Janvier 1996 – Journées Francophones des Langages Applicatifs – JFLA96

Décrire et comparer les implantations de langages fonctionnels

Rémi Douence & Pascal Fradet

IRISA/INRIA, Campus de Beaulieu
35042 Rennes Cedex, France
{douence,fradet}@irisa.fr

Résumé: Nous proposons un cadre formel pour décrire et comparer les implantations de langages fonctionnels. Nous décrivons le processus de compilation comme une suite de transformations de programmes dans le cadre fonctionnel. Les choix fondamentaux de mise en œuvre ainsi que les optimisations s'expriment naturellement comme des transformations différentes. Les avantages de cette approche sont de décomposer et de structurer la compilation, de simplifier les preuves de correction et de permettre des comparaisons formelles en étudiant chaque transformation ou leur composition. Nous nous concentrons ici sur l'appel par valeur et décrivons trois implantations différentes: la Cam [7], Tabac un compilateur basé sur [13] et une version stricte de la machine de Krivine [19].

1 Introduction

Un des sujets les plus étudiés concernant les langages fonctionnels est leur mise en œuvre. Depuis le célèbre article de P.J. Landin [18], il y a plus de trente ans, une pléthore de nouvelles machines abstraites ou techniques de compilation a été proposée. La liste des machines abstraites fonctionnelles comprend (entre autres) la SECD [18], la Fam [6], la Cam [7], la CMCM [20], la Tim [11], la Zam [19], la G-machine [15] et la machine de Krivine [8]. D'autres implantations ne sont pas décrites par une machine abstraite mais par des collections de transformations ou de techniques de compilation. Citons par exemple, les compilateurs basés sur la conversion CPS [1][13][17]. De plus, de nombreux papiers proposent des optimisations adaptées à une machine ou une approche spécifique [3][4][16]. A la vue de cette myriade de travaux différents, certaines questions s'imposent: Quels sont les choix fondamentaux ? Quels sont leurs avantages respectifs ? Quels sont précisément les points communs et différences entre deux compilateurs donnés ? L'optimisation conçue pour telle machine abstraite ne pourrait elle pas être adaptée pour telle autre machine ? On trouve comparativement très peu de papiers consacrés à ces questions. Malgré le besoin évident, il y n'a eu, à notre connaissance,

aucune approche globale visant à décrire, classifier et comparer les mises en œuvre de langages fonctionnels.

Le but de notre travail est d'aider à pallier cette lacune. Notre approche consiste à exprimer dans un cadre commun le processus de compilation comme une succession de transformations de programmes. Ce cadre commun est constitué d'une collection de langages intermédiaires sous-ensembles du λ-calcul. Une implantation est décrite comme une série de transformations $\Lambda \stackrel{T1}{\rightarrow} \Lambda_1 \stackrel{T2}{\rightarrow} \ldots \stackrel{Tn}{\rightarrow} \Lambda_n$ chacune compilant une tâche en convertissant une expression d'un langage intermédiaire dans le suivant. Le dernier langage Λ_n est composé d'expressions fonctionnelles particulièrement simples (combinateurs + flot de contrôle explicite) qui peuvent être vues comme du code machine. Un des avantages de cette approche est de décomposer et de structurer le processus de compilation. On peut, par exemple, montrer que deux implantations apparemment très différentes partagent un choix de mise en œuvre. L' approche facilite de plus les preuves de correction et les comparaisons. La correction de chaque étape peut être montrée indépendamment et se résume à la preuve d'une transformation dans le cadre fonctionnel. Des comparaisons formelles peuvent s'envisager sous la forme d'étude de complexité des transformations ou de leurs combinaisons.

La tâche est évidemment immense et nous nous limitons ici à l'appel par valeur et aux λ-expressions pures. Dans un article précédent [9] nous avons décrit les principaux choix de compilation existants pour les langages stricts. Nous y avons aussi indiqué comment la compilation de l'appel par nom (avec ou sans information de nécessité des arguments), des opérateurs primitifs et des structures de données pouvaient se décrire. La conception de machines hybrides (i.e. mélangeant plusieurs choix de compilation) est également mentionné. Nous ne traitons pas ici ces extensions et le lecteur intéressé pourra se reporter à cet article ou à sa version étendue [10].

Dans ce papier, nous représentons le cadre formel de l'approche et considérons son application à la description de trois implantations: la Cam, Tabac (TrAnsformation-BAsed Compiler) un compilateur CPS basé sur [13] et une version stricte de la machine de Krivine (proposée dans [19]). Le cadre général utilisé pour modéliser les implantations est décrit en section 2. La section 3 (resp. section 4) présente pour chaque machine son choix de compilation de l'appel par valeur (resp. de la β-réduction). Chaque section se conclue par une rapide comparaison des différentes options. Les deux étapes suivantes (compilation des transferts de contrôle et représentation des piles) conduisent à du code machine et sont décrites dans la section 5. Nous concluons par un rapide survol des travaux existants et indiquons nos axes de recherche actuels (section 6).

2 Cadre formel

Nous ne considérons que des λ-expressions pures et notre langage source Λ est défini par la grammaire $E ::= x \mid \lambda x.E \mid E_1\ E_2$. La plupart des choix fondamentaux de mise en œuvre peuvent se décrire dans ce cadre. Les deux étapes qui influencent le plus l'implantation sont la compilation du schéma d'évaluation (i.e. la recherche du prochain rédex) et de la β-réduction (i.e. la gestion de l'environnement). Un autre étape est la mise en œuvre des ruptures de séquences (calls & returns). La séquence $\Lambda \xrightarrow{\mathcal{V}} \Lambda_s \xrightarrow{\mathcal{A}} \Lambda_e \xrightarrow{\mathcal{S}} \Lambda_k$ présentée dans ce papier reflète ces trois étapes par trois transformations et trois langages intermédiaires. Le premier langage intermédiaire, Λ_s, restreint l'utilisation de l'application et explicite le schéma d'évaluation à l'aide d'une fonction de séquencement. Le second langage, Λ_e, restreint l'utilisation des variables et explicite la gestion de l'environnement à l'aide de combinateurs. Le troisième langage, Λ_k, code les transferts de contrôle à l'aide d'instructions d'appel et de retour de fonction. Ce dernier langage peut être vu comme du code machine. Nous présentons en détail le langage séquentiel Λ_s ; les autres ajoutent simplement des restrictions et sont décrits plus brièvement.

