed as the rank of the variable associated with the **mkbind** occurrence in the static environment ρ. For example, in

$$\mathcal{A}c\ [\![\lambda_s y.E]\!]\ (((),x_2),x_1),x_0) = \textbf{mkbind} \circ \mathcal{A}c\ [\![E]\!]\ ((((),x_2),x_1),x_0),y)$$

we have $\mathit{next} = \mathit{rank}\ y\ ((((),x_2),x_1),x_0),y) = 4$, and y is stored in the fourth cell of the environment. The maximum size of each vector could be statically calculated too. To simplify the presentation, we leave these administrative tasks implicit.

There are several abstractions according to the time of the copies. We present only the rules differing from $\mathcal{A}g$. A first solution (Figure 10) is to copy the environment just before adding a new binding (as in [11]). From the first step we know that n-ary functions ($\lambda_s x_1 \ldots \lambda_s x_n.E$) are fully 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.

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

Figure 10 Copy at Function Entry ($\mathcal{A}c1$ Abstraction)

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

Example. $\mathcal{A}c1\ [\![\lambda_s x_1.\lambda_s x_0.\textbf{push}_s\ E_1 \circ x_1]\!]\ \rho = \textbf{Copy}\ \overline{\rho} \circ \textbf{mkbind}^2 \circ \textbf{dupl}_e \circ$

$$\textbf{push}_s\ (\mathcal{A}c1\ [\![E]\!]\ ((\overline{\rho},x_1),x_0))) \circ \textbf{mkclos} \circ \textbf{swap}_{se} \circ \textbf{access}_1 \circ \textbf{appclos}$$

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

A second solution (Figure 11) is to copy the environment when building and opening closures (as in [13]). 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).

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

Figure 11 Copy at Closure Building and Opening ($\mathcal{A}c2$ Abstraction)

A refinement of this last option 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 specialized environments ($\overline{\rho}_G$++$\overline{\rho}_L$) is copied. The code for variables has now to specify which environment is accessed. Although the transformation scheme remains similar, every rule must be redefined to take into account the two environments.

$$\mathcal{A}c3\ [\![E_1 \circ E_2]\!]\ \rho_L\ \rho_G = \mathbf{dupl2}_e \circ \mathcal{A}c3\ [\![E_1]\!]\ \rho_L\ \rho_G \circ \mathbf{swap2}_{se} \circ \mathcal{A}c3\ [\![E_2]\!]\ \rho_L\ \rho_G$$

$$\mathcal{A}c3\ [\![\mathbf{push}_s\ E]\!]\ \rho_L\ \rho_G = \mathbf{Copy2}\ (\overline{\rho}_G{+}{+}\overline{\rho}_L) \circ \mathbf{push}_s(\mathbf{push}_e\ () \circ \mathcal{A}c3\ [\![E]\!]\ ()\ \overline{\rho}_L{+}{+}\overline{\rho}_G) \circ \mathbf{mkclos}$$

$$\mathcal{A}c3\ [\![\lambda_s x.E]\!]\ \rho_L\ \rho_G = \mathbf{mkbind2} \circ \mathcal{A}c3\ [\![E]\!]\ \rho_L\ (\rho_G,x)$$

$$\mathcal{A}c3\ [\![x_i]\!]\ (\ldots((\rho_L,x_i),x_{i-1})\ldots,x_0)\ \rho_G = \mathbf{getlocal} \circ \mathbf{access}_i \circ \mathbf{appclos}$$

$$\mathcal{A}c3\ [\![x_i]\!]\ \rho_L\ (\ldots((\rho_G,x_i),x_{i-1})\ldots,x_0) = \mathbf{getglobal} \circ \mathbf{access}_i \circ \mathbf{appclos}$$

with $\mathbf{dupl2}_e = \lambda_e e_l.\lambda_e e_g.\mathbf{push}_e\ e_g \circ \mathbf{push}_e\ e_l \circ \mathbf{push}_e\ e_g \circ \mathbf{push}_e\ e_l$

$\mathbf{swap2}_{se} = \lambda_s x.\lambda_e e_l.\lambda_e e_g.\mathbf{push}_s\ x \circ \mathbf{push}_e\ e_g \circ \mathbf{push}_e\ e_l$

$\mathbf{mkbind2} = \lambda_e e_l.\lambda_e e_g.\lambda_s x.\mathbf{push}_e\ e_g \circ \mathbf{push}_s\ x \circ \mathbf{push}_e\ e_l \circ \mathbf{mkbind}$

$\mathbf{getlocal} = \lambda_e e_l.\lambda_e e_g.\mathbf{push}_e\ el$ $\qquad$ $\mathbf{getglobal} = \lambda_e e_l.\lambda_e e_g.\mathbf{push}_e\ e_g$

Figure 12 Abstraction with Local Environments ($\mathcal{A}c3$ Abstraction)

Local environments are not compatible with $\mathcal{V}m$: $\mathcal{A}c3\ [\![\mathbf{grab}_s\ E]\!]$ 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.

4.4 Comparison

Assuming each basic combinator can be implemented in constant time, the size of the abstracted expressions gives an approximation of the overhead entailed by the encoding of the β-reduction. It is easy to show that $\mathcal{A}_s$ entails a code expansion which is quadratic with respect to the size of the source expression. More precisely

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

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, $\lambda x_1 \ldots \lambda x_{n\lambda}.x_1 \ldots x_{n\lambda}$. 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

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

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 [30]. Balancing expressions reduces this upper bound to O(*n*log*n*) [17]. 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 [26] section 4.2.6).

4.5 A family of abstractions

Starting from different properties (such as $\mathcal{A}_{g_s}\,[\![E]\!]\,\rho = \mathbf{swap}_n \circ \mathcal{A}_g\,[\![E]\!]\,\rho$ for example) a collection of abstractions can be derived form $\mathcal{A}_g$. These variations introduce different environment manipulation schemes avoiding stacks elements reordering (swap-less), environment duplication (dupl-less), environment building (mkbind-less) or closure building (mkclos-less). We present here six of them and specify which ones are suited for shared or copied environments. Further study would be necessary to define rules dealing with $\mathbf{grab}_s$, define rules in presence of copied environments, and compare or mix such variations. The transformations introduce indexed combinators (which are generalizations of previously used combinators) and use the notion of arity :

Definition 13 *An expression E of type* $\sigma_1 \to \ldots \to \sigma_n \to R\sigma$ *is said to have arity n.*

4.5.1 A swap-less abstraction

This variation suppresses the occurrences of $\mathbf{swap}_{se}$ in $\mathcal{AG}$. Once pushed, the references to environments stay at a fixed distance from the bottom of the stack until they are popped (the references are no more **swap**ped). It can be used with shared environments as well as with copied ones. $\mathcal{AG}_s$ is derived from the equation:

$\mathcal{AG}_s [\![E]\!] \rho = \mathbf{swap}_n \circ \mathcal{AG} [\![E]\!] \rho$ *hence* $\mathcal{AG} [\![E]\!] \rho = \mathbf{store}_n \circ \mathcal{AG}_s [\![E]\!] \rho$ (n arity of E)

$\mathcal{AG}_s : \Lambda_s \to env \to \Lambda_e$

$\mathcal{AG}_s [\![E_1 \circ E_2]\!] \rho = \mathbf{dupl}_n \circ \mathcal{AG}_s [\![E_1]\!] \rho \circ \mathcal{AG}_s [\![E_2]\!] \rho$ (n arity of $E_1 \circ E_2$)

$\mathcal{AG}_s [\![\mathbf{push}_s\ E]\!] \rho = \mathbf{push}_s\ (\mathbf{repl}_n \circ \mathcal{AG}_s [\![E]\!] \rho) \circ \mathbf{mkclos}$ (n arity of E)

$\mathcal{AG}_s [\![\lambda_s x.E]\!] \rho = \mathbf{mkbind}_n \circ \mathcal{AG}_s [\![E]\!] (\rho,x)$ (n arity of E)

$\mathcal{AG}_s [\![x_i]\!] (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{take}_{n,i} \circ \mathbf{appclos}$ (n arity of x_i)

with $\mathbf{swap}_n = \lambda_s x_1 \ldots \lambda_s x_n.\lambda_e e.\mathbf{push}_s\ x_n \circ \ldots \circ \mathbf{push}_s\ x_1 \circ \mathbf{push}_e\ e$

$\mathbf{store}_n = \lambda_e e.\lambda_s x_1 \ldots \lambda_s x_n.\mathbf{push}_e\ e \circ \mathbf{push}_s\ x_n \circ \ldots \circ \mathbf{push}_s\ x_1$

$\mathbf{dupl}_n = \lambda_s x_1 \ldots \lambda_s x_n.\lambda_e e.\mathbf{push}_e\ e \circ \mathbf{push}_s\ x_n \circ \ldots \circ \mathbf{push}_s\ x_1 \circ \mathbf{push}_e\ e$

$\mathbf{repl}_n = \lambda_e e.\lambda_s x_1 \ldots \lambda_s x_n.\lambda_e e'.\mathbf{push}_e\ e \circ \mathbf{push}_s\ x_n \circ \ldots \circ \mathbf{push}_s\ x_1$

$\mathbf{mkbind}_n = \lambda_s y.\lambda_s x_1 \ldots \lambda_s x_n.\lambda_e e.\mathbf{push}_s\ y \circ \mathbf{push}_e\ e \circ \mathbf{mkbind} \circ \mathbf{push}_s\ x_n \circ \ldots \circ \mathbf{push}_s\ x_1$

$\mathbf{take}_{n,i} = \lambda_s x_1 \ldots \lambda_s x_n.\lambda_e e.\mathbf{push}_e\ e \circ \mathbf{push}_s\ x_n \circ \ldots \circ \mathbf{push}_s\ x_1 \circ \mathbf{push}_e\ e \circ \mathbf{access}_i$

Figure 13 "Swap-less" Abstraction ($\mathcal{AG}_s$)

The indexed combinators enjoy many properties allowing code transformations. Let us present only two of them:

$\mathbf{store}_n \circ \mathbf{swap}_n \circ E = E$

$\mathbf{swap}_n \circ (\mathbf{dupl}_0 \circ E_1 \circ \mathbf{swap}_1 \circ E_2) = \mathbf{dupl}_n \circ \mathbf{swap}_0 \circ E_1 \circ \mathbf{swap}_{n+1} \circ E_2$

So, the dual of $\mathcal{AG}_s [\![E]\!] \rho = \mathbf{swap}_n \circ \mathcal{AG} [\![E]\!] \rho$ is $\mathcal{AG} [\![E]\!] \rho = \mathbf{store}_n \circ \mathcal{AG}_s [\![E]\!] \rho$ (first property). The $\mathcal{AG}_s$ abstraction and the others following variations, are correct by construction. To illustrate how $\mathcal{AG}_s$ is derived from $\mathcal{AG}$, let us take the rule for compositions:

$\mathcal{AG}_s [\![E_1 \circ E_2]\!] \rho = \mathbf{swap}_n \circ \mathcal{AG} [\![E_1 \circ E_2]\!] \rho$ ($\mathcal{AG}_s$ property)

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

$= \mathbf{swap}_n \circ (\mathbf{dupl}_0 \circ \mathcal{AG} [\![E_1]\!] \rho \circ \mathbf{swap}_1 \circ \mathcal{AG} [\![E_2]\!] \rho)$ ($\mathbf{swap}_{n'}$ $\mathbf{dupl}_n$ definitions)

$= \mathbf{dupl}_n \circ \mathbf{swap}_0 \circ \mathcal{Ag}\,[\![E_1]\!]\rho \circ \mathbf{swap}_{n+1} \circ \mathcal{Ag}\,[\![E_2]\!]\rho$ (**swap**$_n$, **dupl**$_n$ properties)

$= \mathbf{dupl}_n \circ \mathcal{Ag}_s\,[\![E_1]\!]\rho \circ \mathbf{swap}_{n+1} \circ \mathcal{Ag}\,[\![E_2]\!]\rho$ (folding (E_1 is 0-ary))

$= \mathbf{dupl}_n \circ \mathcal{Ag}_s\,[\![E_1]\!]\rho \circ \mathcal{Ag}_s\,[\![E_2]\!]\rho$ (folding (E_2 is n+1-ary))

4.5.2 A dupl-less abstraction

This variation suppresses the occurrences of $\mathbf{dupl}_e$ in $\mathcal{Ag}\,[\![E_1 \circ E_2]\!]$. Duplications are postponed until really needed (in closure building or opening). This can change the order of magnitude of the depth of the environment stack needed to reduce an expression. For example, if $E \equiv (\ldots(x_n \circ x_{n-1})\ldots \circ x_2) \circ x_1$, the depth of the environment stack will be n for $\mathcal{Ag}\,[\![E]\!]\rho$ and 1 for $\mathcal{Ag}_d\,[\![E]\!]\rho$. $\mathcal{Ag}_d$ is derived from the equation:

$\mathcal{Ag}_d\,[\![E]\!]\rho = \mathbf{copy}_n \circ \mathcal{Ag}\,[\![E]\!]\rho$ (dually: $\mathcal{Ag}\,[\![E]\!]\rho = \mathcal{Ag}_d\,[\![E]\!]\rho \circ \mathbf{pop}_1$) ($n$ arity of E)

Note that $\mathbf{copy}_n$ is a generalized $\mathbf{dupl}_e$ ($\mathbf{copy}_0 = \mathbf{dupl}_e$).

