

A Taxonomy of Functional Language Implementations Part II: Call-by-Name, Call-by-Need and Graph Reduction

Rémi Douence, Pascal Fradet

▶ **To cite this version:**

Rémi Douence, Pascal Fradet. A Taxonomy of Functional Language Implementations Part II: Call-by-Name, Call-by-Need and Graph Reduction. [Research Report] RR-3050, INRIA. 1996. inria-00073642

HAL Id: inria-00073642
https://inria.hal.science/inria-00073642

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

A Taxonomy of Functional Language Implementations

Part II : Call-by-Name, Call-by-Need and Graph Reduction

Rémi Douence - Pascal Fradet

N° 3050

Novembre 1996

THEME 2

Génie logiciel et calcul symbolique



ISSN 0249-6399



A Taxonomy of Functional Language Implementations

Part II: Call-by-Name, Call-by-Need and Graph Reduction

Rémi Douence* and Pascal Fradet**

Thème 2 — Génie logiciel et calcul symbolique

Projet Lande

Publication interne n˚3050 — Novembre 1996 — 36 pages

Abstract: In Part I [5], we proposed an approach to formally describe and compare functional languages implementations. We focused on call-by-value and described well-known compilers for strict languages. Here, we complete our exploration of the design space of implementations by studying call-by-name, call-by-need and graph reduction. We express the whole compilation process as a succession of program transformations in a common framework. At each step, different transformations model fundamental choices or optimizations. We describe and compare the diverse alternatives for the compilation of the call-by-name strategy in both environment and graph-based models. The different options for the compilation of β-reduction described in [5] can be applied here as well. Instead, we describe other possibilities specific to graph reduction. Call-by-need is nothing but call-by-name with redex sharing and update. We present how sharing can be expressed in our framework and we describe different update schemes. We finally classify some well-known call-by-need implementations.

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

(Résumé : tsvp)

* douence@irisa.fr
** fradet@irisa.fr

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

Une Taxonomie des implantations des langages fonctionnels

Partie II : Appel par nom, appel par nécessité et réduction de graphe

Résumé : Dans la première partie de ce travail [5], nous avons proposé une approche pour formellement décrire et comparer les implantations de langages fonctionnels. Nous avons appliqué cette approche à l'étude des mises en œuvre de l'appel par valeur. Ici, nous poursuivons notre exploration des techniques de mise en œuvre en étudiant l'appel par nom, l'appel par nécessité et la réduction de graphe. Nous décrivons le processus de compilation comme une suite de transformations de programmes dans le cadre fonctionnel. Les choix fondamentaux de mise en œuvre ainsi que les optimisations s'expriment naturellement comme des transformations différentes. Nous décrivons et comparons les choix de compilation de l'appel par nom dans les modèles à environnement et à réduction de graphes. Les différentes options de compilation de la β-réduction décrits dans [5] restent valides ici. Au lieu de cela, nous décrivons de nouvelles possibilités plus spécifiques à la réduction de graphes. L'appel par nécessité n'est rien d'autre qu'un raffinement de l'appel par nom intégrant le partage et la mise à jour d'expressions. Nous présentons comment le partage peut s'exprimer dans notre cadre et décrivons deux procédés de mise à jour. Enfin, nous cataloguons plusieurs implantations connues de l'appel par nécessité.

Mots-clé : Compilation, optimisations, transformations de programmes, λ-calcul, combinateurs, réduction de graphes.

1 Introduction

In part I [5], we proposed an approach to precisely describe and compare functional languages implementations. We focused on call-by-value and described well-known compilers for strict languages. Here, we complete our exploration of the design space of implementations by studying call-by-name, call-by-need and graph reduction. Our approach is to express in a common framework the whole compilation process as a succession of program transformations. The framework considered is a hierarchy of intermediate languages all of which are subsets of the lambda-calculus. Our description of an implementation consists of a series of transformations $\Lambda \xrightarrow{\tau 1} \Lambda_1 \xrightarrow{\tau 2} \ldots \xrightarrow{\tau m} \Lambda_n$ each one compiling a particular task by mapping an expression from one intermediate language into another. The last language Λ_n consists of functional expressions which can be seen as machine code (essentially, combinators with explicit sequencing and calls). For each step, different transformations are designed to represent fundamental choices or optimizations. A benefit of this approach is to structure and decompose the implementation process. Two seemingly disparate implementations can be found to share some compilation steps.

The two steps which cause the greatest impact on the compiler structure are the implementation of the reduction strategy (searching for the next redex) and the environment management (compilation of β-reduction). For call-by-need, another important step is the implementation of redex sharing and update whereas graph reduction brings the issue of graph representation.

As in part I, we concentrate on pure λ-expressions and our source language Λ is described by the grammar $E ::= x \mid \lambda x.E \mid E_1\, E_2$. Most fundamental choices can be described for this simple language. In section 2, we recall briefly the framework used to model the compilation process. In section 3, we present the alternatives to compile call-by-name. The compilation of control in graph reducers is peculiar. A separate, long sub-section (3.3), is devoted to this point. This section ends with some comparisons, notably a study of the relationship between the compilation of control in the environment and graph-based models. The different options for the compilation of β-reduction described in [5] can be applied here as well. Instead, we describe some other possibilities commonly used with graph reduction (section 4). The problem of the representation of sharing and implementation of updates is addressed in section 5. Finally, in section 6, we classify some well-known call-by-need implementations. Proofs of the properties stated in this paper are sketched in the annex.

Note that this report is not intended to be read independently. Even if important points are briefly recalled, we expect the reader to have some knowledge of Part I [5] or, at least, of its short version [4].

2 General Framework

We quickly review the unified framework used in this paper. A more thorough presentation can be found in [5].

Each compilation step is represented by a transformation from an intermediate language to another one closer to a machine code. The whole implementation process is described via a transformation sequence. The transformation sequence presented in this paper involves four intermediate languages ($\Lambda \rightarrow \Lambda_s \rightarrow \Lambda_e \rightarrow \Lambda_k \rightarrow \Lambda_h$) and describes the whole implementation process.

The first phase is the compilation of control which is described by transformations from Λ to Λ_s. The intermediate language Λ_s is of the form

Λ_s $\quad E ::= x \mid \mathbf{push}_s\ E \mid \lambda_s x.E \mid E_1 \circ E_2$

Intuitively, $\circ$ is a sequencing operator and $E_1 \circ E_2$ can be read "evaluate E_1 then evaluate E_2", $\mathbf{push}_i\ E$ returns E as a result and $\lambda_i x.E$ binds the previous intermediate result to x before evaluating E. The pair (λ_s, $\mathbf{push}_s$) specifies a component storing intermediate results (e.g. a data stack). 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 not relevant anymore (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 β-reduction. The language Λ_e excludes unrestricted uses of variables which are now only needed to define macro-combinators. The encoding of environment management is made possible using a new pair of combinators ($\mathbf{push}_e$, λ_e). They behave exactly as $\mathbf{push}_s$ and λ_s ; they just act on a (at least conceptually) different component (e.g. a stack of environments). So, Λ_e is of the form

Λ_e $\quad E ::= x \mid \mathbf{push}_s\ E \mid \lambda_s x.E \mid \mathbf{push}_e\ E \mid \lambda_e x.E \mid E_1 \circ E_2$

Transformations from Λ_e to Λ_k describe 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 storing return addresses. Since call-by-name or graph reduction do not introduce new choices *w.r.t.* control transfers, we do not describe this compilation step again. A last transformation family ($\mathcal{H}$) from Λ_k to Λ_h adds 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. This last language can be seen as a machine code.

The intermediate languages are subsets of λ-expressions, therefore substitution and the notion of free or bound variables are the same as in λ-calculus. The basic combinators satisfy the following properties:

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

(η_i) $\quad \lambda_i x.(\mathbf{push}_i\ x \circ E) = E$ $\quad$ *if x does not occur free in E*

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

We consider only reduction rules corresponding to the classical β-reduction:

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

As with all standard implementations, we are only interested in modeling weak reductions. Sub-expressions inside $\mathbf{push}_i$'s and λ_i's are not considered as redexes and from here on we write "redex" (resp. reduction, normal form) for weak redex (resp. weak reduction, weak normal form). A key property of the framework is that:

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

Transformations of source programs will produce expressions denoting results (i.e. which can be reduced to expressions of the form $\mathbf{push}_i\ F$). In order to express laws more easily or to distinguish blocks of instructions, it is convenient to restrict Λ_i using a type system (Figure 1).

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

Figure 1 Λ_s typed subset (Λ_s^σ)

The type system restricts expressions $E_1 \circ E_2$ so that E_1 must denote a result (i.e. has type $\mathsf{R}_i\sigma$, R_i being a type constructor) and E_2 must denote a function taking its argument from the *i* component.

The framework enjoys also many algebraic laws useful to transform the functional code or to prove the correctness or equivalence of program transformations such as

if x does not occur free in F	$(\lambda_i x.E) \circ F = \lambda_i x.(E \circ F)$	(L1)
$\forall E_1{:}\mathsf{R}_i\sigma$, *if x does not occur free in* E_2	$E_1 \circ (\lambda_i x.(E_2 \circ E_3)) = E_2 \circ (E_1 \circ (\lambda_i x.E_3))$	(L2)
$\forall E_1{:}\mathsf{R}_i\sigma, E_2{:}\mathsf{R}_j\sigma$ *and* $x \not\equiv y$	$E_1 \circ (E_2 \circ (\lambda_j x.\lambda_i y.E_3)) = E_2 \circ (E_1 \circ (\lambda_i y.\lambda_j x.E_3))$	(L3)

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 pattern-matching $\lambda_i(x_1,\ldots,x_n).E$.

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 do not have to be chosen until the very last step. Nevertheless, in order to provide some intuition, we give here possible definitions in terms of standard λ-expressions.

The most natural definition for the sequencing combinator is $\circ = \lambda xyz.x\ (y\ z)$. The pairs of combinators (λ_i, $\mathbf{push}_i$) can be seen as encoding a component of an underlying abstract machine and their definitions specify the state transitions. We can choose to keep the components separate or merge (some of) them.

Keeping the components separate brings new properties such as

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

allowing code motion and simplifications. Possible definitions (*c, s, e* being fresh variables) follow:

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

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

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

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

Then, the reduction of our expressions can be seen as state transitions of an abstract machine with five components (code, data stack, environment stack, control stack, heap), e.g.:

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

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

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

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

and the reduction of our expressions are of the form

$$\mathbf{push}_i\ N\ C\ Z \rightarrow C\ (Z,N)$$

Let us point out that our use of the term “abstract machines” should not suggest a layer of interpretation. At the end of the compilation process, we get a realistic assembly code and the “abstract machines” resemble real stack machines.