2.1 Le langage séquentiel Λ_s

Λ_s est défini à l'aide des combinateurs $\circ$, $\mathbf{push}_s$, et $\lambda_s x.\ E^*$.

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

On peut remarquer que la syntaxe de Λ_s ne comporte pas d'applications générales. La propriété importante du langage est que le choix du rédex n'est plus sémantiquement significatif (tous les rédex sont utiles). La compilation du schéma d'évaluation repose sur cette propriété. La transformation associée ($\Lambda \xrightarrow{\mathcal{V}} \Lambda_s$) explicite l'ordre d'évaluation à l'aide du combinateur $\circ$. Intuitivement, $\circ$ est un opérateur de séquencement et $E_1 \circ E_2$ évalue E_1 *puis* évalue E_2, $\mathbf{push}_s\ E$ rend E en résultat et $\lambda_s x.\ E$ lie à x le résultat intermédiaire précédent avant d'évaluer E. La paire de combinateurs ($\mathbf{push}_s$, λ_s) spécifie une structure de données pour mémoriser les résultats intermédiaires (e.g. une pile de données).

On peut donner diverses définitions à ces combinateurs (cf. la sous-section 5.2). Nous n'optons pas pour une définition précise ici mais nous imposons que leurs définitions satisfassent l'équivalent de la β- et η-conversion :

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

$$(\eta_s) \qquad \lambda_s x.(\mathbf{push}_s\ x \circ E) = E \qquad \textit{si } x \textit{ n'est pas libre dans } E$$

---

* Cette construction n'est pas à proprement parler un combinateur, mais peut se définir comme un combinateur appliqué à $\lambda x.E$.

Le langage Λ_s est un sous-ensemble du λ-calcul et la définition de la substitution ainsi que la notion de variable libre ou liée restent identiques. Comme il va de soi pour l'opérateur de séquencement impératif ";", nous imposons l'associativité du combinateur $\circ$. Cette propriété est tout spécialement utile pour transformer les programmes.

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

Par la suite, nous omettons souvent les parenthèses et écrivons, par exemple, $\mathbf{push}_s\ E \circ \lambda_s x.F \circ G$ pour $(\mathbf{push}_s\ E) \circ (\lambda_s x.(F \circ G))$.

2.2 Un sous-ensemble typé

Toutes les expressions de Λ_s ne sont pas intéressantes pour notre propos. Les transformations de Λ vers Λ_s ne produisent que des expressions dénotant des résultats (i.e. qui se réduisent en des expressions de la forme $\mathbf{push}_s\ F$). Afin de pouvoir caractériser de telles expressions ou d'exprimer certaines lois algébriques, il est utile de considérer un sous-ensemble typé de Λ_s. Les restrictions imposées par le système de type portent sur la combinaison des résultats et des fonctions. Le typage impose que dans une composition $E_1 \circ E_2$, E_1 dénote un résultat (i.e. a comme type $\mathsf{R}\sigma$, R étant un constructeur de type) et E_2 dénote une fonction.

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

Figure 1 Sous-ensemble typé de Λ_s

Ceci n'entraîne aucune restriction de type sur le langage source Λ. On peut, par exemple, autoriser les types réflexifs comme $\alpha{=}\alpha{\to}\alpha$ pour typer toute λ-expression source.

2.3 Réduction

Nous considérons une seule règle de réduction correspondant à la β-réduction:

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

Comme toutes les mises en œuvre classiques, nous ne traitons que les réductions faibles. Les sous-expressions sous les λ_s ou les $\mathbf{push}_s$ ne sont pas réductibles et nous utilisons par la suite rédex (resp. réduction, forme normale) pour rédex faible (resp. réduction faible, forme normale faible).

Il est facile de voir que, dans Λ_s, deux rédex sont forcément disjoints. La β_s-réduction étant linéaire à gauche, le système de réécriture est orthogonal et donc confluent. De plus, tout rédex est nécessaire (une réécriture ne peut supprimer un rédex) donc tous les ordres d'évaluation (faibles) sont normalisants.

Propriété 1 *Si une expression fermée* $E{:}\mathsf{R}\sigma$ *(resp.* $E{:}\ \sigma \to_s \tau$*) a une forme normale, alors* $E \overset{*}{\rightarrow} \mathbf{push}_s\ V$ *(resp.* $E \overset{*}{\rightarrow} \lambda_s x.\ F$*)*

Nous ne détaillons pas les preuves ici ; le lecteur intéressé peut se reporter à [10] qui présente l'approche formelle de façon plus détaillée.

2.4 Lois

Ce langage dispose d'une collection de lois algébriques utiles pour transformer le code fonctionnel (e.g. exprimer des optimisations) ou montrer la correction ou l'équivalence de transformations. Nous en présentons ici deux:

(L1) *si x n'est pas libre dans F* $\qquad (\lambda_s x.E) \circ F = \lambda_s x.\ (E \circ F)$

(L2) $\forall E_1{:}\mathsf{R}\sigma$, $\forall E_2{:}\mathsf{R}\tau$, *si x n'est pas libre dans* E_2

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

Ces lois permettent de réorganiser le code en le faisant sortir ou rentrer des corps de fonctions ou en intervertissant l'évaluation de deux résultats intermédiaires. Leur correction peut être montrée très facilement. Par exemple, (L1) est valide puisque x n'est pas libre dans $(\lambda_s x.E)$ ni, par hypothèse, dans F et

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

Dans la suite, nous introduisons d'autres règles au fur et à mesure pour exprimer des optimisations de transformations particulières.