$\mathcal{Ag}_d : \Lambda_s \to env \to \Lambda_e$

$\mathcal{Ag}_d\,[\![E_1 \circ E_2]\!]\rho = \mathcal{Ag}_d\,[\![E_1]\!]\rho \circ \mathbf{swap}_{se} \circ \mathcal{Ag}_d\,[\![E_2]\!]\rho$ (n arity of $E_1 \circ E_2$)

$\mathcal{Ag}_d\,[\![\mathbf{push}_s\,E]\!]\rho = \mathbf{push}_s\,(\mathcal{Ag}_d\,[\![E]\!]\rho \circ \mathbf{pop}_1) \circ \mathbf{mkclos}_d$ (n arity of E)

$\mathcal{Ag}_d\,[\![\lambda_s x.E]\!]\rho = \mathbf{mkbind} \circ \mathcal{Ag}_d\,[\![E]\!](\rho,x) \circ \mathbf{brkbind}_1$ (n arity of E)

$\mathcal{Ag}_d\,[\![x_i]\!](\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{copy}_n \circ \mathbf{access}_i \circ \mathbf{appclos}$ (n arity of x_i)

with $\mathbf{copy}_n = \lambda_e e.\lambda_s x_1 \ldots \lambda_s x_n.\mathbf{push}_e\,e \circ \mathbf{push}_s\,x_n \circ \ldots \circ \mathbf{push}_s\,x_1 \circ \mathbf{push}_e\,e$

$\mathbf{pop}_n = \lambda_s x_1 \ldots \lambda_s x_n.\lambda_e e.\mathbf{push}_s\,x_n \circ \ldots \circ \mathbf{push}_s\,x_1$

$\mathbf{mkclos}_d = \lambda_e e.\lambda_s x.\mathbf{push}_e\,e \circ \mathbf{push}_e\,e \circ \mathbf{push}_s\,x \circ \mathbf{mkclos}$

$\mathbf{brkbind}_n = \lambda_s x_1 \ldots \lambda_s x_n.\lambda_e e.\mathbf{push}_e\,e \circ \mathbf{brkbind} \circ \mathbf{push}_s\,x_n \circ \ldots \circ \mathbf{push}_s\,x_1$

with the property $(\mathbf{push}_s\,X \circ \mathbf{push}_e\,E \circ \mathbf{mkbind}) \circ \mathbf{brkbind} = \mathbf{push}_e\,E$

Figure 14 "Dupl-less" Abstraction ($\mathcal{Ag}_d$)

This abstraction algorithm exploits the sequencing encoded in compositions. Instead of saving and restoring the environment (as in $\mathcal{Ag}\,[\![E_1 \circ E_2]\!]$), it is passed to E_1 which may add new bindings but has to remove them (as expressed by the nesting of **mkbind** and $\mathbf{brkbind}_1$ in the $\mathcal{Ag}_d\,[\![\lambda_s x.E]\!]\rho$ rule) before passing the environment to E_2. This transformation can be used with shared or copied environments.

$\mathcal{Ag}_d$ may be inefficient since the environment must be stored behind the context ($\mathbf{copy}_n$) before each closure evaluation. However with most compilation schemes, closures arity is

never greater than 1 (they are only unevaluated basic values or unary functions). In this case, $\mathbf{copy}_n$ is a basic (constant time) operation.

4.5.3 A mkbind-less abstraction

This variation postpones the occurrences of **mkbind** in $\mathcal{Ag}$. So, the environments are unfolded in the data stack. This avoids an indirection and provides a direct access to values. In the following, m denotes the length of the environment ρ. $\mathcal{Ag}_{\mathrm{b}}$ is derived from the equation:

$\mathcal{Ag}_{\mathrm{b}}\,[\![E]\!]\,\rho = \mathbf{push}_{\mathrm{e}}\ \mathbf{init} \circ \mathbf{mkbind}^m \circ \mathcal{Ag}\,[\![E]\!]\,\rho$ (dually $\mathcal{Ag}\,[\![E]\!]\,\rho = \mathbf{pscl}_m \circ \mathcal{Ag}_{\mathrm{b}}\,[\![E]\!]\,\rho$)

$\mathcal{Ag}_{\mathrm{b}} : \Lambda_{\mathrm{s}} \to env \to \Lambda_{\mathrm{e}}$

$\mathcal{Ag}_{\mathrm{b}}\,[\![E_1 \circ E_2]\!]\,\rho = \mathbf{dupls}_{0,m} \circ \mathcal{Ag}_{\mathrm{b}}\,[\![E_1]\!]\,\rho \circ \mathbf{swaps}_{1,m} \circ \mathcal{Ag}_{\mathrm{b}}\,[\![E_2]\!]\,\rho$

$\mathcal{Ag}_{\mathrm{b}}\,[\![\mathbf{push}_{\mathrm{s}}\ E]\!]\,\rho = \mathbf{push}_{\mathrm{s}}\ (\mathbf{pscl}_m \circ \mathcal{Ag}_{\mathrm{b}}\,[\![E]\!]\,\rho) \circ \mathbf{mkclos}_m$

$\mathcal{Ag}_{\mathrm{b}}\,[\![\lambda_{\mathrm{s}}x.E]\!]\,\rho = \mathcal{Ag}_{\mathrm{b}}\,[\![E]\!]\,(\rho,x)$

$\mathcal{Ag}_{\mathrm{b}}\,[\![x_i]\!]\,(\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{flsh}_{i,m} \circ \mathbf{appclos}$

with $\mathbf{dupls}_{n,m} = \lambda_{\mathrm{s}}x_1\ldots\lambda_{\mathrm{s}}x_n.\lambda_{\mathrm{s}}y_0\ldots\lambda_{\mathrm{s}}y_m.\mathbf{push}_{\mathrm{s}}\ y_m \circ \ldots \circ \mathbf{push}_{\mathrm{s}}\ y_0 \circ$
$\mathbf{push}_{\mathrm{s}}\ x_n \circ \ldots \circ \mathbf{push}_{\mathrm{s}}\ x_1 \circ \mathbf{push}_{\mathrm{s}}\ y_m \circ \ldots \circ \mathbf{push}_{\mathrm{s}}\ y_0$

$\mathbf{swaps}_{n,m} = \lambda_{\mathrm{s}}x_1\ldots\lambda_{\mathrm{s}}x_n.\lambda_{\mathrm{s}}y_0\ldots\lambda_{\mathrm{s}}y_m.\mathbf{push}_{\mathrm{s}}\ x_n \circ \ldots \circ \mathbf{push}_{\mathrm{s}}\ x_1 \circ$
$\mathbf{push}_{\mathrm{s}}\ y_m \circ \ldots \circ \mathbf{push}_{\mathrm{s}}\ y_0$

$\mathbf{mkclos}_m = \lambda_{\mathrm{s}}x.\mathbf{push}_{\mathrm{e}}\ () \circ \mathbf{mkbind}^m \circ \mathbf{push}_{\mathrm{s}}\ x \circ \mathbf{mkclos}$

$\mathbf{pscl}_m = \lambda_{\mathrm{e}}e.(\mathbf{push}_{\mathrm{e}}\ e \circ \mathbf{access}_0) \circ \ldots \circ (\mathbf{push}_{\mathrm{e}}\ e \circ \mathbf{access}_m)$

$\mathbf{flsh}_{i,m} = \lambda_{\mathrm{s}}y_0\ldots\lambda_{\mathrm{s}}y_i\ldots\lambda_{\mathrm{s}}y_m.\mathbf{push}_{\mathrm{s}}\ y_i$

Figure 15 "Mkbind-less" Abstraction ($\mathcal{Ag}_{\mathrm{b}}$)

The bindless scheme and its variations (4.5.5 for example), are suited to copied environment schemes. Indeed, $\mathbf{dupls}_{0,m}$, closure building ($\mathbf{mkclos}_m$) and opening ($\mathbf{pscl}_m$) necessarily copy the environment.

With the previous definitions, the component e is only used temporary to fold and unfold the closure environments. It can be completely suppressed with the two alternative definitions:

$\mathbf{mkclos}_m = \lambda_{\mathrm{s}}x.\lambda_{\mathrm{s}}y_0\ldots\lambda_{\mathrm{s}}y_m.\mathbf{push}_{\mathrm{s}}\ (\mathbf{push}_{\mathrm{s}}\ y_m \circ \ldots \circ \mathbf{push}_{\mathrm{s}}\ y_0 \circ x)$

$\mathbf{pscl}_m = \mathbf{Id}_{\mathrm{s}}$ *with* $\mathbf{Id}_{\mathrm{s}} \circ E = \mathbf{Id}_{\mathrm{s}}$

This transformation unfolds the environment in the data stack. With a more general property : $\mathcal{AG}_{\mathrm{b}}[\![E]\!]\rho = \mathbf{move\text{-}etos}_m \circ \mathbf{push}_{\mathrm{e}}() \circ \mathbf{mkbind}^m \circ \mathcal{AG}[\![E]\!]\rho$, the environments are unfolded in the environment stack, so that it becomes a closure stack [13] ($\mathbf{move\text{-}etos}_m = \lambda_{\mathrm{e}} y_0 \ldots \lambda_{\mathrm{e}} y_m.\mathbf{push}_{\mathrm{s}}\, y_m \circ \ldots \circ \mathbf{push}_{\mathrm{s}}\, y_0$)

The environments can also be unfolded in the data stack by merging the components *e* and *s*. This choice takes within the component instantiation step (see section 5.2).

4.5.4 A dupl-less, swap-less abstraction

This variation mixes the effects of the dupl-less and swap-less transformations. It can be used with shared or copied environments. It is derived from $\mathcal{AG}_{\mathrm{d}}$ using the equation:

$\mathcal{AG}_{\mathrm{ds}}[\![E]\!]\rho\, n = \mathbf{swap}_n \circ \mathcal{AG}_{\mathrm{d}}[\![E]\!]\rho = \mathbf{swap}_n \circ \mathbf{copy}_n \circ \mathcal{AG}[\![E]\!]\rho = \mathbf{dupl}_n \circ \mathcal{AG}[\![E]\!]\rho$

(dually $\mathcal{AG}[\![E]\!]\rho = \mathbf{store}_n \circ \mathcal{AG}_{\mathrm{ds}}[\![E]\!]\rho \circ \mathbf{pop}_1$) *with n is the arity of E*

Actually, the equation used for the derivation is the more general:

$\mathcal{AG}_{\mathrm{ds}}[\![E]\!]\rho\, k = \mathbf{dupl}_k \circ \mathcal{AG}[\![E]\!]\rho$ *with k the arity of E*

$\mathcal{AG}_{\mathrm{ds}} : \Lambda_{\mathrm{s}} \to env \to int \to \Lambda_{\mathrm{e}}$

$\mathcal{AG}_{\mathrm{ds}}[\![E_1 \circ E_2]\!]\rho\, k = \mathcal{AG}_{\mathrm{ds}}[\![E_1]\!]\rho\, k \circ \mathcal{AG}_{\mathrm{ds}}[\![E_2]\!]\rho\,(k+1)$

$\mathcal{AG}_{\mathrm{ds}}[\![\mathbf{push}_{\mathrm{s}}\, E]\!]\rho\, k = \mathbf{push}_{\mathrm{s}}(\mathbf{store}_n \circ \mathcal{AG}_{\mathrm{ds}}[\![E]\!]\rho\, n \circ \mathbf{pop}_1) \circ \mathbf{mkclos}_{\mathrm{d},k}$ (n arity of E)

$\mathcal{AG}_{\mathrm{ds}}[\![\lambda_{\mathrm{s}} x.E]\!]\rho\, k = \mathbf{mkbind}_k \circ \mathcal{AG}_{\mathrm{ds}}[\![E]\!](\rho,x)\,(k-1) \circ \mathbf{brkbind}_{k\text{-}n\text{-}1}$ (n arity of E)

$\mathcal{AG}_{\mathrm{ds}}[\![x_i]\!](\ldots((\rho,x_i),x_{i-1})\ldots,x_0)\, k = \mathbf{take}_{n,i} \circ \mathbf{appclos}$ (n arity of x_i)

with $\mathbf{mkclos}_{\mathrm{d},k} = \lambda_{\mathrm{s}} x \lambda_{\mathrm{s}} x_0 \ldots \lambda_{\mathrm{s}} x_k.\lambda_{\mathrm{e}} e.\mathbf{push}_{\mathrm{e}}\, e \circ \mathbf{push}_{\mathrm{s}}\, x_0 \circ \ldots \circ \mathbf{push}_{\mathrm{s}}\, x_m \circ \mathbf{push}_{\mathrm{s}}\, x \circ \mathbf{push}_{\mathrm{e}}\, e \circ \mathbf{mkclos}$

Figure 16 "Dupl-less, Swap-less" Abstraction ($\mathcal{AG}_{\mathrm{ds}}$)

As explained in 4.5.2, with most compilation schemes, closures arity is never greater than 1. In these cases, the combinator $\mathbf{store}_n$ in $\mathcal{AG}_{\mathrm{ds}}$, which inserts an element in a stack, is a constant time operation ($\mathbf{store}_1$).