3 Compilation of Control

We focus here on the compilation of the call-by-name reduction strategy. Call-by-need is only a refinement involving redex sharing and update which is described in section 5. We first present the two main choices taken by environment-based implementations: the push-enter and the eval-apply models. Even if the push-enter and the eval-apply models can be adopted by graph reduction as well, these implementations are at first so different that we treat them apart. The graph-based implementations use an interpretative implementation of the reduction strategy and are presented in 3.3.

3.1 The push-enter model

Contrary to call-by-value, the most natural choice to implement call-by-name is the push-enter model. In call-by-name, a function is evaluated only when applied to an argument. They do not have to be considered as results. An application can be compiled by applying the unevaluated function right away to its unevaluated argument. This option is taken by Tim [6], the Krivine abstract machine (Mak) [3] and graph-based implementations.

The transformation $\mathcal{N}m$ formalizes this choice and is described in Figure 2.

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

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

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

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

Figure 2 Compilation of Call-by-Name in the Push-Enter Model ($\mathcal{N}m$)

Variables are bound to arguments which 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 its function.

The correctness of $\mathcal{N}m$ is stated by Property 2 which establishes that the reduction of transformed expressions ($\overset{*}{\blacktriangleright}$) simulates the call-by-name reduction ($\underset{cbn}{\Rightarrow}$) of source λ-expressions.

Property 2 $\forall E \ closed \in \Lambda, E \underset{cbn}{\Rightarrow} V \Leftrightarrow \mathcal{N}m \ [\![E]\!] \overset{*}{\blacktriangleright} \mathcal{N}m \ [\![V]\!]$

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

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

The choice of redex in Λ_s does not matter anymore. The illicit (in call-by-name) reduction $E \rightarrow (\lambda x.x)(\lambda z.z)$ cannot occur within $\mathcal{N}m \ [\![E]\!]$. This redex is no more a weak redex in the Λ_s expression. ❐

If $\mathcal{N}_m$ is arguably the simplest way to compile call-by-name, it makes however 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. As it stands, $\mathcal{N}_m$ makes it impossible to distinguish results of closures (which have to be updated) from regular functions (which are applied right away). We already have encountered this same problem in the compilation of call-by-value within the push-enter model [5]. The solution is similar and consists in using marks. The $\mathcal{N}_{ml}$ transformation (Figure 3) still compiles call-by-name but adds the necessary combinators to deal with marks and updates.

$\mathcal{N}_{ml} : \Lambda \rightarrow \Lambda_s$

$\mathcal{N}_{ml}\ [\![x]\!] = x$

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

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

Figure 3 Compilation of Call-by-Name with Marks in the Push-Enter Model ($\mathcal{N}_{ml}$)

For call-by-name, it is sufficient that $\mathbf{grab}_s\ E$ verifies the reduction rule

$$\mathbf{push}_s\ V \circ \mathbf{grab}_s\ E \rightarrow \mathbf{push}_s\ V \circ E$$

and $\mathcal{N}_{ml}$ would verify the analogue of Property 2. For call-by-need, we introduce a mark ε supposed to be a distinguishable value along with the reduction rule

$$\mathbf{push}_s\ \varepsilon \circ \mathbf{grab}_s\ E \rightarrow \mathbf{push}_s\ E$$

Combinator $\mathbf{grab}_s$ and the mark ε are identical to those introduced to compile call-by value in [5]. They can be defined in Λ_s but, in practice, $\mathbf{grab}_s$ would be implemented using a conditional which tests the presence of a mark.

It is now easy to insert updates. For example, a callee update scheme can take the form:

$\mathcal{U}_{callee}\ [\![\mathbf{push}_s\ E]\!] = \mathbf{push}_s\ (\mathbf{push}_s\ @ \circ \mathbf{push}_s\ \varepsilon \circ \mathcal{U}_{callee}\ [\![E]\!] \circ \mathbf{updt} \circ \mathbf{resume}_s)$

with $\mathbf{resume}_s = \lambda_s x.\mathbf{grab}_s\ x$

The closure stores its own address and a mark before its evaluation. The normal form of the closure, of the form $\mathbf{grab}_s(\lambda_s x.E)$, takes the mark and passes $(\lambda_s x.E)$ to **updt** which performs the update according to the address, then $\mathbf{resume}_s$ resumes the global reduction. The different options for updating closures are detailed in section 5.

3.2 The eval-apply model

In this scheme, applications $E_1\ E_2$ are compiled by returning E_2, evaluating E_1 and finally applying the evaluated function to the unevaluated argument. So, this choice considers λ-abstractions as results and is formalized by the transformation $\mathcal{N}_a$ in Figure 4.

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

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

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

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

Figure 4 Compilation of Call-by-Name in the Eval-Apply Model ($\mathcal{N}a$)

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

Property 3 $\forall E \ closed \in \Lambda, E \underset{cbn}{\Rightarrow} V \Leftrightarrow \mathcal{N}a \ [\![E]\!] \overset{*}{\rightarrow} \mathcal{N}a \ [\![V]\!]$

It is clearly useless to store a function to apply it immediately after. This optimization is expressed by the same law as the one used to simplify expressions produced by $\mathcal{V}a$ [5].

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

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

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

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

Like $\mathcal{N}m$, transformation $\mathcal{N}a$ may produce expressions such as $\mathbf{push}_s \ E_1 \circ \ldots \circ \mathbf{push}_s \ E_n$ which require a stack to store intermediate results. To get a stackless variant of $\mathcal{N}a$, the rule for compositions should be changed into:

$\mathcal{N}a_{sf} \ [\![E_1 \ E_2]\!] = \mathbf{push}_s \ (\mathcal{N}a_{sf} \ [\![E_2]\!]) \circ (\lambda_s a.\mathcal{N}a_{sf} \ [\![E_1]\!] \circ (\lambda_s f.\mathbf{push}_s \ a \circ f))$

With this variant, the component on which $\mathbf{push}_s$ and λ_s act may be a single register.

The implementation of call-by-need does not require any modification of $\mathcal{N}a$. For example, a caller update scheme can take the form

$$\mathcal{N}af \ [\![x]\!] = x \circ \mathbf{updt}$$

The result of a closure evaluation is of the form $\mathbf{push}_s(\lambda_s x.E)$ and is thus passed to **updt** (see section 5.2).

3.3 Graph Reduction

Graph-based implementations manipulate a graph representation of the source λ-expression. The reduction consists in rewriting the graph more or less interpretatively. 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 im-

plementations consider only 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. Others choices such as call-by-value graph reducers and the eval-apply model are briefly described in section 3.3.4.

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 and its reduction via an interpreter. The transformation $\mathcal{G}$ (Figure 5) produces an expression which builds a graph (for now, only a tree) when reduced. This transformation is common to all the graph reduction schemes we describe afterwards.

$\mathcal{G} : \Lambda \rightarrow \Lambda_s$

$\mathcal{G}\ [\![x]\!] = \mathbf{push}_s\ x \circ \mathbf{mkVar}_s$

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

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

Figure 5 Generic Graph Building Code ($\mathcal{G}$)

The three new combinators $\mathbf{mkVar}_s$, $\mathbf{mkFun}_s$ and $\mathbf{mkApp}_s$ take their arguments on the s component and return graph nodes (respectively variable, function and application nodes) on s. 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 $\mathcal{G}\ [\![E]\!] \circ \mathbf{unwind}_s$. In this setting, the push-enter and eval-apply models of compilation of call-by-value and call-by-name are specified by defining the interactions of $\mathbf{unwind}_s$ with the three graph builders ($\mathbf{mkVar}_s$, $\mathbf{mkFun}_s$, $\mathbf{mkApp}_s$).

These combinators can take several definitions implying different concrete representations of the graph. We present one possible definition in the next section but several others are discussed in section 3.3.3.

3.3.2 Call-by-name: the push-enter model

This option is defined by the three properties in Figure 6.

($\mathcal{GNm}$ 1) $\quad (E \circ \mathbf{mkVar}_s) \circ \mathbf{unwind}_s = E \circ \mathbf{unwind}_s$

($\mathcal{GNm}$ 2) $\quad V \circ (\mathbf{push}_s\ F \circ \mathbf{mkFun}_s) \circ \mathbf{unwind}_s = (V \circ F) \circ \mathbf{unwind}_s$

($\mathcal{GNm}$ 3) $\quad (E_2 \circ E_1 \circ \mathbf{mkApp}_s) \circ \mathbf{unwind}_s = E_2 \circ E_1 \circ \mathbf{unwind}_s$

Figure 6 Properties of Graph Combinators for the Call-by-Name Push-Enter Model

These properties can be explained intuitively as:

- ($\mathcal{GNm}$ 1) The reduction of a variable node amounts to reduce the graph which 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 [14], it represents indirection nodes.
- ($\mathcal{GNm}$ 2) The reduction of a function node amounts to apply the function to its argument and to reduce 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.
- ($\mathcal{GNm}$ 3) The reduction of an application node amounts to store the argument graph and to reduce the function graph.

Figure 7 presents one possible instance of the graph combinators.

$\mathbf{mkVar}_s = \lambda_s x.\mathbf{push}_s\ x$

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

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

$\mathbf{unwind}_s = \mathbf{app} = \lambda_s x.x$

Figure 7 Instantiation of Graph Combinators According to $\mathcal{GNm}$ (Option Node-as-Code)

Here, the graph is not encoded by data structures but by code performing the needed actions. It simplifies the interpreter which just has to trigger a code; that is, $\mathbf{unwind}_s$ boils down to an application. It is easy to check that these definitions verify the conditions ($\mathcal{GNm}$ 1), ($\mathcal{GNm}$ 2), and ($\mathcal{GNm}$ 3). Moreover, the definition of $\mathbf{mkVar}_s$ (the identity function in Λ_s) makes clear that indirection chains can be collapsed. That is to say:

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

With this combinator instantiation, the graph is represented by closures. A more classical representation, based on data structures, arises with instantiations presented in section 3.3.3. The correctness of $\mathcal{GNm}$ is stated by Property 4.