2.5 Survol des étapes de compilation

Avant de décrire plus formellement les trois implantations choisies, nous faisons un rapide survol des différentes étapes, des choix de mise en œuvre et des autres langages intermédiaires.

La première étape est la compilation du schéma d'évaluation qui est décrite par une famille de transformations ($\mathcal{V}$) de Λ vers Λ_s. La paire de combinateurs ($\mathbf{push}_s$, λ_s) spécifie une structure de données pour mémoriser les résultats intermédiaires (e.g. une pile de données). La principale alternative est d'utiliser des opérations d'application explicites (modèle "eval-apply") ou d'utiliser des marques (modèle "push-enter").

La famille de transformations ($\mathcal{A}$) de Λ_s vers Λ_e modélise la compilation de la β-réduction. Le langage Λ_e ne comprend plus un ensemble infini de varia-

bles et introduit la paire ($\mathbf{push}_e$, λ_e). Ces combinateurs se comportent comme $\mathbf{push}_s$ et λ_s et vérifient l'équivalent des propriétés (β_s, η_s). Ils agissent sur un composant conceptuellement différent (e.g. une pile d'environnements). La principale alternative est entre des environnements chaînés (favorisant la construction) et des environnements copiés (favorisant l'accès).

Une dernière transformation ($\mathcal{S}$) de Λ_e vers Λ_k est utilisée pour compiler les transferts de contrôle. Le langage Λ_k introduit la paire ($\mathbf{push}_k$, λ_k) qui spécifie un composant pour mémoriser les adresses de retour.

De nombreuses optimisations peuvent être exprimées dans ces différents langages. Nous en décrirons certaines ; néanmoins notre but est ici de modéliser fidèlement différentes machines abstraites, qui dans leurs formes épurées font peu d'optimisations. Des combinateurs, définis à l'aide de $\mathbf{push}_x$ et λ_x, sont décrits en même temps que les transformations. Pour alléger l'écriture, nous nous permettons d'utiliser des facilités syntaxiques comme les tuples $(x_1,\ldots,x_n)$ ou le filtrage $\lambda_x(x_1,\ldots,x_n).E$.

Cette hiérarchie de langages est une abstraction utile pour structurer et décrire les mises en œuvre. Il ne faut pas perdre de vue que chaque langage est en fait un sous-ensemble du λ-calcul. Un point notable est que nous n'avons pas à opter dès le départ pour une définition spécifique des combinateurs ($\circ$, $\mathbf{push}_x$,…). Nous imposons juste qu'elles respectent les propriétés (β_x) et (η_x). Les définitions peuvent être choisies en dernier lieu ; plusieurs choix possibles sont donnés en section 5.2.

3 Compilation du schéma d'évaluation

Les deux options de compilation du schéma d'évaluation sont l'utilisation d'application explicites ou l'utilisation de marques. La première option est la technique standard introduite dans la SECD et adoptée par la Cam et Tabac. La seconde fut évoquée dans [11] et utilisée dans la machine de Krivine stricte (que nous nommerons Maks) proposée dans [19].

3.1 La Cam

Les applications $E_1\ E_2$ sont mises en œuvre en évaluant la fonction E_1, l'argument E_2, et en appliquant le résultat de E_1 au résultat de E_2.

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

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

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

$\mathcal{V}a_L\ [\![E_1\ E_2]\!] = \mathcal{V}a_L\ [\![E_1]\!] \circ \mathcal{V}a_L\ [\![E_2]\!] \circ \mathbf{app}_L$ avec $\mathbf{app}_L = \lambda_s x.\lambda_s f.\ \mathbf{push}_s\ x \circ f$

Figure 2 Compilation de l'appel par valeur gauche-à-droite avec applications ($\mathcal{V}a_L$)

La Cam implante la version gauche-à-droite de l'appel par valeur et la transformation associée est décrite en Figure 2. Les règles peuvent s'expliquer intuitivement en lisant "retourne la valeur" pour $\mathbf{push}_s$, "évaluer" for $\mathcal{V}a_L$, "puis" pour $\circ$ et "appliquer" pour $\mathbf{app}_L$. $\mathcal{V}a_L$ produit des expressions bien typées. Sa correction s'exprime par la Propriété 2 qui établit que la réduction des expressions transformées ($\overset{*}{\rightarrow}$) simule la réduction en appel par valeur (*CBV*) des λ-expressions sources.

Propriété 2 $\forall E$ *fermée, V forme normale, CBV(E)*$\equiv V \Leftrightarrow \mathcal{V}a_L[\![E]\!] \overset{*}{\rightarrow} \mathcal{V}a_L[\![V]\!]$

Il est clairement inutile de rendre une fonction en résultat pour l'appliquer immédiatement après. Cette optimisation peut s'exprimer par la loi suivante

(L3) $E : R\sigma \quad \mathbf{push}_s\, F \circ E \circ \mathbf{app}_L = E \circ F$

Pour illustrer ce style d'optimisation, prenons le cas courant d'une fonction appliquée à tous ses arguments $(\lambda x_1.\ldots\lambda x_n.E_0)\ E_1 \ldots E_n$ alors

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

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

$\circ\ \mathcal{V}a_L[\![E_n]\!] \circ \mathbf{app}_L$

$= \mathcal{V}a_L[\![E_1]\!] \circ (\lambda_s x_1 \ldots \mathbf{push}_s(\lambda_s x_n.\mathcal{V}a_L[\![E_0]\!])\ldots) \circ \ldots \circ \mathcal{V}a_L[\![E_n]\!] \circ \mathbf{app}_L$ (L3)

$= \mathcal{V}a_L[\![E_1]\!] \circ (\lambda_s x_1.\mathcal{V}a_L[\![E_2]\!] \circ (\lambda_s x_2 \ldots \mathcal{V}a_L[\![E_n]\!] \circ (\lambda_s x_n.\mathcal{V}a_L[\![E_0]\!])\ldots))$ (L1)(L3)

$= \mathcal{V}a_L[\![E_n]\!] \circ \ldots \circ \mathcal{V}a_L[\![E_1]\!] \circ \lambda_s x_1 \ldots \lambda_s x_n.\mathcal{V}a_L[\![E_0]\!]$ (L2),(assoc)

Tous les combinateurs $\mathbf{app}_L$ ont été supprimés et nous avons évité la construction de *n* fermetures intermédiaires correspondant aux *n* fonctions unaires de l'expression $\lambda x_1.\ldots\lambda x_n.E_0$. La généralisation de cette optimisation constitue la phase de décurryfication présente dans de nombreuses mises en œuvre. On peut noter que dans notre cadre, $\lambda_s x_1.\ldots\lambda_s x_n.E$ dénote une fonction qui sera toujours appliquée à *n* arguments (autrement, il y aurait des $\mathbf{push}_s$ entre les λ_s).