In much the same way, a swap-less bind-less abstraction and a dupl-less mkbind-less abstraction can be defined.

4.5.5 A dupl-less, swap-less, mkbind-less abstraction

This variation mixes the effects of $\mathcal{AG}_{\mathrm{ds}}$ and $\mathcal{AG}_{\mathrm{b}}$. The environments are unfolded in the stack at a fixed place (from the bottom of the stack) where they grow and shrink according to the

bindings scope. As previously said, the mkbind-less abstractions are not well suited for shared environments. In the following, m denotes the length of the environment ρ. $\mathcal{Ag}_{\text{dsb}}$ is derived from the equation:

$\mathcal{Ag}_{\text{dsb}}\ [\![E]\!]\ \rho\ k = \mathbf{dupls}_{k,m} \circ \mathbf{push}_{\text{e}}\ () \circ \mathbf{mkbind}^m \circ \mathcal{Ag}\ [\![E]\!]\ \rho$ *with k the arity of E*

(dually $\mathcal{Ag}\ [\![E]\!]\ \rho = \mathbf{pscl}_m \circ \mathbf{stores}_{p,m} \circ \mathcal{Ag}_{\text{dsb}}\ [\![E]\!]\ \rho \circ \mathbf{flushs}_{1,m}$ *with p is the arity of E*)

$\mathcal{Ag}_{\text{dsb}} : \Lambda_{\text{s}} \rightarrow env \rightarrow int \rightarrow \Lambda_{\text{e}}$

$\mathcal{Ag}_{\text{dsb}}\ [\![E_1 \circ E_2]\!]\ \rho\ k = \mathcal{Ag}_{\text{dsb}}\ [\![E_1]\!]\ \rho\ k \circ \mathcal{Ag}_{\text{dsb}}\ [\![E_2]\!]\ \rho\ (k{+}1)$

$\mathcal{Ag}_{\text{dsb}}\ [\![\mathbf{push}_{\text{s}}\ E]\!]\ \rho\ k = \mathbf{push}_{\text{s}}(\mathbf{pscl}_m \circ \mathbf{stores}_{p,m} \circ \mathcal{Ag}_{\text{dsb}}\ [\![E]\!]\ \rho \circ \mathbf{flushs}_{1,m}) \circ \mathbf{mkclos}_{k,m}$

(p arity of E)

$\mathcal{Ag}_{\text{dsb}}\ [\![\lambda_{\text{s}}x.E]\!]\ \rho\ k = \mathbf{stores}_{k+m,1} \circ \mathcal{Ag}_{\text{dsb}}\ [\![E]\!]\ (\rho,x)\ (k-1) \circ \mathbf{flushs}_{k+1,1}$

$\mathcal{Ag}_{\text{dsb}}\ [\![x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0)\ k = \mathbf{stores}_{k,m} \circ \mathbf{dupls}_{k+m-i,1} \circ \mathbf{appclos}$ (n arity of x_i)

with $\mathbf{stores}_{n,m} = \lambda_{\text{s}}y_0\ldots\lambda_{\text{s}}y_{\text{m}}.\lambda_{\text{s}}x_1\ldots\lambda_{\text{s}}x_n.\mathbf{push}_{\text{s}}\ y_m \circ \ldots \circ \mathbf{push}_{\text{s}}\ y_0 \circ \mathbf{push}_{\text{s}}\ x_n \circ \ldots \circ \mathbf{push}_{\text{s}}\ x_1$

$\mathbf{flushs}_{n,m} = \lambda_{\text{s}}x_1\ldots\lambda_{\text{s}}x_n.\lambda_{\text{s}}y_0\ldots\lambda_{\text{s}}y_{\text{m}}.\mathbf{push}_{\text{s}}\ x_n \circ \ldots \circ \mathbf{push}_{\text{s}}\ x_1$

Figure 17 "Dupl-less, Swap-less, Bind-less" Abstraction ($\mathcal{Ag}_{\text{dsb}}$)

4.5.6 A mkclos-less abstraction

This variation suppresses the occurrences of **mkclos** in $\mathcal{Ag}$. Closures are not allocated anymore. Both the code and the environment references are manipulated on stacks until they are bound in an environment (as in TIM [11]). The macro combinator $\mathbf{rmvmkclos}_n$ transforms the environment structure from a list of closures to a list of unfolded closures (i.e. a list of alternating codes and environments) whereas $\mathbf{addmkclos}_n$ does the inverse operation. $\mathcal{Ag}_{\text{m}}$is derived from the equation:

$\mathcal{Ag}_{\text{m}}\ [\![E]\!]\ \rho = \mathbf{addmkclos}_n \circ \mathcal{Ag}\ [\![E]\!]\ \rho \circ \mathbf{brkclos}$ *with n = length* ρ

(dually $\mathcal{Ag}\ [\![E]\!]\ \rho = \mathbf{rmvmkclos}_n \circ \mathcal{Ag}_{\text{dsb}}\ [\![E]\!]\ \rho \circ \mathbf{mkclos}$)

$\mathcal{Ag}_{\text{m}} : \Lambda_{\text{s}} \rightarrow env \rightarrow \Lambda_{\text{e}}$

$\mathcal{Ag}_{\text{m}}\ [\![E_1 \circ E_2]\!]\ \rho = \mathbf{dupl}_e \circ \mathcal{Ag}_{\text{m}}\ [\![E_1]\!]\ \rho \circ \mathbf{swapclos} \circ \mathcal{Ag}_{\text{m}}\ [\![E_2]\!]\ \rho$

$\mathcal{Ag}_{\text{m}}\ [\![\mathbf{push}_{\text{s}}\ E]\!]\ \rho = \mathbf{push}_{\text{s}}\ (\mathcal{Ag}_{\text{m}}\ [\![E]\!]\ \rho)$

$\mathcal{Ag}_{\text{m}}\ [\![\lambda_{\text{s}}x.E]\!]\ \rho = \mathbf{mkbind} \circ \mathbf{mkbind}_e \circ \mathcal{Ag}_{\text{s}}\ [\![E]\!]\ (\rho,x)$

$\mathcal{Ag}_{\text{m}}\ [\![x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{dupl}_{\text{e}} \circ \mathbf{access}_{2i} \circ \mathbf{stoe} \circ \mathbf{swap}_{\text{se}} \circ \mathbf{access}_{2i+1} \circ \mathbf{app}$

with $\mathbf{push}_e\ e \circ \mathbf{push}_s\ x \circ \mathbf{mkclos} \circ \mathbf{brkclos} = \mathbf{push}_e\ e \circ \mathbf{push}_s\ x$

$\mathbf{etos} = \lambda_e e.\mathbf{push}_s\ e \qquad \mathbf{stoe} = \lambda_s x.\mathbf{push}_e\ x$

$\mathbf{swapclos} = \lambda_s cx.\lambda_e ce.\lambda_e e.\mathbf{push}_e\ ce \circ \mathbf{push}_s\ cx \circ \mathbf{push}_e\ e$

$\mathbf{mkbind}_e = \lambda_e e.\lambda_e x.\mathbf{push}_s\ x \circ \mathbf{push}_e\ e \circ \mathbf{mkbind}$

$\mathbf{rmvmkclos}_n = \lambda_e e.(\mathbf{push}_e\ e \circ \mathbf{access}_0 \circ \mathbf{brkclos} \circ \mathbf{etos}) \circ \ldots$

$\circ\ (\mathbf{push}_e\ e \circ \mathbf{access}_n \circ \mathbf{brkclos} \circ \mathbf{etos}) \circ \mathbf{push}_e\ () \circ \mathbf{mkbind}^{2n}$

$\mathbf{addmkclos}_n = \lambda_e e.(\mathbf{push}_e\ e \circ \mathbf{access}_0 \circ \mathbf{push}_e\ e \circ \mathbf{access}_1 \circ \mathbf{stoe} \circ \mathbf{mkclos}) \circ \ldots$

$\circ\ (\mathbf{push}_e\ e \circ \mathbf{access}_{2n} \circ \mathbf{push}_e\ e \circ \mathbf{access}_{2n+1} \circ \mathbf{stoe} \circ \mathbf{mkclos})$

$\circ\ \mathbf{push}_e\ () \circ \mathbf{mkbind}^n$

Figure 18 "Mkclos-less" Abstraction ($\mathcal{Ag}_m$)

Although this abstraction manipulates more data and needs deeper stacks and longer environments than the previous ones, it suppresses one indirection level of closure manipulation. Instantiated with a copy scheme, $\mathcal{Ag}_m$ has a bigger memory allocation granularity than the others abstractions. However, it duplicates "closures" (the code and environment references). This can cause trouble to implement the update operation of lazy languages (e.g. see TIM [26]).

In order to compare these different options it would be imperative to determine the cost of each indexed combinator (constant or linear). The following indexed combinators (Figure 19) have a constant time implementation (assuming a stack machine). It may be useful to express all the previous indexed combinators in terms of these ones. However, the complexity may also depend whether the components are merged or kept separate (see 5.2).

$\mathbf{read}_n = \lambda_i x_1 \ldots \lambda_i x_n.\lambda_i y.\mathbf{push}_i\ y \circ \mathbf{push}_i\ x_n \circ \ldots \circ \mathbf{push}_i\ x_1 \circ \mathbf{push}_i\ y$

$\mathbf{write}_n = \lambda_i y.\lambda_i x_1 \ldots \lambda_i x_n.\mathbf{push}_i\ y \circ \mathbf{push}_i\ x_{n-1} \circ \ldots \circ \mathbf{push}_i\ x_1$

$\mathbf{flush}_n \circ E = \lambda_i x_1 \ldots \lambda_i x_n.E \qquad with\ i \equiv e,s,k$

Figure 19 O(1) Indexed Combinators

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 saves the code following the function call using $\mathbf{push}_k$, and returns to it with $\mathbf{rts}_i$ ($= \lambda_i x.\lambda_k f.\mathbf{push}_i\ x \circ f$ and $i \equiv s,e$) when the function ends (as in [13][19][20]). Intuitively these combinators can be seen as implementing a control stack. Compared to Λ_e, Λ_k-expressions do not have $x \circ E$ code sequences.

$\mathcal{S} : \Lambda_e \rightarrow \Lambda_k$ *with* $i \equiv s,e$

$\mathcal{S}[\![E_1 \circ E_2]\!] = \mathbf{push}_k\ (\mathcal{S}[\![E_2]\!]) \circ \mathcal{S}[\![E_1]\!]$

$\mathcal{S}[\![\mathbf{push}_i\ E]\!] = \mathbf{push}_i\ (\mathcal{S}[\![E]\!]) \circ \mathbf{rts}_i$ *with* $\mathbf{rts}_i = \lambda_i x.\lambda_k k.\mathbf{push}_i\ x \circ k$

$\mathcal{S}[\![\lambda_i x.E]\!] = \lambda_i x.\mathcal{S}[\![E]\!]$

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

$\mathcal{S}[\![(E_1, E_2)]\!] = (\mathcal{S}[\![E_1]\!], \mathcal{S}[\![E_2]\!])$ $\mathcal{S}[\![()]\!] = ()$

Figure 20 General Compilation of Control Transfers ($\mathcal{S}$)

The correctness $\mathcal{S}$ of is stated by Property 14.

