A Systematic Study of Functional Language Implementations

RÉMI DOUENCE and PASCAL FRADET
INRIA/IRISA

We introduce a unified framework to describe, relate, compare, and classify functional language implementations. The compilation process is expressed as a succession of program transformations in the common framework. At each step, different transformations model fundamental choices. 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. This approach also paves the way to formal comparisons by making it possible to estimate the complexity of individual transformations or compositions of them. Our study aims at covering the whole known design space of sequential functional language implementations. In particular, we consider call-by-value, call-by-name, and call-by-need reduction strategies as well as environment- and graph-based implementations. We describe for each compilation step the diverse alternatives as program transformations. In some cases, we illustrate how to compare or relate compilation techniques, express global optimizations, or hybrid implementations. We also provide a classification of well-known abstract machines.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Functional Programming; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures*; D.3.4 [**Programming Languages**]: Processors—*code generation*; *compilers*; *optimization*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Abstract machines, compilers, combinators, functional programming, program transformation

1. INTRODUCTION

One of the most studied issues concerning functional languages is their implementation. Since Landin's [1964] seminal proposal, 30 years ago, a plethora of new abstract machines and compilation techniques have been proposed. The list of existing abstract machines includes the SECD [Landin 1964], the Cam [Cousineau et al. 1987], the CMCM [Lins 1987], the Tim

Authors' address: INRIA/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France; email: {douence; fradet}@irisa.fr.

[Fairbairn and Wray 1987], the Zam [Leroy 1990], the G-machine [Johnson 1987], and the Krivine-machine [Crégut 1991]. Other implementations are not described via an abstract machine but as a collection of transformations or compilation techniques such as compilers based on continuation passing style (CPS) [Appel 1992; Fradet and Le Métayer 1991; Kranz et al. 1986; Steele 1978]. Furthermore, numerous papers present optimizations often adapted to a specific abstract machine or a specific approach [Argo 1989; Burn et al. 1988; Joy et al. 1985]. Looking at these myriad 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 [Lins et al. 1992; Peyton Jones and Lester 1992] but, to the best of our knowledge, no global approach to study implementations.

The goal of this article is to fill this gap by introducing a unified framework to describe, relate, compare, and classify functional language implementations. Our approach is to express the whole compilation process as a succession of program transformations. The common framework considered here is a hierarchy of intermediate languages, all of which are subsets of the lambda calculus. Our description of an implementation consists of a series of transformations $\Lambda \xrightarrow{\mathcal{T}1} \Lambda_1 \xrightarrow{\mathcal{T}2} \ldots \xrightarrow{\mathcal{T}n} \Lambda_n$, each one compiling a particular task by mapping an expression from one intermediate language into another. The last language Λ_n consists of functional expressions that can be seen as assembly 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 also has 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. Our approach also paves the way to formal comparisons by estimating the complexity of individual transformations or compositions of them.

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 using this simple language. The two steps that cause the greatest impact on the compiler are the implementation of the reduction strategy (searching for the next redex) and the environment management (compilation of the β-reduction). Other steps include the implementation of control transfers (calls and returns), the implementation of closure sharing and update (implied by the call-by-need strategy), the representation of components such as the data stack or environments, and various optimizations.

In Section 2 we describe the framework used to model the compilation process. In Section 3 we present the alternatives to compile the reduction strategy (i.e., call-by-value and call-by-name). The compilation of control used by graph reducers is peculiar. A separate section (3.3) is dedicated to this point. Section 3 ends with a comparison of two compilation techniques of call-by-value and a study of the relationship between the compilation of control in the environment and graph-based models. Section 4 (resp., Section 5) describes the different options to compile the β-reduction (resp., the control transfers). Call-by-need is nothing but call-by-name with redex sharing and update, and in Section 6 we present how it can be expressed in our framework. Section 7 embodies our study in a taxonomy of classical functional implementations. In Section 8 we outline some extensions and applications of the framework. Section 9 is devoted to a review of related work, and Section 10 concludes by indicating directions for future research.

In order to alleviate the presentation, some more involved material such as proofs, variants of transformations, and other technical details have been kept out of the main text. We refer the motivated reader to the (electronically published) Appendix. A previous conference paper [Douence and Fradet 1995] concentrates on call-by-value and can be used as a short introduction to this work. Additional details can also be found in two companion technical reports [Douence and Fradet 1996a; 1996b] and a Ph.D. thesis [Douence 1996].

2. GENERAL FRAMEWORK

Each compilation step is represented by a transformation from an intermediate language to another one that is closer to machine code. In this article, the whole implementation process is described via a transformation sequence $\Lambda \xrightarrow{\mathcal{T}1} \Lambda_s \xrightarrow{\mathcal{T}2} \Lambda_e \xrightarrow{\mathcal{T}3} \Lambda_k \xrightarrow{\mathcal{T}4} \Lambda_h$ starting with Λ and involving four intermediate languages (very close to each other). This framework possesses several benefits:

—It has a *strong formal basis*. Each intermediate language can be seen either as a formal system with its own conversion rules or as a subset of the λ-calculus by defining its constructs as λ-expressions. The intermediate languages share many laws and properties, the most important being that every reduction strategy is normalizing. These features facilitate program transformations, correctness proofs, and comparisons.

—It is (relatively) *abstract*. Since we want to model implementations completely and precisely, the intermediate languages must come closer to an assembly language as we progress in the description. The framework nevertheless possesses many abstract features that do not lessen its precision. The combinators of the intermediate languages and their conversion rules allow a more abstract description of notions such as instructions, sequencing, stacks, etc., than an encoding as λ-expressions. As a consequence, the compilation of control is expressed more abstractly

$$
\begin{array}{ll}
\Lambda_s & E ::= x \mid E_1 \circ E_2 \mid \mathbf{push}_s\ E \mid \lambda_s x.E \\
\Lambda_e & E ::= x \mid E_1 \circ E_2 \mid \mathbf{push}_s\ E \mid \lambda_s x.E \mid \mathbf{push}_e\ E \mid \lambda_e x.E \\
\Lambda_k & E ::= x \mid E_1 \circ E_2 \mid \mathbf{push}_s\ E \mid \lambda_s x.E \mid \mathbf{push}_e\ E \mid \lambda_e x.E \\
 & \quad \mid \mathbf{push}_k\ E \mid \lambda_k x.E \\
\Lambda_h & E ::= x \mid E_1 \circ E_2 \mid \mathbf{push}_s\ E \mid \lambda_s x.E \mid \mathbf{push}_e\ E \mid \lambda_e x.E \\
 & \quad \mid \mathbf{push}_k\ E \mid \lambda_k x.E \mid \mathbf{push}_h\ E \mid \lambda_h x.E
\end{array}
$$

Fig. 1. The intermediate languages.

than using CPS expressions, and the implementation of components (e.g., data stack, environment stack) is a separate step.

—It is *modular*. Each transformation implements one compilation step and can be defined independently of the former steps. Transformations implementing different steps are freely composed to specify implementations. Transformations implementing the same step represent different choices and can be compared.

—It is *extendable*. New intermediate languages and transformations can be defined and inserted into the transformation sequence to model new compilation steps (e.g., register allocation).

2.1 Overview

The first step is the compilation of control which is described by transformations from Λ to Λ_s. The intermediate language Λ_s (Figure 1) is defined using the combinators $\circ$, $\mathbf{push}_s$, and a new form of λ-abstraction $\lambda_s x.E$. 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. The pair ($\mathbf{push}_s$, λ_s) specifies a component (noted s) storing intermediate results (e.g., a data stack). So, $\mathbf{push}_s$ and λ_s can be seen as "store" and "fetch" in s.

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 weak redex is no longer relevant: all weak redexes are needed. This is the key point to view transformations from Λ to Λ_s as compiling the evaluation strategy.

Transformations from Λ_s to Λ_e are used to compile the β-reduction. The language Λ_e excludes unrestricted uses of variables that are now needed only to define macrocombinators. The encoding of environment management is made possible using the new pair ($\mathbf{push}_e$, λ_e). They behave exactly as $\mathbf{push}_s$ and λ_s; they just act on a (at least conceptually) different component e (e.g., a stack of environments).

Transformations from Λ_e to Λ_k describe the compilation of control transfers. The language Λ_k makes calls and returns explicit. It introduces the pair ($\mathbf{push}_k$, λ_k) which specifies a component k storing return addresses.

The last transformations from Λ_k to Λ_h add a memory component in order to express closure sharing and updating. The language Λ_h introduces the pair ($\mathbf{push}_h$, λ_h) which specifies a global heap h. The expressions of this last language can be read as assembly code.

$$\begin{array}{ll} (\text{assoc}) & (E_1 \circ E_2) \circ E_3 = E_1 \circ (E_2 \circ E_3) \\ (\beta_i) & (\mathbf{push}_i\ F) \circ (\lambda_i x.E) = E[F/x] \\ (\eta_i) & \lambda_i x.(\mathbf{push}_i\ x \circ E) = E \quad \textit{if } x \textit{ does not occur free in } E \end{array}$$

Fig. 2. Conversion rules in Λ_i (for $i \in \{s,e,k,h\}$).

2.2 Conversion Rules

The substitution and the notion of free or bound variables are the same as in the λ-calculus. The basic combinators can be given different definitions (possible definitions are given in Section 2.5). We do not pick specific ones at this point; we simply impose the associativity of sequencing and that the combinators satisfy the equivalent of β- and η-conversions (Figure 2). We consider only reduction rules corresponding to the classical β-reduction:

$$(\mathbf{push}_i\ F) \circ (\lambda_i x.E) \rightarrowtriangle E[F/x]$$

As with all standard implementations, we are only interested in modeling weak reductions. In our framework, a weak redex is a redex that does not occur inside an expression of the form $\mathbf{push}_i E$ or $\lambda_i x.E$. Weak reduction does not reduce under the $\mathbf{push}_i$'s or λ_i's, and from here on we write "redex" (resp., reduction, normal form) for weak redex (resp., weak reduction, weak normal form).

The following example illustrates β_i-reduction (note that $\mathbf{push}_s\ F \circ \lambda_s z.G$ is not a (weak) redex of the global expression):

$$\mathbf{push}_e\ E \circ \mathbf{push}_s(\mathbf{push}_s\ F \circ \lambda_s z.G) \circ \lambda_s x.\lambda_e y.\mathbf{push}_s(\mathbf{push}_e\ y \circ x)$$

$$\rightarrowtriangle \mathbf{push}_e\ E \circ \lambda_e y.\mathbf{push}_s(\mathbf{push}_e\ y \circ \mathbf{push}_s\ F \circ \lambda_s z.G)$$

$$\rightarrowtriangle \mathbf{push}_s(\mathbf{push}_e\ E \circ \mathbf{push}_s\ F \circ \lambda_s z.G)$$

Any two redexes are clearly disjoint, and the β_i-reductions are left-linear; so the term-rewriting system is orthogonal, hence confluent [Klop 1992]. Alternatively, it is very easy to show that the relation $\rightarrowtriangle$ is strongly confluent, therefore confluent (see Appendix A). Furthermore, any redex is needed (a rewrite cannot suppress a redex) thus the following property:

PROPERTY 1. *All Λ_i-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

All the expressions of the intermediate languages can be given a meaning as λ-expressions (Section 2.5). Using conversion rules such as (assoc) the same expression can be represented differently. For example, one can write

$$\frac{}{\Gamma \cup \{x:\sigma\} \vdash x:\sigma} \qquad \frac{\Gamma \vdash E:\sigma}{\Gamma \vdash \mathbf{push}_i\ E : \mathsf{R}_i\sigma}$$

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

Fig. 3. Λ_i typed subset (Λ_i^σ) (for $i \in \{s,e,k,h\}$).

equivalently

$$\mathbf{push}_s\ E_1 \circ (\mathbf{push}_s\ E_2 \circ \lambda_s x.\lambda_s y.E_3)$$

or

$$(\mathbf{push}_s\ E_1 \circ \mathbf{push}_s\ E_2) \circ \lambda_s x.\lambda_s y.E_3.$$

This flexibility is very useful for transforming or reshaping the code. However, unrestricted transformations may lose information about the structure of the expression. Many laws and transformations (e.g., see laws (L2) and (L3) in Section 2.4 or transformation $\mathcal{H}c$ in Section 6.1) rely on the fact that a subexpression denotes a result (i.e., can be reduced to an expression of the form $\mathbf{push}_i\ E$) or a function (i.e., can be reduced to an expression of the form $\lambda_i x.E$). If we allow subexpressions such as ($\mathbf{push}_s\ E_1 \circ \mathbf{push}_s\ E_2$) that neither denote a result nor a function, fewer laws and transformations can be expressed. It is therefore convenient to restrict Λ_i using a type system (Figure 3).

The restrictions enforced by the type system are on how results and functions are combined in Λ_i. For example, the composition $E_1 \circ E_2$ is restricted so that E_1 denotes a result (i.e., has type $\mathsf{R}_i\sigma$, R_i being a type constructor) and E_2 denotes a function. The type system restricts the set of normal forms (which in general includes expressions such as $\mathbf{push}_i\ E_1 \circ \mathbf{push}_j\ E_2$), and we have the following natural facts (see Appendix B).

PROPERTY 2. *If a closed expression* $E:\mathsf{R}_i\sigma$ *has a normal form, then* $E \overset{*}{\rightarrow} \mathbf{push}_i\ V$. *If a closed expression* $E:\sigma \rightarrow_i \tau$ *has a normal form, then* $E \overset{*}{\rightarrow} \lambda_i x.F$.

So, the reduction of any well-typed expression $A \circ F$ either reaches an expression of the form $\mathbf{push}_i\ A' \circ \lambda_i x.F'$ or loops.

Our transformations implementing compilation steps will produce well-typed expressions denoting results, and during all the compilation processes, the compiled program will be well typed. Typing is used to maintain some structure in the expression and does not impose any restrictions on source λ-expressions (Appendix B). It should be regarded as a syntactic tool, not a semantic one. Ill-typed Λ_i-expressions have a meaning in terms of λ-expressions as well (see Section 2.5).