Property 4 $\forall E\ closed \in \Lambda,\ E \xrightarrow{cbn} V \Rightarrow \mathcal{G}\ [\![E]\!] \circ \mathbf{unwind}_s = \mathcal{G}\ [\![V]\!] \circ \mathbf{unwind}_s$

The equality symbol (instead of a reduction symbol) on the *rhs* of the implication comes from the indirections which may remain in the normal form of $\mathcal{G}\ [\![E]\!] \circ \mathbf{unwind}_s$. Actually, a more precise property would be: $\mathcal{G}\ [\![E]\!] \circ \mathbf{unwind}_s$ reduces to an expression X which after removal of indirection chains is syntactically equal to the graph of $\mathcal{G}\ [\![V]\!] \circ \mathbf{unwind}_s$.

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

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

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

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

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

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

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

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

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 $\mathcal{G}\ [\![\lambda z.z]\!] \circ \mathbf{unwind}_s$ 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 $\mathbf{unwind}_s$. ❐

A naive implementation of call-by-need is possible without introducing marks. The graph construction and reduction are distinct operations. This makes it possible to systematically update the graph [9]. For example, the reduction of an application node would store its address. The definition of $\mathbf{mkApp}_s$ would be of the form

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

The address would be updated after the construction of the reduced graph, that is

$$\mathbf{mkFun}_s = \lambda_s f.\mathbf{push}_s\ (\lambda_s x.\lambda_s @.\mathbf{push}_s\ @ \circ (\mathbf{push}_s\ x \circ f) \circ \mathbf{updt} \circ \mathbf{unwind}_s)$$

This update scheme is the G-Machine's and is described in section 5.3.1.

Such a scheme performs many useless updates some of which can be detected by simple syntactic criteria or a sharing analysis. In order to benefit from this kind of information, it must be possible to perform selective updates. However, like with $\mathcal{N}m$, this facility imposes the introduction of marks [1]. The combinator $\mathbf{mkFun}_s$ is changed to test the presence of a mark using $\mathbf{grab}_s$.

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

Updatable/shared nodes can now be distinguished using a new kind of application node $\mathbf{mkAppS}_s$ defined as

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

The role of the mark ε is to suspend the reduction before the update (see section 5.3.2).

3.3.3 Alternate graph representations

A graph and its associated reducer can be seen as an abstract data type with different implementations [11]. We have already used one encoding which represents nodes by code (i.e. closures). We present here two others solutions. Both of them verify properties ($\mathcal{GNm}$ 1), ($\mathcal{GNm}$ 2) and ($\mathcal{GNm}$ 3) and, therefore, implement a push-enter model of the compilation of call-by-name.

A first natural solution is to represent the graph by a (for now) tree-like data structure. We introduce three data constructors **VarNode**, **FunNode** and **AppNode** and the interpreter $\mathbf{unwind}_s$ is defined as a case. Their definitions is given in Figure 8 in an Haskell-like syntax.

$\mathbf{mkVar}_s = \lambda_s x.\mathbf{push}_s\ (\mathbf{VarNode}\ x)$

$\mathbf{mkFun}_s = \lambda_s f.\mathbf{push}_s\ (\mathbf{FunNode}\ f)$

$\mathbf{mkApp}_s = \lambda_s x_1.\lambda_s x_2.\ \mathbf{push}_s\ (\mathbf{AppNode}\ x_1\ x_2)$

$\mathbf{unwind}_s = \lambda_s n.\ \mathbf{case}\ n\ \mathbf{of}$

$(\mathbf{VarNode}\ e) \rightarrow \mathbf{push}_s\ e \circ \mathbf{unwind}_s$

$(\mathbf{FunNode}\ e) \rightarrow \lambda_s a.(\mathbf{push}_s\ a \circ e) \circ \mathbf{unwind}_s$

$(\mathbf{AppNode}\ e_1\ e_2) \rightarrow \mathbf{push}_s\ e_2 \circ \mathbf{push}_s\ e_1 \circ \mathbf{unwind}_s$

Figure 8 Instantiation of Graph Combinators According to $\mathcal{GNm}$ (Option Node-as-Constructor)

The second solution is a refinement which is used by the G-machine. Each node encloses in addition the code to be executed when it is unwound. The interpreter $\mathbf{unwind}_s$ just executes this code and does not have to perform a dynamic test.

$\mathbf{mkVar}_s = \lambda_s x.\mathbf{push}_s\ (\mathbf{tagVar}_s, x)$ with $\mathbf{tagVar}_s = \lambda_s x.\mathbf{push}_s\ x \circ \mathbf{unwind}_s$

$\mathbf{mkFun}_s = \lambda_s f.\mathbf{push}_s\ (\mathbf{tagFun}_s, f)$ with $\mathbf{tagFun}_s = \lambda_s f.\lambda_s x.(\mathbf{push}_s\ x \circ f) \circ \mathbf{unwind}_s$

$\mathbf{mkApp}_s = \lambda_s x_1.\lambda_s x_2.\mathbf{push}_s\ (\mathbf{tagApp}_s, (x_1, x_2))$

with $\mathbf{tagApp}_s = \lambda_s(x_1,x_2).\mathbf{push}_s\ x_2 \circ \mathbf{push}_s\ x_1 \circ \mathbf{unwind}_s$

$\mathbf{unwind}_s = \lambda_s(tag, data).\mathbf{push}_s\ data \circ tag$

Figure 9 Instantiation of Graph Combinators According to $\mathcal{GNm}$ (Option G-Machine)

This representation may appear very different of the first option presented in Figure 7. However, at the end of the compilation process, both solutions come close. For example, af-

ter the compilation of β-reduction, the "node-as-code" option represents nodes by closures which resemble data-structures used by the G-machine. The efficiency of these two options is comparable.

In fact, the G-machine uses a slightly more complex representation. Instead of the address of a code, each node contains the address of a jump table. The reason is that, in the G-machine, many operations (other than $\mathbf{unwind}_s$) can be applied to a node (e.g. "eval" to reduce strictly the graph or "print" to display it). These operations are defined for each type of node and stored in tables.

3.3.4 Graph reduction, eval-apply model and call-by-value

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. We examine each problem in turn. In both cases, the same graph has to be built and the transformation $\mathcal{G}$ (of section 3.3.1) is reused. Only the graph reduction (expressed by the combinators properties) has to be adapted.

The eval-apply model for a call-by-name graph reducer can be defined by the properties of Figure 10.

($\mathcal{GN}a$ 1) $\quad (E \circ \mathbf{mkVar}_s) \circ \mathbf{unwind}_s = E \circ \mathbf{unwind}_s$

($\mathcal{GN}a$ 2) $\quad (\mathbf{push}_s\ F \circ \mathbf{mkFun}_s) \circ \mathbf{unwind}_s = \mathbf{push}_s\ F$

($\mathcal{GN}a$ 3) $\quad (E_2 \circ E_1 \circ \mathbf{mkApp}_s) \circ \mathbf{unwind}_s = E_2 \circ (E_1 \circ \mathbf{unwind}_s) \circ \mathbf{appFun}_s$

($\mathcal{GN}a$ 4) $\quad E \circ (\mathbf{push}_s\ F) \circ \mathbf{appFun}_s = (E \circ F) \circ \mathbf{unwind}_s$

Figure 10 Properties of Graph Combinators for the Call-by-Name Eval-Apply Model

The rule ($\mathcal{GN}a$ 2) makes it clear that a function is considered as a result (there is no $\mathbf{unwind}_s$ in the *rhs* to continue the reduction). The rule ($\mathcal{GN}a$ 3) specifies that application nodes must be reduced by evaluating the function E_1 and applying ($\mathbf{appFun}_s$) the result to the unevaluated argument E_2. Property ($\mathcal{GN}a$ 4) defines the new application combinator $\mathbf{appFun}_s$. The "node-as-code" instantiation of the graph combinators is described in Figure 11.

$\mathbf{mkVar}_s = \lambda_s x.\mathbf{push}_s\ x$ $\qquad$ $\mathbf{mkFun}_s = \lambda_s f.\mathbf{push}_s\ (\mathbf{push}_s\ f)$

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

$\mathbf{appFun}_s = \mathbf{app} \circ \mathbf{unwind}_s$ $\qquad$ $\mathbf{unwind}_s = \mathbf{app} = \lambda_s x.x$

Figure 11 Instantiation of Graph Combinators According to $\mathcal{GN}a$ (Option Node-as-Code)

In the definition of **appFun**$_s$, **app** calls the function of the **mkFun**$_s$ node which builds a graph whose reduction is triggered by **unwind**$_s$. It is easy to check that Property 4 holds with these definitions as well.

The eval-apply model for a call-by-value graph reducer can be defined by the following properties.

($\mathcal{GV}a1$) (**push**$_s$ E o **mkVar**$_s$) o **unwind**$_s$ = **push**$_s$ E

($\mathcal{GV}a2$) (**push**$_s$ F o **mkFun**$_s$) o **unwind**$_s$ = **push**$_s$ F

($\mathcal{GV}a3$) (E_2 o E_1 o **mkApp**$_s$) o **unwind**$_s$ = E_2 o **unwind**$_s$ o E_1 o **unwind**$_s$ o **appFun**

($\mathcal{GV}a4$) E o **push**$_s$ F o **appFun**$_s$ = (E o F) o **unwind**$_s$

Figure 12 Properties of Graph Combinators for the Call-by-Value Eval-Apply Model

Functions and variables, which are bound to evaluated values, are returned as results (($\mathcal{GV}a1$), ($\mathcal{GV}a2$)). The two **unwind**$_s$ in rule ($\mathcal{GV}a1$) express the evaluation of the function and its argument before application.

3.4 Comparisons

As we already shown in [5], it is possible to compare the complexity of each compilation step. Here, we compare briefly the complexity of $\mathcal{N}a$ and $\mathcal{N}m$. Then, we exhibit the precise relationship between the environment and graph approaches. In particular, we show how to derive the transformation $\mathcal{N}m$ from $\mathcal{G}$ and the properties ($\mathcal{GN}m$).