Property 14 $\forall E \in \Lambda_e^{\sigma}$ *closed, N a normal form,* $E \overset{*}{\rightarrow} N \Rightarrow \mathcal{S}[\![E]\!] \overset{*}{\rightarrow} \mathcal{S}[\![N]\!]$

An optimized version of $\mathcal{S}$ for $\mathcal{Ag}$ can easily be derived. For example:

$\mathcal{S}[\![\mathbf{dupl}_e \circ E_1 \circ \mathbf{swap}_{se} \circ E_2]\!]$

$= \mathcal{S}[\![\lambda_e e.\mathbf{push}_e\ e \circ (\mathbf{push}_e\ e \circ E_1) \circ \lambda_s x.\lambda_e e.\mathbf{push}_s\ x \circ \mathbf{push}_e\ e \circ E_2]\!]$ $(\mathbf{dupl}_e, \mathbf{swap}_{se})$

$= \lambda_e e.\mathbf{push}_e\ e \circ \mathbf{push}_k\ (\lambda_s x.\lambda_e e.\mathbf{push}_s\ x \circ \mathbf{push}_e\ e \circ \mathcal{S}[\![E_2]\!]) \circ \mathbf{push}_e\ e \circ \mathcal{S}[\![E_1]\!]$ $(\mathcal{S})$

$= \lambda_e e.\mathbf{push}_e\ e \circ \mathbf{push}_e\ e \circ \mathbf{push}_k\ (\lambda_s x.\lambda_e e.\mathbf{push}_s\ x \circ \mathbf{push}_e\ e \circ \mathcal{S}[\![E_2]\!]) \circ$

$\lambda_k k.\lambda_e e.\mathbf{push}_k\ k \circ \mathbf{push}_e\ e \circ \mathcal{S}[\![E_1]\!]$ $(\beta_s),(\beta_e)$

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

We finally get:

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

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

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

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

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

$\mathcal{S}\ [\![\mathbf{access}_i]\!] = \mathbf{access}_i \circ \mathbf{rts}_s$

Figure 21 Compilation of Control Transfers ($\mathcal{S}$)

The Λ_k expressions have no more **appclos** ∘ E code sequences. 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 with the following rule :

$$\mathbf{push}_k\ E_1 \circ \mathbf{push}_s\ E_2 \circ \mathbf{rts}_s = \mathbf{push}_s\ E_2 \circ E_1$$

To get a real machine code a further step would be to introduce labels to name sequences of code (such as E in $\mathbf{push}_x\ E$).

An alternative to $\mathcal{S}$ is to use a transformation $\mathcal{SL}$ between the control and the abstraction phases. It transforms the expression into continuation passing style. The continuation encodes return addresses and will be abstracted in the environment as an ordinary variable. This solution, known as stackless, is chosen in the New Jersey SML 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*.

$\mathcal{SL} : \Lambda_s \rightarrow \Lambda_s$

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

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

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

$\mathcal{SL}\ [\![x]\!] = x$

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

Figure 22 CPS-like Compilation of Control ($\mathcal{SL}$)

The "top level" expression must be called with an initial identity continuation ($\mathbf{Id}_s = \lambda_s x.\mathbf{push}_s\ x$). The following optimization removes unnecessary manipulations of the continuation k :

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

The correctness of $\mathcal{SL}$ is stated by Property 15.

Property 15 $\forall E \in \Lambda_s^\sigma$ *closed, N a normal form,* $E \overset{*}{\rightarrow} N \Rightarrow \mathbf{push}_s\ K \circ \mathcal{SL}[\![E]\!] \overset{*}{\rightarrow} \mathbf{push}_s\ K \circ \mathcal{SL}[\![N]\!]$

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 of composition: $\circ = \lambda xyz.x\ (y\ z)$.

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

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

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

$$\lambda_k x.X = \lambda c.\lambda s.\lambda e.\lambda(k,x).X\ c\ s\ e\ k \qquad \mathbf{push}_k\ N = \lambda c.\lambda s.\lambda e.\lambda k.c\ s\ e\ (k,N)$$

Then, the reduction of our expressions can be seen as state transitions of an abstract machine, e.g. :

$$\mathbf{push}_s\ N\ C\ S\ E\ K \rightarrow C\ (S,N)\ E\ K$$

$$\mathbf{push}_e\ N\ C\ S\ E\ K \rightarrow C\ S\ (E,N)\ K$$

$$\mathbf{push}_k\ N\ C\ S\ E\ K \rightarrow C\ S\ E\ (K,N)$$

A second option is to merge all components. Here, administrative combinators remain necessary. The underlying abstract machine has only two components (the code and a data-environment-control stack).

$$\lambda_s x.X = \lambda_e x.X = \lambda_k x.X = \lambda c.\lambda(z,x).X\ c\ z$$

$$\mathbf{push}_s\ N = \mathbf{push}_e\ N = \mathbf{push}_k\ N = \lambda c.\lambda z.c\ (z,N) \qquad \text{and} \qquad \mathbf{push}_x\ N\ C\ Z \rightarrow C\ (Z,N)$$

As previously claimed, the transformations can be compared on code size expansion assuming that combinators have a constant time operation. These comparisons must take into

account the components instantiation step which can change the size of combinators. For example $\mathbf{swap}_{se}$ becomes useless (of null size) when the component *s* and the component *e* are not merged together. In the same way, $\mathbf{copy}_n$ has a O(*n*) cost when *s* and *e* are merged together and a O(1) cost when they are kept separate.

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 [11].

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

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

$$\mathcal{V}[\![n]\!] = \mathbf{push}_s\ n$$

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

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

$$\mathcal{V}[\![head]\!] = \mathbf{head}_s$$

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

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

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

Figure 23 An Extension with Constants, Primitive Operators and Lists

Figure 23 describes a possible extension using the data stack to store constants and tagged cells of lists.

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 24 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.

$\mathcal{N}a : \Lambda \rightarrow \Lambda_s$

$\mathcal{N}a \ [\![x]\!] = x$

$\mathcal{N}a \ [\![\lambda x.E]\!] = \mathbf{push}_s \ (\lambda_s x.\mathcal{N}a \ [\![E]\!])$

$\mathcal{N}a \ [\![E_1 \ E_2]\!] = \mathbf{push}_s \ (\mathcal{N}a \ [\![E_2]\!]) \circ \mathcal{N}a \ [\![E_1]\!] \circ \mathbf{app}$

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

$\mathcal{N}m \ [\![x]\!] = x$

$\mathcal{N}m \ [\![\lambda x.E]\!] = \lambda_s x.\mathcal{N}m \ [\![E]\!]$

$\mathcal{N}m \ [\![E_1 \ E_2]\!] = \mathbf{push}_s(\mathcal{N}m \ [\![E_2]\!]) \circ \mathcal{N}m \ [\![E_1]\!]$

Figure 24 Two Transformations for Call-by-Name ($\mathcal{N}a$ & $\mathcal{N}m$)

The transformation $\mathcal{N}m$ is simpler and avoids some overhead of $\mathcal{N}a$. On the other hand, making $\mathcal{N}m$ lazy is problematic: it needs marks to be able to update closures [11][8][28]. This is exactly the same problem as with $\mathcal{V}m$; without marks we cannot know if a function represents a result or has to be applied. In the first case, a call-by-value implementation has to return it ($\mathcal{V}m$) whereas a call-by-need implementation has to update a closure ($\mathcal{N}m$).

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 $\mathcal{N}a$ can be extended as follows

$$\mathcal{N}a \ [\![\underline{x}]\!] = \mathbf{push}_s \ x \qquad\qquad \mathcal{N}a \ [\![\underline{E}_1 \ E_2]\!] = \mathcal{N}a \ [\![E_2]\!] \circ \mathcal{N}a \ [\![E_1]\!] \circ \mathbf{app}$$

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 [27]) 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 [25]), by modifying the continuation (as in [13]). 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 25 states the main design choices structuring several classical implementations. There are cosmetic differences between our descriptions and the real implementations. Also, some extensions and optimizations are not described here.

Let us state precisely the differences for the categorical abstract machine. Let $\mathcal{CAM} = \mathcal{A}s \bullet \mathcal{V}a_L$, 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*) ∘ **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 $\lambda_s(x,e).\mathbf{push}_e\ e \circ \mathbf{bind} \circ x$, which is exactly CAM's sequence **Cons;App**.

The strict Krivine abstract machine ($\mathcal{SKAM}$) compiles control using the push-enter model ([20] pp. 27). This simple machine has served as the basis of the Zinc abstract machine ([20]). Starting from $\mathcal{SKAM} = \mathcal{S} \bullet \mathcal{A}s \bullet \mathcal{V}m$, we get:

$\mathcal{SKAM}\ [\![x_i]\!]\ \rho = \mathbf{grab}_k\ (\mathbf{fst}^i \circ \mathbf{snd} \circ \mathbf{appclos})$

$\mathcal{SKAM}\ [\![\lambda x.E]\!]\ \rho = \mathbf{grab}_k\ (\mathbf{bind} \circ \mathcal{SKAM}\ [\![E]\!]\ (\rho,x))$

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

$\qquad\qquad \circ \mathbf{push}_s\ \varepsilon \circ \mathbf{swap}_{se} \circ \mathcal{SKAM}\ [\![E_1]\!]\ \rho$

with $\mathbf{push}_s\ \varepsilon \circ \mathbf{push}_e\ e \circ \mathbf{grab}_k\ F \Rightarrow \mathbf{push}_e\ e \circ \mathbf{push}_s\ F \circ \mathbf{mkclos} \circ \mathbf{rts}_s$

and $E : R\sigma \qquad E \circ \mathbf{push}_e\ e \circ \mathbf{grab}_k\ F \Rightarrow E \circ \mathbf{push}_e\ e \circ F$

The sequence **dupl**$_e$ ∘ **push**$_k$ (**swap**$_{se}$ ∘ E_2) ∘ **swap**$_{se}$ ∘ **push**$_s$ ε ∘ **swap**$_{se}$ ∘ E_1 is equivalent to the $\mathcal{SKAM}$ sequence **Reduce** E_2 **;** E_1 which uses the same stack to store the copy (**dupl**$_e$) of the environment, the return code E_2 and the mark. The main difference comes from the $\mathcal{SKAM}$ instruction **Grab** which is a merge of **bind** with a recursive version of **grab**$_k$ (see 3.2.2). So, the $\mathcal{SKAM}$ code is **Grab ;** *E* rather than **grab**$_k$ (**bind** ∘ *E*) and **Access(*i*)** (= **fst**^{i} ∘ **snd** ∘ **appclos**) rather than **grab**$_k$ (**fst**^{i} ∘ **snd** ∘ **appclos**).

Compiler	Transformations				Components
SECD	$\mathcal{V}a$	*Id*	$\mathcal{A}s$	$\mathcal{S}$	s (e ≡ k)
CAM	$\mathcal{V}a_L$	*Id*	$\mathcal{A}s$	*Id*	s ≡ e
SKAM	$\mathcal{V}m$	*Id*	$\mathcal{A}s$	$\mathcal{S}$	s ≡ e ≡ k
SML -NJ	$\mathcal{V}a_f$	$\mathcal{S}l$	*(Ac3+As)*	*Id*	s e *(reg isters)*
TABA C (cbv)	$\mathcal{V}a$	*Id*	$\mathcal{A}c2_{\text{dsb}}$	$\mathcal{S}$	(s ≡ e) k
TABA C (cbn)	$\mathcal{N}a$	*Id*	$\mathcal{A}c2_{\text{dsb}}$	$\mathcal{S}$	(s ≡ e) k
TIM (cbn)	$\mathcal{N}m$	*Id*	$\mathcal{A}c1_{\text{m}}$	*Id*	s e

Figure 25 Several Classical Compilation Schemes

Let us quickly review the other differences between Figure 25 and real implementations. The SECD machine [19] 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 [14]. 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 [13] and has greatly inspired this study. It implements strict or non-strict languages by program transformations. The environments are unfolded in the environment/data stack with a mk-bind-less, dupl-less, swap-less (see 4.5.5) version of $\mathcal{A}c2$. The call-by-name TIM [11] unfolds closures in the environment as mentioned in 4.5.6. The environment copying included in the transformation $\mathcal{A}c1$ have 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 $T ::= a \mid m \mid T_1 \overset{a/m}{\rightarrow} T_2$ with a (resp. m) for apply (resp. marks) mode. Intuitively a function E: $\alpha \overset{\delta}{\rightarrow} \beta$ 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.