2.4 Laws

This framework possesses a number of algebraic laws that are useful to transform the functional code or to prove the correctness or equivalence of

program transformations such as

$$\text{If } x \text{ does not occur free in } F \quad (\lambda_i x.E) \circ F = \lambda_i x.(E \circ F). \tag{L1}$$

For all E_1: $\mathsf{R}_i\sigma$, if x does not occur free in E_2

$$E_1 \circ (\lambda_i x.(E_2 \circ E_3)) = E_2 \circ (E_1 \circ (\lambda_i x.E_3)). \tag{L2}$$

For all E_1: $\mathsf{R}_i\sigma$, E_2: $\mathsf{R}_j\tau$ and $x \not\equiv y$

$$E_1 \circ (E_2 \circ (\lambda_j x.\lambda_i y.E_3)) = E_2 \circ (E_1 \circ (\lambda_i y.\lambda_j x.E_3)). \tag{L3}$$

These rules permit code to be moved inside or outside function bodies or to invert the evaluation order of two intermediate results (which is correct because we consider only purely functional expressions (Appendix C)). To illustrate the conversion rules at work, let us prove the law (L1). Note that x does not occur free in $(\lambda_i x.E)$ nor, by hypothesis, in F and in the following:

$$\begin{aligned}
(\lambda_i x.E) \circ F &= \lambda_i x.\mathbf{push}_i\ x \circ ((\lambda_i x.E) \circ F) && (\eta_i)\\
&= \lambda_i x.((\mathbf{push}_i\ x \circ (\lambda_i x.E)) \circ F) && (\text{assoc})\\
&= \lambda_i x.(E[x/x] \circ F) && (\beta_i)\\
&= \lambda_i x.(E \circ F). && (\text{subst})
\end{aligned}$$

Even if using some rules or laws (e.g., (assoc) or (L1)) may lead to untyped programs, we still may use them as long as the final program is well typed. For example, a closed and well-typed expression

$$(\mathbf{push}_s\ V \circ (\lambda_s x.\mathbf{push}_s\ E)) \circ (\lambda_s y.F)$$

can be transformed using (assoc) and (L1) into the well-typed expression

$$\mathbf{push}_s\ V \circ \lambda_s x.(\mathbf{push}_s\ E \circ (\lambda_s y.F)).$$

To simplify the presentation, we often omit parentheses and write, for example, $\mathbf{push}_i\ E \circ \lambda_i x.F \circ G$ for $(\mathbf{push}_i\ E) \circ (\lambda_i x.(F \circ G))$. We also use syntactic sugar such as tuples $(x_1, \ldots, x_n)$ and simple pattern matching $\lambda_i(x_1, \ldots, x_n).E$.

2.5 Instantiation

The intermediate languages Λ_i are subsets 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_i), (η_i), and (assoc). Definitions can be chosen only after the last compilation step. This feature allows us to shift from the β_i-reduction in Λ_i to a

state-machine-like expression reduction. Moreover, it permits us to specify the implementation of components independently of the other steps. For example, we may eventually choose to implement the data component s and the environment component e either as a single stack or as two separate ones. In Section 7 we present an example of instantiation for the Cam.

In order to provide some intuition, we nevertheless give here some possible definitions in terms of standard λ-expressions. The most natural definition for the sequencing combinator is $\circ = \lambda abc.a\ (b\ c)$, i.e., $E_1 \circ E_2 = \lambda c.E_1\ (E_2\ c)$. The (fresh) variable c can be seen as a continuation and implements the sequencing.

The pairs of combinators (λ_i, $\mathbf{push}_i$) can be seen as encoding a component of an underlying abstract machine and their definitions as specifying the state transitions. A sequence of code such as $\mathbf{push}_i\ E_1 \circ \ldots \circ \mathbf{push}_i\ E_n \circ \ldots$ suggests that the underlying machine must possess a component i (such as a stack, list, tree, or vector) in order to store intermediate results. We can choose to keep the components separate or merge (some of) them.

Keeping all the components separate leads to the following possible definitions (c, s, e, k, h being fresh variables):

$$\mathbf{push}_s\ N = \lambda c.\lambda s.\lambda e.\lambda k.\lambda h.c(s,N)e\ k\ h$$

$$\lambda_s x.X = \lambda c.\lambda(s,x).\lambda e.\lambda k.\lambda h.X\ c\ s\ e\ k\ h$$

$$\mathbf{push}_e\ N = \lambda c.\lambda s.\lambda e.\lambda k.\lambda h.c\ s(e,N)k\ h$$

$$\lambda_e x.X = \lambda c.\lambda s.\lambda(e,x).\lambda k.\lambda h.X\ c\ s\ e\ k\ h$$

$$\mathbf{push}_k\ N = \lambda c.\lambda s.\lambda e.\lambda k.\lambda h.c\ s\ e(k,N)h$$

$$\lambda_k x.X = \lambda c.\lambda s.\lambda e.\lambda(k,x).\lambda h.X\ c\ s\ e\ k\ h$$

$$\mathbf{push}_h\ N = \lambda c.\lambda s.\lambda e.\lambda k.\lambda h.c\ s\ e\ k(h,N)$$

$$\lambda_h x.X = \lambda c.\lambda s.\lambda e.\lambda k.\lambda(h,x).X\ c\ s\ e\ k\ h$$

Then, the reduction (using classical β-reduction and normal order) of our expressions can be seen as state transitions of an abstract machine with five components (code, data stack, environment stack, control stack, heap), for example,

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

$$\mathbf{push}_h\ N\ C\ S\ E\ K\ H \rightarrow C\ S\ E\ K(H,N).$$

According to the definition of $\circ$, the rewriting rule for sequencing is

$$(E_1 \circ E_2)C\ S\ E\ K\ H \rightarrow E_1(E_2\ C)S\ E\ K\ H.$$

Note that C plays the role of a continuation. A code can be seen as a state transformer of type

$$(data \to env \to control \to heap \to Ans) \to data \to env \to control \to heap \to Ans.$$

To be reduced, a code is applied to an initial continuation (e.g., **id**), initial (empty) data, environment and control components, and an initial heap.

Keeping some components separate brings new properties such as

$$\mathbf{push}_i\ E \circ \mathbf{push}_j\ F = \mathbf{push}_j\ F \circ \mathbf{push}_i\ E \qquad \text{if } i \neq j,$$

allowing code motion and simplifications.

A second option is to merge all the components. The underlying abstract machine has only two components (the code and a data-environment-control-heap stack). Possible definitions are

$$\mathbf{push}_s\ N = \mathbf{push}_e\ N = \mathbf{push}_k\ N = \mathbf{push}_h\ N = \lambda c.\lambda z.c(z,N)$$

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

and the reduction of expressions is of the form $\mathbf{push}_i\ N\ C\ Z \to C\ (Z,N)$ for $i \in \{s,e,k,h\}$.

Let us point out that our use of the term "abstract machines" should not suggest a layer of interpretation. The abstraction only consists of the use of components and generic code. At the end of the compilation process, we get realistic assembly code, and the "abstract machines" resemble real machines.

3. COMPILATION OF CONTROL

We focus here on the compilation of the call-by-value and the call-by-name reduction strategies. Call-by-need is only a refinement of call-by-name involving redex sharing and update. It is described in Section 6. We first present the two main choices taken by environment-based implementations. Following Peyton Jones' [1992] terminology, these two options are named the *eval-apply* model (presented in Section 3.1) and the *push-enter* model (presented in Section 3.2). The graph-based implementations use an interpretative implementation of the reduction strategy. They are presented in Section 3.3. Finally, we compare the eval-apply and the push-enter schemes for call-by-value, and we relate environment machines and graph reducers.

3.1 The Eval-Apply Model

In the eval-apply model, a λ-abstraction is considered as a result, and the application of a function to its argument is an explicit operation. This

$$\gamma_a : \Lambda \to \Lambda_s$$
$$\gamma_a[\![x]\!] = \mathbf{push}_s\ x$$
$$\gamma_a[\![\lambda x.E]\!] = \mathbf{push}_s(\lambda_s x.\ \gamma_a[\![E]\!])$$
$$\gamma_a[\![E_1\ E_2]\!] = \gamma_a[\![E_2]\!] \circ \gamma_a[\![E_1]\!] \circ \mathbf{app} \qquad \text{with } \mathbf{app} = \lambda_s f.f$$

Fig. 4. Compilation of right-to-left call-by-value in the eval-apply model (γ_a).

model is the most natural choice to implement call-by-value where functions can be evaluated as arguments.

3.1.1 *Call-by-Value.* 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. Normal forms denote results; so λ-abstractions and variables (which, in strict languages, are always bound to normal forms) are transformed into results (i.e., $\mathbf{push}_s\ E$). The compilation of right-to-left call-by-value is formalized by the transformation γ_a in Figure 4.

This compilation choice is taken by the SECD machine [Landin 1964] and the Tabac compiler [Fradet and Le Métayer 1991]. The rules can be explained intuitively by reading "return the value" for $\mathbf{push}_s$, "evaluate" for γ_a, "then" for $\circ$, and "apply" for **app**. Although environment management is tackled only in Section 4, it is also useful to keep in mind that a Λ_s-expression returning a function (such as $\mathbf{push}_s\ (\lambda_s x.E)$) will involve building a closure (i.e., a data structure containing the function and an environment recording the values of its free variables).

Strictly speaking, γ_a does not enforce a right-to-left evaluation ($\gamma_a[\![E_1]\!]$ could be reduced before $\gamma_a[\![E_2]\!]$). However, after instantiation, the normal order of reductions will enforce the sequencing nature of "$\circ$." It is easy to check that γ_a produces well-typed expressions of result type $\mathsf{R}_s\sigma$ (see Appendix D).

The correctness of γ_a is stated by Property 3 which establishes that the reduction ($\overset{*}{\rightarrow}$) of transformed programs simulates the call-by-value reduction ($\underset{cbv}{\longrightarrow}$) of source λ-expressions (Appendix E). As it is standard, we consider that the source program (i.e., the global expression) is a closed Λ-expression.

PROPERTY 3. *For all closed Λ-expressions E, $E \underset{cbv}{\longrightarrow} V$ if and only if $\gamma_a[\![E]\!] \overset{*}{\rightarrow} \gamma_a[\![V]\!]$.*

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

$$\mathbf{push}_s\ E \circ \mathbf{app} = E \qquad (\mathbf{push}_s\ E \circ \lambda_s f.f =_{\beta_s} f[E/f] = E) \qquad \text{(L4)}$$

Example. Let $E \equiv (\lambda x.x)((\lambda y.y)(\lambda z.z))$; after simplifications, we get

$$\gamma_a[\![\mathrm{E}]\!] \equiv \mathbf{push}_s(\lambda_s z.\mathbf{push}_s\ z) \circ (\lambda_s y.\mathbf{push}_s\ y) \circ (\lambda_s x.\mathbf{push}_s\ x)$$
$$\rightarrow \mathbf{push}_s(\lambda_s z.\mathbf{push}_s\ z) \circ (\lambda_s x.\mathbf{push}_s\ x)$$
$$\rightarrow \mathbf{push}_s(\lambda_s z.\mathbf{push}_s\ z) \equiv \gamma_a[\![\lambda z.z]\!].$$

$$\mathcal{N}a: \Lambda \rightarrow \Lambda_s$$
$$\mathcal{N}a[\![x]\!] = x$$
$$\mathcal{N}a[\![\lambda x.E]\!] = \mathbf{push}_s(\lambda_s x.\mathcal{N}a[\![E]\!])$$
$$\mathcal{N}a[\![E_1\ E_2]\!] = \mathbf{push}_s(\mathcal{N}a[\![E_2]\!]) \circ \mathcal{N}a[\![E_1]\!] \circ \mathbf{app} \qquad \text{with } \mathbf{app} = \lambda_s f.f$$

Fig. 5. Compilation of call-by-name in the eval-apply model ($\mathcal{N}a$).

The source expression has two redexes $(\lambda x.x)((\lambda y.y)(\lambda z.z))$ and $(\lambda y.y)(\lambda z.z)$, but only the latter can be chosen by a call-by-value strategy. In contrast, $\mathcal{V}a[\![E]\!]$ has only the compiled version of $(\lambda y.y)(\lambda z.z)$ as redex. The illicit (in call-by-value) reduction $E \rightarrow (\lambda y.y)(\lambda z.z)$ cannot occur within $\mathcal{V}a[\![E]\!]$. This illustrates the fact that the reduction strategy has been compiled and that the choice of redex in Λ_s is not semantically relevant.

The law (L4) is central in the implementation of *uncurrying* (e.g., see Appel [1992]). To illustrate a simple case of uncurrying, let us take the 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

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

Using (L4), (assoc), and (L1) this expression can be simplified into

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

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 $\lambda x_1 \ldots \lambda x_n.E_0$. An important point to note is that, in Λ_s, $\lambda_s x_1 \ldots \lambda_s x_n.E$ always denotes an n-ary function, that is to say a function that will be applied to at least n arguments (otherwise there would be $\mathbf{push}_s$'s between the λ_s's).

There exist several variants of $\mathcal{V}a$ such as $\mathcal{V}a_L$ (used by the Cam) which implements a left-to-right call-by-value or $\mathcal{V}a_f$ (used by the SML-NJ compiler) which does not assume a data stack and disallows several pushes in a row (Appendix F).

3.1.2 *Call-by-Name.* For call-by-name in the eval-apply model, applications $E_1\ E_2$ are compiled by returning E_2, evaluating E_1, and finally applying the evaluated function to the unevaluated argument. This choice is implemented by the call-by-need version of the Tabac compiler [Fradet and Le Métayer 1991], and it is described by the transformation $\mathcal{N}a$ in Figure 5. The correctness of $\mathcal{N}a$ is stated by Property 4 which establishes that the reduction of transformed expressions ($\overset{*}{\blacktriangleright}$) simulates the call-by-name reduction ($\underset{cbn}{\longrightarrow}$) of source λ-expressions.

PROPERTY 4. *For all closed* Λ*-expressions* E, $E \xrightarrow[cbn]{} V$ *if and only if* $\mathcal{N}a[\![E]\!] \overset{*}{\blacktriangleright} \mathcal{N}a[\![V]\!]$.

Example. Let $E \equiv (\lambda x.x)((\lambda y.y)(\lambda z.z))$; after simplifications, we get

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

$$\blacktriangleright \mathbf{push}_s(\mathbf{push}_s(\lambda_s z.z)) \circ \lambda_s y.y$$

$$\blacktriangleright \mathbf{push}_s(\lambda_s z.z) \equiv \mathcal{N}a[\![\lambda z.z]\!].$$

The illicit (in call-by-name) reduction $E \rightarrow (\lambda x.x)(\lambda z.z)$ cannot occur within $\mathcal{N}a[\![E]\!]$.

Like $\mathcal{Y}a$, the transformation $\mathcal{N}a$ has a variant that does not assume a data stack (i.e., disallows several pushes in a row) (Appendix G).

3.2 The Push-Enter Model

In the eval-apply model, the straightforward compilation of a function expecting n arguments produces a code building n closures. In practice, much of this overhead can be removed by uncurrying, but this optimization is not always possible for functions passed as arguments. The main motivation of the push-enter model is to avoid useless closure buildings. In the push-enter model, unevaluated functions are applied right away, and application is an implicit operation.

3.2.1 *Call-by-Value.* Instead of evaluating the function and its argument and then applying the results as in the eval-apply model, another solution is to evaluate the argument and to apply the unevaluated function right away. With call-by-value, a function can also be evaluated as an argument. 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 performing dynamic tests. It is implemented in Zinc [Leroy 1990].

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

$$\mathbf{push}_s\ \varepsilon \circ \mathbf{grab}_s\ E \blacktriangleright \mathbf{push}_s\ E;$$

that is, a mark is present; the function E is returned; and

$$\mathbf{push}_s\ V \circ \mathbf{grab}_s\ E \blacktriangleright \mathbf{push}_s\ V \circ E \quad (V \not\equiv \varepsilon);$$

that is, no mark is present, and the function E is applied to its argument V.

$$\mathcal{Y}_m : \Lambda \rightarrow \Lambda_s$$
$$\mathcal{Y}_m[\![x]\!] = \mathbf{grab}_s\ x$$
$$\mathcal{Y}_m[\![\lambda x.E]\!] = \mathbf{grab}_s(\lambda_s x.\ \mathcal{Y}_m[\![E]\!])$$
$$\mathcal{Y}_m[\![E_1\ E_2]\!] = \mathbf{push}_s\ \varepsilon \circ \mathcal{Y}_m[\![E_2]\!] \circ \mathcal{Y}_m[\![E_1]\!]$$

Fig. 6. Compilation of right-to-left call-by-value in the push-enter model ($\mathcal{Y}_m$).