3.4.1 Push-enter *versus* eval-apply

The $\mathcal{N}m$ scheme is simpler than $\mathcal{N}a$. It avoids the overhead of returning a function that always has to be applied in call-by-name. $\mathcal{N}m$ builds less closures and is more space and time efficient than the corresponding $\mathcal{N}a$ code. The size of the transformed expressions gives an approximation of the overhead entailed by the encoding of the control. It is easy to show that $\mathcal{N}a$ and $\mathcal{N}m$ produce a linear code expansion with respect to the size of the source expression. More precisely:

$$\text{if } \mathcal{S}ize\,(E) = n \text{ then } \mathcal{S}ize\,(\mathcal{N}a\ [\![E]\!]) \leq 3n_v+n_\lambda-2$$

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 $n_v=n-1$. This upper bound can be approached with, for example, $\lambda x_1.x_1\ \ldots\ x_1$. For $\mathcal{N}m$, the size of transformed expressions is strictly smaller. More precisely:

$$\text{if } \mathcal{S}ize\,(E) = n \text{ then } \mathcal{S}ize\,(\mathcal{N}m\ [\![E]\!]) \leq 2n_v+n_\lambda-1$$

However, when call-by-need is considered, $\mathcal{N}_{ml}$ must introduce marks and dynamic tests in order to be able to update redexes. We have

$$\text{if } \mathcal{S}_{ize}(E) = n \text{ then } \mathcal{S}_{ize}(\mathcal{N}_{ml}[\![E]\!]) \leq 2n_v + 2n_\lambda - 1$$

On the other hand, $\mathcal{N}_a$ can directly insert update instructions. The comparison between $\mathcal{N}_a$ and $\mathcal{N}_{ml}$ is very similar to the comparison between $\mathcal{V}_a$ and $\mathcal{V}_m$ [5]. $\mathcal{N}_a$ and $\mathcal{N}_{ml}$ perform the same updates, however a code produced by $\mathcal{N}_{ml}$ builds less closures than the corresponding $\mathcal{N}_a$ code at the price of dynamic tests. A mark can be represented by one bit and $\mathcal{N}_{ml}$ is likely to be, on average, less greedy on space resources. As with call-by-value, time efficiency depends a lot of the respective costs of closure buildings and combinators such as $\mathbf{grab}_s$ but also of the proportion of updatable closures. Many implementations use analysis to detect unshared redexes which has not to be updated. The more redexes are detected unshared, the less closures will have to be built by $\mathcal{N}_{ml}$. As in [5], a more detailed comparison could be conducted using symbolic costs and probabilities.

The graph reduction schemes $\mathcal{GN}_m$ and $\mathcal{GN}_a$ share the same differences as $\mathcal{N}_{ml}$ with $\mathcal{N}_a$. However, the overhead induced by the graph construction and the interpreted reduction is likely to mitigate these differences.

3.4.2 Graph *versus* environment

Even if their starting points are utterly different, graph reducers and environment machines can be related. We illustrate this fact by comparing 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 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 of graph building for 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 to change property ($\mathcal{GN}_m$ 1) by

$$(\mathcal{GN}_m\ 1) \qquad (\mathbf{push}_s E \circ \mathbf{mkVar}_s) \circ \mathbf{unwind}_s = E$$

We can now show that $\mathcal{N}_m\ [\![E]\!]$ is nothing else that the specialization of $\mathbf{unwind}_s$ with respect to the graph of E; that is:

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

This property can be shown by structural induction.

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

  $= \mathbf{push}_s\ x \circ \mathbf{mkVar}_s \circ \mathbf{unwind}_s\ = x = \mathcal{N}_m\ [\![x]\!]$ (def.$\mathcal{G}$),($\mathcal{GN}_m$ 1),(def.$\mathcal{N}_m$)

- Note that ($\mathcal{GN}_m$ 2) holds for all expressions V and could be written

  $$\lambda_s v.\ \mathbf{push}_s\ v \circ (\mathbf{push}_s\ F \circ \mathbf{mkFun}_s) \circ \mathbf{unwind}_s = \lambda_s v.\ (\mathbf{push}_s\ v \circ F) \circ \mathbf{unwind}_s$$

  Using (assoc) and (η_s) we get $(\mathbf{push}_s\ F \circ \mathbf{mkFun}_s) \circ \mathbf{unwind}_s = F \circ \mathbf{unwind}_s$ ($\mathcal{GN}_m$ 2')

  So $\mathcal{G}\ [\![\lambda x.E]\!] \circ \mathbf{unwind}_s$

  $= (\mathbf{push}_s(\lambda_s x.\mathcal{G}\ [\![E]\!])\ \circ \mathbf{mkFun}_s) \circ \mathbf{unwind}_s$ (def. $\mathcal{G}$)

  $= (\lambda_s x.\mathcal{G}\ [\![E]\!])\ \circ \mathbf{unwind}_s = (\lambda_s x.\mathcal{G}\ [\![E]\!] \circ \mathbf{unwind}_s)$ ($\mathcal{GN}_m$ 2'), (L1)

  $= (\lambda_s x.\mathcal{N}_m\ [\![E]\!]) = \mathcal{N}_m\ [\![\lambda x.E]\!]$ (induction hypothesis),(def. $\mathcal{N}_m$)

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

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

  $= \mathbf{push}_s\ (\mathcal{G}\ [\![E_2]\!] \circ \mathbf{unwind}_s) \circ \mathcal{G}\ [\![E_1]\!] \circ \mathbf{unwind}_s$ ($\mathcal{GN}_m$ 3)

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

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

One can show similar properties between the environment based transformations $\mathcal{N}_{ml}$, $\mathcal{N}_a$, $\mathcal{V}_m$, and $\mathcal{V}_a$ and the corresponding graph reducers $\mathcal{GN}_{ml}$, $\mathcal{GN}_a$, $\mathcal{GV}_m$, and $\mathcal{GV}_a$.

These properties show that, as far as the compilation of control is concerned, environment based transformations are more efficient than their graph counterpart. However, optimized graph reducers avoid as much as possible interpretative scans of the graph or graph building and come close 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 in replacing variables by combinators. Compared to Λ_s, Λ_e adds the pair ($\mathbf{push}_e$, λ_e) encoding a new environment component *e*, and uses variables only to define combinators.

The different options for the compilation of β-reduction described in [5] can be applied for call-by-name as well. We do not recall them here. Instead, we focus on abstraction algorithms usually used by graph reduction. One of the most claimed advantage of graph reduction is to be well suited to parallel execution because it has no global environment structure. So, in our framework, the transformations modeling the compilation of β-reduction for graph reducers should not use the *e* component. We present two transformations to model the environment management used in the G-machine[9] and in the SKI-machine[14]. We first introduce general transformations which can be used along with any previous compilation of control. Then, we specialize them to model accurately model existing graph reduction implementations. Finally, we briefly compare them.

4.1 A G-machine-like abstraction algorithm

The G-machine uses an abstraction algorithm close to the transformation $\mathcal{Ag}_{\text{dsb}}$ already described in [5]. The transformation $\mathcal{Ag}_{\text{dsb}}$ specifies an environment manipulation scheme avoiding stack elements reordering ($\mathbf{swap}_{se}$-less), environment duplication ($\mathbf{dupl}_e$-less), and environment building (**mkbind**-less). The environments are unfolded (as sequences of closures) in the stack at a fixed place where they grow and shrink according to the binding scope. We present here a slight variation of $\mathcal{Ag}_{\text{dsb}}$ (Figure 13) which uses the *s* component to store the environment instead of *e*. This transformation introduces indexed combinators and uses the notion of arity:

Definition *An expression E of type* $\sigma_1 \rightarrow_s \ldots \rightarrow_s \sigma_n \rightarrow_s \mathbf{R}_s\sigma$*, is said to have arity n.*

The transformation uses a compile time environment ρ, and an index *k* which represents the number of closures stacked on the current environment (i.e. the environment depth). The main call for a closed expression *E* of arity *p* is: $\mathcal{Ag}_{\text{dsb}'}\ [\![E]\!]()\ p$. In the following, *m* denotes the length of the environment ρ.

$\mathcal{Ag}_{\text{dsb}'} : \Lambda_s \rightarrow env \rightarrow int \rightarrow \Lambda_e$

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

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

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

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

Figure 13 Abstraction Algorithm Dupl-less, Swap-less and Bind-less ($\mathcal{Ag}_{\text{dsb}'}$)

$\mathcal{Ag}_{\text{dsb'}}$ needs five new combinators to express the environment manipulations. In the first rule, both elements of a sequence use the same environment ρ, but E_1 produces a result, so the environment rank of E_2 is k+1. The second rule builds a closure (**mkclos**), which inserts the environment under the arguments (**stores**), evaluates E, then flushes the environment (**flushs**). The rule for λ_s-abstractions adds a new binding to the environment (**stores**), evaluates E, then removes it (**flushs**). Finally, the last rule accesses a closure (**dupls**) and calls it (**appclos**). Figure 13 gives the definition of these (macro)combinators in Λ_s.

$$\mathbf{stores}_{n,m} = \lambda_s y_1 \ldots \lambda_s y_m . \lambda_s x_1 \ldots \lambda_s x_n . \mathbf{push}_s\ y_m \circ \ldots \circ \mathbf{push}_s\ y_1 \circ \mathbf{push}_s\ x_n \circ \ldots \circ \mathbf{push}_s\ x_1$$

$$\mathbf{flushs}_{n,m} = \lambda_s x_1 \ldots \lambda_s x_n . \lambda_s y_1 \ldots \lambda_s y_m . \mathbf{push}_s\ x_n \circ \ldots \circ \mathbf{push}_s\ x_1$$

$$\mathbf{dupls}_{n,m} = \lambda_s x_1 \ldots \lambda_s x_n . \lambda_s y_1 \ldots \lambda_s y_m . \mathbf{push}_s\ y_m \circ \ldots \circ \mathbf{push}_s\ y_1 \circ$$
$$\mathbf{push}_s\ x_n \circ \ldots \circ \mathbf{push}_s\ x_1 \circ \mathbf{push}_s\ y_m \circ \ldots \circ \mathbf{push}_s\ y_1$$

$$\mathbf{mkclos}_{k,m} = \lambda_s c . \lambda_s x_1 \ldots \lambda_s x_k . \lambda_s y_1 \ldots \lambda_s y_m . \mathbf{push}_s\ y_m \circ \ldots \circ \mathbf{push}_s\ y_1 \circ$$
$$\mathbf{push}_s\ x_k \circ \ldots \circ \mathbf{push}_s\ x_1 \circ \mathbf{push}_s\ (\mathbf{push}_s\ y_m \circ \ldots \circ \mathbf{push}_s\ y_1 \circ c)$$

$$\mathbf{appclos} = \mathbf{app}$$

Figure 14 $\mathcal{Ag}_{\text{dsb'}}$ **Indexed Combinators Definitions**

The difference between $\mathcal{Ag}_{\text{dsb'}}$ and $\mathcal{Ag}_{\text{dsb}}$ lies in $\mathbf{mkclos}_{k,m}$ which does not fold an environment in e. The proofs of correctness are similar.