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

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

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

with

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

Figure 26 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 26) just adds $\mathbf{push}_s\ \varepsilon$ 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 $\mathbf{grab}_s$ 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 [23]).

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

Another solution uses environments mixing lists and vectors (as in [29]).

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

$\mathcal{Mix}_{\mathcal{A}}\ [\![x_i]\!]\ (\ldots(\rho,\rho_i),\ldots,\rho_0) = \mathbf{access}^l_i \circ \mathcal{Mix}_{\mathcal{A}}\ [\![x_i]\!]\ \rho_i$ *with* x_i *in* ρ_i

$\mathcal{Mix}_{\mathcal{A}}\ [\![x_i]\!]\ [\rho:\rho_i:\ldots:\rho_0] = \mathbf{access}^v_i \circ \mathcal{Mix}_{\mathcal{A}}\ [\![x_i]\!]\ \rho_i$ *with* x_i *in* ρ_i

$\mathcal{Mix}_{\mathcal{A}}\ [\![x_i]\!]\ (\ldots(\rho,x_i),\ldots,x_0) = \mathbf{access}^l_i \circ \mathbf{appclos}$

$\mathcal{Mix}_{\mathcal{A}}\ [\![x_i]\!]\ [\rho:x_i:\ldots:x_0] = \mathbf{access}^v_i \circ \mathbf{appclos}$

with $\mathbf{access}^l_i$ *is the* $\mathbf{access}_i$ *version which access a list*

and $\mathbf{access}^v_i$ *is the* $\mathbf{access}_i$ *version which access a vector*

Figure 27 Hybrid Abstraction (extract)

Each λ-abstraction is annotated by a new mixed environment structure θ and ⊕ (∈ {*v*,*l*}) which indicates how to bind the current value (as a vector "*v*" or as a link "*l*"). Mixed structures are built by $\mathbf{mkbind}^v$, $\mathbf{mkbind}^l$ and the macro-combinator **Mix** which copies and restructures the environment ρ according to the annotation θ (Figure 27). Paths to values are now expressed by sequences of $\mathbf{access}^l_i$ and $\mathbf{access}^v_i$. The abstraction algorithm distinguishes vectors from lists in the compile time environment using constructors ":" and ",".

9 Conclusion

We have presented a framework to describe, prove and compare functional implementation techniques and optimizations (see Figure 3 in 2.6 for a summary). Our first intermediate language Λ_s bears strong similarities with CPS-expressions. Indeed, if we take combinator definitions (**DEF1**) (section 2.6) we naturally get Fischer's CPS transformation [12] 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. We see Λ_s as a powerful and more abstract framework than CPS to express different reduction strategies. As pointed out by Hatcliff & Danvy [15], Moggi's computational metalanguage [24] 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 $\mathbf{push}_s\ E$ 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) [24]. 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 [31] or operational semantics [14] [28]. They aim at providing 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][26] and CMCM and TIM [22]. 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 , 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.

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] O. Danvy. Back to direct style. In *Proc. of ESOP'92*, LNCS Vol. 582, pp. 130-150, 1992.

[10] R. Douence and P. Fradet. Towards a taxonomy of functional language implementations. In *Proc of PLILP'95*, LNCS 982, pp. 27-44, 1995.

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

[18] 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.

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

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

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

[22] 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.

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

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

[25] 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.

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

[27] 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.

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

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

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

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

Annex

A Proofs of Property 2, Property 3, Property 4 and Property 5 (§ 2.3)

Property 2. It is clearly sufficient to show the property for one reduction step. The proof for the inductive rules such as $E \blacktriangleright N \Rightarrow E \circ F \blacktriangleright N \circ F$ is obvious. The interesting rule is the β_s-reduction and the proof boils down to the proof of $\Gamma \vdash F : \sigma$ and $\Gamma \cup \{x{:}\sigma\} \vdash E : \tau \Rightarrow \Gamma \vdash E\,[F/x] : \tau$. This is shown by structural induction.

- $E \equiv x$ then $\quad \sigma \equiv \tau$ and $x[F/x] \equiv F$ so $\Gamma \vdash F{:}\sigma \Rightarrow \Gamma \vdash E\,[F/x](\equiv F) : \tau\ (\equiv \sigma)$
- $x \notin E$ (i.e. $E \equiv y \not\equiv x$ or $E \equiv \lambda_s x.E'$) then $\Gamma \cup \{x{:}\sigma\} \vdash E : \tau \Rightarrow \Gamma \vdash E[F/x]\ (\equiv E) : \tau$
- $E \equiv \lambda_s z.E'$ $(z \not\equiv x)$ then

  $\Gamma \cup \{x{:}\sigma\} \vdash \lambda_s z.E' : \tau\ (\equiv \tau_1 \rightarrow_s \tau_2) \Leftrightarrow \Gamma \cup \{x{:}\sigma\} \cup \{z{:}\tau_1\} \vdash E' : \tau_2$
  since $z \not\equiv x$, $\Gamma \cup \{z{:}\tau_1\} \cup \{x{:}\sigma\} \vdash E' : \tau_2$ and since the definition of substitution enforces z not to occur free in F (by variable renaming or convention) $\Gamma \vdash F : \sigma \Rightarrow \Gamma \cup \{z{:}\tau_1\} \vdash F : \sigma$. So, by induction hypothesis, $\Gamma \cup \{z{:}\tau_1\} \vdash E'\,[F/x] : \tau_2$ which implies $\Gamma \vdash \lambda_s z.E'\,[F/x] : \tau_1 \rightarrow_s \tau_2$.
- $E \equiv E_1 \circ E_2$ then

  $\Gamma \cup \{x{:}\sigma\} \vdash E_1 \circ E_2 : \tau \Rightarrow \Gamma \cup \{x{:}\sigma\} \vdash E_1{:}\ \mathsf{R}\tau_1$ and $\Gamma \cup \{x{:}\sigma\} \vdash E_2{:}\ \tau_1 \rightarrow_s \tau$. Using $\Gamma \vdash F : \sigma$ and the induction hypothesis we get $\Gamma \cup \{x{:}\sigma\} \vdash E_1[F/x]{:}\ \mathsf{R}\tau_1$ and $\Gamma \cup \{x{:}\sigma\} \vdash E_2[F/x]{:}\ \tau_1 \rightarrow_s \tau$ so $\Gamma \cup \{x{:}\sigma\} \vdash (E_1 \circ E_2)\,[F/x]{:}\ \tau$
- $E \equiv \mathbf{push}_s\, E'$ then

  $\Gamma \cup \{x{:}\sigma\} \vdash \mathbf{push}_s\, E'{:}\ \tau(\equiv \mathsf{R}\tau_1) \Rightarrow \Gamma \cup \{x{:}\sigma\} \vdash E'{:}\ \tau_1 \Rightarrow \Gamma \cup \{x{:}\sigma\} \vdash E'[F/x]{:}\ \tau_1$ (by induction hypothesis) $\Rightarrow \Gamma \cup \{x{:}\sigma\} \vdash \mathbf{push}_s\, E'[F/x]{:}\ \mathsf{R}\tau_1(\equiv \tau)$ ❒

Property 3. Structural induction. We have to show that an expression ${E_1}^{\mathsf{R}\sigma} \circ {E_2}^{\sigma \rightarrow_s \tau}$ is reducible. If $E_1 \equiv \mathbf{push}_s\, E$ then either $E_2 \equiv \lambda_s x.F$ (and $E_1 \circ E_2$ is a redex) or $E_2 \equiv E'_2 \circ E''_2$ and by hypothesis E_2 has a redex (thus $E_1 \circ E_2$ is reducible). Otherwise $E_1 \equiv E'_1 \circ E''_1$ and by hypothesis E_1 has a redex (thus $E_1 \circ E_2$ is reducible). ❒

Property 4. If $E{:}\mathsf{R}\tau$ has a normal form N then $E \overset{*}{\blacktriangleright} N$. By Property 2, $N{:}\mathsf{R}\tau$ and by Property 3 (N is not reducible) $N \equiv \mathbf{push}_s\, V$, so $E \overset{*}{\blacktriangleright} \mathbf{push}_s\, V$. Same thing with $E{:}\sigma \rightarrow_s \tau$ ❒

Property 5. Induction on the reduction tree. Evident if E is canonical (by the implicit rule $N \succ N$). If $E \equiv E_1 \circ E_2$, since all reduction strategies are normalizing :

$E \overset{*}{\blacktriangleright} N \Leftrightarrow E_1 \overset{*}{\blacktriangleright} \mathbf{push}_s\, V$ and $E_2 \overset{*}{\blacktriangleright} \lambda_s x.F$ and $F[V/x] \overset{*}{\blacktriangleright} N$

or $E_1 \overset{*}{\blacktriangleright} N_1$ and $E_2 \overset{*}{\blacktriangleright} N_2$ and $(N_1 \not\equiv \mathbf{push}_s\, V$ or $N_1 \not\equiv \lambda_s x.F)$ (i.e. $N \equiv N_1 \circ N_2$)

$\Leftrightarrow E_1 \succ \mathbf{push}_s\ V$ and $E_2 \succ \lambda_s x.F$ and $F[V/x] \succ N$

or $E_1 \succ N_1$ and $E_2 \succ N_2$ $(N_1 \not\equiv \mathbf{push}_s\ V$ or $N_1 \not\equiv \lambda_s x.F)$ (by induction hypothesis)

$\Leftrightarrow E \succ N$

In the typed case, the closed expression $E \equiv E_1 \circ E_2$, E_1 (resp. E_2) reduces to $\mathbf{push}_s\ V$ (resp. $\lambda_s x.F$) (Property 4), so the first inference rule of the natural semantics is sufficient. ❒

B Proofs of Laws (§ 2.4)

Laws are valid in their generality only within the corresponding of a classical consistent extension of the λ-calculus (identification of unsolvable terms and ω-rule):

(Ω_s) If M and N do not have a (weak) normal form then $M = N$

(ω_s) Let $\Gamma \cup \{z{:}\sigma\} \vdash M, N{:}\tau$ if $\forall Z{:}\sigma$ closed $M[Z/z] = N[Z/z]$ then $M = N$

Intuitively, the motivation behind this extension is that our only concern is that two equal terms behave the same during the reduction. That is, we accept to replace an expression by another as long as they are equal after their free variables are instantiated or to replace a looping expression by another looping expression.

Law (L2) If E_1 does not have a normal form then both expressions $E_1 \circ (\lambda_s x.E_2 \circ E_3)$ and $E_2 \circ E_1 \circ (\lambda_s x.E_3)$ will not have normal forms. They can be seen as equal (Ω_s).Otherwise, let $z_1,\ldots, z_n$ the free variables of $E_1 \circ (\lambda_s x.E_2 \circ E_3)$ then $\forall Z_1,\ldots, Z_n$ closed

$(E_1 \circ (\lambda_s x.E_2 \circ E_3))[Z_1,\ldots, Z_n/z_1,\ldots, z_n]$

$= E_1[Z_1,\ldots, Z_n/z_1,\ldots, z_n] \circ (\lambda_s x.E_2\ [Z_1,\ldots, Z_n/z_1,\ldots, z_n] \circ E_3[Z_1,\ldots, Z_n/z_1,\ldots, z_n])$

$E_1[Z_1,\ldots, Z_n/z_1,\ldots, z_n]$ is closed. By Property 4, there exists N such that $E_1[Z_1,\ldots,Z_n/z_1,\ldots, z_n] = \mathbf{push}_s\ N$ so

$= \mathbf{push}_s\ N \circ (\lambda_s x.E_2\ [Z_1,\ldots, Z_n/z_1,\ldots, z_n] \circ E_3[Z_1,\ldots, Z_n/z_1,\ldots, z_n])$

$= E_2[Z_1,\ldots, Z_n/z_1,\ldots, z_n] \circ E_3\ [Z_1,\ldots, Z_n/z_1,\ldots, z_n][N/x]$ (β_s) and x is not free in E_2

$= E_2[Z_1,\ldots, Z_n/z_1,\ldots, z_n] \circ \mathbf{push}_s\ N \circ (\lambda_s x.E_3[Z_1,\ldots, Z_n/z_1,\ldots, z_n])$ (β_s)

$= (E_2 \circ E_1 \circ (\lambda_s x.E_3))\ [Z_1,\ldots, Z_n/z_1,\ldots, z_n]$

So $(E_1 \circ (\lambda_s x.E_2 \circ E_3)) = (E_2 \circ E_1 \circ (\lambda_s x.E_3))$ (ω_s)

- Law (L3) Similar. ❒

C Proof of Property 6 and Property 7

Property 6. Induction on the reduction tree. We have to prove that if condition (C1) is verified then Property 6 holds. Since E is typed either E is a normal form and the property holds trivially either $E \equiv E_1 \circ E_2$ and $E_1 \overset{*}{\rightarrow} \mathbf{push}_s\ V$ and $E_2 \overset{*}{\rightarrow} \lambda_s x.F$ and $F[V/x] \overset{*}{\rightarrow} N$. By induction hypothesis $E_1\ X_1 \ldots X_n \overset{*}{\rightarrow} \mathbf{push}_s\ V\ X_1 \ldots X_n$, $E_2\ X_1 \ldots X_n \overset{*}{\rightarrow} \lambda_s x.F\ X_1 \ldots X_n$ and $F[V/x]X_1 \ldots X_n \overset{*}{\rightarrow} N\ X_1 \ldots X_n$. Thus, using condition (C1), $E_1 \circ E_2\ X_1 \ldots X_n \overset{*}{\rightarrow} F[V/x]\ X_1 \ldots X_n \overset{*}{\rightarrow} N\ X_1 \ldots X_n$. ❐

Property 7. We need the following lemma, whose proof is analogous to Lemma 17 (see below) for Λ_s.