Nous présentons maintenant un exemple de code produit par $\mathcal{V}a_L$ et détaillons sa réduction.

Exemple. Soit $E \equiv \lambda x.\lambda y.y\ (\lambda z.z\ x)$ alors après simplification par (L3) on a

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

Prenons $\mathbf{I} \equiv \lambda t.\ t$ et $\mathcal{V}a_L[\![I]\!] \equiv \mathbf{push}_s\ (\lambda_s t.\ \mathbf{push}_s\ t) \equiv \mathbf{push}_s\ \mathbf{I}_s$ alors

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

$\circ\ \mathbf{push}_s\ \mathbf{I}_s \circ \mathbf{app}_L \circ \mathbf{push}_s\ \mathbf{I}_s \circ \mathbf{app}_L$

$\overset{*}{\rightarrow}$ $\mathbf{push}_s(\lambda_s y.\ \mathbf{push}_s(\lambda_s z.\ \mathbf{push}_s\ \mathbf{I}_s \circ z) \circ y) \circ \mathbf{push}_s\ \mathbf{I}_s \circ \mathbf{app}_L$

$\overset{*}{\rightarrow}$ $\mathbf{push}_s(\lambda_s z.\ \mathbf{push}_s\ \mathbf{I}_s \circ z) \circ \mathbf{I}_s$

$\rightarrow$ $\mathbf{push}_s(\lambda_s z.\ \mathbf{push}_s\ \mathbf{I}_s \circ z) = \mathcal{V}a_L\ [\![\lambda z.z\ \mathbf{I}]\!]$

Une fermeture est construite pour chaque argument auquel E est appliqué ($\mathbf{push}_s$ devant $\lambda_s x$ et $\lambda_s y$). ❒

3.2 Tabac

Le compilateur Tabac est un produit dérivé de notre travail dans [13] et est l'inspirateur de cette étude. Il compile des programmes stricts ou paresseux par transformations de programmes. L'appel par valeur choisi par Tabac est de type droite-à-gauche. La transformation associée $\mathcal{V}a$ est identique à $\mathcal{V}a_L$ excepté la règle pour la composition qui devient

$$\mathcal{V}a\ [\![E_1\ E_2]\!] = \mathcal{V}a\ [\![E_2]\!] \circ \mathcal{V}a\ [\![E_1]\!] \circ \mathbf{app} \qquad \text{avec} \qquad \mathbf{app} = \lambda_s f.\ f$$

En fait, $\mathcal{V}a$ et $\mathcal{V}a_L$ peuvent être dérivées l'une de l'autre. Elles ont les mêmes propriétés de correction ; notamment la Propriété 2 reste valide pour $\mathcal{V}a$.

La décurryfication s'exprime plus simplement que dans le cas précédent à l'aide de la règle (L4)

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

Exemple. Pour l'exemple précédent, on obtient après simplification par (L4)

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

Ici encore, une fermeture est construite pour chaque argument auquel E est appliqué ($\mathbf{push}_s$ devant $\lambda_s x$ et $\lambda_s y$). ❒

3.3 La Maks

Au lieu d'évaluer la fonction, l'argument et de les appliquer, une autre solution est d'évaluer l'argument et d'appliquer directement la fonction non évaluée. Cette mise en œuvre est tout à fait naturelle pour l'appel par nom où une fonction n'est évaluée que quand elle est appliquée. En appel par valeur, une fonction peut aussi être évaluée en tant qu'argument. Dans ce cas, elle ne peut être directement appliquée et doit être rendue en résultat. Il faut donc un moyen à la fonction de déterminer si son argument est présent ou non: c'est le rôle des marques. La marque ε est supposée être une valeur distinguable des autres et les fonctions $\lambda x.E$ sont transformées en expressions de la forme $\mathbf{grab}_s\ (\lambda_s x.E')$. Après l'évaluation d'une fonction, un test est effectué: si une marque est présente (règle $\mathrm{grab}_s 1$), il n'y a pas d'argument disponible, et la

fonction est rendue en résultat (une fermeture est construite) sinon l'argument est présent (règle grab_s2) et la fonction peut s'appliquer.

$$(\text{grab}_s1) \qquad \mathbf{push}_s\ \varepsilon \circ \mathbf{grab}_s\ F \rightarrow \mathbf{push}_s\ F$$

et $$(\text{grab}_s2) \qquad \mathbf{push}_s\ V \circ \mathbf{grab}_s\ F \rightarrow \mathbf{push}_s\ V \circ F \qquad (V \not\equiv \varepsilon)$$

Cette technique évite de construire certaines fermetures au prix de tests dynamiques. Le combinateur $\mathbf{grab}_s$ et la marque ε pourrait se définir en Λ_s. En pratique, $\mathbf{grab}_s$ est implanté à l'aide d'une conditionnelle qui teste la présence d'une marque. La transformation de l'appel par valeur droite-à-gauche avec marques est décrite dans la Figure 3.

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

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

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

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

Figure 3 Compilation de l'appel par valeur droite-à-gauche avec marques ($\mathcal{V}m$)

La correction de $\mathcal{V}m$ s'exprime par la Propriété 3 qui établit que la réduction des expressions transformées simule la réduction en appel par valeur (*CBV*) des λ-expressions sources.