The combinator $\mathbf{grab}_s$ and the mark ε can be defined in Λ_s (Appendix H). In practice, $\mathbf{grab}_s$ is implemented using a conditional testing the presence of a mark. The transformation for right-to-left call-by-value is described in Figure 6.

The correctness of $\mathcal{Y}_m$ is stated by Property 5.

PROPERTY 5. *For all closed Λ-expressions E, $E \xrightarrow[cbv]{} V$ if and only if $\mathcal{Y}_m[\![E]\!] \overset{*}{\rightarrow} \mathcal{Y}_m[\![V]\!]$.*

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

$$\mathcal{Y}_m[\![E]\!] \equiv \mathbf{push}_s\ \varepsilon \circ \mathbf{push}_s(\lambda_s z.\mathbf{grab}_s\ z) \circ (\lambda_s y.\mathbf{grab}_s\ y) \circ (\lambda_s x.\mathbf{grab}_s\ x)$$
$$\rightarrow \mathbf{push}_s\ \varepsilon \circ \mathbf{grab}_s(\lambda_s z.\mathbf{grab}_s\ z) \circ (\lambda_s x.\mathbf{grab}_s\ x)$$
$$\rightarrow \mathbf{push}_s(\lambda_s z.\mathbf{grab}_s\ z) \circ (\lambda_s x.\mathbf{grab}_s\ x)$$
$$\rightarrow \mathbf{grab}_s(\lambda_s z.\mathbf{grab}_s\ z) \equiv \mathcal{Y}_m[\![\lambda z.z]\!].$$

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 $\mathcal{Y}_m$ is subject to the same kind of optimizations as $\mathcal{Y}_a$. Uncurrying and related optimizations can be expressed based on the reduction rules of $\mathbf{grab}_s$ and (L2).

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, other closely related transformations using marks exist (Appendix I).

3.2.2 *Call-by-Name.* Contrary to call-by-value, the most natural choice to implement call-by-name is the push-enter model. In call-by-name, functions are evaluated only when applied to an argument. Functions do not have to be considered as results. This option is taken by Tim [Fairbairn and Wray 1987], the Krivine machine [Crégut 1991], and graph-based implementations (see Section 3.3.2). The transformation $\mathcal{N}_m$ formalizes this choice; it is described in Figure 7.

Variables are bound to arguments that must be evaluated when accessed. Functions are not returned as results but assume that their argument is present. Applications are transformed by returning the unevaluated argument to the function. The correctness of $\mathcal{N}_m$ is stated by Property 6.

$$\mathcal{N}_m : \Lambda \rightarrow \Lambda_s$$
$$\mathcal{N}_m[\![x]\!] = x$$
$$\mathcal{N}_m[\![\lambda x.E]\!] = \lambda_s x.\mathcal{N}_m[\![E]\!]$$
$$\mathcal{N}_m[\![E_1\ E_2]\!] = \mathbf{push}_s(\mathcal{N}_m[\![E_2]\!]) \circ \mathcal{N}_m[\![E_1]\!]$$

Fig. 7. Compilation of call-by-name in the push-enter model ($\mathcal{N}_m$).

PROPERTY 6. *For all closed Λ-expressions E, $E \xrightarrow[cbn]{} V$ if and only if* $\mathcal{N}_m[\![E]\!] \overset{*}{\rightarrow} \mathcal{N}_m[\![V]\!]$.

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

$$\mathcal{N}_m[\![E]\!] \equiv \mathbf{push}_s(\mathbf{push}_s(\lambda_s z.z) \circ \lambda_s y.y) \circ \lambda_s x.x$$
$$\rightarrow \mathbf{push}_s(\lambda_s z.z) \circ \lambda_s y.y$$
$$\rightarrow \lambda_s z.z \equiv \mathcal{N}_m[\![\lambda z.z]\!].$$

Arguably, $\mathcal{N}_m$ is the simplest way to compile call-by-name. However, it makes the compilation of call-by-need problematic. After the evaluation of an unevaluated expression bound to a variable (i.e., a closure), a call-by-need implementation updates it by its normal form. Contrary to $\mathcal{N}_a$, $\mathcal{N}_m$ makes it impossible to distinguish results of closures (which have to be updated) from regular functions (which are applied right away). This problem is solved, as in $\mathcal{V}_m$, with the help of marks. We come back to this issue in Section 6.

Transformations from Λ to Λ_s share the goal of compiling control with CPS transformations [Fischer 1972; Plotkin 1975]. Actually, with a properly chosen instantiation of the combinators, the transformation $\mathcal{V}_{a_f}$ is nothing but Fischer's [1972] CPS transformation (Appendix J). As for CPS-expressions [Danvy 1992], it is also possible to design an inverse transformation mapping Λ_s-expressions back to Λ-expressions (Appendix K).

3.3 Graph Reduction

Graph-based implementations manipulate a graph representation of the source λ-expression. The reduction consists of rewriting the graph more or less interpretively. One of the motivations of this approach is to elegantly represent sharing which is ubiquitous in call-by-need implementations. So, even if call-by-value can be envisaged, well-known graph-based implementations only consider call-by-need. In the following, we focus on the push-enter model for call-by-name which is largely adopted by existing graph reducers. Its refinement into call-by-need is presented in Section 6.2.2.

3.3.1 *Graph Building.* As before, the compilation of control is expressed by transformations from Λ to Λ_s. However, this step is now divided in two parts: the graph construction, then its reduction via an interpreter. The transformation $\mathcal{G}$ (Figure 8) produces an expression that builds a graph

$$\mathcal{G}: \Lambda \rightarrow \Lambda_s$$
$$\mathcal{G}[\![x]\!] = \mathbf{push}_s\ x \circ \mathbf{mkVar}_s$$
$$\mathcal{G}[\![\lambda x.E]\!] = \mathbf{push}_s(\lambda_s x.\mathcal{G}[\![E]\!]) \circ \mathbf{mkFun}_s$$
$$\mathcal{G}[\![E_1\ E_2]\!] = \mathcal{G}[\![E_2]\!] \circ \mathcal{G}[\![E_1]\!] \circ \mathbf{mkApp}_s$$

Fig. 8. Generic graph-building code ($\mathcal{G}$).

(for now, only a tree) when reduced. The three new combinators $\mathbf{mkVar}_s$, $\mathbf{mkFun}_s$, and $\mathbf{mkApp}_s$ take their arguments from the s component and return graph nodes (resp., variable, function, and application nodes) on s. The following condition formalizes the fact that the reduction of $\mathcal{G}[\![E]\!]$ is just the graph construction that terminates and yields a result in the s component.

($\mathcal{C}ond\mathcal{G}$) For all Λ-expressions E, $\mathcal{G}[\![E]\!] \overset{*}{\blacktriangleright} \mathbf{push}_s\ V$.

The graph is scanned and reduced using a small interpreter denoted by the combinator $\mathbf{unwind}_s$. After the compilation of control, the global expression is of the form $\mathcal{G}[\![\mathrm{E}]\!] \circ \mathbf{unwind}_s$. This transformation is common to all the graph reduction schemes we describe. The push-enter or eval-apply models of the compilation of call-by-value or call-by-name can be specified simply by defining the interactions of $\mathbf{unwind}_s$, with the three graph builders $\mathbf{mkVar}_s$, $\mathbf{mkFun}_s$, and $\mathbf{mkApp}_s$.

3.3.2 *Call-by-Name: The Push-Enter Model.* This option is defined by the three following conditions:

($\mathcal{GN}m1$) $(E \circ \mathbf{mkVar}_s) \circ \mathbf{unwind}_s = E \circ \mathbf{unwind}_s$

($\mathcal{GN}m2$) $V \circ (\mathbf{push}_s\ F \circ \mathbf{mkFun}_s) \circ \mathbf{unwind}_s = (V \circ F) \circ \mathbf{unwind}_s$

($\mathcal{GN}m3$) $(E_2 \circ E_1 \circ \mathbf{mkApp}_s) \circ \mathbf{unwind}_s = E_2 \circ E_1 \circ \mathbf{unwind}_s$.

These conditions can be explained intuitively as follows:

—($\mathcal{GN}m1$) The reduction of a variable node amounts to reducing the graph that has been bound to the variable. The combinator $\mathbf{mkVar}_s$ may seem useless, since it is bypassed by $\mathbf{unwind}_s$. However, when call-by-need is considered, $\mathbf{mkVar}_s$ is needed to implement updating without losing sharing properties. As the combinator **I** in Turner [1979a], it represents indirection nodes.

—($\mathcal{GN}m2$) The reduction of a function node amounts to applying the function to its argument and to reducing the resulting graph. This rule makes the push-enter model clear. The reduction of the function node does not return the function F as a result, but immediately applies it.

—($\mathcal{GN}m3$) The reduction of an application node amounts to storing the argument graph and to reducing the function graph.

$$\begin{aligned}\mathbf{mkVar}_s &= \lambda_s x.\mathbf{push}_s\ x\\ \mathbf{mkFun}_s &= \lambda_s f.\mathbf{push}_s(\lambda_s a.(\mathbf{push}_s\ a \circ f) \circ \mathbf{unwind}_s)\\ \mathbf{mkApp}_s &= \lambda_s x_1.\lambda_s x_2.\mathbf{push}_s(\mathbf{push}_s\ x_2 \circ x_1)\\ \mathbf{unwind}_s &= \mathbf{app} = \lambda_s x.x\end{aligned}$$

Fig. 9. Instantiation of graph combinators according to $\mathcal{GN}m$ (option node-as-code).

Figure 9 presents one possible instance of the graph combinators. Here, the graph is not encoded by data structures but by code performing the needed actions. For example, $\mathbf{mkFun}_s$ takes a function f and returns a code (i.e., builds a closure) that will evaluate the function f applied to its argument a using $\mathbf{unwind}_s$ whereas $\mathbf{mkApp}_s$ takes two expressions x_1 and x_2 and returns a code that will apply x_1 to x_2. This encoding simplifies the interpreter which just has to trigger a code; that is, $\mathbf{unwind}_s$ is just an application. It is easy to check that these definitions verify the conditions ($\mathcal{C}ond\mathcal{G}$), ($\mathcal{GN}m1$), ($\mathcal{GN}m2$), and ($\mathcal{GN}m3$). Moreover, the definition of $\mathbf{mkVar}_s$ (the identity function in Λ_s) makes it clear that indirection chains can be collapsed, i.e.,

$$\forall E \in \Lambda,\ \mathcal{G}[\![E]\!] \circ \mathbf{mkVar}_s = \mathcal{G}[\![E]\!]. \qquad \text{(L5)}$$

With this combinator instantiation, the graph is represented by closures. More classical representations, based on data structures, are mentioned in Section 3.3.3. The correctness of $\mathcal{G}$ with respect to conditions $\mathcal{GN}m$ is stated by Property 7 (Appendix L).

PROPERTY 7. *Let* ($\mathcal{C}ond\mathcal{G}$), ($\mathcal{GN}m1$), ($\mathcal{GN}m2$), ($\mathcal{GN}m3$), *and* (L5) *hold; then for all closed* Λ*-expressions* E, *if* $E \xrightarrow[cbn]{} V$, *then* $\mathcal{G}[\![E]\!] \circ \mathbf{unwind}_s = \mathcal{G}[\![V]\!] \circ \mathbf{unwind}_s$.

Compared to the corresponding properties for the previous transformations ($\mathcal{V}a$, $\mathcal{N}a$, $\mathcal{V}m$, $\mathcal{N}m$), Property 7 is expressed using equality instead of reduction ($\overset{*}{\rightarrow}$). This is because the normal form of $\mathcal{G}[\![E]\!] \circ \mathbf{unwind}_s$ may contain indirection nodes ($\mathbf{mkVar}_s$) and is not, in general, syntactically identical to $\mathcal{G}[\![V]\!] \circ \mathbf{unwind}_s$. Actually, $\mathcal{G}$ verifies a stronger (but less easily formalized) property than Property 7: $\mathcal{G}[\![E]\!] \circ \mathbf{unwind}_s$ reduces to an expression X which, after removal of indirection chains, is syntactically equal to the graph of $\mathcal{G}[\![V]\!]$.

Example. Let $E \equiv (\lambda x.x)((\lambda y.y)(\lambda z.z))$ and

$$I_w \equiv (\lambda_s a.(\mathbf{push}_s\ a \circ (\lambda_s w.\mathbf{push}_s\ w \circ \mathbf{mkVar}_s)) \circ \mathbf{unwind}_s);$$

then

$$\mathcal{G}[\![E]\!] \circ \mathbf{unwind}_s$$

$$\equiv (\mathcal{G}[\![\lambda z.z]\!] \circ \mathcal{G}[\![\lambda y.y]\!] \circ \mathbf{mkApp}_s) \circ \mathcal{G}[\![\lambda x.x]\!] \circ \mathbf{mkApp}_s \circ \mathbf{unwind}_s$$

$$\overset{*}{\rightarrow} \mathbf{push}_s(\mathbf{push}_s(\mathbf{push}_s\ I_z \circ I_y) \circ I_x) \circ \mathbf{unwind}_s$$

$$\rightarrow \mathbf{push}_s(\mathbf{push}_s\ I_z \circ I_y) \circ (\lambda_s a.\ (\mathbf{push}_s\ a \circ (\lambda_s x.\mathbf{push}_s\ x \circ \mathbf{mkVar}_s)) \circ \mathbf{unwind}_s)$$

$$\overset{*}{\rightarrow} \mathbf{push}_s\ (\mathbf{push}_s\ I_z \circ I_y) \circ \mathbf{unwind}_s$$

$$\rightarrow \mathbf{push}_s\ I_z \circ (\lambda_s a.\ (\mathbf{push}_s\ a \circ (\lambda_s y.\mathbf{push}_s\ y \circ \mathbf{mkVar}_s)) \circ \mathbf{unwind}_s)$$

$$\overset{*}{\rightarrow} (\mathbf{push}_s\ I_z \circ \mathbf{mkVar}_s) \circ \mathbf{unwind}_s \rightarrow \mathbf{push}_s\ I_z \circ \mathbf{unwind}_s.$$

In this example, there is no indirection chain, and the result is syntactically equal to the graph of the source normal form. That is, $\mathbf{push}_s\ I_z \circ \mathbf{unwind}_s$ is exactly $\mathscr{G}[\![\lambda z.z]\!] \circ \mathbf{unwind}_s$ after the few reductions corresponding to graph construction.

The first sequence of reductions corresponds to the graph construction. Then $\mathbf{unwind}_s$ scans the (leftmost) spine (the first $\mathbf{push}_s$ represents an application node). The graph representing the function $(\lambda x.x)$ is applied. The result is the application node $\mathbf{push}_s(\mathbf{push}_s\ I_z \circ I_y)$ which is scanned by $\mathbf{unwind}_s$. Then, the reduction proceeds in the same way until it reaches the normal form.