Lemma 16 $[\![E[F/x]]\!]^{-1} = [\![E]\!]^{-1}\ [\ [\![F]\!]^{-1}/x]$

We have to check Property 7 for each case of the definition of equality in Λ_s, that is

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

  $[\![(\mathbf{push}_s\ F) \circ (\lambda_s x.E)]\!]^{-1} = (\lambda x.\ [\![E]\!]^{-1})\ [\![F]\!]^{-1} =_\beta [\![E]\!]^{-1}\ [\ [\![F]\!]^{-1}/x] = [\![E[F/x]]\!]^{-1}$ (Lemma 16)

- $\lambda_s x.(\mathbf{push}_s\ x \circ E) = E$ *if x does not occur free in E*

  $[\![\lambda_s x.(\mathbf{push}_s\ x \circ E)]\!]^{-1} = (\lambda x.\ [\![E]\!]^{-1}\ x) =_\eta E$ (x does not occur free in E implies x does not occur free in $[\![E]\!]^{-1}$).

- $E_2 = F \Rightarrow E_2 \circ E_1 = F \circ E_1$

  $[\![E_2 \circ E_1]\!]^{-1} = [\![E_1]\!]^{-1}\ [\![E_2]\!]^{-1}$ by induction hypothesis $E_2 = F \Rightarrow [\![E_2]\!]^{-1} = [\![F]\!]^{-1}$ thus

  $[\![E_1]\!]^{-1}\ [\![E_2]\!]^{-1} = [\![E_1]\!]^{-1}\ [\![F]\!]^{-1} = [\![F \circ E_1]\!]^{-1}$

- same thing for $E_1 = F \Rightarrow E_2 \circ E_1 = E_1 \circ F$, $E = F \Rightarrow \mathbf{push}_s\ E = \mathbf{push}_s\ F$ and $E = F \Rightarrow \lambda_s x.E = \lambda_s x.F$. ❐

D Generic Substitution Lemma

This lemma is useful for several proofs. A context $\mathbf{X}[]$ is said to be closed if for all expressions E, F and variable x, $\mathbf{X}[E]\ [F/x] \equiv \mathbf{X}[E\ [F/x]]$ (i.e. a closed context does not introduce free variables nor does it bind free variables).

Lemma 17 *Let* $\mathbf{X}\ []$, $\mathbf{Y}\ []$, $\mathbf{Z}\ [][]$ *be closed contexts and* $\mathcal{T}$ *a transformation such that*

$\mathcal{T}\ [\![x]\!] = \mathbf{X}\ [x]$ $\quad \mathcal{T}\ [\![\lambda x.E]\!] = \mathbf{Y}\ [\lambda x.\mathcal{T}\ [\![E]\!]\ ]$ $\quad \mathcal{T}\ [\![E_1\ E_2]\!] = \mathbf{Z}\ [\mathcal{T}\ [\![E_1]\!]]\ [\ \mathcal{T}\ [\![E_2]\!]\ ]$

then for all E and F such that $\mathcal{T}\ [\![F]\!] \equiv \mathbf{X}[F']$ $\quad \mathcal{T}\ [\![E[F/x]]\!] \equiv \mathcal{T}\ [\![E]\!]\ [F'/x]$

Proof. By structural induction.

- $E \equiv x$ $\quad \mathcal{T}\ [\![x[F/x]]\!] \equiv \mathcal{T}\ [\![F]\!] \equiv \mathbf{X}\ [F'] \equiv \mathbf{X}\ [x[F'/x]] \equiv (\mathbf{X}\ [x])[F'/x] \equiv \mathcal{T}\ [\![x]\!]\ [F'/x]$

since **X** closed

- $x \notin E \quad \mathcal{T}\ [\![E[F/x]]\!] \equiv \mathcal{T}\ [\![E]\!] \equiv \mathcal{T}\ [\![E]\!]\ [F'/x]$ since $\mathcal{T}$ does not introduce free variables
- $E \equiv \lambda z.E' \ (z \not\equiv x) \qquad \mathcal{T}\ [\![(\lambda z.E')[F/x]]\!] \equiv \mathcal{T}\ [\![\lambda z.(E'[F/x])]\!] \equiv \mathbf{Y}\ [\lambda z.\mathcal{T}[\![E'[F/x]]\!]]$

  $\equiv \mathbf{Y}\ [\lambda z.\mathcal{T}[\![E']\!]\ [F'/x])]$ by induction hypothesis

  $\equiv \mathbf{Y}\ [\lambda z.\mathcal{T}[\![E']\!]\ ]\ [F'/x]$ since **Y** closed

  $\equiv \mathcal{T}\ [\![\lambda z.E']\!]\ [F'/x]$
- $E \equiv E_1\ E_2 \qquad \mathcal{T}\ [\![(E_1\ E_2)[F/x]]\!] \equiv \mathcal{T}\ [\![(E_1\ [F/x])\ (E_2\ [F/x])]\!]$

  $\equiv \mathbf{Z}\ [\mathcal{T}\ [\![E_1\ [F'/x]]\!]]\ [\mathcal{T}[\![E_2\ [F/x]]\!]]$

  $\equiv \mathbf{Z}\ [\mathcal{T}\ [\![E_1]\!]\ [F'/x]]\ [\ \mathcal{T}\ [\![E_2]\!]\ [F'/x]]$ by induction hypothesis

  $\equiv \quad \mathbf{Z}\ [\mathcal{T}\ [\![E_1]\!]]\ [\mathcal{T}\ [\![E_2]\!]]\ [F'/x]$ since **Z** closed

  $\equiv \mathcal{T}\ [\![E_1\ E_2]\!]\ [F'/x]$ ❒

In particular, the transformations $\mathcal{V}a$, $\mathcal{V}a_L$ and $\mathcal{V}m$ verify the conditions of the lemma. So, we have

$\mathcal{V}a_{(L)}\ [\![E[F/x]]\!] \equiv \mathcal{V}a_{(L)}\ [\![E]\!]\ [F'/x]$ if $\mathcal{V}a_{(L)}\ [\![F]\!] \equiv \mathbf{push}_s\ F'$

$\mathcal{V}m\ [\![E[F/x]]\!] \equiv \mathcal{V}m\ [\![E]\!]\ [F'/x]$ if $\mathcal{V}m\ [\![F]\!] \equiv \mathbf{grab}\ F'$

E Proof of Property 8

We prove the stronger property let E an expression with free variables $\{x_1 \ldots x_n\}$ such that $\{x_1{:}\alpha_1, \ldots x_n{:}\alpha_n\} \vdash E{:}\sigma$ then $\{x_1{:}\overline{\alpha}_1, \ldots x_n{:}\overline{\alpha}_n\} \vdash \mathcal{V}a\ [\![E]\!] : \mathsf{R}\overline{\sigma}$.

Proof. By structural induction.

- $E \equiv x_i \qquad \{x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n\} \vdash E{:}\alpha_i$ then $\{x_1{:}\overline{\alpha}_1, \ldots, x_n{:}\overline{\alpha}_n\} \vdash \mathbf{push}_s\ x_i\ (\equiv \mathcal{V}a\ [\![x_i]\!]) : \mathsf{R}\overline{\alpha}_i$
- $E \equiv \lambda z.E' \ \{x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n\} \vdash E{:}\sigma \to \tau$ that is $\{x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n\} \cup \{z{:}\sigma\} \vdash E'{::}\tau$.

By induction hypothesis, $\{x_1{:}\overline{\alpha}_1, \ldots x_n{:}\overline{\alpha}_n\} \cup \{z{:}\overline{\sigma}\} \vdash \mathcal{V}a\ [\![E']\!]: \mathsf{R}\overline{\tau}$

and $\{x_1{:}\overline{\alpha}_1, \ldots x_n{:}\overline{\alpha}_n\} \vdash \lambda_s z.\mathcal{V}a\ [\![E']\!]: \overline{\sigma} \to_s \mathsf{R}\overline{\tau}\ (\equiv \overline{\sigma \to \tau})$

hence $\{x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n\} \vdash \mathbf{push}_s\ (\lambda_s z.\mathcal{V}a\ [\![E']\!])\ (\equiv \mathcal{V}a\ [\![\lambda z.E']\!]): \mathsf{R}(\overline{\sigma \to \tau})$

- $E \equiv E_1\ E_2$ $\{x_1{:}\alpha_1, \ldots x_n{:}\alpha_n\} \vdash E_1{:}\sigma \to \tau$ and $\{x_1{:}\alpha_1, \ldots x_n{:}\alpha_n\} \vdash E_2{:}\sigma$

By induction hypothesis,

$\{x_1{:}\overline{\alpha}_1,\ldots\ x_n{:}\overline{\alpha}_n\} \vdash \nu_a [\![E_1]\!]\colon \mathsf{R}(\overline{\sigma \to \tau})$ and $\{x_1{:}\overline{\alpha}_1,\ldots\ x_n{:}\overline{\alpha}_n\} \vdash \nu_a [\![E_2]\!]\colon \mathsf{R}\,\overline{\sigma}$

and $\vdash \mathbf{app}\colon (\overline{\sigma \to \tau}) \to_s (\overline{\sigma \to \tau})$ thus $\{x_1{:}\overline{\alpha}_1,\ldots\ x_n{:}\overline{\alpha}_n\} \vdash \nu_a [\![E_1]\!] \circ \mathbf{app} : \overline{\sigma \to \tau}$

and $\{x_1{:}\overline{\alpha}_1,\ldots\ x_n{:}\overline{\alpha}_n\} \vdash \nu_a [\![E_2]\!] \circ \nu_a [\![E_1]\!] \circ \mathbf{app} : \mathsf{R}\overline{\tau}$ ❒

F Proof of Property 9

We first need the following lemma

Lemma 18 $\forall E$ *closed* $\in \Lambda\ \nu_a [\![E]\!] \succ X \Rightarrow \exists N \in \Lambda$ *such that* $\nu_a [\![N]\!] \equiv X$

Proof. If $E \equiv \lambda x.F$ then $N{\equiv}E$. If $E \equiv E_1\ E_2$ then $\nu_a [\![E]\!] \equiv \nu_a [\![E_2]\!] \circ \nu_a [\![E_1]\!] \circ \mathbf{app}$. By Property 8 and Property 4 $\nu_a [\![E]\!] \succ \mathbf{push}_s\ X$ so there must be a derivation $\nu_a [\![E_2]\!] \succ \mathbf{push}_s\ V'$, $\nu_a [\![E_1]\!] \succ \mathbf{push}_s\ (\lambda_s x.F')$ and $F'[V'/x] \succ \mathbf{push}_s\ X$. By induction hypothesis, there are V such that $\nu_a [\![V]\!] \equiv \mathbf{push}_s\ V'$and Z such that $\nu_a [\![Z]\!] \equiv \mathbf{push}_s(\lambda_s x.F')$ (i.e. $Z \equiv \lambda x.F$ with $\nu_a [\![F]\!] \equiv F'$). So $F'[V'/x] \equiv \nu_a [\![F]\!][V'/x] \equiv \nu_a [\![F[V/x]]\!]$ (Lemma 17) and from $\nu_a [\![F[V/x]]\!] \succ \mathbf{push}_s\ X$ we deduce by induction hypothesis that there is N such that $\nu_a [\![N]\!] \equiv \mathbf{push}_s\ X$. ❒

Call-by-value reduction is described by the following natural semantics (with V and N normal forms):

$$\frac{E_1 \xRightarrow{cbv} \lambda x.F \qquad E_2 \xRightarrow{cbv} V \qquad F[V/x] \xRightarrow{cbv} N}{E_1\ E_2 \xRightarrow{cbv} N}$$

The proof of Property 9 is on the shape of the reduction trees.