Example. Let us consider the expression $\lambda_s x_1 . \lambda_s x_0 . \mathbf{push}_s\ E \circ x_1$, with p the arity of E, then

$$\mathcal{Ag}_{\text{dsb'}}\ [\![\lambda_s x_1 . \lambda_s x_0 . \mathbf{push}_s\ E \circ x_1]\!]\ ()\ 2$$
$$= \mathbf{stores}_{1,1} \circ \mathbf{stores}_{1,1} \circ \mathbf{push}_s(\mathbf{stores}_{p,2} \circ \mathcal{Ag}_{\text{dsb'}}\ [\![E]\!]\ (((),x_1),x_0)\ p \circ \mathbf{flushs}_{1,2}) \circ \mathbf{mkclos}_{0,2}$$
$$\circ\ \mathbf{dupls}_{2,1} \circ \mathbf{appclos} \circ \mathbf{flushs}_{2,1} \circ \mathbf{flushs}_{1,1}$$

The (originally empty) environment is under the two arguments of the expression. These two arguments are added to the environment ($\mathbf{stores}_{1,1} \circ \mathbf{stores}_{1,1}$). Then a closure of E is built ($\mathbf{mkclos}_{0,2}$). The value of x_1 is accessed ($\mathbf{dupls}_{2,1}$) and evaluated (**appclos**). Finally, x_0 then x_1 are removed ($\mathbf{flushs}_{2,1} \circ \mathbf{flushs}_{1,1}$) from the environment which is now empty and (conceptually) under the result of $\mathbf{push}_s\ E \circ x_1$. ❒

This transformation can easily be optimized. For example, variables are bound to closures. With the original rules, $\mathcal{Ag}_{\text{dsb'}}\ [\![\mathbf{push}_s\ x]\!]$ would build yet another closure. This useless "boxing" can be avoided by the following rule:

$$\mathcal{Ag}_{\text{dsb'}}\ [\![\mathbf{push}_s\ x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0)\ k = \mathbf{dupls}_{k+m-i,1}$$

Others optimizations are described in [5].

4.2 A SKI-like abstraction algorithm

Some abstraction algorithms do not use the environment notion, but encode separately every substitution. A simple algorithm [14] uses only three combinators {**S**, **K**, **I**} but is inefficient *w.r.t.* code expansion. Different refinements, which use extended combinators families (e.g. {**S**, **K**, **I**, **B**, **C**, **S'**, **B'**, **C'**}), have been proposed [15] [2] [10]. They usually lower the complexity of code expansion from exponential with {**S**, **K**, **I**} to quadratic or even O(*nlogn*). We describe only the SKI abstraction algorithm in our framework. It should be clear that the optimized versions could be expressed as easily.

The transformation $\mathcal{SKI}\ [\![E]\!]\ x$ suppresses the free occurrences of x in the Λ_s-expression E using the combinators set {**Ss**, **Ks**, **Is**}. Their role is to distribute and propagate lazily the values in the compiled expression.

$\mathcal{SKI} : \Lambda_s \rightarrow var \rightarrow \Lambda_e$

$\mathcal{SKI}\ [\![E]\!]\ x = \mathbf{push}_s\ E \circ \mathbf{Ks}$, if E does not contain free occurrences of x

$\mathcal{SKI}\ [\![E_1 \circ E_2]\!]\ x = \mathbf{push}_s\ (\mathcal{SKI}\ [\![E_1]\!]\ x) \circ \mathbf{push}_s\ (\mathcal{SKI}\ [\![E_2]\!]\ x) \circ \mathbf{Ss}$

$\mathcal{SKI}\ [\![\mathbf{push}_s\ E]\!]\ x = \mathbf{push}_s\ (\mathcal{SKI}\ [\![E]\!]\ x) \circ \mathbf{mkPush}$

$\mathcal{SKI}\ [\![\lambda_s y.E]\!]\ x = \mathcal{SKI}\ [\![\mathcal{SKI}\ [\![E]\!]\ y]\!]\ x$

$\mathcal{SKI}\ [\![x]\!]\ x = \mathbf{Is}$

Figure 15 Abstraction Algorithm SKI ($\mathcal{SKI}$)

An argument is either propagated to the sub-expressions (**Ss**), cancelled (**Ks**) or kept (**Is**). A fourth combinator (**mkPush**) propagates the argument inside $\mathbf{push}_s$'s. These combinators are defined in Λ_s as follows:

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

$\mathbf{mkPush} = \lambda_s e.\lambda_s x.\mathbf{push}_s\ (\mathbf{push}_s\ x \circ e)$

Figure 16 $\mathcal{SKI}$ Combinator Definitions

The correctness of the transformation $\mathcal{SKI}$ is stated by Property 5.

Property 5 $\forall E \in \Lambda_s, \mathbf{push}_s\ x \circ \mathcal{SKI}\ [\![E]\!]\ x = E$

The usual optimizations can be expressed. For example, **S** (**K** E) **I** = E and **S** (**K** E_2) (**K** E_1) = **K** (E_2 E_1) corresponds respectively to the rules

$$\mathbf{push}_s\ \mathbf{Is} \circ \mathbf{mkPush} \circ \mathbf{push}_s\ (\mathbf{push}_s\ E \circ \mathbf{Ks}) \circ \mathbf{Ss} = E$$

$$\mathbf{push}_s\ (\mathbf{push}_s\ E_1 \circ \mathbf{Ks}) \circ \mathbf{push}_s\ (\mathbf{push}_s\ E_2 \circ \mathbf{Ks}) \circ \mathbf{Ss} = \mathbf{push}_s\ (E_1 \circ E_2) \circ \mathbf{Ks}$$

whose correction can be easily established using combinator definitions.

The abstraction algorithm could be extended with the combinators **B**, **C**, **S'**, **B'**, **C'** in order to get a more compact code. For example, **S'** can be encoded by

$$\mathbf{Ss'} = \lambda_s e_3.\lambda_s e_2.\lambda_s e_1.\lambda_s x.(\mathbf{push}_s\ x \circ e_1) \circ (\mathbf{push}_s\ x \circ e_2) \circ e_3$$

4.3 Specializations

The transformations $\mathcal{Ag}_{\text{dsb}'}$ and $\mathcal{SKI}$ can be applied to all Λ_s-expressions. In particular, they can be composed with the transformations for the compilation of the graph reduction control (section 3.3). The resulting code, although correct, does not always model accurately the classical compilation schemes of the G- or SKI-machine. In order to do so, we specialize them to the code produced by $\mathcal{G}\ [\![]\!]$. That is to say, we consider only expressions of the form

$$\mathbf{push}_s\ x_i \circ \mathbf{mkVar}_s \qquad \mathbf{push}_s\ (\lambda_s x.E) \circ \mathbf{mkFun}_s \qquad E_1 \circ E_2 \circ \mathbf{mkApp}_s$$

It is of course unnecessary to abstract the graph combinators. The specialized version of $\mathcal{Ag}_{\text{dsb}'}$ is:

$$\mathcal{Ag}_{\text{dsb}'}\ [\![\mathbf{push}_s\ x_i \circ \mathbf{mkVar}_s]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0)\ k = \mathbf{dupls}_{k+m-i,1} \circ \mathbf{mkVar}_s$$

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

$$\mathcal{Ag}_{\text{dsb}'}\ [\![\mathbf{push}_s\ (\lambda_s x.E) \circ \mathbf{mkFun}_s]\!]\ \rho\ k$$

$$= \mathbf{push}_s(\mathbf{stores}_{p,m} \circ \mathcal{Ag}_{\text{dsb}'}\ [\![\lambda_s x.E]\!]\ \rho\ p \circ \mathbf{flushs}_{1,m}) \circ \mathbf{mkclos}_{k,m} \circ \mathbf{mkFun}_s$$

This last rule must be modified since, in graph reduction, closures are encoded by graphs. The combinator $\mathbf{mkclos}_{k,m}$ is replaced by graph builder code and the rule for functions is:

$$\mathcal{Ag}_{\text{dsb}'}\ [\![\mathbf{push}_s\ (\lambda_s x.E) \circ \mathbf{mkFun}_s]\!]\ \rho\ k = \mathbf{dupls}_{k,m} \circ \mathbf{push}_s(\mathbf{stores}_{p,m} \circ \mathcal{Ag}_{\text{dsb}'}\ [\![\lambda_s x.E]\!]\ \rho\ p \circ \mathbf{flushs}_{1,m}) \circ \mathbf{mkFun}_s\ (\circ\ \mathbf{mkApp}_s)^m$$

These two styles of closure representation ($\mathbf{mkclos}_{k,m}$ or graph) are equivalent when unwound; that is:

$$\mathbf{push}_s(\mathbf{stores}_{p,m} \circ \mathcal{Ag}_{\text{dsb}'}\ [\![\lambda_s x.E]\!]\ \rho\ p \circ \mathbf{flushs}_{1,m}) \circ \mathbf{mkclos}_{k,m} \circ \mathbf{mkFun}_s \circ \mathbf{unwind}_s =$$

$$\mathbf{dupls}_{k,m} \circ \mathbf{push}_s(\mathbf{stores}_{p,m} \circ \mathcal{Ag}_{\text{dsb}'}\ [\![\lambda_s x.E]\!]\ \rho\ p \circ \mathbf{flushs}_{1,m}) \circ \mathbf{mkFun}_s\ (\circ\ \mathbf{mkApp}_s)^m \circ \mathbf{unwind}_s$$

A preliminary step of the G-machine is to transform (λ-lift) source functions into supercombinators [8]. This can be directly modeled by inserting copies of the environment (like the $\mathcal{A}_{c1}$ scheme in [5]). The **mkbind**-less variations (like $\mathcal{Ag}_{\text{dsb}'}$) perform these copies (using **mkclos** and **stores**) and, in a way, integrate the λ-lifting process. The produced code models the usual G-machine instructions, for example: $\mathbf{dupls}_{k+m-i,1}$ for **PUSH(***k*+*m*-*i***)**, $\mathbf{mkApp}_s$ for **MKAP**, $\mathbf{mkFun}_s$ for **PUSHFUN** *f*, and $\mathbf{flushs}_{1,m+1}$ for **SLIDE (***m*+1**)**.

In the same way, a straightforward application of $\mathcal{SKI}$ to graph reduction code does not model accurately the SKI-machine. For example, the derived code would propagate the value of variables before building the graph. The SKI-machine does the reverse. The easiest way to model precisely the SKI abstraction algorithm is to define a new ad-hoc transformation (Figure 17).