Because of the interpretive essence of the graph reduction, a naive implementation of call-by-need is possible without introducing marks (as opposed to $\mathscr{N}_m$ in Section 3.2.2). Such a scheme performs many useless updates, some of which can be detected by simple syntactic criteria or a sharing analysis. An optimized implementation, performing selective updates, can be defined by introducing marks. These two points are presented in Section 6.2.2.

3.3.3 *Other Choices.* A graph and its associated reducer can be seen as an abstract data type with different implementations [Peyton Jones 1987]. We have already used one encoding that represents nodes by code (i.e., closures). Another natural solution is to represent the graph by a data structure. It amounts to introducing three data constructors **VarNode**, **FunNode**, and **AppNode** and to defining the interpreter $\mathbf{unwind}_s$ by a case expression. A refinement, exploited by the G-machine, is to enclose in nodes the code to be executed when it is unwound. Adding code in data structures comes very close to the solution using closures described in Figure 9. The interpreter $\mathbf{unwind}_s$ can just execute the code and does not have to perform a dynamic test. In any case, the new combinator definitions should still verify the $\mathscr{GN}_m$ properties in order to implement a push-enter model of the compilation of call-by-name.

By far, the most common use of graph reduction is the implementation of call-by-need in the push-enter model. However, the eval-apply model or the compilation of call-by-value can be expressed as well. These choices are specified by redefining the interactions of $\mathbf{unwind}_s$ with the three graph

builders ($\mathbf{mkVar}_s$, $\mathbf{mkFun}_s$, $\mathbf{mkApp}_s$). In each case, it amounts to defining new properties such as ($\mathcal{GN}m1$), ($\mathcal{GN}m2$), and ($\mathcal{GN}m3$).

More details on these alternate choices can be found in Douence and Fradet [1996b].

3.4 Comparisons

We compare the efficiency of codes produced by transformations $\mathcal{Y}a$ (eval-apply CBV) and $\mathcal{Y}m$ (push-enter CBV). Then we exhibit the precise relationship between the environment and graph approaches. In particular, it is shown how to derive the transformation $\mathcal{N}m$ from $\mathcal{G}$ and the properties ($\mathcal{GN}mi$). We take only these two examples to show the advantages of a unified framework in terms of formal comparisons. It should be clear that such comparisons could be carried on for other transformations and compilation steps.

3.4.1 $\mathcal{Y}a$ versus $\mathcal{Y}m$. Let us first emphasize that our comparisons focus on finding complexity upper bounds. They do not take the place of benchmarks which are still required to take into account complex implementation aspects (e.g., interactions with memory cache or the garbage collector).

A code produced by $\mathcal{Y}m$ builds less closures than the corresponding $\mathcal{Y}a$-code. Since a mark can be represented by one bit (e.g., in a bit stack parallel to the data stack), $\mathcal{Y}m$ is likely to be, on average, more efficient with respect to space resources. Concerning time efficiency, the size of compiled expressions provides a first approximation of the cost 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 $\mathcal{Y}x = \mathcal{Y}a$ or $\mathcal{Y}m$, we have

$$\text{If } \mathcal{S}ize(E) = n, \text{ then } \mathcal{S}ize(\mathcal{Y}x[\![E]\!]) < 3n.$$

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., $\mathbf{push}_s\ E$), and *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

$$\mathcal{C}ost(\mathcal{Y}a[\![E]\!]) = n_\lambda\, clos + n_v\, push + (n_v - 1)\, app$$

and

$$\mathcal{C}ost(\mathcal{Y}m[\![E]\!]) = (n_\lambda + n_v)\, grab + (n_v - 1)\, push.$$

The benefit of $\mathcal{Y}m$ over $\mathcal{Y}a$ is to sometimes replace a (useless) closure construction by a test. When a closure has to be built, $\mathcal{Y}m$ involves a useless test compared to $\mathcal{Y}a$. So if *clos* is comparable to the cost of a test (e.g., when

returning a closure amounts to building a pair as in Section 4.1.2) $\mathcal{V}_m$ will produce more expensive code than $\mathcal{V}_a$. If closure building is not a constant time operation (as in Section 4.1.3), $\mathcal{V}_m$ can be arbitrarily better than $\mathcal{V}_a$. Actually, it can change the program complexity in contrived cases. In practice, however, the situation is not so clear. When no mark is present, $\mathbf{grab}_s$ is implemented by a test followed by an **app**. If a mark is present, the test is followed by a $\mathbf{push}_s$ (i.e., a closure building for λ-abstractions). So, we have

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

with p (resp., $\bar{p}$) representing the likelihood ($p + \bar{p} = 1$) of the presence (resp., absence) of a mark that depends on the program. The best situation for $\mathcal{V}_m$ is when no closure has to be built, i.e., $p = 0$ and $\bar{p} = 1$. If we take some reasonable hypothesis such as $test = app$ and $n_\lambda < n_v < 3n_\lambda$, we find that the cost of closure construction must be 3 to 5 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 7 or 8 *app*.

We are led to conclude that $\mathcal{V}_m$ should be considered only when closure building is potentially costly (such as the $\mathcal{A}c2$ transformation in Section 4.1.3 which builds closures by copying part of the environment). 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. In Douence and Fradet [1996a] we present how $\mathcal{V}_a$ and $\mathcal{V}_m$ could be mixed.

3.4.2 *Environment Machine versus Graph Reducer.* Even if their starting points are utterly different, graph reducers and environment machines can be related. This has been done for specific implementations such as Peyton Jones and Lester [1992] which shows how to transform a G-machine into a Tim. We focus here on the compilation of control and compare the transformation $\mathcal{N}_m$ with the $\mathcal{GN}_m$ approach to graph reduction.

The two main departures of graph reduction from the environment approach are the following:

—*The potentially useless graph constructions*: For example, the rule $\mathcal{G}[\![E_1\ E_2]\!] = \mathcal{G}[\![E_2]\!] \circ \mathcal{G}[\![E_1]\!] \circ \mathbf{mkApp}_s$ builds a graph for E_2 even if E_2 is never reduced (i.e., if it is not needed). On the other hand, $\mathcal{N}_m$ suspends all operations (such as variable instantiation) on E_2 by building a closure ($\mathcal{N}_m[\![E_1\ E_2]\!] = \mathbf{push}_s\ (\mathcal{N}_m[\![E_2]\!]) \circ \mathcal{N}_m[\![E_1]\!]$).

—*The interpretative nature of graph reduction*: Even in the "node-as-code" instantiation, each application node ($\mathbf{mkApp}_s$) is "interpreted" by $\mathbf{unwind}_s$. In the environment family no interpreter is needed, and this approach can be seen as the specialization of the interpreter $\mathbf{unwind}_s$ according to the source graph built by $\mathcal{G}[\![\,]\!]$.

In order to formalize these two points, we first change the rule for graph building in the case of applications by

$$\mathcal{G}[\![E_1\ E_2]\!] = \mathbf{push}_s(\mathcal{G}[\![E_2]\!] \circ \mathbf{unwind}_s) \circ \mathcal{G}[\![E_1]\!] \circ \mathbf{mkApp}_s.$$

This corresponds to a lazy graph construction where the graph argument is built only if needed. In particular, variables will be bound to unbuilt graphs. This new kind of graph entails replacing property ($\mathcal{GN}_m1$) with

$$(\mathcal{GN}_m1) \quad (\mathbf{push}_s\ E \circ \mathbf{mkVar}_s) \circ \mathbf{unwind}_s = E.$$

We can now show that $\mathcal{N}_m[\![E]\!]$ is merely the specialization of $\mathbf{unwind}_s$ with respect to the graph of E, i.e.,

$$\mathcal{N}_m[\![E]\!] = \mathcal{G}[\![E]\!] \circ \mathbf{unwind}_s.$$

For example, the specialization for the application case is

$\mathcal{G}[\![E_1\ E_2]\!] \circ \mathbf{unwind}_s$

$= \mathbf{push}_s(\mathcal{G}[\![E_2]\!] \circ \mathbf{unwind}_s) \circ \mathcal{G}[\![E_1]\!] \circ \mathbf{mkApp}_s \circ \mathbf{unwind}_s$ (unfolding $\mathcal{G}$)

$= \mathbf{push}_s(\mathcal{G}[\![E_2\ ]\!] \circ \mathbf{unwind}_s) \circ \mathcal{G}[\![E_1]\!] \circ \mathbf{unwind}_s$ ($\mathcal{GN}_m3$)

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

$= \mathcal{N}_m[\![E_1\ E_2]\!].$ (folding $\mathcal{N}_m$)

This property shows that, as far as the compilation of control is concerned, environment-based transformations are more efficient than their graph counterparts. However, optimized graph reducers avoid as much as possible interpretative scans of the graph or graph building and are similar to environment-based implementations.

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 of replacing variables with combinators. Compared to Λ_s, Λ_e adds the pair ($\mathbf{push}_e$, λ_e) encoding an environment component, and it uses variables only to define combinators. Graph reducers use specific (usually environmentless) transformations. We express in our framework the **SKI** abstraction algorithm (Section 4.2).

4.1 Environment-Based Abstractions

In the λ-calculus, the β-reduction is defined as a textual substitution. In environment-based implementations, substitutions are compiled by storing the value to be substituted in a data structure (an environment). Values

$$\mathcal{A}g : \Lambda_s \to env \to \Lambda_e$$
$$\mathcal{A}g[\![E_1 \circ E_2]\!]\rho = \mathbf{dupl}_e \circ \mathcal{A}g[\![E_1]\!]\rho \circ \mathbf{swap}_{se} \circ \mathcal{A}g[\![E_2]\!]\rho$$
$$\mathcal{A}g[\![\mathbf{push}_s\ E]\!]\rho = \mathbf{push}_s(\mathcal{A}g[\![E]\!]\rho) \circ \mathbf{mkclos}$$
$$\mathcal{A}g[\![\lambda_s x.E]\!]\rho = \mathbf{mkbind} \circ \mathcal{A}g[\![E]\!](\rho, x)$$
$$\mathcal{A}g[\![x_i]\!](\ldots((\rho, x_i), x_{i-1})\ldots, x_0) = \mathbf{access}_i \circ \mathbf{appclos}$$

Fig. 10. A generic abstraction ($\mathcal{A}g$).

are then accessed in the environment only when needed. This technique can be compared with the activation records used by imperative language compilers. The main choice is using list-like (shared) environments or vector-like (copied) environments. For the latter choice, there are several transformations, depending on when the environments are copied.

4.1.1 *A Generic Abstraction.* The denotational-like transformation $\mathcal{A}g$ (Figure 10) is a generic abstraction that is specialized to model several choices in the following sections. It introduces an environment from which the values of variables are stored and fetched. The transformation is done with respect to a compile-time environment ρ (initially empty for a closed expression). We denote x_i as the variable occurring at the ith entry in the environment.

$\mathcal{A}g$ needs six new combinators to express environment saving and restoring ($\mathbf{dupl}_e$, $\mathbf{swap}_{se}$), closure building and calling (**mkclos**, **appclos**), access to values ($\mathbf{access}_i$), and adding a binding (**mkbind**).

The first combinator pair ($\mathbf{dupl}_e$, $\mathbf{swap}_{se}$) is defined in Λ_e by

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

Note that $\mathbf{swap}_{se}$ is needed only if s and e are implemented by a single component. In our approach, this choice is made in the final implementation step (see Section 2.5). If eventually e and s are implemented by, say, two distinct stacks, then new algebraic simplifications become valid; in particular, $\mathbf{swap}_{se}$ can be removed (its definition as a λ-expression will be the identity function).

The closure combinators (**mkclos**, **appclos**) can have different definitions in Λ_e as long as they satisfy the condition

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

That is, evaluating a closure made of the function X and environment E amounts to evaluating X with the environment E. For example, two possible definitions are

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

or

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

$$\textbf{mkbind} = \lambda_e e.\lambda_s x.\textbf{push}_e(e, x) \qquad \textbf{access}_i = \textbf{fst}^i \circ \textbf{snd}$$
$$\textbf{fst} = \lambda_e(e, x).\textbf{push}_e\ e \qquad \textbf{snd} = \lambda_e(e, x).\textbf{push}_s\ x$$
$$\textit{with} \quad \mathbf{c}^i = \mathbf{c} \circ \ldots \circ \mathbf{c}\ (i \text{ times})$$

Fig. 11. Combinator instantiation for shared environments ($\mathscr{A}_s$).

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 it is used in the rest of the article.

In the same way, the environment combinators (**mkbind**, $\textbf{access}_i$) can have several instantiations in Λ_e. Let us denote $\textbf{comb}^i$ as the sequence $\textbf{comb} \circ \ldots \circ \textbf{comb}$ (i times); then the definitions of **mkbind** and $\textbf{access}_i$ must satisfy the condition

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

This property simply says that adding $i + 1$ bindings $X_i, \ldots, X_0$ in an environment E then accessing the ith value is equivalent to returning X_i directly. Examples of definitions for **mkbind** and $\textbf{access}_i$ appear in Figures 11 and 12.

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

$$\mathscr{A}_g[\![E \circ \textbf{app}]\!]\rho = \mathscr{A}_g[\![E]\!]\rho \circ \textbf{appclos}$$

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

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

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

Whether this new rule duplicates the closure or only its address depends on the memory management (Section 6). In call-by-need, one has to make sure that $\textbf{access}_i$ returns the address of the closure, since closure duplication may entail a loss of sharing.

4.1.2 *Shared Environments.* A first choice is to instantiate $\mathscr{A}_g$ with linked environments. 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 of the variable to its binding λ_s (i.e., its de Bruijn index). This specialization, denoted $\mathscr{A}_s$, is used by the Cam [Cousineau et al. 1987], the SECD [Landin 1964], and the strict and lazy versions of the Krivine machine [Leroy 1990; Crégut 1991].

$\mathbf{mkbind} = \lambda_e e.\lambda_s x.\mathbf{push}_e(e[next] := x)$ $\quad\quad$ $\mathbf{access}_i = \lambda_e e.\mathbf{push}_s(e[i])$
where $e[next] := x$ *adds the value x in the first empty cell of the vector e*

Fig. 12. Combinator instantiation for abstraction with copied environments (Ac_i).

Specializing $\mathscr{A}_g$ into $\mathscr{A}_s$ amounts to defining the environment combinators as in Figure 11.

Example.

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

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

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

The correctness of $\mathscr{A}_s$ is stated by Property 8 (Appendix M).

PROPERTY 8. *For all closed well-typed* Λ_s*-expressions E,* $\mathbf{push}_e\ () \circ \mathscr{A}_s[\![E]\!]\ () = E$.

4.1.3 *Copied Environments.* Another choice is to provide a constant access time. In this case, the structure of the environment must be a vector of closures. A code copying the environment (an $O(length\ \rho)$ operation) has to be inserted in $\mathscr{A}_g$ in order to avoid links. This scheme is less prone to space leaks, since it permits suppressing useless variables during copies.