Propriété 3 $\forall E$ *fermée, V forme normale, CBV(E)*$\equiv V \Leftrightarrow \mathcal{V}m[\![E]\!] \overset{*}{\rightarrow} \mathcal{V}m[\![V]\!]$

Les règles de réduction de $\mathbf{grab}_s$ amènent deux nouvelles lois:

(L5) $\qquad \mathbf{push}_s\ \varepsilon \circ \mathbf{grab}_s\ F = \mathbf{push}_s\ F$

(L6) $E : R\sigma \qquad E \circ \mathbf{grab}_s\ F = E \circ F$

Exemple. Soit de nouveau l'expression $E \equiv \lambda x.\lambda y.y\ (\lambda z.z\ x)$, alors après simplification par (L5) et (L6) on obtient

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

Une fermeture n'est pas nécessairement construite pour chaque argument auquel E est appliqué ($\mathbf{grab}_s$ devant $\lambda_s x.$ et $\lambda_s y.$ en décide dynamiquement). ❐

Dans [19], la règle associée aux variables est $\mathcal{V}m[\![x]\!] = x$ alors que la définition de **Grab** est récursive (on a $\mathbf{push}_s\ \varepsilon \circ \mathbf{Grab}\ F = \mathbf{push}_s\ (\mathbf{Grab}\ F)$ au lieu de (L5)). Notre solution reste proche mais permet plus de simplifications.

Comme précédemment, quand une fonction est syntaxiquement appliquée à n arguments, le code peut être optimisé pour éviter n tests dynamiques. En fait, $\mathcal{V}m$ bénéficie du même style d'optimisations que $\mathcal{V}a$.

Un ordre d'évaluation de gauche à droite n'a pas grand sens avec cette option. L'idée des marques est d'éviter la construction de fermeture en testant si l'argument est présent. L'argument doit donc être évalué avant la fonction.

3.4 Comparaison

Notre cadre permet d'exhiber des complexités théoriques, de trouver des exemples pathologiques, ou encore d'argumenter formellement sur les avantages et inconvénients des techniques de compilation. Il va sans dire que ce style de comparaison ne se substitue pas aux expérimentations, toujours utiles pour prendre en compte des aspects plus complexes comme, par exemple, les interactions avec le cache ou le GC.

Nous donnons ici quelques éléments de comparaison des codes produits par les transformations précédentes (i.e. $\mathcal{V}m$ vs. $\mathcal{V}a_L$ ou $\mathcal{V}a$). Les expressions transformées sont sujettes aux mêmes optimisations et nous examinons seulement le code non optimisé. Les expressions produites par $\mathcal{V}m$ construisent moins de fermetures que le code correspondant produit par $\mathcal{V}a$ et une marque peut se coder par un bit dans une pile de bits parallèlement à la pile de d'arguments. La transformation $\mathcal{V}m$ produit un code vraisemblablement moins demandeur d'espace mémoire en moyenne. Pour ce qui concerne l'efficacité, la taille des expressions compilées donne une bonne approximation du surcoût de gestion de l'ordre d'évaluation introduit par les transformations. Il est facile de montrer que dans tous les cas l'expansion de code est linéaire en fonction de la taille de l'expression source. Plus précisément, si $\mathcal{T}aille[\![E]\!] = n$ alors $\mathcal{T}aille(\mathcal{V}[\![E]\!]) < 3\,n$ pour $\mathcal{V} = \mathcal{V}a$, $\mathcal{V}a_L$ ou $\mathcal{V}m$. Une comparaison plus précise est possible en associant aux combinateurs introduits un coût caractérisant la consommation espace ou temps. Si on prends *push* pour le coût d'empiler une variable ou une marque, *clos* pour le coût d'une construction de fermeture (i.e. $\mathbf{push}_S\ E$), *app* et *grab* pour le coût des combinateurs correspondants, et enfin n_λ et n_v pour le nombre de λ-abstractions et d'occurrences de variables dans l'expression source, on a :

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

et $$\mathcal{C}oût(\mathcal{V}m[\![E]\!]) = (n_\lambda + n_v)\ grab + (n_v\text{-}1)\ push$$

L'avantage de $\mathcal{V}m$ sur $\mathcal{V}a$ est de remplacer parfois une construction de fermeture et un **app** par un test et un **app**. Si *clos* est comparable au coût d'un test (par exemple, quand rendre une fermeture se résume à la construction d'une paire comme en 4.1) $\mathcal{V}m$ produira un moins bon code que $\mathcal{V}a$. Si la construction de fermeture n'est pas en temps constant (comme en section 4.2) $\mathcal{V}m$ peut être arbitrairement meilleur que $\mathcal{V}a$. En fait, la complexité du code peut même changer pour des programmes pathologiques. En pratique, la situation n'est pas si claire et le gain éventuel apporté par $\mathcal{V}m$ dépend de la probabilité

de présence d'une marque et du coût moyen d'une construction de fermeture par rapport à un test.

4 Compilation de la β-réduction

Cette transformation met en œuvre la substitution. Les variables sont remplacées par des combinateurs qui manipulent des environnements. La valeur d'une variable est extraite de l'environnement quand nécessaire. Grâce à la portée lexicale, les chemins d'accès aux valeurs dans l'environnement sont statiques. Comparé à Λ_s, Λ_e ajoute la paire (**push**$_e$, λ_e) qui est utilisée pour coder une pile d'environnements.

4.1 La Cam

La transformation $\mathcal{A}s$ (qui rappelle une sémantique dénotationnelle) est largement utilisée par les machines abstraites [7][18][19]. La structure de l'environnement est un arbre de fermetures et une fermeture est ajoutée à l'environnement en temps constant. D'un autre coté, l'accès aux valeurs se fait en suivant une chaîne d'indirections. La complexité temps de l'accès à une variable est O(n) où n est le nombre de λ_s depuis l'occurrence de la variable jusqu'à son λ_s liant (i.e. son nombre de Bruijn). La transformation $\mathcal{A}s$ (Figure 4) est faite relativement à un environnement statique ρ constitué de paires. L'entier i dans x_i dénote le rang de la variable dans l'environnement.