Axioms.

($\Rightarrow$) If E is not reducible it is of the form $\lambda x.F$ (E is closed) and $\nu_a [\![\lambda x.F]\!] \equiv \mathbf{push}_s\ (\lambda_s x.\nu_a [\![F]\!])$ which is not reducible.

($\Leftarrow$) If $\nu_a [\![E]\!]$ is not reducible then E is of the form $\lambda x.F$. Indeed, since E is closed, the only alternative would be $E \equiv (\lambda x.F)\ E_1\ \ldots\ E_n$ but then $\nu_a [\![E]\!]$ would be reducible (there would be the redex $\mathbf{push}_s\ (\lambda_s x.\nu_a [\![F]\!]) \circ \mathbf{app}$). So E is not reducible.

Induction.

($\Rightarrow$) E is reducible, that is, $E \equiv E_1\ E_2$, $E_1 \xRightarrow{cbv} \lambda x.F$, $E_2 \xRightarrow{cbv} V$ and $F[V/x] \xRightarrow{cbv} N$. By induction hypothesis, we have $\nu_a [\![E_1]\!] \succ \nu_a [\![\lambda x.F]\!]$, $\nu_a [\![E_2]\!] \succ \nu_a [\![V]\!]$ and $\nu_a [\![F[V/x]]\!] \succ \nu_a [\![N]\!]$. Since V is closed $\nu_a [\![V]\!] \equiv \mathbf{push}_s\ V'$and, by Lemma 17, $\nu_a [\![F]\!][V'/x] \equiv \nu_a [\![F[V/x]]\!]$, we have $\nu_a [\![E_2]\!] \succ \mathbf{push}_s\ V'$, $\nu_a [\![E_1]\!] \circ \mathbf{app} \succ \lambda_s x.\nu_a [\![F]\!]$ and $\nu_a [\![F]\!][V'/x] \succ \nu_a [\![N]\!]$ therefore, $\nu_a [\![E_1\ E_2]\!] \equiv \nu_a [\![E_2]\!] \circ \nu_a [\![E_1]\!] \circ \mathbf{app} \succ \nu_a [\![N]\!]$.

($\Leftarrow$) $\nu_a [\![E]\!]$ is reducible, that is, $E \equiv E_1\ E_2$ and $\nu_a [\![E]\!] \succ N'$. Since $\nu_a [\![E]\!]$ is well-typed (Property 8), the reduction tree must be of the form $\nu_a [\![E_2]\!] \succ \mathbf{push}_s\ V'$, $\nu_a [\![E_1]\!] \succ \mathbf{push}_s\ (\lambda_s x F')$ and $F'[V'/x] \succ N'$. By Lemma 18 we know that there is Vsuch that $\nu_a [\![V]\!] \equiv \mathbf{push}_s\ V'$, Z such that $\nu_a [\![Z]\!] \equiv \mathbf{push}_s\ (\lambda x.F)$, (i.e. $Z \equiv \lambda x.F$with $\nu_a [\![F]\!] \equiv F'$) and N such that ν_a

$[\![N]\!] \equiv N'$. So, by induction hypothesis, $E_1 \xrightarrow[cbv]{} \lambda x.F$, $E_2 \xrightarrow[cbv]{} V$. By Lemma 17, $\mathcal{V}_a\ [\![F]\!]\,[V'/x] \equiv \mathcal{V}_a\ [\![F[V/x]]\!]$ and, by induction hypothesis, $F[V/x] \xrightarrow[cbv]{} N$, thus $E \xrightarrow[cbv]{} N$. ❒

G Proof of Property 10

A technical problem with $\mathbf{grab}_s$ is that it is not well-typed: it returns a result or applies a function depending on marks. However, expressions are composed in a very regular way and it is not complicated to extend the type system to accept this style of expressions (this extension can be used to detect result expressions as needed by law (L7)).

$$\frac{\Gamma \vdash E : \sigma}{\Gamma \vdash \mathbf{grab}_s\, E : \mathsf{G}\sigma} \qquad \frac{\Gamma \vdash E_2 : \mathsf{G}\sigma}{\Gamma \vdash \mathbf{push}_s\, \varepsilon \circ E_2 : \mathsf{R}\sigma} \qquad \frac{\Gamma \vdash E_1 : \mathsf{R}\sigma \quad \Gamma \vdash E_2 : \mathsf{G}(\sigma \to_c \tau)}{\Gamma \vdash E_1 \circ E_2 : \tau}$$

The proof of the property could be done in the same way as for Property 9. We would have to show that the needed properties and lemmas holds with the new type system. Here, we exhibit an alternative proof very close to the previous one but not relying on types. Like the correctness proof of $\mathcal{V}_a$, the proof relies on the fact that normal forms of $\mathcal{V}_m\ [\![E]\!]$ are canonical and of the form $\mathcal{V}_m\ [\![N]\!]$ (corresponding of Lemma 18).

Lemma 19 *A closed normal form $\mathcal{V}_m\ [\![E]\!]$ is of the form $\mathbf{grab}_s(\lambda_s x.\mathcal{V}_m\ [\![F]\!])$*

True for $\mathcal{V}_m\ [\![\lambda x.F]\!]$. The only other case is $E \equiv E_1\, E_2$. We show by structural induction that the expression $\mathcal{V}_m\ [\![E_1\, E_2]\!]$ is reducible. If $E_2 \equiv \lambda_s x.F$ then $\mathbf{push}_s\, \varepsilon \circ \mathbf{grab}_s(\lambda_s x.\mathcal{V}_m\ [\![F]\!])$ is reducible. If $E_2 \equiv E'_2 \circ E''_2$ by hypothesis $\mathcal{V}_m\ [\![E_2]\!]$ is reducible (thus $\mathcal{V}_m\ [\![E_1\, E_2]\!]$ is reducible). ❒

Lemma 20 *Normal forms of closed expressions $\mathcal{V}_m\ [\![E]\!]$ have the form $\mathcal{V}_m\ [\![N]\!]$*

We prove by induction on the shape of the reduction trees that if $\mathcal{V}_m\ [\![E]\!]$ has a normal form then there is a normal form $\mathcal{V}_m\ [\![N]\!]$ such that $\mathcal{V}_m\ [\![E]\!] \overset{*}{\rightarrow} \mathcal{V}_m\ [\![N]\!]$. Let $E \equiv E_1\, E_2$ (only case where $\mathcal{V}_m\ [\![E]\!]$ is reducible) then $\mathcal{V}_m\ [\![E_1\, E_2]\!] \equiv \mathbf{push}_s\, \varepsilon \circ \mathcal{V}_m\ [\![E_2]\!] \circ \mathcal{V}_m\ [\![E_1]\!]$. $\mathcal{V}_m\ [\![E_1\, E_2]\!]$ has a normal form only if $\mathcal{V}_m\ [\![E_2]\!]$ and $\mathcal{V}_m\ [\![E_1]\!]$ have normal forms so, by hypothesis $\mathcal{V}_m\ [\![E_2]\!] \overset{*}{\rightarrow} \mathcal{V}_m\ [\![V]\!]$ and $\mathcal{V}_m\ [\![E_1]\!] \overset{*}{\rightarrow} \mathcal{V}_m\ [\![W]\!]$. Lemma 19 implies $\mathcal{V}_m\ [\![V]\!] \equiv \mathbf{grab}_s V'$ and $\mathcal{V}_m\ [\![W]\!] \equiv \mathbf{grab}_s(\lambda_s x.\mathcal{V}_m\ [\![F]\!])$ so $\mathcal{V}_m\ [\![E_1\, E_2]\!] \overset{*}{\rightarrow} \mathcal{V}_m\ [\![F]\!]\,[V'/x] \equiv \mathcal{V}_m\ [\![F[V/x]]\!]$ (Lemma 17). But $\mathcal{V}_m\ [\![E_1\, E_2]\!]$ has a normal form only if $\mathcal{V}_m\ [\![F[V/x]]\!]$ has one. So, by induction hypothesis, $\mathcal{V}_m\ [\![F[V/x]]\!] \overset{*}{\rightarrow} \mathcal{V}_m\ [\![N]\!]$ hence $\mathcal{V}_m\ [\![E_1\, E_2]\!] \overset{*}{\rightarrow} \mathcal{V}_m\ [\![N]\!]$. ❒

We can now tackle the proof of Property 10 by induction on the shape of the reduction trees.

Axioms.

($\Rightarrow$) If E is not reducible it is of the form $\lambda x.F$ (E is closed) and $\mathcal{V}_m\ [\![\lambda x.F]\!] \equiv \mathbf{grab}_s(\lambda_s x.\mathcal{V}_m\ [\![F]\!])$ which is not reducible.

($\Leftarrow$) If $\mathcal{V}_m\ [\![E]\!]$ is not reducible then $\mathcal{V}_m\ [\![E]\!] \equiv \mathbf{grab}_s(\lambda_s x.\mathcal{V}_m\ [\![F]\!])$ (Lemma 19). So E must be of the form $\lambda x.F$ and is not reducible.

Induction.

($\Rightarrow$) E is reducible that is, $E \equiv E_1\ E_2$, $E_1 \xrightarrow[cbv]{} \lambda x.F$, $E_2 \xrightarrow[cbv]{} V$ and $F[V/x] \xrightarrow[cbv]{} N$. By induction hypothesis $\mathcal{V}m\ [\![E_1]\!] \overset{*}{\rightarrow} \mathcal{V}m\ [\![\lambda x.F]\!]$, $\mathcal{V}m\ [\![E_2]\!] \overset{*}{\rightarrow} \mathcal{V}m\ [\![V]\!]$ and $\mathcal{V}m\ [\![F[V/x]]\!] \overset{*}{\rightarrow} \mathcal{V}m\ [\![N]\!]$ and since V is closed $\mathcal{V}m\ [\![V]\!] \equiv \mathbf{grab}_s\ V'$ (Lemma 19) So $\mathbf{push}_s\ \varepsilon \circ \mathcal{V}m\ [\![E_2]\!] \overset{*}{\rightarrow} \mathbf{push}_s\ \varepsilon \circ \mathbf{grab}_s\ V' \overset{*}{\rightarrow} \mathbf{push}_s\ V'$ and $\mathcal{V}m\ [\![E_1\ E_2]\!] \equiv \mathbf{push}_s\ \varepsilon \circ \mathcal{V}m\ [\![E_2]\!] \circ \mathcal{V}m\ [\![E_1]\!] \overset{*}{\rightarrow} \mathbf{push}_s\ V' \circ \mathbf{grab}_s\ (\lambda_s x.\mathcal{V}m\ [\![F]\!]) \overset{*}{\rightarrow} \mathcal{V}m\ [\![F]\!]\ [V'/x] \equiv \mathcal{V}m\ [\![F[V/x]\ ]\!]$ (Lemma 17) $\overset{*}{\rightarrow} \mathcal{V}m\ [\![N]\!]$

($\Leftarrow$) $\mathcal{V}m\ [\![E]\!]$ is reducible, that is $E \equiv E_1\ E_2$ and $\mathcal{V}m\ [\![E_1\ E_2]\!] \equiv \mathbf{push}_s\ \varepsilon \circ \mathcal{V}m\ [\![E_2]\!] \circ \mathcal{V}m\ [\![E_1]\!]$. Since $\mathcal{V}m\ [\![E]\!]$ has a normal form $\mathcal{V}m\ [\![N]\!]$, a possible strategy is to reduce first $\mathcal{V}m\ [\![E_1]\!]$ and $\mathcal{V}m\ [\![E_2]\!]$ to normal forms. Thus, by Lemma 20, we have $\mathcal{V}m\ [\![E_2]\!] \overset{*}{\rightarrow} \mathcal{V}m\ [\![V]\!]$, $\mathcal{V}m\ [\![E_1]\!] \overset{*}{\rightarrow} \mathcal{V}m\ [\![W]\!]$. Further, by Lemma 19, we know that $\mathcal{V}m\ [\![V]\!] \equiv \mathbf{grab}\ V'$ and $\mathcal{V}m\ [\![W]\!] \equiv \mathbf{grab}\ (\lambda_s x.\mathcal{V}m\ [\![F]\!])$ thus $\mathcal{V}m\ [\![E]\!] \equiv \mathbf{push}_s\ \varepsilon \circ \mathcal{V}m\ [\![E_2]\!] \circ \mathcal{V}m\ [\![E_1]\!] \overset{*}{\rightarrow} \mathcal{V}m\ [\![F]\!]\ [V'/x] \equiv \mathcal{V}m\ [\![F[V/x]]\!]$ (Lemma 17) and from $\mathcal{V}m\ [\![E]\!] \overset{*}{\rightarrow} \mathcal{V}m\ [\![N]\!]$ we conclude that $\mathcal{V}m\ [\![F[V/x]]\!] \overset{*}{\rightarrow} \mathcal{V}m\ [\![N]\!]$. So, by induction hypothesis $E_1 \xrightarrow[cbv]{} \lambda x.F$, $E_2 \xrightarrow[cbv]{} V$ and $F[V/x] \xrightarrow[cbv]{} N$ hence $E \xrightarrow[cbv]{} N$. ❐

H Proof of Property 12

In order to prove $\mathbf{push}_e() \circ \mathcal{A}s\ [\![E]\!]\ () = E$, we prove by induction the more general property:

$$\mathbf{push}_e\ \rho \circ \mathcal{A}s\ [\![E]\!]\ \rho = E \qquad \text{with } \rho = (\ldots((),x_n)\ldots,x_0) \text{ and } FV(E) = \{x_0,\ldots,x_n\}$$

where $FV(E)$ is the set of free variables of E.