The macrocombinator **Copy** ρ produces code performing this copy according to ρ's structure.

$$\mathbf{Copy}(\cdots(),x_n),\cdots,x_0)$$

$$= (\mathbf{dupl}_e \circ \mathbf{access}_n \circ \mathbf{swap}_{se}) \circ \ldots \circ (\mathbf{dupl}_e \circ \mathbf{access}_1 \circ \mathbf{swap}_{se})$$

$$\circ\ \mathbf{access}_0 \circ \mathbf{push}_e() \circ \mathbf{mkbind}^{n+1}.$$

The combinators $\mathbf{dupl}_e$ and $\mathbf{swap}_{se}$ are needed to pass the environment to each $\mathbf{access}_i$ which will store each value of the environment in s. With all the values in s, a fresh copy of the environment can be built (using $\mathbf{push}_e\ ()$ ∘ $\mathbf{mkbind}^{n+1}$). If we still see the structure of the environment as a tree of closures, the effect of **Copy** ρ is to prevent sharing. Environments can thus be represented by vectors. The combinator **mkbind** now adds a binding in a vector, and $\mathbf{access}_i$ becomes a constant time operation (Figure 12).

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

$$\mathscr{A}_c[\![\lambda_s y.E]\!](((((),x_2),x_1),x_0) = \mathbf{mkbind} \circ \mathscr{A}_c[\![E]\!](((((),x_2),x_1),x_0),y)$$

we have $next = rank\ y\ (((((),x_2),x_1),x_0),y) = 4$, and y is stored in the fourth cell of the environment. The maximum size of each vector can be

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

Fig. 13. Copy at function entry ($\mathscr{A}c1$).

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 them by indicating only the rules that differ from $\mathscr{A}g$. A first solution (Figure 13) is to copy the environment just before adding a new binding (as in Fairbairn and Wray [1987] and Plasmeijer and van Eekelen [1993]). From the first compilation 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. The transformation $\mathscr{A}c1$ produces (possibly oversized) environments that can be shared by several closures but only as a whole. So, there must be an indirection when accessing the environment. The environment $\bar{\rho}$ represents ρ restricted to variables occurring free in the subexpression E.

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

$$\mathbf{push}_s(\mathscr{A}c1[\![E]\!]((\bar{\rho},x_1),x_0)) \circ \mathbf{mkclos} \circ \mathbf{swap}_{se} \circ \mathbf{access}_1 \circ \mathbf{appclos}.$$

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

A second solution (Figure 14) is to copy the environment when building and opening closures (as in Fradet and Le Métayer [1991]). 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). The transformation $\mathscr{A}c2$ produces environments that cannot be shared but may be accessed directly (they can be packaged with a code pointer to form a closure).

A refinement of this last option, the $\mathscr{A}c3$ abstraction (see Appendix N), is to copy the environment only when building closures. Variations of $\mathscr{A}c3$ are used in the SML-NJ compiler [Appel 1992] and the spineless tagless G-machine [Peyton Jones 1992]. In order to be able to add new bindings after closure opening, an additional local environment is needed.

Starting from different properties a collection of abstractions can be systematically derived from $\mathscr{A}g$. Some of these abstractions are new; some have already been used in well-known implementations. For example, starting from the equation $\mathscr{A}g_{\mathrm{s}}\ [\![E]\!]\ \rho = \mathbf{swap}_n \circ \mathscr{A}g\ [\![E]\!]\ \rho$ one can derive the swapless transformation $\mathscr{A}g_{\mathrm{s}}$. With this variation, the references to environments stay at a fixed distance from the bottom of the stack until they are popped (the references are no longer **swap**ped). These variations introduce different environment manipulation schemes avoiding stack element reordering (**swap**less),

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

Fig. 14. Copy at closure building and opening ($\mathscr{A}c2$).

environment duplication (**dupl**less), environment building (**mkbind**less), or closure building (**mkclos**less) (Appendix O).

4.1.4 *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 $\mathscr{A}$ entails a code expansion which is quadratic with respect to the size of the source expression. More precisely

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

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

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

which makes clear that the efficiency of $\mathscr{A}c1$ is not dependent on accesses. The two transformations 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 favor building, and copied environments, that favor access. Let us point out that these bounds are related to the quadratic growth implied by Turner's [1979a] abstraction algorithm. Balancing expressions reduces this upper bound to $O(n \log n)$ [Joy et al. 1985]. It is very likely that this technique could also be applied to λ-expressions to get an $O(n \log n)$ complexity for environment management.

The abstractions also can be compared according to their memory usage. $\mathscr{A}c2$ copies the environment for every closure, where $\mathscr{A}c1$ may share a bigger copy. So, the code generated by $\mathscr{A}c2$ consumes more memory and implies frequent garbage collections, whereas the code generated by $\mathscr{A}c1$ may create space leaks and needs special tricks to plug them (see Peyton Jones and Lester [1991, Section 4.2.6]).

4.2 A SKI Abstraction Algorithm

Some abstraction algorithms do not use the environment notion, but encode every substitution separately. A simple algorithm [Curry and Feys 1958] uses only three combinators {**S**, **K**, **I**} but is inefficient with respect to code expansion. Different refinements, which use extended combinator families (e.g., {**S**, **K**, **I**, **B**, **C**, **S′**, **B′**, **C′**}), have been proposed [Joy et al. 1985; Turner 1979a; 1979b]. They usually lower the complexity of code expansion from

$$\mathcal{S}ki: \Lambda_s \rightarrow var \rightarrow \Lambda_e$$
$$\mathcal{S}ki[\![E]\!]x = E \circ (\mathbf{push}_s\ \mathbf{Ks} \circ \mathbf{mkFun}_s) \circ \mathbf{mkApp}_s \qquad x \text{ not free in } E$$
$$\mathcal{S}ki[\![E_1 \circ E_2 \circ \mathbf{mkApp}_s]\!]x$$
$$= \mathcal{S}ki[\![E_1]\!]x \circ (\mathcal{S}ki[\![E_2]\!]x \circ (\mathbf{push}_s\ \mathbf{Ss} \circ \mathbf{mkFun}) \circ \mathbf{mkApp}_s) \circ \mathbf{mkApp}_s$$
$$\mathcal{S}ki[\![\mathbf{push}_s(\lambda_s y.E) \circ \mathbf{mkFun}_s]\!]x = \mathcal{S}ki[\![\mathcal{S}ki[\![E]\!]y]\!]x$$
$$\mathcal{S}ki[\![\mathbf{push}_s x \circ \mathbf{mkVar}_s]\!]x = \mathbf{push}_s\ \mathbf{Is} \circ \mathbf{mkFun}_s$$

Fig. 15. Abstraction SKI ($\mathcal{S}ki$).

exponential with {**S**, **K**, **I**} to quadratic or even $O(n \log n)$. We describe only the SKI abstraction algorithm in this article. It should be clear that the optimized versions could be expressed as easily in our framework.

It is possible to define a transformation $\mathcal{S}ki[\![E]\!]\,x$ that can be applied to all Λ_s-expressions [Douence and Fradet 1996b]. In particular, it can be composed with the transformations for the compilation of graph reduction control (Section 3.3). The resulting code, although correct, does not accurately model the classical compilation scheme of the SKI-machine. The easiest way to model it precisely is to define a transformation specialized for graph code (Figure 15).

The **Ss**, **Ks**, and **Is** combinators build or select a graph. They can be defined as

$$\mathbf{Ss} = \lambda_s e_2.\lambda_s e_1.\lambda_s x.(\mathbf{push}_s\ x \circ \mathbf{push}_s\ e_1 \circ \mathbf{mkApp}_s)$$
$$\circ (\mathbf{push}_s\ x \circ \mathbf{push}_s\ e_2 \circ \mathbf{mkApp}_s) \circ \mathbf{mkApp}_s$$

$$\mathbf{Ks} = \lambda_s e.\lambda_s x.\mathbf{push}_s\ e \qquad \mathbf{Is} = \lambda_s x.\mathbf{push}_s\ x$$

In the same way, the transformation $\mathcal{A}g_{dsb}$ (a **dupl**less, **swap**less, and **mkbind**less abstraction algorithm) can be specialized for graph code [Douence and Fradet 1996b]. It would then precisely model the classical abstraction of the G-machine [Johnsson 1987].

5. COMPILATION OF CONTROL TRANSFERS

A conventional machine executes linear sequences of basic instructions. In our framework, reducing expressions of the form **appclos** $\circ$ E involves evaluating a closure and then returning to E. We have to make calls and returns explicit. We present here two solutions.

The first solution, adopted by most implementations, is to save the return address on a call stack k. The transformation $\mathcal{S}$ (Figure 16) 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* $i \equiv s$ or e) when the function ends. Intuitively these combinators can be seen as implementing a control stack. Compared to Λ_e, Λ_k = expressions do not have **appclos** $\circ$ E code sequences. The correctness of $\mathcal{S}$ is stated by Property 9.

$$\begin{array}{ll}
\mathcal{S}: \Lambda_e \rightarrow \Lambda_k & \textit{with } i \equiv s, e \\
\mathcal{S}[\![E_1 \circ E_2]\!] = \mathbf{push}_k(\mathcal{S}[\![E_2]\!]) \circ \mathcal{S}[\![E_1]\!] & \\
\mathcal{S}[\![\mathbf{push}_i\ E]\!] = \mathbf{push}_i(\mathcal{S}[\![E]\!]) \circ \mathbf{rts}_i & \textit{with } \mathbf{rts}_i = \lambda_i x.\lambda_k k.\mathbf{push}_i\ x \circ k \\
\mathcal{S}[\![\lambda_i x.E]\!] = \lambda_i x.\mathcal{S}[\![E]\!] & \\
\mathcal{S}[\![x]\!] = x &
\end{array}$$

Fig. 16. General compilation of control transfers ($\mathcal{S}$).

PROPERTY 9. *For all closed well-typed Λ_e-expressions E and N being a normal form,*

$$\textit{if } E \overset{*}{\rightarrow} N, \textit{ then } \mathcal{S}[\![E]\!] \overset{*}{\rightarrow} \mathcal{S}[\![N]\!].$$

An optimized version of $\mathcal{S}$ for the different previous transformations could easily be derived. For example, we get

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

The second solution is to use a transformation $\mathcal{S}\ell$ between the control and the abstraction phases ($\mathcal{S}\ell$: $\Lambda_s \rightarrow \Lambda_s$). It transforms the expression into CPS. The continuation k encodes return addresses and will be abstracted as an ordinary variable. Let us present only two transformation rules:

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

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

The first rule replaces returns by continuation calls, and the second rule encodes the return stack of $\mathcal{S}$ by a continuation composition. This solution is used in the SML-NJ compiler [Appel 1992].

6. SHARING AND UPDATES

The call-by-need strategy is an optimization of the call-by-name strategy that shares and updates closures. In order to express sharing, we introduce a memory component to store closures. The evaluation of an unevaluated argument amounts to accessing a closure in the memory, reducing it, and updating the memory with its normal form. This way, every argument is reduced at most once. The new intermediate language Λ_h adds to Λ_k the combinator pair ($\mathbf{push}_h$, λ_h) that specifies a memory component h. This component is represented and accessed via a heap pointer. A first transformation $\mathcal{H}_e$ from Λ_k to Λ_h threads the component h in which closures are allocated and accessed. Then we express updating and present several options specific to graph reduction.

$$
\begin{array}{lll}
\mathcal{H}_c\colon \Lambda_k \rightarrow \Lambda_h & \text{with } i \equiv s, e \text{ or } k & \text{and } h \text{ a fresh variable} \\
\mathcal{H}_c[\![E_1 \circ E_2]\!] = \mathcal{H}_c[\![E_1]\!] \circ \mathcal{H}_c[\![E_2]\!] & & \\
\mathcal{H}_c[\![\lambda_i x.E]\!] = \lambda_h h.\lambda_i x.\mathbf{push}_h\ h \circ \mathcal{H}_c[\![E]\!] & \text{with } i \equiv s, e \text{ or } k & \\
\mathcal{H}_c[\![\mathbf{push}_i\ E]\!] = \mathbf{Store}[\mathcal{H}_c[\![E]\!]] & \text{if } i \equiv s \text{ and } E\colon \mathsf{R}_s\sigma & \\
= \lambda_h h.\mathbf{push}_i(\mathcal{H}_c[\![E]\!]) \circ \mathbf{push}_h\ h & \text{otherwise } (i \equiv s, e \text{ or } k) & \\
\mathcal{H}_c[\![x]\!] = \mathbf{Call}[x] & \text{if } x\colon \mathsf{R}_s\tau \text{ bound by } \lambda_s x. & \\
= x & \text{otherwise} &
\end{array}
$$

$$
\begin{aligned}
\text{with } \mathbf{Store}[E] \equiv\ & \lambda_h h.\mathbf{push}_h\ h \circ \mathbf{alloc} \circ \lambda_h h.\lambda_s a. \\
& \mathbf{push}_s\ E \circ \mathbf{push}_s\ a \circ \mathbf{push}_h\ h \circ \mathbf{write} \circ \\
& \lambda_h h.\mathbf{push}_s\ a \circ \mathbf{push}_h\ h \\
\mathbf{Call}[E] \equiv\ & \lambda_h h.\mathbf{push}_s\ E \circ \mathbf{push}_h\ h \circ \mathbf{read} \circ \lambda_s y.\mathbf{push}_h\ h \circ y \\
\mathbf{alloc} =\ & \lambda_h(\mathit{heap}, \mathit{free}).\mathbf{push}_s\ \mathit{free} \circ \mathbf{push}_h(\mathit{heap}, \mathit{free} + 1) \\
\mathbf{write} =\ & \lambda_h(\mathit{heap}, \mathit{free}).\lambda_s \mathit{add}.\lambda_s \mathit{val}.\mathbf{push}_h((\mathit{heap}, \{\mathit{add}, \mathit{val}\}), \mathit{free}) \\
\mathbf{read} =\ & \lambda_h((\mathit{heap}, \{\mathit{add}_1, \mathit{val}\}), \mathit{free}).\lambda_s \mathit{add}_2.\ \mathit{if}\ \mathit{add}_1 = \mathit{add}_2 \\
& \mathit{then}\ \mathbf{push}_s\ \mathit{val}\ \mathit{else}\ \mathbf{push}_h(\mathit{heap}, \mathit{free}) \circ \mathbf{push}_s\ \mathit{add}_2 \circ \\
& \mathbf{read}
\end{aligned}
$$

Fig. 17. Introducing a heap where closures are allocated and accessed ($\mathcal{H}c$).

6.1 Introduction of a Heap

The transformation $\mathcal{H}_c$ (Figure 17) introduces a new component h, which encodes a heap threaded through the expression. Throughout the reduction of such an expression, there is only one reference to the heap (i.e., h is single-threaded [Schmidt 1986]).

The transformed expression $\mathcal{H}_c[\![E]\!]$ takes the heap as an argument and returns the heap as the result. The last two rules of $\mathcal{H}_c$ are responsible for making closure allocation and access explicit. In our framework, constructions of updatable closures are of the form $\mathbf{push}_s\ E$ with E:$\mathsf{R}_s\sigma$, and accesses of updatable closures are of the form x:$\mathsf{R}_s\tau$ where x is bound by a λ_s. These rules use two contexts. The context $\mathbf{Store}[E]$ can be read as: allocate a new cell in the heap, write the code E in this cell, and return its address a and the heap. The context $\mathbf{Call}[E]$ can be read as: access the expression stored in the heap in the cell of address E, and then reduce it (with the heap as an argument). Henceforth, the argument of a function is a closure address rather than the closure itself. A closure address is represented by an integer, and the heap is represented by a pair made of a list of written cells and the address of the next free cell $((\mathit{tail}, \{\mathit{add}, \mathit{val}\}), \mathit{free})$. The initial empty heap is denoted **emptyH** and is defined as $((),0)$. The three combinators **alloc**, **write**, and **read** perform basic heap manipulations. Since h is single-threaded, these combinators can be implemented efficiently as constant time operators on a mutable data structure.