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

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

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

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

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

Figure 4 Abstraction avec des environnements partagés ($\mathcal{A}s$)

$\mathcal{A}s$ nécessite sept nouveaux combinateurs pour exprimer la sauvegarde et la restauration des environnements (**dupl**$_e$, **swap**$_{se}$), la construction et l'ouverture des fermetures (**mkclos**, **appclos**), l'accès aux valeurs (**fst**, **snd**), et enfin l'ajout d'une fermeture à l'environnement (**bind**). Ils sont définis dans Λ_e par

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

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

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

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

La correction de $\mathcal{A}s$ est exprimée par la Propriété 4 ($\mathcal{R}$ est une fonction changeant la représentation des fermetures de (c, e) en $\mathbf{push}_e\ e \circ c$).

Propriété 4 $\forall E\ \ \mathcal{R}[\![\mathbf{push}_e\ () \circ \mathcal{A}s[\![E]\!]\ ()]\!] =_\beta E$

La transformation $\mathcal{A}s$ peut être optimisée en ajoutant la règle :

$\mathcal{A}s[\![\lambda_s x.E]\!]\ \rho = \mathbf{pop}_{se} \circ \mathcal{A}s[\![E]\!]\ \rho$ *si x n'est pas libre dans E*

avec $\mathbf{pop}_{se} = \lambda_e e.\ \lambda_s x.\ \mathbf{push}_e\ e$

Les variables sont liées aux fermetures stockées dans les environnements. Avec les règles initiales, $\mathcal{A}s[\![\mathbf{push}_s x_i]\!]$ construirait encore une nouvelle fermeture. Cet emboîtement ("boxing") inutile peut être évité grâce à la règle:

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

Dans le cas de la Cam, il est utile d'introduire un combinateur représentant la version abstraite de $\mathbf{app}_L$: $\mathbf{appclos}_L = \lambda_s x.\lambda_s(f,e).\ \mathbf{push}_s\ x \circ \mathbf{push}_e\ e \circ f$

On pose $Cam\ [\![E]\!]\ \rho = \mathcal{A}s\ (\mathcal{V}a_L\ [\![E]\!])\ \rho$ et on obtient:

$Cam : \Lambda \rightarrow env \rightarrow \Lambda_e$

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

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

$Cam\ [\![E_1\ E_2]\!]\ \rho = \mathbf{dupl}_e \circ Cam\ [\![E_1]\!]\ \rho \circ \mathbf{swap}_{se} \circ Cam\ [\![E_2]\!]\ \rho \circ \mathbf{appclos}_L$

Figure 5 Schéma de compilation de la Cam ($\mathcal{A}s \circ \mathcal{V}a_L$)

Il existe des différences cosmétiques entre notre description et la machine réelle. Les combinateurs **fst**, **snd, dupl**$_e$ et **swap**$_{se}$ correspondent aux **Fst**, **Snd**, **Push** et **Swap** de la Cam. La séquence $\mathbf{push}_s\ (E) \circ \mathbf{mkclos}$ est équivalente au **Cur**(E) de la Cam. La seule différence réelle vient de la position de **bind** (au début de chaque fermeture dans notre cas). En décalant ce combinateur jusqu'à l'endroit où les fermetures sont évaluées (i.e. en le fusionnant avec $\mathbf{appclos}_L$), nous obtenons $\lambda_s x.\lambda_s(f,e).\ \mathbf{push}_s\ x \circ \mathbf{push}_e\ e \circ \mathbf{bind} \circ f$, qui est exactement la séquence **Cons;App** de la Cam.

Exemple. Soit $E \equiv (\lambda x.\lambda y.\ y\ (\lambda z.z\ x))$ alors

$Cam\ [\![E]\!]\ () \equiv \mathbf{push}_s(\mathbf{bind} \circ \mathbf{push}_s(\mathbf{bind} \circ \mathbf{dupl}_e \circ \mathbf{snd} \circ \mathbf{swap}_{se} \circ \mathbf{push}_s(\mathbf{bind} \circ \mathbf{dupl}_e \circ \mathbf{snd} \circ \mathbf{swap}_{se} \circ \mathbf{fst}^2 \circ \mathbf{snd} \circ \mathbf{appclos}_L) \circ \mathbf{mkclos} \circ \mathbf{appclos}_L) \circ \mathbf{mkclos}) \circ \mathbf{mkclos}$

Une fermeture est construite pour chaque argument auquel E est appliqué. L'accès aux variables suit une chaîne d'indirections (e.g. $\mathbf{fst}^2 \circ \mathbf{snd}$ pour x dans l'abstraction $(\lambda z.z\ x)$). Les environnements associés aux fermetures contien-

nent des valeurs inutiles (e.g. *y* pour la fermeture de l'abstraction ($\lambda z.z\ x$) où *y* n'apparaît pas libre). ❒

Notons que le combinateur **mkclos** peut être évité par une abstraction qui déplie les paires (code,env) dans les environnements comme dans Tim [11].

4.2 Tabac

Un autre choix consiste à garantir un temps d'accès constant [1][13]. Dans ce cas, la structure de l'environnement doit être un vecteur de fermetures. Un code qui copie les environnements (une opération en O(*longueur* ρ)) doit être inséré dans $\mathcal{A}s$ de manière à éviter le partage et les indirections.

Le macro combinateur **Copy** ρ génère un code qui copie un environnement conformément à la structure de ρ.

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

Les combinateurs $\mathbf{get}_i$ sont des abréviations de $\mathbf{fst}^i \circ \mathbf{snd}$. Néanmoins, si les environnements sont représentés par des vecteurs, $\mathbf{get}_i$ peut être considéré comme une opération en temps constant et **bind** peut être vu comme ajoutant un nouvel élément à un vecteur.