$\mathcal{SKI}' : \Lambda_s \to var \to \Lambda_e$

$\mathcal{SKI}' \ [\![E]\!] \ x = E \circ (\mathbf{push}_s \ \mathbf{Ks} \circ \mathbf{mkFun}_s) \circ \mathbf{mkApp}_s$ $\quad$ x not free in E

$\mathcal{SKI}' \ [\![E_1 \circ E_2 \circ \mathbf{mkApp}_s]\!] \ x$

$= \mathcal{SKI}' \ [\![E_1]\!] \ x \circ (\mathcal{SKI}' \ [\![E_2]\!] \ x \circ (\mathbf{push}_s \ \mathbf{Ss} \circ \mathbf{mkFun}) \circ \mathbf{mkApp}_s) \circ \mathbf{mkApp}_s$

$\mathcal{SKI}' \ [\![\mathbf{push}_s \ (\lambda_s y.E) \circ \mathbf{mkFun}_s]\!] \ x = \mathcal{SKI}' \ [\![\mathcal{SKI}' \ [\![E]\!] \ y]\!] \ x$

$\mathcal{SKI}' \ [\![\mathbf{push}_s \ x \circ \mathbf{mkVar}_s]\!] \ x = \mathbf{push}_s \ \mathbf{Is} \circ \mathbf{mkFun}_s$

Figure 17 Abstraction SKI ($\mathcal{SKI}'$)

The new versions of the **Ss**, **Ks**, **Is** combinators build or select a graph. They can be defined as:

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

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

4.4 Comparison

With the hypothesis that every combinator can be implemented by a constant time operation, the size of transformed expressions gives a measure of the overhead entailed by the compilation of β-reduction. It is then possible to compare the different compilation techniques of β-reduction by evaluating the complexity of the corresponding transformations in terms of code expansion. It is easy to show that $\mathcal{Ag}_{\text{dsb}'}$ entails a code expansion which is linear with respect to the size of the source expression. More precisely:

$$\text{If} \quad \mathcal{Size} \ (E) = s \text{ then } \mathcal{Size} \ (\mathcal{Ag}_{\text{dsb}'} \ (\mathcal{N}m \ [\![E]\!])) \leq 6n_v + 2n_\lambda - 4$$

with n_λ the number of λ-abstractions and n_v the number of variables occurrences ($s = n_\lambda + n_v$) of the source expression. However, assuming a standard stack machine where the component s is implemented as a data stack, $\mathbf{stores}_{n,m}$ and $\mathbf{flushs}_{n,m}$ have a $O(m+n)$ cost and $\mathbf{dupls}_{n,m}$ and $\mathbf{mkclos}_{n,m}$ have a $O(m)$ cost, where n is the number of closures stacked on the environment (max n_v), and m is the environment length (max n_λ). In this case, the size function must be weighted according to the indexed combinators costs. For $\mathcal{Ag}_{\text{dsb}'}$, with the common hypothesis that closure arity is not greater than 1, we get:

$$\text{If} \quad \mathcal{Size} \ (E) = \text{s then } \mathcal{Cost} \ (\mathcal{Ag}_{\text{dsb}'} \ (\mathcal{N}m \ [\![E]\!])) \leq 2n_\lambda^2 + 5n_\lambda n_v - 7n_\lambda + 2n_v$$

The formula makes clear that the environment length (n_λ) is the predominant criterion for the efficiency of $\mathcal{Ag}_{dsb'}$. This expression reach a maximum with n_λ=5*s*/6-3/2. The upper bound can be approached with, for example, the expression $\lambda x_1 \ldots \lambda x_n . x_n$.

The same kind of study can be conducted for $\mathcal{SKI}$. In this case, the hypothesis that combinators can be implemented in constant time is sound. It is then easy to prove that the simple version presented entails an exponential code expansion:

$$\text{If } \mathcal{Size}\ (E) = s \text{ then } \mathcal{Size}\ (\mathcal{SKI}\ (\mathcal{N}m\ [\![E]\!])) \leq (5(n_v\text{-}1))^{n\lambda} + (2n_v)^{n\lambda}$$

The optimized versions (with {**S**, **K**, **I**, **B**, **C**, **S'**, **B'**, **C'**}) induces a quadratic ($O(n^2)$) code expansion. Balancing source expressions reduces this upper bound to $O(n \log n)$ [2]. This last result is optimal with a finite set of combinators. This balancing technique could be applied to $\mathcal{Ag}_{dsb'}$ as well to get a $O(n \log n)$ complexity for environment management.

5 Closure Sharing and Updates

The call-by-need strategy is an optimization of the call-by-name strategy which 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 access a closure in the memory, to reduce it, and to update the memory with the closure normal form. This way, every argument are reduced at most once.

We choose to apply this compilation step after the compilation of the sequence breaks (see [5] for a description of this step). A new intermediate language Λ_h adds to Λ_k the combinator pair ($\mathbf{push}_h$, λ_h) which specifies a memory component h. This component is represented and accessed via a heap pointer and should logically be instantiated as a separated component.

A first transformation $\mathcal{Hc}$ from Λ_k to Λ_h threads the component h where closures are allocated and accessed. Then, two possible implementations of updates are expressed and we present several options specific to graph reduction.

5.1 Introduction of a heap

The transformation $\mathcal{Hc}$ (Figure 18) 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 [13]). The transformed expression $\mathcal{Hc}\ [\![E]\!]$ takes the heap as argument and is reduced exactly as E except that it also returns the heap as result.

$\mathcal{Hc} : \Lambda_k \rightarrow \Lambda_h$ with $i \equiv s,e$ or k and h a fresh variable

$\mathcal{Hc}\ [\![E_1 \circ E_2]\!] = \mathcal{Hc}\ [\![E_1]\!] \circ \mathcal{Hc}\ [\![E_2]\!]$

$\mathcal{Hc}\ [\![\mathbf{push}_i\ E]\!] = \lambda_h h.\mathbf{push}_i\ (\mathcal{Hc}\ [\![E]\!]) \circ \mathbf{push}_h\ h$

$\mathcal{Hc}\ [\![\lambda_i x.E]\!] = \lambda_h h.\lambda_i x.\mathbf{push}_h\ h \circ \mathcal{Hc}\ [\![E]\!]$

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

Figure 18 Introducing a Heap ($\mathcal{Hc}$)

For now, there is no interaction between the reduction process and h content. To express closure allocation and access, addresses are represented by integers and the heap is represented by a pair made of a list of written cells {address, value} and the address of the next free cell. The initial empty heap is noted **emptyH** and is defined as ((),0). We introduce three combinators performing basic heap manipulations:

alloc = $\lambda_h(heap,free).\mathbf{push}_s\ free \circ \mathbf{push}_h\ (heap,free+1)$

write = $\lambda_h(heap,free).\lambda_s add.\lambda_s val.\mathbf{push}_h\ ((heap,\{add,val\}),\ free)$

$$\textbf{read} = \lambda_h((heap,\{add_1,val\}),free).\lambda_s add_2.\ \ if\ \ add_1 = add_2\ then\ \textbf{push}_s\ val$$
$$else\ \textbf{push}_h\ (heap,free)\ \circ\ \textbf{push}_s\ add_2\ \circ\ \textbf{read}$$

Figure 19 Heap Manipulation Combinators (alloc, write and read)

It is sufficient to change $\mathcal{H}c$ rules for $\textbf{push}_s\ E$ and x to make closure allocation and access explicit (Figure 20). In our framework, constructions of updatable closures are of the form $\textbf{push}_s\ E$ with $E{:}\mathbf{R}_s\sigma$. The motivation for this criterion is that there may be closures of the form $\textbf{push}_s\ E$ where E denotes a function (as $\mathcal{N}m$ produces). In this case, E does not yield a result and that forbids updating. Accesses of updatable closures are of the form $x : \mathbf{R}_s\tau$ where x is bound by a λ_s. The reason is that there may be expressions of the form $\lambda_e x \ldots x \ldots$ where x denotes a result but not a closure (not in the s component).

$\mathcal{H}c : \Lambda_k \rightarrow \Lambda_h$ with $E : \mathbf{R}_s\sigma$ and $x : \mathbf{R}_s\tau$ bound by $\lambda_s x$.

$\mathcal{H}c\ [\![\textbf{push}_s\ E]\!] = \textbf{Store}[\mathcal{H}c\ [\![E]\!]\ ]$

with $\textbf{Store}\ [E] \equiv \lambda_h h.\textbf{push}_h\ h\ \circ\ \textbf{alloc}\ \circ\ \lambda_h h.\lambda_s a.$

$\textbf{push}_s\ E\ \circ\ \textbf{push}_s\ a\ \circ\ \textbf{push}_h\ h\ \circ\ \textbf{write}\ \circ\ \lambda_h h.\ \textbf{push}_s\ a\ \circ\ \textbf{push}_h\ h$

$\mathcal{H}c\ [\![x]\!] = \textbf{Call}[x]$

with $\textbf{Call}\ [E] \equiv \lambda_h h.\textbf{push}_s\ E\ \circ\ \textbf{push}_h\ h\ \circ\ \textbf{read}\ \circ\ \lambda_s y.\ \textbf{push}_h\ h\ \circ\ y$

Figure 20 Allocating and Accessing Closures ($\mathcal{H}c$)

The context **Store**[E] can be read as: allocate a new cell in the heap, write the code E in this cell, return its address a and the heap. The context **Call**[E] can be read as: access the expression stored in the heap in the cell of address E, then reduce it (with the heap as an argument). Henceforth, the argument of a function is a closure address rather than the closure itself. Since h is single-threaded, the combinators **alloc**, **write** and **read** 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 used by the previous compilation steps. When a combinator does not create nor call a closure, the transformation $\mathcal{H}c$ threads the heap without interaction. For example, for the combinator **flushs** introduced by the abstraction $\mathcal{A}g_{\text{dsb}'}$, we get:

$$\textbf{flushs}_{h\ n,m} = \mathcal{H}c\ [\![\textbf{flushs}_{n,m}]\!] = \mathcal{H}c\ [\![\lambda_s x_1 \ldots \lambda_s x_n.\lambda_s y_1 \ldots \lambda_s y_m.\textbf{push}_s\ x_n\ \circ \ldots \circ\ \textbf{push}_s\ x_1]\!]$$
$$= \lambda_h h.\lambda_s x_1 \ldots \lambda_s x_n.\lambda_s y_1 \ldots \lambda_s y_m.\textbf{push}_s\ x_n\ \circ \ldots \circ\ \textbf{push}_s\ x_1\ \circ\ \textbf{push}_h\ h$$

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

$$\textbf{mkclos}_{h\ k,m} = \mathcal{H}c\ [\![\textbf{mkclos}_{k,m}]\!]$$

$$= \mathcal{H}c\ [\![\lambda_s x.\lambda_s x_1 \ldots \lambda_s x_k.\lambda_s y_1 \ldots \lambda_s y_m.\mathbf{push}_s\, y_m \circ \ldots \circ \mathbf{push}_s\, y_1 \circ$$

$$\mathbf{push}_s\, x_k \circ \ldots \circ \mathbf{push}_s\, x_1 \circ \mathbf{push}_s\, (\mathbf{push}_s\, y_m \circ \ldots \circ \mathbf{push}_s\, y_1 \circ x)]\!]$$

$$= \lambda_h h.\lambda_s x.\lambda_s x_1 \ldots \lambda_s x_k.\lambda_s y_1 \ldots \lambda_s y_m.\mathbf{push}_s\, y_m \circ \ldots \circ \mathbf{push}_s\, y_1 \circ$$

$$\mathbf{push}_s\, x_k \circ \ldots \circ \mathbf{push}_s\, x_1 \circ \mathbf{push}_h\, h \circ \mathbf{Store}[\mathbf{push}_s\, y_m \circ \ldots \circ \mathbf{push}_s\, y_1 \circ x]$$

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

The transformation $\mathcal{H}c$ handles closures but it could also be used to model memory management for others components. For example, the transformation $\mathcal{A}s$ ([5]) represents environment as trees (using pairs) and one may want to express the implicit allocations needed to store such trees.