We can apply the transformation $\mathcal{H}_c$ to get new versions of the combinators introduced by the previous compilation steps. When a combinator neither creates nor calls a closure, the transformation $\mathcal{H}_c$ threads the heap without interaction. For example, for the combinator $\mathbf{dupl}_e$ introduced by the abstraction $\mathcal{A}_g$, we get

$$\mathbf{dupl}_{eh} = \mathcal{H}_c[\![\mathbf{dupl}_e]\!] = \lambda_h h.\lambda_e e.\mathbf{push}_e\ e \circ \mathbf{push}_e\ e \circ \mathbf{push}_h\ h.$$

$$\mathcal{U}_{callee} : \Lambda_k \to \Lambda_h \qquad \text{with } E : \mathsf{R}_s\sigma$$
$$\mathcal{U}_{callee}[\![\mathbf{push}_s\ E]\!] = \mathbf{Store}[\mathbf{push}_s\ a \circ \mathbf{swap}_{sh} \circ \mathcal{U}_{callee}[\![E]\!] \circ \mathbf{updt}]$$
$$\text{with}\quad \mathbf{swap}_{sh} = \lambda_s a.\lambda_h h.\mathbf{push}_s\ a \circ \mathbf{push}_h\ h$$
$$\text{and}\quad \mathbf{updt} = \lambda_h h.\lambda_s b.\lambda_s a.\mathbf{push}_s(\lambda_h h.\mathbf{push}_s\ b \circ \mathbf{push}_h\ h) \circ \mathbf{push}_s\ a \circ \mathbf{push}_h\ h \circ \mathbf{write} \circ \lambda_h h.\mathbf{push}_s\ b \circ \mathbf{push}_h\ h$$

Fig. 18. Callee closure update ($\mathcal{U}_{callee}$).

On the other hand, combinators such as **appclos** and **mkclos** create or call closures. So, their transformed definitions use **Call** and **Store**:

$$\mathbf{appclos}_h = \mathcal{H}_c[\![\mathbf{appclos}]\!] = \mathcal{H}_c[\![\lambda_s x.x]\!] = \lambda_h h.\lambda_s x.\mathbf{push}_h\ h \circ \mathbf{Call}[x]$$

$$\mathbf{mkclos}_h = \mathcal{H}_c[\![\mathbf{mkclos}]\!]$$
$$= \lambda_h h.\lambda_s x.\lambda_e e.\mathbf{push}_h\ h \circ \mathbf{Store}[\lambda_h h.\mathbf{push}_e\ e \circ \mathbf{push}_h\ h \circ x]$$

6.2 Updating

The transformation $\mathcal{H}_c$ only makes memory management explicit. A heap-stored closure is still reduced every time it is accessed. The call-by-need strategy updates the heap-allocated closures with their normal forms.

The main choice is whether the update is performed by the caller (i.e., by the code from which the closure is accessed) or by the callee (i.e., by the code of the closure itself). The caller update scheme updates a closure every time it is accessed, whereas the callee-update scheme updates closures only the first time they are accessed: once in normal form, other accesses will not entail further (useless) updates. This last scheme is more efficient and is implemented by all the realistic, environment-based implementations. We model only callee updates here.

6.2.1 *Callee Update.* In order to have self-updating closures, the transformation $\mathcal{U}_{callee}$ (Figure 18) changes the rule of $\mathcal{H}_c$ for $\mathbf{push}_s\ E$. It introduces a combinator **updt** which takes as its arguments the heap h, the address b of the result, and the address a of the closure to be updated. It returns the address b and a new heap where the cell a contains an indirection to b. The combinator $\mathbf{swap}_{sh}$ reorders the address x and the heap.

A closure is allocated in the heap when it is created as in $\mathcal{H}_c$, but its code is modified. The closure now stores its own address ($\mathbf{push}_s\ a$), and its evaluation is followed by **updt**. Note that a is a variable bound in the context **Store**[] (see the definition of **Store**) and denotes the address of a freshly allocated cell. Of course, when E is already (syntactically) in normal form, the simple rule $\mathcal{U}_{callee}[\![\mathbf{push}_s\ E]\!] = \mathbf{Store}[\mathcal{U}_{callee}[\![E]\!]]$ suffices. Thus, a closure is updated at most once (i.e., after the first access) because the compiled code of its normal form ($\mathcal{H}_c[\![\mathbf{push}_s\ N]\!]$) contains no **updt**.

The callee-update scheme can be used with $\mathcal{N}_m$. However, as noted in Section 3.2.2, marks have to be inserted in expressions to suspend the reduction before performing an update. The rule for λ-abstractions becomes

$$\mathcal{N}_m[\![\lambda x.E]\!] = \mathbf{grab}_s(\lambda_s x.\mathcal{N}_m[\![E]\!]),$$

and $\mathcal{U}_{callee}$ is specialized for the push-enter model as follows:

$$\mathcal{U}_{callee}[\![\mathbf{push}_s\ E]\!]$$

$$= \mathbf{Store}[\mathbf{push}_s\ a \circ \mathbf{swap}_{sh} \circ \mathbf{push}_s\ \varepsilon \circ \mathbf{swap}_{sh} \circ \mathcal{U}_{callee}[\![E]\!] \circ$$

$$\mathbf{updt} \circ \mathbf{resume}_h]$$

$$\text{with } \mathbf{resume}_h = \lambda_h h.\lambda_s x.\mathbf{push}_s\ h \circ \mathbf{grab}_h\ x.$$

An evaluation context is isolated by inserting a mark ε after the update address ($\mathbf{push}_s\ a$), and $\mathbf{resume}_h$ resumes the reduction once the update has been performed. The combinator $\mathbf{grab}_h$ is defined by $\mathcal{H}_c[\![\mathbf{grab}_s]\!]$. Marks are used by Tim [Fairbairn and Wray 1987], Clean [Plasmeijer and van Eekelen 1993], the Krivine Machine [Crégut 1991], and the spineless tagless G-machine [Peyton Jones 1992]. The codes produced by $\mathcal{N}_a$ and $\mathcal{N}_m$ have the same update opportunities. As in call-by-name, the call-by-need version of $\mathcal{N}_m$ may prevent building unnecessary intermediate closures.

6.2.2 *Updating and Graph Reduction.* The previous transformations can be employed to transform the call-by-name graph reduction schemes into call-by-need. Here we present two updating techniques (spineless and spine variations) that have been introduced for the G-machine.

The spineless G-machine [Burn et al. 1988] updates only selected application nodes. Unwinding application nodes entails stacking either their address (updatable) or only the argument address (nonupdatable). So, in general, the complete leftmost spine of the graph does not appear in the stack. The code must annotate updatable nodes, and marks are necessary to dynamically detect when an update must be performed. Updatable nodes are distinguished using the combinator $\mathbf{mkAppS}_s$ which has the same definition as $\mathbf{mkApp}_s$, and $\mathbf{mkFun}_s$ must be redefined to detect marks:

$$\mathbf{mkAppS}_s = \lambda_s x_1.\lambda_s x_2.\mathbf{push}_s(\mathbf{push}_s\ x_2 \circ x_1)$$

$$\mathbf{mkFun}_s = \lambda_s f.\mathbf{push}_s(\mathbf{grab}_s(\lambda_s a.(\mathbf{push}_s\ a \circ f) \circ \mathbf{unwind}_s))$$

The transformation $\mathcal{U}_{callee}$ for the push-enter model can be applied to the graph constructors. For $\mathbf{mkAppS}_s$ we get

$$\mathcal{U}_{callee}[\![\mathbf{mkAppS}_s]\!]$$
$$= \lambda_h h.\lambda_s x_1.\lambda_s x_2.\mathbf{Store}[\mathbf{push}_s\ a \circ \mathbf{swap}_{sh} \circ \mathbf{push}_s\ \varepsilon \circ \mathbf{swap}_{sh}$$
$$\circ\ \mathcal{U}_{callee}[\![\mathbf{push}_s\ x_2 \circ x_1\ ]\!] \circ \mathbf{updt} \circ \mathbf{resume}_h].$$

As suggested in Section 3.3.2, the use of marks is not mandatory for expressing updating in the G-machine [Johnsson 1987] where graph building and graph reduction are separate steps. Application nodes must stack their addresses as they are unwound; then updates can be systematically inserted between each graph building and reduction step. However, this naive scheme (that we call the spine variation) cannot be expressed using the previous transformations. Indeed, the canonical definition of $\mathbf{mkApp}_s$ for $\mathcal{GM}$ is

$$\mathbf{mkApp}_s = \lambda_s x_1.\lambda_s x_2.\mathbf{push}_s(\mathbf{push}_s\ x_2 \circ x_1)$$
$$\text{where } \mathbf{push}_s\ x_2 \circ x_1\colon \sigma_1 \rightarrow_s \sigma_2.$$

Since $\mathcal{H}_c$ shares only expressions of the form $\mathbf{push}_s\ E$ with $E{:}\mathsf{R}_s\sigma$, application nodes will not be considered for updating with this definition of $\mathbf{mkApp}_s$. In order to model the G-machine scheme, a new transformation should be defined (see $\mathcal{U}_{spine}$ in Douence and Fradet [1996b]).

The introduction of the threaded memory component in our functional intermediate code makes formal manipulations more complicated. For example, a property ensuring that the reduction of $\mathcal{H}_c[\![E]\!]$ simulates the reduction of E should use a decompilation transformation in order to replace the addresses in reduced expressions by their actual values that lie in the heap. This prevented us from finding a simple and convincing formulation of correctness properties for the transformations presented in this section.

7. CLASSICAL FUNCTIONAL IMPLEMENTATIONS

The description of the compilation process is now complete. A compiler can be described by a simple composition of transformations. Figure 19 states the main design choices structuring several classical implementations. There are cosmetic differences between our descriptions and the real implementations. Some descriptions of the literature leave the compilation of control transfers implicit (e.g., the Cam and Tim). Also, some extensions and optimizations are not described here.

Let us describe precisely our modeling of the categorical abstract machine and state the differences with the description in Cousineau et al. [1987]. The Cam implements the left-to-right call-by-value strategy using the eval-apply model and has linked environments. In our framework, this is expressed

Compiler	*Transformations*				*Components*
	$\Lambda \rightarrow$	$\Lambda_s \rightarrow$	$\Lambda_e \rightarrow$	$\Lambda_k \rightarrow \Lambda_h$	
Cam	$\mathcal{V}a_L$	$\mathcal{A}s$	*Id*	*Id*	s ≡ e
Clean	*Nml*	$\mathcal{A}c1$	*Id*	*UCallee*	s e k h
G-machine	*GNm*	$\mathcal{A}c1_{\mathrm{dsb}'}$	*Id*	*USpine*	s e k h
Spineless G-machine	*GNml*	$\mathcal{A}c1_{\mathrm{dsb}'}$	*Id*	*Ucallee*	s e k h
Spineless tagless G-machine	*Nml*	$\mathcal{A}c3$	*Id*	*UCallee*	(s ≡ k) e h
Mak(cbn)	*Nml*	$\mathcal{A}s$	*Id*	*UCallee*	s ≡ e ≡ k h
Maks(cbv)	$\mathcal{V}m$	$\mathcal{A}s$	$\mathcal{S}$	*Id*	s ≡ e ≡ k
Secd	$\mathcal{V}a$	$\mathcal{A}s$	$\mathcal{S}$	*Id*	s (e ≡ k)
Ski-machine	*GNm*	*Ski*	*Id*	*USpine*	s h
Sml-Nj	$\mathcal{V}a_f$	$\mathcal{A}c3$	$\mathcal{S}\ell$	*Id*	s e (registers)
Tabac(cbv)	$\mathcal{V}a$	$\mathcal{A}c2_{\mathrm{dsb}}$	$\mathcal{S}$	*Id*	(s ≡ e) k
Tabac(cbn)	*Na*	$\mathcal{A}c2_{\mathrm{dsb}}$	$\mathcal{S}$	*UHybrid*	(s ≡ e) k h
Tim	*Nml*	$\mathcal{A}c1_{\mathrm{m}}$	*Id*	*Ucallee*	s e k h

Fig. 19. Several classical compilation schemes.

as $\mathcal{C}am = \mathcal{A}s \cdot \mathcal{V}a_L$. By simplifying this composition of transformations, we get:

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

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

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

with

$$\mathbf{appclos}_L = \lambda_s x.\lambda_s f.\mathbf{push}_s\ x \circ f.$$

To illustrate its output, let us consider the expression $E \equiv (\lambda x.x)\ ((\lambda y.y)(\lambda z.z))$; then

$$\mathcal{C}am\ [\![E]\!] = \mathbf{dupl}_e \circ \mathbf{push}_s\ C_1 \circ \mathbf{mkclos} \circ \mathbf{swap}_{se} \circ \mathbf{dupl}_e \circ \mathbf{push}_s\ C_1$$
$$\circ\ \mathbf{mkclos} \circ \mathbf{swap}_{se} \circ \mathbf{push}_s\ C_1 \circ \mathbf{mkclos} \circ \mathbf{appclos}_L \circ \mathbf{appclos}_L$$

with

$$C_1 \equiv \mathbf{mkbind} \circ \mathbf{snd}.$$

The code is made of two linear code sequences, each of them composed of combinators that can be implemented by a few standard assembly instructions. The minor step consisting of naming code fragments has been left implicit. By instantiating the combinators, we get the rules of the machine. In the Cam, both components s and e are merged; the instantiation is therefore

$$\circ = \lambda abc.a(b\ c) \qquad \mathbf{push}_s\ N = \mathbf{push}_e\ N = \lambda c.\lambda z.c(z,N)$$

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

The definitions of the (macro) combinators follow:

$$\mathbf{dupl}_e = \lambda_e e.\mathbf{push}_e\ e \circ \mathbf{push}_e\ e = \lambda c.\lambda(z,e).c((z,e),e)$$

$$\mathbf{mkbind} = \lambda_e e.\lambda_s x.\mathbf{push}_e(e,x) = \lambda c.\lambda((z,e),x).c(z,(e,x))$$

$$\mathbf{snd} = \lambda_e(e,x).\mathbf{push}_s\ x = \lambda c.\lambda(z,(e,x)).c(z,x)$$

If these combinators are considered as the basic instructions of an abstract machine, their definitions imply the following state transitions:

$\mathbf{dupl}_e$	$C(Z,E)$	$\rightarrow$	$C((Z,E),E)$
mkbind	$C((Z,E),X)$	$\rightarrow$	$C(Z,(E,X))$
snd	$C(Z,(E,X))$	$\rightarrow$	$C(Z,X)$

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

Figure 19 gathers our modelings of 13 implementations of strict or lazy functional languages. It refers to a few transformations not described in this article, but which can be found in Douence and Fradet [1996a; 1996b].