Dans [9], nous avons identifié trois abstractions selon l'instant où sont réalisées les copies. La solution choisie par Tabac (Figure 6) est de copier l'environnement à la construction et à l'ouverture des fermetures. La copie à l'ouverture permet d'ajouter simplement de nouvelles liaisons dans une mémoire contiguë (l'environnement doit rester un vecteur). Nous présentons seulement les règles qui différent du schéma $\mathcal{A}s$.

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

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

Figure 6 Abstraction avec des environnements copiés ($\mathcal{A}c$)

L'environnement $\overline{\rho}$ représente ρ restreint aux variables libres de la sous-expression *E*. On pose: $\mathcal{T}abac\ [\![E]\!]\ \rho = \mathcal{A}c\ (\mathcal{V}a[\![E]\!])\ \rho$ et on a alors:

$\mathcal{T}abac : \Lambda \rightarrow env \rightarrow \Lambda_e$

$\mathcal{T}abac\ [\![x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{get}_i$

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

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

Figure 7 Schéma de compilation de Tabac ($\mathcal{A}c \circ \mathcal{V}a$)

Exemple. Soit $E \equiv (\lambda x.\lambda y.\ y\ (\lambda z.z\ x))$ alors

$\mathcal{T}abac\ [\![E]\!]\ () \equiv$ **Copy** () ∘ **push**$_s$(**Copy** () ∘ **bind** ∘ **Copy** ((),x) ∘ **push**$_s$(**Copy** ((),x) ∘ **bind** ∘ **dupl**$_e$ ∘ **Copy** ((),x) ∘ **push**$_s$(**Copy** ((),x) ∘ **bind** ∘ **dupl**$_e$ ∘ **get**$_1$ ∘ **swap**$_{se}$ ∘ **get**$_0$ ∘ **appclos**) ∘ **mkclos** ∘ **swap**$_{se}$ ∘ **get**$_0$ ∘ **appclos**) ∘ **mkclos**) ∘ **mkclos**

Une fermeture est construite pour chaque argument auquel E est appliqué (**push**$_s$ et **Copy** devant $\lambda_s x.$ et $\lambda_s y.$ et **mkclos**). Les copies de l'environnement sont restreintes aux valeurs nécessaires (e.g. **Copy** ((),x) plutôt que **Copy** (((),x),y) pour la fermeture de l'abstraction ($\lambda z.z\ x$) où y n'apparaît pas libre). L'accès aux variables est direct (e.g. **get**$_1$ pour x dans l'abstraction ($\lambda z.z\ x$)). ❐

Les trois règles précédentes, constituent un schéma simpliste qui n'exploite pas la forme des expressions sources. L'implantation réelle optimise le code à chaque étape (i.e. dans Λ_s et Λ_e) et les environnements sont dépliés dans la pile de données. Ceci peut également se décrire dans notre cadre par des transformations de programmes.

4.3 La Maks

La machine de Krivine stricte utilise la même abstraction que la Cam. Une version abstraite du macro-combinateur **grab**$_s$ doit être produite. Abstraire naïvement la définition de **grab**$_s$ produit une version abstraite inefficace. Aussi nous introduisons ici un nouveau combinateur **grab**$_e$. Et nous rajoutons une règle à $\mathcal{A}s$:

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

Comme **grab**$_s$, le combinateur **grab**$_e$ a deux règles de réduction:

(L7) **push**$_s$ ε ∘ **push**$_e$ e ∘ **grab**$_e$ F ➨ **push**$_e$ e ∘ **push**$_s$ F ∘ **mkclos**

(L8) $E : R\sigma$ $\quad E$ ∘ **push**$_e$ e ∘ **grab**$_e$ F ➨ E ∘ **push**$_e$ e ∘ F

Les variables sont liées aux fermetures stockées dans l'environnement. Avec la règle (L7), $\mathcal{A}s\ [\![\mathbf{grab}_s\ x]\!]$ construit encore une nouvelle fermeture. Cet emboîtement ("boxing") inutile peut être évité grâce à la règle (L9).

(L9) **push**$_s$ ε ∘ **push**$_e$ e ∘ **grab**$_e$ (**fst**^{i} ∘ **snd** ∘ **appclos**) ➨ **push**$_e$ e ∘ **fst**^{i} ∘ **snd**

Le choix de la règle (L7) ou (L9) pour la réduction de **grab**$_e$ F peut être réalisée sans test dynamique sur la forme de F, en introduisant un combinateur **grab**$_e$**'** spécifique aux variables.

Il est bien sur inutile d'abstraire **push**$_s$ ε. Une marque peut être assimilée à une constante et construire une fermeture n'est pas nécessaire dans ce cas. De nouveau, il faut noter que **grab**$_e$ peut être défini dans Λ_e.

On pose $\mathcal{M}aks\ [\![E]\!]\ \rho = \mathcal{A}s\,(\mathcal{V}m[\![E]\!])\ \rho$ et on a alors:

$\mathcal{M}aks : \Lambda \rightarrow env \rightarrow \Lambda_e$

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

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

$\mathcal{M}aks\ [\![E_1\ E_2]\!]\ \rho = \mathbf{dupl}_e \circ \mathbf{push}_s\ \varepsilon \circ \mathbf{swap}_{se} \circ$

$\mathcal{M}aks\ [\![E_2]\!]\ \rho \circ \mathbf{swap}_{se} \circ \mathcal{M}aks\ [\![E_1]\!]\ \rho$

Figure 8 Schéma de compilation de la Maks ($\mathcal{A}s \circ \mathcal{V}m$)

Exemple. Soit $E \equiv (\lambda x.\lambda y.\ y\ (\lambda z.z\ x))$ alors

$\mathcal{M}aks\ [\![E]\!]\ () \equiv \mathbf{grab}_e(\mathbf{bind} \circ \mathbf{grab}_e(\mathbf{bind} \circ \mathbf{dupl}_e \circ \mathbf{push}_s\ (\mathbf{bind} \circ \mathbf{dupl}_e \circ \mathbf{fst}^2 \circ \mathbf{snd} \circ \mathbf{swap}_{se} \circ \mathbf{snd} \circ \mathbf{appclos}) \circ \mathbf{mkclos} \circ \mathbf{swap}_{se} \circ \mathbf{snd} \circ \mathbf{appclos}))$