5.2 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 either 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. This scheme is implemented by a first version of the Krivine abstract machine [3]. The callee-update scheme updates closures only the first time they are accessed. Once in normal form, others accesses will not entail further (useless) updates. This scheme is implemented by all the realistic, environment-based implementations. In both cases, the shared expression is bracketed by codes storing the update address and performing the update.

5.2.1 Caller update

The transformation $\mathcal{U}caller$ specializes the $\mathcal{H}c$ rule for variables in order to update closures after calling (Figure 21).

$\mathcal{U}caller : \Lambda_k \rightarrow \Lambda_h$ with $x : \mathbf{R}_s\tau$ bound by λ_s

$\mathcal{U}caller\ [\![x]\!] = \mathbf{push}_s\, x \circ \mathbf{swap}_{sh} \circ \mathbf{Call}[x] \circ \mathbf{updt}$

with $\mathbf{swap}_{sh} = \lambda_s a.\lambda_h h.\mathbf{push}_s\, a \circ \mathbf{push}_h\, h$

and $\mathbf{updt} = \lambda_h h.\lambda_s b.\lambda_s a.\mathbf{push}_s\, (\lambda_h h.\mathbf{push}_s\, b \circ \mathbf{push}_h\, h) \circ \mathbf{push}_s\, a \circ \mathbf{push}_h\, h \circ \mathbf{write}$

$\circ\, \lambda_h h.\mathbf{push}_s\, b \circ \mathbf{push}_h\, h$

Figure 21 Caller Closure Update ($\mathcal{U}caller$)

This transformation introduces a combinator **updt** which takes as argument 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 the heap where the cell *a* contains now an indirection to *b*. The combinator $\mathbf{swap}_{sh}$ reorders the address *x* and the heap.

For the sake of simplicity, we have presented a generic transformation $\mathcal{U}_{caller}$ which transforms any Λ_k expression. This transformation introduces sequence breaks: a closure call is followed by an update (**Call**[*x*] ∘ **updt**). This is annoying since control transfers have already been compiled. The solution is to specialize $\mathcal{U}_{caller}$ to different forms of codes produced by the transformations $\mathcal{S}$ and $\mathcal{SI}$ which compile control transfers [5]. In doing so, the combinator **updt** can be shifted to the beginning of the code which follows the closure call.

5.2.2 Callee update

The transformation $\mathcal{U}_{callee}$ specializes the $\mathcal{H}_c$ rule for $\mathbf{push}_s\ E$ in order to introduce self updating closures.

$$\mathcal{U}_{callee} : \Lambda_k \rightarrow \Lambda_h \qquad \text{with} \quad E : \mathbf{R}_s\sigma$$

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

Figure 22 Callee Closure Update ($\mathcal{U}_{callee}$)

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 fresh 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 transformation $\mathcal{U}_{callee}$ introduces also sequence breaks ($\mathcal{U}_{callee}\ [\![E]\!] \circ \mathbf{updt}$) which are suppressed in the same way as $\mathcal{U}_{caller}$ by specializing the transformation.

5.2.3 Updating and the push-enter model

The caller update and the callee update schemes can be used with $\mathcal{N}_m$. As noted in section 3.1, marks have to be inserted in the expressions to pause the reduction and insert updating codes. In a caller update scheme, $\mathcal{U}_{caller}$ is specialized as:

$$\mathcal{U}_{caller}\ [\![x]\!] = \mathbf{push}_s\ x \circ \mathbf{swap}_{sh} \circ \mathbf{push}_s\ \varepsilon \circ \mathbf{swap}_{sh} \circ \mathbf{Call}[x] \circ \mathbf{updt} \circ \mathbf{resume}_h$$

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

In the same way, in a callee update scheme, $\mathcal{U}_{callee}$ is specialized as:

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

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

In both cases, an evaluation context is isolated by inserting a mark ε after the update address ($\mathbf{push}_s\ x$ or $\mathbf{push}_s\ a$); the combinator $\mathbf{grab}_h$ is defined by $\mathcal{H}c\ [\![\mathbf{grab}_s]\!]$.

The codes produced by $\mathcal{N}a$ and $\mathcal{N}ml$ have the same update opportunities. As in call-by-name, the $\mathcal{N}ml$ scheme may prevent from building unnecessary intermediate closures.

5.3 Updating and graph reduction

The previous transformations can be used to transform the call-by-name graph reduction schemes into call-by-need. Here, we present transformations to model two updating techniques (spine and spineless variations) which has been introduced for the G-machine. The spine variation, the standard technique for the G-machine, does not fit with our modeling and an ad-hoc transformation must be defined. The spineless variation is found by using $\mathcal{U}callee$.

5.3.1 G-machine

As suggested in section 3.3.2, the use of marks is not mandatory to express updating in the G-machine. Graph building and graph reduction are separate steps. Updates can be systematically inserted between each graph building and reduction step. However this scheme cannot be expressed using the previous transformations. The canonical definition of $\mathbf{mkApp}_s$ for $\mathcal{GN}m$ is:

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

Since $\mathcal{H}c$ shares only expressions of the form $\mathbf{push}_s\ E$ with $E{:}\mathbf{R}_s\sigma$, application nodes will not be considered for updating with this definition of $\mathbf{mkApp}_s$.

It is easy to model the G-machine scheme by a new transformation. We present in Figure 23 only the two most interesting rules:

$$\mathcal{U}spine\ : \Lambda_i \to \Lambda_h$$

$$\mathcal{U}spine\ [\![\mathbf{mkApp}_s]\!] = \mathcal{U}spine\ [\![\lambda_s x_1.\lambda_s x_2.\mathbf{push}_s\ (\mathbf{push}_s\ x_2 \circ x_1)]\!]$$

$$= \lambda_h h.\lambda_s x_1.\lambda_s x_2.\mathbf{push}_h\ h \circ \mathbf{Store}[\mathbf{push}_s\ a \circ \mathbf{swap}_{sh} \circ \mathbf{push}_s\ x_2 \circ \mathbf{swap}_{sh} \circ \mathbf{Call}[x_1]]$$

$$\mathcal{U}spine\ [\![\mathbf{mkFun}_s]\!] = \mathcal{U}spine\ [\![\lambda_s f.\mathbf{push}_s\ (f \circ \mathbf{app})]\!]$$

$$= \lambda_h h.\lambda_s f.\mathbf{Store}[f \circ \mathbf{updt} \circ \lambda_h h.\lambda_s x.\mathbf{push}_h\ h \circ x]$$

Figure 23 Updates in the G-machine ($\mathcal{U}spine$)

An application node is allocated in the heap as it is created. When reduced, its address is stored before stacking the graph argument. In the same way, function nodes are created/al-

located in the heap. When reduced, the function *f* is called; it takes its argument and yields a graph as result. The combinator **updt** takes this graph and the address of the application node to perform the update. Contrary to the standard options, address storing and updating do not bracket the updatable expression anymore. These operations are done independently respectively on $\mathbf{mkApp}_s$ and $\mathbf{mkFun}_s$ nodes. Note that when an application node is unwound, its address (which is necessary for the latter update) and its right son (i.e. the argument) are stacked. Actually, as the G-machine does, it is sufficient to stack the application node addresses (i.e. the leftmost spine) and change accesses by $\mathbf{access}_i \circ \mathbf{accessRightSon}$.

5.3.2 Spineless G-machine

The spineless G-machine [1] updates only selected application nodes. Unwinding application nodes entails to stack either their address (updatable) or only the argument address (non updatable). In general, the complete leftmost spine of the graph does not appears in the stack. The code must annotate updatable nodes and marks are necessary to dynamically detect when an update must be performed (see section 3.3.2). Updatable nodes are distinguished using the combinator $\mathbf{mkAppS}_s$ which has the same definition as $\mathbf{mkApp}_s$.

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

The transformation $\mathcal{U}_{callee}$ for the push-enter model (section 5.2.3) can be applied to these nodes. We get:

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

A caller update variation can be expressed too. This can be done by annotating updatable $\mathbf{mkVar}_s$ nodes.

5.4 Remarks

The comparison of the transformations $\mathcal{U}_{caller}$ and $\mathcal{U}_{callee}$ is easy. Both transformations induce a linear code expansion. It is nevertheless clear that the $\mathcal{U}_{callee}$ scheme which updates closures at most once is more efficient than $\mathcal{U}_{caller}$ which updates closures at every access.

Two techniques could be compared in the graph reduction: the spine push-enter scheme (without marks), and the spineless push-enter scheme (with mark). In practice, the spineless scheme seems to be more efficient than the spine scheme. The inherent interpreting overhead in graph reduction favors the use of dynamic tests (e.g. marks).

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 which lie in the heap.

6 Classical Functional Implementations

Figure 24 states the main design choices structuring several classical call-by-name and call-by-need implementations.