Let us quickly review the differences between Figure 19 and real implementations. The Clean implementation is based on graph rewriting; however, the final code is similar to environment machines (e.g., a closure is encoded by an n-ary node). Our replica is an environment machine that we believe is close. However, the numerous optimizations and especially the lack of clear description (Plasmeijer and van Eekelen [1993] detail only examples of final code) make it difficult to precisely determine the compilation choices.

The G-machine [Johnsson 1987] and the spineless G-machine [Burn et al. 1988] perform only one test for all the arguments of the function (by comparing the arity of the function with the activation record size) whereas our $\mathbf{grab}_s$ combinator performs a test for every argument. So, an n-ary combinator $\mathbf{grabs}_n$ should be introduced.

The spineless tagless G-machine [Peyton Jones 1992] also uses an n-ary version of $\mathbf{grab}_s$ and a local and a global environment. The abstraction with two environments ($\mathscr{A}3$ in our framework) is not directly compatible with $\mathbf{grab}_s$ and extra environment copies must be inserted. The simplest way to model the real machine faithfully would be to introduce a specialized abstraction algorithm.

The Grab instruction of the Krivine abstract machine (Mak) [Crégut 1991; Leroy 1990] is a combination of our $\mathbf{grab}_s$ (in fact, a recursive version; see Appendix I) and **mkbind** combinators.

The SECD machine [Landin 1964] saves environments a bit later than in our scheme. Furthermore, the control stack and the environment stack are grouped into a component called a "dump." The data stack is also (uselessly) saved in the dump. Actually, our replica is closer to the idealized version derived in Hannan [1992].

The SKI-machine [Turner 1979a] reduces a graph made of combinators **S**, **K**, **I**, and application nodes. The graph representing the source expression is totally built at compile time. The machine is made of a recursive interpreter and a data stack to store the unwound spine. Our modeling is close to the somewhat informal description of the SKI-machine in Turner [1979a].

The SML-NJ compiler [Appel 1992] uses only the heap, which is represented in our framework by a unique environment e. It also includes registers and numerous optimizations not described here.

The Tabac compiler is a by-product of our work in Fradet and Le Métayer [1991] and has greatly inspired this study. It implements strict or nonstrict languages by program transformations. Tabac integrated many optimizations that we have not described here.

Our call-by-name Tim description is accurate according to Fairbairn and Wray [1987]. The environment copying included in the transformation $\mathscr{A}c1$ has the same effect as the preliminary lambda-lifting phase of Tim. A n-ary $\mathbf{grab}_s$ should be added to our call-by-need version.

8. EXTENSIONS AND APPLICATIONS

Our framework is powerful enough to handle realistic languages and to model optimizing compilers or hybrid implementations. We illustrate each point in turn. We first present the integration of constants, primitive operators, and data structures; then we take an example of how to express a classical global optimization, and finally we describe a hybrid transformation.

8.1 Constants, Primitive Operators, and Data Structures

We have only considered pure λ-expressions because most fundamental choices can be described through this simple language. Realistic implementations also deal with constants, primitive operators, and data structures that are easily taken into account in our framework.

Concerning basic constants, one question is whether results of basic type are returned in s or whether another component ($\mathbf{push}_b$, λ_b) is introduced. The latter has the advantage of marking a difference between pointers and values that can be exploited by the garbage collector. But in this case, precise type information must also be available at compile time to transform variables and λ-abstractions correctly. In a polymorphic setting, this information is not available in general (a variable x of polymorphic type α can be bound to anything), so constants, functions, and data structures must be stored in s. The fix-point operator, the conditional, and primitive operators acting on basic values are introduced in our language in a

$\mathscr{V}[\![letrec\ f = E]\!] = \mathbf{push}_s(\lambda_s f.\mathscr{V}[\![E]\!]) \circ \mathbf{Y}_s$
with $\mathbf{push}_s\ F \circ \mathbf{Y}_s \rightarrow \mathbf{push}_s(\mathbf{push}_s\ F \circ \mathbf{Y}_s) \circ F$
$\mathscr{V}[\![n]\!] = \mathbf{push}_s\ n$
$\mathscr{V}[\![if\ E_1\ then\ E_2\ else\ E_3]\!] = \mathscr{V}[\![E_1]\!] \circ \mathbf{cond}_s(\mathscr{V}[\![E_2]\!], \mathscr{V}[\![E_3]\!])$
with $\mathbf{push}_s$ True $\circ\ \mathbf{cond}_s(E, F) \rightarrow E$ and $\mathbf{push}_s$ False $\circ\ \mathbf{cond}_s(E, F) \rightarrow F$
$\mathscr{V}[\![E_1 + E_2]\!] = \mathscr{V}[\![E_2]\!] \circ \mathscr{V}[\![E_1]\!] \circ \mathbf{plus}_s$
with $\mathbf{push}_s\ n_2 \circ \mathbf{push}_s\ n_1 \circ \mathbf{plus}_s \rightarrow \mathbf{push}_s\ n_1 + n_2$
$\mathscr{V}[\![head]\!] = \mathbf{head}_s$ with $\mathbf{head}_s = \lambda_s(\mathrm{tag}, h, t).\mathbf{push}_s\ h$
$\mathscr{V}[\![cons\ E_1\ E_2]\!] = \mathscr{V}[\![E_2]\!] \circ \mathscr{V}[\![E_1]\!] \circ$ with $\mathbf{cons}_s = \lambda_s h.\lambda_s t.\mathbf{push}_s(\mathrm{tag}, h, t)$

Fig. 20. An extension with constants, primitive operators, and lists.

straightforward way. The compilation of control using the eval-apply model for these constructs is described in Figure 20.

A naive compilation of β-reduction for letrec expressions yields a code building a closure at each recursive call. Two optimizations exist: building a circular environment or graph and, for environment-based machines, implementing recursive calls to statically known functions by a jump to their address (Appendix P).

As far as data structures are concerned, we can choose to represent them using tags or higher-order functions [Fairbairn and Wray 1987]. Figure 20 describes a possible extension using the data stack to store constants and tagged cells of lists. It just indicates one simple way to accommodate data structures in our framework. The efficient implementation of data structures brings a whole new collection of choices (e.g., see Peyton Jones [1992]) and optimizations (e.g., see Hall [1994] and Shao et al. [1994]). A thorough description of this subject is beyond the scope of this article.

Until now, we considered only pure λ-expressions, and the typing of the source language was not an issue. When constants and data structures are taken into account two cases arise, depending on the typing policy of the source language. If the source language is statically typed, then the code produced by our transformation does not need to be modified (however, supporting polymorphism efficiently involves new and specific optimizations such as unboxing of floats and tuples [Leroy 1992]). For dynamically typed languages, functions, constants, and data structures must carry a type information that will be checked by combinators or primitive operators at run-time.

8.2 Optimizations

Let us take the example of the optimization brought by strictness analysis in call-by-need implementations. It changes the evaluation order and, more interestingly, avoids some thunks using unboxing [Burn and Le Métayer 1996]. If we assume that a strictness analysis has annotated the code $\underline{E}_1\ E_2$ if E_1 denotes a strict function and $\underline{x}$ if the variable is defined by a strict λ-abstraction, then $\mathscr{N}a$ can be optimized as follows:

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

$\mathcal{M}ix\mathcal{A}[\![\lambda_s x.E^{\theta,\oplus}]\!]\rho = \mathbf{Mix}\ \rho\ \theta \circ \mathbf{mkbind}^{\oplus} \circ \mathcal{M}ix\mathcal{A}[\![E]\!](\theta \oplus x)$
$\mathcal{M}ix\mathcal{A}[\![x_i]\!](\ldots(\rho, \rho_i), \ldots, \rho_0) = \mathbf{access}^{l}{}_i \circ \mathcal{M}ix\mathcal{A}[\![x_i]\!]\rho_i$ *with* x_i *in* ρ_i
$\mathcal{M}ix\mathcal{A}[\![x_i]\!][\rho : \rho_i : \ldots : \rho_0] = \mathbf{access}^{v}{}_i \circ \mathcal{M}ix\mathcal{A}[\![x_i]\!]\rho_i$ *with* x_i *in* ρ_i
$\mathcal{M}ix\mathcal{A}[\![x_i]\!](\ldots(\rho, x_i), \ldots, x_0) = \mathbf{access}^{l}{}_i \circ \mathbf{appclos}$
$\mathcal{M}ix\mathcal{A}[\![x_i]\!][\rho : x_i : \ldots : x_0] = \mathbf{access}^{v}{}_i \circ \mathbf{appclos}$
with $\mathbf{access}^{l}{}_i$(*resp.,* $\mathbf{access}^{v}{}_i$) *is the* $\mathbf{access}_i$ *version which accesses a list* (*resp., a vector*)

Fig. 21. Hybrid abstraction (extract).

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 $\mathbf{push}_s\ 2$ is 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 (as in Leroy [1992] or Peyton Jones and Launchbury [1991]) and optimizations (e.g., let-floating [Peyton Jones et al. 1996]) could be expressed as well.

8.3 Hybrid Implementations

The study of the different options showed 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. It is, for example, possible to mix the eval-apply and push-enter models and to design a $\mathcal{V}_a$–$\mathcal{V}_m$ hybrid transformation [Douence and Fradet 1996a]). Here, we describe how to mix shared and copied environments. We suppose that a static analysis has produced an annotated code indicating the chosen mode for each subexpression.

One solution could be to use coercion functions to fit the environment into the chosen structure (list or vector). Instead, we describe a more sophisticated solution (Figure 21) that allows lists and vectors to coexist within environments (as in Shao and Appel [1994]). Motivations for this feature may be to optimize run-time using vectors (resp., links) when access (resp., closure building) cost is predominant or to optimize space usage by using a copy scheme (e.g., vectors) when it eliminates a space leak that would be introduced by linking environments.

Each λ-abstraction is annotated by a new mixed environment structure θ and $\oplus$ ($\in \{v, l\}$) that 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 macrocombinator **Mix** which copies and restructures the environment ρ according to the annotation θ (Figure 21). Paths to values are now ex-

pressed by sequences of $\mathbf{access}_i^l$ and $\mathbf{access}_i^v$. The abstraction algorithm distinguishes vectors from lists in the compile time environment using constructors ":" and ",".

9. RELATED WORK

In this section we review the different formalisms used in the description of functional implementations: the λ-calculus, λ-calculi with explicit substitutions, combinators, and monads. We also present papers comparing specific implementations and the related area of semantic-directed compiler derivation.

Our approach and this article stem from our previous work on compilation of functional languages by program transformation [Fradet and Le Métayer 1991]. Our goal then was to show that the whole implementation process could be described in the functional framework. The two main steps were the compilation of control using a CPS conversion and the compilation of the β-reduction using indexed combinators that could be seen as basic instructions on a stack. We remained throughout within the λ-calculus and did not have to introduce an ad hoc abstract machine. We described only one particular implementation; our main motivation was to make correctness proofs of realistic implementations simpler, not to describe and compare various implementation techniques. The SML-NJ compiler has also been described using program transformations including CPS and closure conversions [Appel 1992]. Other compilers use the CPS transformation to encode the reduction strategy within the λ-calculus [Kranz et al. 1986; Steele 1978]. Encoding implementation issues within the λ-calculus leads to complex expressions (e.g., sequencing is coded as a composition of continuations). The constructors $\mathbf{push}_i$, $\circ$, and λ_i make our framework more abstract and simplify the expressions. The instantiation of these constructors as λ-expressions provides an interesting new implementation step (Section 2.5): the choice of the number and the representation of the components of the underlying abstract machine are kept apart. Within the λ-calculus, one has to choose before describing an implementation whether, for example, data and environments are stored in two separate components or in a single one.

The de Bruijn [1972] λ-calculus, which uses indices instead of variables, has been used as an intermediate language by several abstract machines. As we saw in Section 4.1.2, a de Bruijn index can be seen as the address of a value in the run-time environment. A collection of formalisms, the λ-calculi with explicit substitutions, also emphasizes the environment management and can be seen as calculi of closures [Abadi et al. 1990]. These calculi help formal reasoning on substitution and make some implementation details explicit. However, important implementation choices such as the representation of the environments (lists or vectors) are, in general, not tackled in these formalisms. Hardin et al. [1996] introduce $\lambda\sigma_w$, a weak λ-calculus with explicit substitutions, which can serve as the output language of functional compilers. They describe several abstract

machines in this framework. However, their goal is to exhibit the common points of implementations, not to model existing implementations precisely. Another variant, $\lambda\sigma_w^a$ [Benaissa et al. 1996], can describe sharing and eases the proofs concerning memory management. The $\lambda\sigma_w^a$-expressions stay at a higher level than real machine code, since, for example, sharing is modeled by formal labels and parallel reductions.

A closely related framework used as an intermediate language is combinatory logic [Curry and Feys 1958]. Combinators have been used to encode the compilation of the β-reduction. Some compilation issues are usually not dealt with, such as the representation of environments. Different sets of combinators, such as {**S**, **K**, **I**, **B**, **C**} [Turner 1979a], have been used to define abstraction algorithms for graph reducers [Joy et al. 1985; Lins 1987]. The categorical combinators [Curien 1993] have been used in environment machines such as the Cam [Cousineau 1987] and the Krivine machine [Asperti 1992].

Arising from different roots, our first intermediate language Λ_s is surprisingly close to Moggi's [1991] computational metalanguage. In particular, we may interpret the monadic construct $[E]$ 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). The monadic framework is more abstract. For example, one can write monadic expressions such as

$$\mathbf{let}\_ \Leftarrow \text{writeStack}(X)\mathbf{in}(\mathbf{let}\ e \Leftarrow \text{readEnv}()\mathbf{in}\ E),$$

whereas, in our formalism, we need to reorder data and environment with a **swap** combinator:

$$\mathbf{push}_s\ X \circ \mathbf{swap}_{se} \circ \lambda_e e.E$$

These administrative combinators allow us to merge several components in the instantiation step. The abstract features of monads can be a hindrance to expressing low-level implementation details and to getting closer to a machine code. For example, the monadic call-by-value CPS expression (**let** $a \Leftarrow A$ **in** (**let** $f \Leftarrow F$ **in** $[f\ a]$)) evaluates the argument A, the function F, and returns the application $(f\ a)$, but does not state if the application is reduced before it is returned. In Λ_s, we disallow unrestricted applications and make the previous reduction explicit with an **app** combinator. A key feature of our approach is to describe and structure the compilation process as a composition of individualized transformations. The monadic framework does not appear to be well suited to this purpose, since monads are notoriously difficult to compose. Liang et al. [1995] need complex parameterized monads to describe and compose different compilation steps. The difficulties of composing monads and representing low-level details are serious drawbacks with respect to our goals. Overall, the monadic framework is a general tool for structuring functional programs [Wadler 1992], whereas our small framework has been tailormade to describe implementations.

Except for benchmarks, few functional language implementations have been compared. Some particular compilation steps have been studied. For example, Joy et al. [1985] compare different abstraction algorithms, and Hatcliff and Danvy [1994] express CPS transformations in the monadic framework. A few works explore the relationship between two abstract machines such as CMC and Tim [Lins et al. 1992] and Tim and the G-machine [Peyton Jones and Lester 1992]. Their goal is to show the equivalence between seemingly very different implementations. CMC and Tim are compared by defining transformations between the states of the two machines. The comparison of Tim and the G-machine is more informal, but highlights the relationship between an environment machine and a graph reducer. Also, let us mention Asperti [1992], who provides a categorical understanding of the Krivine machine and an extended Cam, and Crégut [1991], who has studied the relationship between the Tim and the Krivine machine. All these implementation comparisons focus on particular compilation steps or machines, but do not define a global approach to compare implementations.