We will make use of the fact that, if $FV(E) \subseteq \rho$ then $\mathcal{A}s\ [\![E]\!]\ \rho$ is closed (easy to check).

- $E \equiv E_1 \circ E_2$

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

  $= \mathbf{push}_e\ \rho \circ (\mathbf{push}_e\ \rho \circ \mathcal{A}s\ [\![E_1]\!]\ \rho) \circ \lambda_s x.\lambda_e e.\mathbf{push}_s\ x \circ \mathbf{push}_e\ e \circ \mathcal{A}s\ [\![E_2]\!]\ \rho \qquad (\beta_s),(\beta_e)$

  $= \mathbf{push}_e\ \rho \circ E_1 \circ \lambda_s x.\lambda_e e.\mathbf{push}_s\ x \circ \mathbf{push}_e\ e \circ \mathcal{A}s\ [\![E_2]\!]\ \rho$ by induction hypothesis

  $= E_1 \circ \mathbf{push}_e\ \rho \circ \mathcal{A}s\ [\![E_2]\!]\ \rho$ Λ_e version of (L3),(β_e),(η_s)

  $= E_1 \circ E_2$ by induction hypothesis

- $E \equiv \mathbf{push}_s\ V$

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

  $= \mathbf{push}_s\ (\mathbf{push}_e\ \rho \circ \mathcal{A}s\ [\![V]\!]\ \rho\ ])$ **mkclos** def,(β_s),(β_e)

  $= \mathbf{push}_s\ V$ by induction hypothesis

- $E \equiv \lambda_s x.F$

$\mathbf{push}_e\ \rho \circ \mathcal{A}_s\ [\![\lambda_s x.F]\!]\ \rho = \mathbf{push}_e\ \rho \circ \mathbf{bind} \circ \mathcal{A}_s\ [\![F]\!]\ (\rho,x)$

$= \mathbf{push}_e\ \rho \circ \lambda_e e.\lambda_s y.\mathbf{push}_e\ (e,y) \circ \mathcal{A}_s\ [\![F]\!]\ (\rho,x)$ **bind** def.

$= \mathbf{push}_e\ \rho \circ \lambda_e e.\lambda_s x.\mathbf{push}_e\ (e,x) \circ \mathcal{A}_s\ [\![F]\!]\ (\rho,x)$ $\mathcal{A}_s\ [\![F]\!]\ (\rho,x)$ closed and α_s

$= \lambda_s x.\mathbf{push}_e\ (\rho,x) \circ \mathcal{A}_s\ [\![F]\!]\ (\rho,x)$ (β_e)

$= \lambda_s x.F$ by induction hypothesis

- $E \equiv x_i$

$\mathbf{push}_s\ \rho \circ \mathcal{A}_s\ [\![x_i]\!]\ \rho = \mathbf{push}_e\ \rho \circ \mathbf{access}_i \circ \mathbf{appclos}$ with $\rho = (\ldots((),x_n)\ldots,x_0)$

$= \mathbf{push}_s\ x_i \circ \mathbf{appclos}$ $\mathbf{access}_i$ def.,$(\beta_s),(\beta_e)$

$= x_i$ **appclos** def., (β_s) ❐

I Proof of Property 14

First, we introduce a Λ_e type system (with $i \equiv s,e$):

$$\frac{\Gamma \vdash E : \sigma}{\Gamma \vdash \mathbf{push}_i\ E : \mathsf{R}_i\sigma} \qquad \frac{\Gamma \cup \{x{:}\sigma\} \vdash E : \tau}{\Gamma \vdash \lambda_i x.E : \sigma \rightarrow_i \tau} \qquad \frac{\Gamma \vdash E_1 : \mathsf{R}_i\sigma \quad \Gamma \vdash E_2 : \sigma \rightarrow_i \tau}{\Gamma \vdash E_1 \circ E_2 : \tau}$$

Figure 28 Λ_e **typed subset (**Λ_e^σ **)**

The subject reduction property holds for Λ_e and can be shown exactly as Property 2. As a corollary, the reduction to normal form of a Λ_e^σ -expression $E_1 \circ E_2$ is of the form $E_1 \overset{*}{\rightarrow} \mathbf{push}_i\ V$, $E_2 \overset{*}{\rightarrow} \lambda_i x.F$ and $F[V/x] \overset{*}{\rightarrow} N$.

A generic substitution lemma for Λ_e can be defined and proved in the same way than Lemma 17. The instance of this lemma for $\mathcal{S}$ is

Lemma 21 $\forall F,V \in \Lambda_e,\ \mathcal{S}\ [\![F]\!]\ [\mathcal{S}\ [\![V]\!]/x] \equiv \mathcal{S}\ [\![F[V/x]]\!]$

We now can prove Property 14, by induction on the reduction tree. In the following $i \equiv s,e$:

- E is already in normal form, $E \equiv \mathbf{push}_i\ V$ or $E \equiv \lambda_i x.F$, then the property is true.

- E is reducible, $E \equiv E_1 \circ E_2$, and E_1: $\mathsf{R}_i\sigma$ and E_2: $\sigma \rightarrow_i \tau$ then $E_1 \overset{*}{\rightarrow} \mathbf{push}_i\ V$, $E_2 \overset{*}{\rightarrow} \lambda_i x.F$ and $F[V/x] \overset{*}{\rightarrow} N$. So

$\mathcal{S}\ [\![E_1 \circ E_2]\!]$

$\overset{*}{\rightarrow} \mathbf{push}_k\ (\mathcal{S}\ [\![E_2]\!]\ ) \circ \mathcal{S}\ [\![\mathbf{push}_i\ V]\!]$ by induction hypothesis

$\equiv \mathbf{push}_k\ (\mathcal{S}\ [\![E_2]\!]\ ) \circ \mathbf{push}_i\ (\mathcal{S}\ [\![V]\!]\ ) \circ \lambda_i y.\lambda_k k.\mathbf{push}_i\ y \circ k$

$\rightarrow$ $\mathbf{push}_i\ (\mathcal{S}\ [\![V]\!]\ )\ \mathrm{o}\ \mathcal{S}\ [\![E_2]\!]$ $(\beta_k),(\beta_k)$

$\overset{*}{\rightarrow}$ $\mathbf{push}_i\ (\mathcal{S}\ [\![V]\!]\ )\ \mathrm{o}\ \mathcal{S}\ [\![\lambda_i x.F]\!] \equiv \mathbf{push}_i\ (\mathcal{S}\ [\![V]\!]\ )\ \mathrm{o}\ \lambda_i x.\mathcal{S}\ [\![F]\!]\ )$ by induction hypothesis

$\rightarrow$ $\mathcal{S}\ [\![F]\!]\ [\mathcal{S}\ [\![V]\!]\ /x] \equiv \mathcal{S}\ [\![F[V/x]]\!]$ (β_i),(Lemma 21)

$\overset{*}{\rightarrow}$ $\mathcal{S}\ [\![N]\!]$ by induction hypothesis ❐

J Proof of Property 15

The corresponding of Lemma 17 for $\mathcal{SI}$ is:

Lemma 22 $\forall F,V \in \Lambda_s$, $\mathcal{SI}\ [\![F]\!]\ [\mathcal{SI}\ [\![V]\!]\ /x] \equiv \mathcal{SI}\ [\![F[V/x]]\!]$

The proof of Property 15 is by induction on the reduction tree.

- E is already in normal form, $E \equiv \mathbf{push}_s\ V$ or $E \equiv \lambda_s x.F$, then the property is true.
- E is reducible, $E \equiv E_1\ \mathrm{o}\ E_2$, since E is well-typed $E_1 \overset{*}{\rightarrow} \mathbf{push}_s\ V$, $E_2 \overset{*}{\rightarrow} \lambda_s x.F$ and $F[V/x] \overset{*}{\rightarrow} N$.

$\mathbf{push}_s\ K\ \mathrm{o}\ \mathcal{SI}\ [\![E_1\ \mathrm{o}\ E_2]\!] \equiv \mathbf{push}_s\ K\ \mathrm{o}\ \lambda_s k.\mathbf{push}_s\ (\mathbf{push}_s\ k\ \mathrm{o}\ \mathcal{SI}\ [\![E_2]\!]\ )\ \mathrm{o}\ \mathcal{SI}\ [\![E_1]\!]$

$\rightarrow$ $\mathbf{push}_s\ (\mathbf{push}_s\ K\ \mathrm{o}\ \mathcal{SI}\ [\![E_2]\!]\ )\ \mathrm{o}\ \mathcal{SI}\ [\![E_1]\!]$ (β_s)

$\overset{*}{\rightarrow}$ $\mathbf{push}_s\ (\mathbf{push}_s\ K\ \mathrm{o}\ \mathcal{SI}\ [\![E_2]\!]\ )\ \mathrm{o}\ \mathcal{SI}\ [\![\mathbf{push}_s\ V]\!]$ by induction hypothesis

$\equiv \mathbf{push}_s\ (\mathbf{push}_s\ K\ \mathrm{o}\ \mathcal{SI}\ [\![E_2]\!]\ )\ \mathrm{o}\ \lambda_s k.\mathbf{push}_s\ (\mathcal{SI}\ [\![V]\!]\ )\ \mathrm{o}\ k$

$\rightarrow$ $\mathbf{push}_s\ (\mathcal{SI}\ [\![V]\!]\ )\ \mathrm{o}\ \mathbf{push}_s\ K\ \mathrm{o}\ \mathcal{SI}\ [\![E_2]\!]$ (β_s)

$\overset{*}{\rightarrow}$ $\mathbf{push}_s\ (\mathcal{SI}\ [\![V]\!]\ )\ \mathrm{o}\ \mathbf{push}_s\ K\ \mathrm{o}\ \mathcal{SI}\ [\![\lambda_s x.F]\!]$ by induction hypothesis

$\equiv \mathbf{push}_s\ (\mathcal{SI}\ [\![V]\!]\ )\ \mathrm{o}\ \mathbf{push}_s\ K\ \mathrm{o}\ \lambda_s k.\lambda_s x.\mathbf{push}_s\ k\ \mathrm{o}\ \mathcal{SI}\ [\![F]\!]$

$\rightarrow$ $\mathbf{push}_s\ (\mathcal{SI}\ [\![V]\!]\ )\ \mathrm{o}\ \lambda_s x.\mathbf{push}_s\ K\ \mathrm{o}\ \mathcal{SI}\ [\![F]\!]$ (β_s)

$\rightarrow$ $\mathbf{push}_s\ K\ \mathrm{o}\ \mathcal{SI}\ [\![F]\!]\ [\mathcal{SI}\ [\![V]\!]\ /x] \equiv \mathbf{push}_s\ K\ \mathrm{o}\ \mathcal{SI}\ [\![F[V/x]]\!]$ (β_s), (Lemma 22)

$\overset{*}{\rightarrow}$ $\mathbf{push}_s\ K\ \mathrm{o}\ \mathcal{SI}\ [\![N]\!]$ by induction hypothesis ❐

Unité de recherche INRIA Lorraine, technopôle de Nancy-Brabois, 615 rue du jardin botanique, BP 101, 54600 VILLERS-LÈS-NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, domaine de Voluceau, Rocquencourt, BP 105, LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105 LE CHESNAY Cedex (France)

ISSN 0249-6399