Compiler	Transformations			Components
Clean	$\mathcal{N}ml$	$\mathcal{A}c_1$	$\mathcal{U}Callee$	s e k h
G-machine	$\mathcal{G}\mathcal{N}m$	$\mathcal{A}c_{dsb'}$	$\mathcal{U}Spine$	s e k h
G-machine spineless	$\mathcal{G}\mathcal{N}ml$	$\mathcal{A}c_{dsb'}$	$\mathcal{U}Callee$	s e k h
G-machine spineless tagless	$\mathcal{N}ml$	$\mathcal{A}c_3$	$\mathcal{U}Callee$	(s ≡ k) e h
Mak	$\mathcal{N}ml$	$\mathcal{A}s$	$\mathcal{U}Spine$	s ≡ e ≡ k h
SKI-machine	$\mathcal{G}\mathcal{N}m$	$\mathcal{S}\mathcal{K}\mathcal{I}'$	$\mathcal{U}Spine$	s h
Tabac	$\mathcal{N}a$	$\mathcal{A}c2_{dsb}$	$\mathcal{U}Hybr\ id$	(s ≡ e) k h
Tim	$\mathcal{N}ml$	$\mathcal{A}c1_m$	$\mathcal{U}callee$	s e k h

Figure 24 Several Classical Compilation Schemes

There are cosmetic differences between our descriptions and the real implementations. Also, some extensions and optimizations are not described here. Let us quickly review the differences between Figure 24 and real implementations.

The Clean implementation is based on graph rewriting, however the final code is similar to environment machines (for example, a closure is encoded by a n-ary node). The numerous optimizations and especially the lack of clear description ([12] presents mostly examples of final codes) makes the identification of compilation choices difficult.

The G-machine [9] and the spineless G-machine [1] transform the source program in a (super)combinators set. The λ-lifting transformation [8] is modeled by the implicit copies of $\mathcal{A}c_{dsb'}$. A difference is that our $\mathbf{grab}_s$ combinator performs a test for every argument, whereas the real machines perform only one test for all the arguments (by comparison of the arity with the activation record size). So, a n-ary combinator $\mathbf{grabs}_n$ should be introduced. Then, our descriptions would be accurate.

The spineless tagless G-machine [11] does not apply the λ-lifting transformation and uses a local and a global environment. As before, the representation of closures differs and the machine uses a n-ary version of $\mathbf{grab}_s$. The abstraction with two environments is represented in [5] by the $\mathcal{A}c_3$ transformation. As noted in [5], local environments are not directly compatible with $\mathbf{grab}_s$, and extra environment copies must be inserted. So, even specialized to graph reduction, the transformation $\mathcal{A}c_3$ would not model precisely the spineless tagless

G-machine. In this case, the differences cannot be called “cosmetic”. The simplest way to model faithfully the real machine would be to introduce an ad-hoc abstraction algorithm.

The Krivine abstract machine Mak [3] is precisely modeled by the transformations composition: $\mathcal{UC}allee \bullet \mathcal{A}s \bullet \mathcal{N}ml$.

The SKI-machine [14] 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. Even if the description of the SKI-machine in [14] is somewhat informal, we think that our modeling is close to the real machine.

The Tabac compiler is a by-product of our work in [7] and implements strict or non-strict languages by program transformations. Its lazy version uses an hybrid caller-callee update scheme not described in this paper. Tabac use numerous optimizations not described here. Apart from these minor differences, our modeling is faithful.

Our description of the call-by-name version of Tim is accurate according to [6]. However, a n-ary version of $\mathbf{grab}_s$ should be added to our call-by-need version.

7 Conclusion

We have presented a framework to describe, prove and compare functional implementation techniques and optimizations. We have completed previous work on call-by-value, with the description of call-by-name, call-by-need and graph reduction implementations. We have described the different control compilation schemes for environment machines (sections 3.1 and 3.2) and for graph reduction machines (section 3.3). Their description in a unified framework allowed us to formally relate them (3.4.2). We have presented a G-machine-like β-reduction compilation scheme and a SKI-like abstraction algorithm (section 4). Finally, we have expressed closure sharing and updating inherent to the call-by-need strategy (section 5). At each stage, the various options are expressed as transformations and briefly compared.

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

The use of program transformations appeared to be suited to model precisely and completely the compilation process. The unified framework simplifies correctness proofs and makes it possible to reason about the efficiency of the produced code as well as about the complexity of transformations themselves. Actually, these advantages appear clearly only until 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 machine code.

This report along with Part I provides a general taxonomy of known sequential implementations of functional languages. Our main goal was to structure and clarify the design space of functional languages implementations. The exploration is still far from complete; in particular:

- We have suggested how decurryfication, peephole optimizations and unboxing could be expressed [5]. A systematic description of standard optimizations in our framework could be undertaken.
- We have shown, on a few instances, how formal comparisons of transformations can be made. Many interesting comparisons remains to be done.
- New combinations of transformations or the design of hybrid transformations (mixing several compilation schemes) deserve further studies.
- A last step towards 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.

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

References

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

[2] F.W. Burton. A linear space translation of functional programs to Turner combinators. *Information Processing Letters*, pp 201-204, 1984.

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

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

[5] R. Douence and P. Fradet. A taxonomy of functional language implementations. Part I: call-by-value. *INRIA research report 2783*, 1996.

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

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

[8] T. Johnsson. Lambda lifting: transforming programs to recursive equations. *In Proc. of FPCA'85*, LNCS 201, pp. 190-203, 1985.

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

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

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

[12] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[13] D. Schmidt. *Denotational Semantics, a Methodology for Language Development.* W. C. Brown Publishers, 1986.

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

[15] D.A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44, pp. 267-270, 1979.

Annex

Property 1 is shown in [5]. The correctness proofs of and Property 3 are similar to the proofs of the corresponding properties for $\mathcal{V}a$ and $\mathcal{V}m$ [5]. We do not detail these proofs here.

A Proof of Property 4

Call-by-name reduction is described by the following natural semantics:

$$\frac{E_1 \underset{cbn}{\rightarrow} \lambda x.F \qquad F[E_2/x] \underset{cbn}{\rightarrow} N}{E_1\ E_2 \underset{cbn}{\rightarrow} N} \qquad N \text{ normal form}$$

The proof of Property 4 is on the shape of the reduction trees.

Axioms.

If E is not reducible, it is of the form $\lambda x.F$ (E is closed). We have then $E \equiv V$ and the property is trivially verified.

Induction.

If E is reducible, that is, $E \equiv E_1\ E_2$, $E_1 \underset{cbn}{\rightarrow} \lambda x.F$ and $F[E_2/x] \underset{cbn}{\rightarrow} N$. By induction hypothesis, we have $\mathcal{G}\ [\![E_1]\!] \circ \mathbf{unwind}_s = \mathcal{G}\ [\![\lambda x.F]\!] \circ \mathbf{unwind}_s$ and $\mathcal{G}\ [\![F[E_2/x]]\!] \circ \mathbf{unwind}_s = \mathcal{G}\ [\![N]\!] \circ \mathbf{unwind}_s$ Moreover, it is easy to prove by structural induction on F that

Lemma 6 $(\mathcal{G}\ [\![F]\!][\ \mathcal{G}\ [\![E_2]\!]/\ \mathbf{push}_s\ x]) \circ \mathbf{unwind}_s = (\mathcal{G}\ [\![F[\ E_2/x]]\!]) \circ \mathbf{unwind}_s$

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

$= \mathcal{G}\ [\![E_2]\!] \circ \mathcal{G}\ [\![E_1]\!] \circ \mathbf{unwind}_s$ ($\mathcal{GNM}$ 3)

$= \mathcal{G}\ [\![E_2]\!] \circ \mathcal{G}\ [\![\lambda x.F]\!] \circ \mathbf{unwind}_s$ induction hypothesis

$\equiv \mathcal{G}\ [\![E_2]\!] \circ \mathbf{push}_s(\lambda_s x.\mathcal{G}\ [\![F]\!]) \circ \mathbf{mkFun}_s \circ \mathbf{unwind}_s$ (def. $\mathcal{G}$)

$= (\mathcal{G}\ [\![E_2]\!] \circ \lambda_s x.\mathcal{G}\ [\![F]\!]) \circ \mathbf{unwind}_s$ ($\mathcal{GNM}$ 2)

$= (\mathcal{G}\ [\![F]\!][\ \mathcal{G}\ [\![E_2]\!]/\ \mathbf{push}_s\ x]) \circ \mathbf{unwind}_s$ (β)

$= (\mathcal{G}\ [\![F[\ E_2/x]]\!]) \circ \mathbf{unwind}_s$ (Lemma 6)

$= \mathcal{G}\ [\![N]\!] \circ \mathbf{unwind}_s$ induction hypothesis ❒

B Proof of Property 5

We prove $\forall E \in \Lambda_s$, $\mathbf{push}_s\ x \circ \mathcal{SKI}\ [\![E]\!]\ x = E$ by structural induction on E.

- $E \equiv E_1 \circ E_2$

  $\mathbf{push}_s\ x \circ \mathcal{SKI}\ [\![E_1 \circ E_2]\!]\ x$

  $= \mathbf{push}_s\ x \circ \mathbf{push}_s\ (\mathcal{SKI}\ [\![E_1]\!]\ x) \circ \mathbf{push}_s\ (\mathcal{SKI}\ [\![E_2]\!]\ x) \circ \mathbf{Ss}$ def. $\mathcal{SKI}$

  $= (\mathbf{push}_s\ x \circ \mathcal{SKI}\ [\![E_1]\!]\ x) \circ \mathbf{push}_s\ x \circ \mathcal{SKI}\ [\![E_2]\!]\ x$ def. **Ss**

  $= E_1 \circ E_2$ induction hypothesis

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

  $\mathbf{push}_s\ x \circ \mathcal{SKI}\ [\![\mathbf{push}_s\ V]\!]\ x$

  $= \mathbf{push}_s\ x \circ \mathbf{push}_s\ (\mathcal{SKI}\ [\![V]\!]\ x) \circ \mathbf{mkPush}$ def. $\mathcal{SKI}$

  $= \mathbf{push}_s\ (\mathbf{push}_s\ x \circ \mathcal{SKI}\ [\![V]\!]\ x)$ def. **mkPush**

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

- $E \equiv F$ and F has no x free occurrence

  $\mathbf{push}_s\ x \circ \mathcal{SKI}\ [\![F]\!]\ x = \mathbf{push}_s\ x \circ \mathbf{push}_s\ F \circ \mathbf{Ks} = \mathbf{push}_s\ F$ def. $\mathcal{SKI}$, def. **Ks**

- $E \equiv x$

  $\mathbf{push}_s\ x \circ \mathcal{SKI}\ [\![x]\!]\ x = \mathbf{push}_s\ x \circ \mathbf{Is} = x$ def. $\mathcal{SKI}$, def. **Is** ❒



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

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

ISSN 0249-6399