Related work also includes the derivation of abstract machines from denotational or operational semantics. Starting from a denotational semantics with continuations, Wand [1982] compiles the β-reduction using combinators and linearizes expressions in sequences of abstract code. The semantics of the program is translated into a sequence representing the code and a machine to execute it. In our approach, semantics or machines do not appear explicitly. Hannan [1992] and Sestoft [1994] start from a "big step" (natural) operational semantics, incrementally suppress ambiguities (e.g., impose a left-to-right reduction order), and refine complex operation (e.g., β-reduction), until they get a "small step" (structural) operational semantics. Some of the refinement steps have to deal with operations specific to their framework (e.g., suppressing unification). Meijer [1992] uses program algebra to calculate some simple compilers from a denotational semantics via a series of refinements. All these derivation techniques aim at providing a methodology to formally develop implementations from semantics. Their focus is on the refinement process and correction issues, and usually they describe the derivation of a single implementation. Not surprisingly, the derived compilers do not model existing implementations precisely. They are best described as idealized rather than sophisticated or optimized implementations. Comparisons of implementation choices seem harder with a description based on semantics refinement than with a description by program transformations. Also, some choices seem difficult to naturally obtain by derivation (e.g., the push-enter model for call-by-value). On the other hand, these semantics-based methodologies can potentially be applied to any language that can be described in their semantics framework.

10. CONCLUSION

Let us review the implementation choices encountered in our study. The most significant choice for the compilation of control is using the eval-apply

model ($\mathcal{V}a$, $\mathcal{N}a$) or the push-enter model ($\mathcal{V}m$, $\mathcal{N}m$). There are other minor options such as stackless variations ($\mathcal{V}a_f$, $\mathcal{N}a_f$) or right-to-left versus left-to-right call-by-value. We have shown that the transformations employed by graph reducers can be seen as interpretive versions of the environment-based transformations. For the compilation of β-reduction, the main choice is using environmentless (e.g., SKI) abstraction algorithms, list-like (shared) environments ($\mathcal{A}s$), or vector-like (copied) environments ($\mathcal{A}c$). For the last choice, there are several transformations depending on the way environments are copied ($\mathcal{A}c1$, $\mathcal{A}c2$, $\mathcal{A}c3$). Actually, a complete family of generic transformations modeling different managements of the environment stack can be derived. For control transfers, one can introduce a return address stack or use CPS conversion. Self-updatable closures (i.e., callee update) are the standard way to implement updating, but graph reduction brings other options.

Our approach focuses on (but is not restricted to) the description and comparison of fundamental options. The transformations are designed to model a precise compilation step; they are generic with respect to the other steps. It is then not surprising that, often, simple compositions of transformations do not accurately model real implementations whose design is more ad hoc than generic. In most cases, the differences are nevertheless superficial and it is sufficient to specialize the transformations to obtain existing implementations.

The use of program transformations appears to be well suited to precisely and completely modeling the compilation process. Many standard optimizations (uncurrying, unboxing, hoisting, peephole optimizations) can be expressed as program transformations as well. This unified framework simplifies correctness proofs. For example, we do not explicitly introduce an abstract machine, and therefore we do not have to prove that its operational definition is coherent with the semantics of the language (as in Plotkin [1975] and Lester [1989]). Program transformations make it possible to reason about the efficiency of the produced code as well as about the complexity of transformations themselves. Actually, these advantages appear clearly before the last compilation step. The introduction of a threaded state seriously complicates program manipulations and correctness proofs. This is not surprising because our final code is similar to a real assembly code.

Our main goal was to structure and clarify the design space of functional language implementations. The exploration is still far from complete. There are still many avenues for further research:

—It would be interesting to give a concrete form to our framework by implementing all the transformations presented. This compiler construction workbench would make it possible to implement a wide variety of implementations just by composing transformations. This would be useful for trying completely new associations of compilation choices and for assessing the implementations and optimizations in practice.

—A last step toward high-quality machine code would be the modeling of register allocation. This could be done via the introduction of another component: a vector of registers.

—A systematic description of standard optimizations and program transformations should be undertaken. A benefit would be to clarify the impact of a program transformation, depending on the implementation choices. Let us consider, for example, λ-lifting, a controversial transformation [Johnsson 1987; Meijer and Paterson 1991]. Intuitively, λ-lifting can be beneficial for implementations using linked environments. Indeed, in this case, its effect is to shorten accesses to variables by performing copies. Whether the gain is worth the cost depends on how many times a variable is accessed. We believe that this question could be studied and settled in our framework. Also, proving the correctness of optimizations based on static analyses is a difficult (and largely neglected) problem [Burn and Le Métayer 1996]. Expressing these optimizations as program transformations in our unified framework should make this task easier.

—Another research direction is the design of hybrid transformations (mixing several compilation schemes). We hinted at a solution to mix copied and linked environments in Section 8.3 and a solution to mix the eval-apply and the push-enter model in Douence and Fradet [1996b]. Other hybrid transformations as well as the analyses needed to make these transformations worthwhile have yet to be devised. Without the help of a formal framework, such transformations would be quite difficult to design and prove correct. The description of previously unknown compositions of transformations, the mechanical derivation of new abstraction algorithms, and hybrid transformations all indicate that our approach can also suggest new implementation techniques.

—Many interesting formal comparisons of transformations remain to be done. At the moment, we have just compared a few couples of transformations ($\mathcal{V}a$ and $\mathcal{V}m$, $\mathcal{N}a$ and $\mathcal{N}m$ [Douence and Fradet 1996b], $\mathcal{A}s$ and $\mathcal{A}c1$). It might be the case that a specific choice for a compilation step designates a best candidate for the compilation of another step. This could be established by comparing compositions of transformations (e.g., $\mathcal{A}s \cdot \mathcal{V}a$ and $\mathcal{A}c1 \cdot \mathcal{V}a$).

We believe that the accomplished work already shows that our framework is expressive and powerful enough to tackle these problems.

ACKNOWLEDGMENTS

We are grateful to Luke Hornof and Daniel Le Métayer for their comments on several versions of this article. Thanks are also due to the referees who provided helpful suggestions.

ONLINE-ONLY APPENDIX

Appendices A–P are available online only and are retrievable from the citation page for this article:

http://www.acm.org/pubs/citations/journals/toplas/1998-20-2/p344-douence

Alternative instructions on how to obtain online-only appendices are given on the back inside cover of current issues of ACM TOPLAS or on the ACM TOPLAS web page:

http://www.acm.org/toplas

REFERENCES

ABADI, M., CARDELLI, L., CURIEN, P. L., AND LEVY, J. J. 1990. Explicit substitutions. In *Proceedings of POPL 1990*. ACM, New York, 31–46.

APPEL, A. W. 1992. *Compiling with Continuations*. Cambridge University Press, Cambridge, Mass.

ARGO, G. 1989. Improving the three instruction machine. In *Proceedings of FPCA '89*, 100–115.

ASPERTI, A. 1992. A categorical understanding of environment machines. *J. Funct. Program. 2*, 1, 23–59.

BARENDREGT, H. P. 1981. *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam.

BARENDREGT, H. P. AND HEMERIK, K. 1990. Types in lambda calculi and programming languages. In *Proceedings of the 3rd European Symposium on Programming*. Lecture Notes in Computer Science, vol. 432. Springer-Verlag, Berlin, 1–35.

BENAISSA, Z., LESCANNE, P., AND ROSE, K. H. 1996. Modeling sharing and recursion for weak reduction strategies using explicit substitution. In *Proceedings of PLILP 1996*, 393–407.

BURN, G. AND LE MÉTAYER, D. 1996. Proving the correctness of compiler optimisations based on a global analysis. *J. Funct. Program. 6*, 1, 75–110.

BURN, G., PEYTON JONES, S. L., AND ROBSON, J. D. 1988. The spineless G-machine. In *Proceedings of LFP '88*, 244–258.

COUSINEAU, G., CURIEN, P.-L., AND MAUNY, M. 1987. The categorical abstract machine. *Sci. Comput. Program. 8*, 2, 173–202.

CRÉGUT, P. 1991. Machines à environnement pour la réduction symbolique et l'évaluation partielle. Ph.D. thesis, Université Paris VII, Paris, France.

CURIEN, P. L. 1993. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Birkhauser.

CURRY, H. B. AND FEYS, R. 1958. *Combinatory Logic*, Vol. 1, North-Holland.

DANVY, O. 1992. Back to direct style. In *Proceedings of ESOP '92*. Lecture Notes in Computer Science, vol. 582. Springer-Verlag, Berlin, 130–150.

DE BRUJIN, N. G. 1972. λ-calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to Church–Rosser theorem. *Indagationes Math. 34*, 381–392.

DOUENCE, R. 1996. Décrire et comparer les mises en œuvre de langages fonctionnels. Ph.D. thesis, University of Rennes I, Rennes, France.

DOUENCE, R. AND FRADET, P. 1995. Towards a taxonomy of functional language implementations. In *Proceedings of PLILP '95*. Lecture Notes in Computer Science, vol. 982. Springer-Verlag, Berlin, 27–44.

DOUENCE, R. AND FRADET, P. 1996a. A taxonomy of functional language implementations. Part I: Call-by-value. INRIA Res. Rep. 2783, INRIA, Rennes, France.

DOUENCE, R. AND FRADET, P. 1996b. A taxonomy of functional language implementations. Part II: Call-by-name, call-by-need, and graph reduction. INRIA Res. Rep. 3050, INRIA, Rennes, France.

FAIRBAIRN, J. AND WRAY, S. 1987. Tim: A simple, lazy abstract machine to execute supercombinators. In *Proceedings of FPCA '87*. Lecture Notes in Computer Science, vol. 274. Springer-Verlag, Berlin, 34–45.

FISCHER, M. J. 1972. Lambda-calculus schemata. In *Proceedings of the ACM Conference on Proving Properties about Programs. SIGPLAN Not. 7*, 1, 104–109.

FRADET, P. AND LE MÉTAYER, D. 1991. Compilation of functional languages by program transformation. *ACM Trans. Program. Lang. Syst. 13*, 1, 21–51.

HALL, C. 1994. Using Hindley–Milner type inference to optimise list representation. In *Proceedings of LFP '94*, 162–172.

HANNAN, J. 1992. From operational semantics to abstract machines. *Math. Struct. Comput. Sci. 2*, 4, 415–459.

HARDIN, T., MARANGET, L., AND PAGANO, B. 1996. Functional back-ends within the lambda-sigma calculus. In *Proceedings of ICFP 1996*, 25–33.

HATCLIFF, J. AND DANVY, O. 1994. A generic account of continuation-passing styles. In *Proceedings of POPL '94*. ACM, New York, 458–471.

JOHNSSON, T. 1987. Compiling lazy functional languages. Ph.D. thesis, Chalmers Univ. of Technology, Göteborg, Sweden.

JOY, M. S., RAYWARD-SMITH, V. J., AND BURTON, F. W. 1985. Efficient combinator code. *Comp. Lang. 10*, 3/4, 211–224.

KLOP, J. W. 1992. Term rewriting systems. In *Handbook of Logic in Computer Science*. Vol. 2, Oxford University Press, New York, 2–108.

KRANZ, D., KESLEY, R., REES, J., HUDAK, P., PHILBIN, J., AND ADAMS, N. 1986. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Not. 21*, 7, 219–233.

LANDIN, P. J. 1964. The mechanical evaluation of expressions. *Comput. J. 6*, 4, 308–320.

LEROY, X. 1990. The Zinc experiment: An economical implementation of the ML language. INRIA Tech. Rep. 117, INRIA, Rennes, France.

LEROY, X. 1992. Unboxed objects and polymorphic typing. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, New York, 177–188.

LESTER, D. 1989. Combinator graph reduction: A congruence and its application. Ph.D. thesis, Oxford Univ., Oxford, U.K.

LIANG, S., HUDAK, P., AND JONES, M. P. 1995. Monad transformers and modular interpreters. In *Proceedings of POPL '95*. ACM, New York, 333–343.

LINS, R. D. 1987. Categorical multi-combinators. In *Proceedings of FPCA '87*. Lecture Notes in Computer Science, vol. 274. Springer-Verlag, Berlin, 60–79.

LINS, R., THOMPSON, S., AND PEYTON-JONES, S. L. 1992. On the equivalence between CMC and TIM. *J. Funct. Program. 4*, 1, 47–63.

MEIJER, E. 1992. Calculating compilers. Ph.D. thesis, Katholieke Universiteit Nijmegen, Netherlands.

MEIJER, E. AND PATERSON, R. 1991. Down with lambda lifting. Unpublished paper. Copies available at: {erik@cs.kun.nl}.

MOGGI, E. 1991. Notions of computation and monads. *Inf. Comput. 93*, 55–92.

PEYTON JONES, S. L. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, N.J.

PEYTON JONES, S. L. 1992. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *J. Funct. Program. 2*, 2, 127–202.

PEYTON JONES, S. L. AND LESTER, D. 1992. *Implementing Functional Languages, A Tutorial*. Prentice-Hall, Englewood Cliffs, N.J.

PEYTON JONES, S. L. AND LAUNCHBURY, J. 1991. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of FPCA '91*. Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Berlin, 636–666.

PEYTON JONES, S. L., PARTAIN, W., AND SANTOS, A. 1996. Let-floating: Moving bindings to give faster programs. In *Proceedings of ICFP '96*, 1–12.

PLASMEIJER, R. AND VAN EEKELEN, M. 1993. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Reading, Mass.

PLOTKIN, G. D. 1975. Call-by-name, call-by-value and the lambda-calculus. *TCS 1*, 2, 125–159.

SCHMIDT, D. 1986. *Denotational Semantics, A Methodology for Language Development*. W. C. Brown, Dubuque, Iowa.
SESTOFT, P. 1994. Deriving a lazy abstract machine. Tech. Rep. 1994-146, Technical Univ. of Denmark.
SHAO, Z. AND APPEL, A. 1994. Space-efficient closure representations. In *Proceedings of LFP '94*, 150–161.
SHAO, Z., REPPY, J., AND APPEL, A. 1994. Unrolling lists. In *Proceedings of LFP '94*, 185–195.
STEELE, G. L. 1978. Rabbit: A compiler for Scheme. Tech. Rep. AI-TR-474, MIT, Cambridge, Mass.
TURNER, D. A. 1979a. A new implementation technique for applicative languages. *Softw. Pract. Exper. 9*, 31–49.
TURNER, D. A. 1979b. Another algorithm for bracket abstraction. *J. Symbol. Logic 44*, 267–270.
WADLER, P. 1992. The essence of functional programming. In *Proceedings of POPL '92*. ACM, New York, 1–14.
WAND, M. 1982. Deriving target code as a representation of continuation semantics. *ACM Trans. Program. Lang. Syst. 4*, 3, 496–517.

Received February 1997; revised July 1997; accepted September 1997