Une fermeture n'est pas nécessairement construite pour chaque argument auquel E est appliqué ($\mathbf{grab}_e$ devant **bind** en décide dynamiquement). Comme pour la Cam, l'accès aux variables implique le parcours d'une chaîne d'indirections et les environnements associés aux fermetures contiennent des valeurs inutiles. ❒

Il existe ici encore des différences cosmétiques entre notre description et celle de [19]. Le combinateur **Access**(n) est équivalent à la séquence $\mathbf{fst}^n \circ \mathbf{snd} \circ \mathbf{appclos}$, et **Push**(E) est équivalent à $\mathbf{dupl}_e \circ \mathbf{push}_s\ E \circ \mathbf{mkclos} \circ \mathbf{swap}_{se}$. La forme **Grab**;E correspond à $\mathbf{grab}_e(\mathbf{bind} \circ E) \circ \mathbf{swap}_{se}$, et la forme **Reduce**(E) correspond à la séquence $\mathbf{dupl}_e \circ \mathbf{push}_s\ \varepsilon \circ \mathbf{swap}_{se} \circ E$. De plus, ces deux dernières formes manipulent explicitement les adresses de retour. Cette gestion explicite du contrôle est introduite dans la section 5.1.

Une optimisation introduite dans [8], peut se décrire dans notre cadre, où $\lambda_s x_1 \ldots \lambda_s x_n.E$ dénote une fonction toujours appliquée à au moins n arguments. Les fermetures peuvent dans ce cas être regroupées dans l'environnement, et les indirections correspondantes supprimées sans aucune perte de partage. La structure en liste de l'environnement devient localement un vecteur, et les accès aux variables sont modifiés en conséquence.

4.4 Comparaison

La taille de l'expression abstraite donne une première approximation du surcoût introduit par le codage de la β-réduction. Il est facile de montrer que l'expansion du code est quadratique en fonction de la taille de l'expression source (ces bornes sont à rapprocher de la croissance quadratique impliquée par l'algorithme d'abstraction de Turner [29]). Plus précisément :

si $\mathcal{T}aille\ [\![E]\!] = n$ alors $\mathcal{T}aille(\mathcal{A}s(\mathcal{V}a\ [\![E]\!])) \leq n_\lambda n_v - n_v + 6n + 6$

avec n_λ le nombre de λ-abstractions et n_v le nombre d'occurrences de variables ($n = n_\lambda + n_v$) de l'expression source. Cette expression atteint un maximum pour $n_v = (n-1)/2$. Le produit $n_\lambda n_v$ indique que l'efficacité de $\mathcal{A}s$ dépend aussi bien du nombre d'accès (n_v) que de leurs longueurs (n_λ). Pour $\mathcal{A}c$, on a

si $\mathcal{T}aille\ [\![E]\!] = n$ alors $\mathcal{T}aille(\mathcal{A}c(\mathcal{V}a\ [\![E]\!])) \leq 6n_\lambda^2 - 6n_\lambda + 7n + 6$

ce qui montre que l'efficacité de $\mathcal{A}c$ ne dépend pas des accès. Par exemple l'expression source $\lambda x_1 \ldots \lambda x_n.\ x_n \ldots x_1$ produit n-1 fermetures (une pour chaque application partielle potentielle) avec des environnements de taille 1,2,…n-1. Avec $\mathcal{A}c$, l'expansion de code est quadratique puisque les codes copiant les environnements nécessitent (de l'ordre de) 1,2,…n-1 instructions.

Les abstractions ont le même ordre de complexité, néanmoins l'une peut être mieux adaptée que l'autre à une expression source donnée. Ces complexités mettent en évidence la principale différence entre les environnements partagés qui favorisent la construction et les environnements copiés qui favorisent l'accès. Les abstractions pourraient aussi être comparées vis-à-vis de leur consommation mémoire.

5 Compilation vers du code machine

Dans cette section, nous rendons explicite les transferts de contrôle et proposons des définitions pour les combinateurs. Après ces étapes, les expressions transformées peuvent être vues comme du code machine réaliste.

5.1 Un schéma de compilation commun

Une machine conventionnelle exécute du code linéaire où chaque instruction est basique. Le code obtenu à l'étape précédente s'en rapproche, mais il reste à rendre les appels et retours explicites. Dans notre cadre, réduire des expressions de la forme **appclos** ∘ *E* implique d'évaluer une fermeture, puis de retourner à *E*. Certaines machines abstraites n'explicitent pas cette étape.

La transformation $\mathcal{S}$ décrite Figure 9 modélise la sauvegarde et restauration des adresses de retour. Elle est commune aux trois machines présentées. Elle explicite la sauvegarde de l'adresse de retour (i.e. le code suivant un appel de fonction) en utilisant $\mathbf{push}_k$, et les retours avec $\mathbf{rts}_s$ ($= \lambda_s x.\lambda_k f.\ \mathbf{push}_s\ x \circ f$). Intuitivement ces combinateurs peuvent être vus comme implantant une pile de contrôle. Comparé à Λ_e, les expressions dans Λ_k ne comportent plus de séquences **appclos** ∘ *E*.

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

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

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

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

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

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

Figure 9 Compilation des transferts de contrôle ($\mathcal{S}$)

Le combinateur $\mathbf{swap}_{ke} = \lambda_k x.\ \lambda_e e.\ \mathbf{push}_k\ x \circ \mathbf{push}_e\ e$ est nécessaire pour intégrer le nouveau composant *k* avec les autres. Le code résultant peut être simplifié afin d'éviter les ruptures de séquence inutiles. Un pas de plus vers du code machine réel serait d'introduire des labels pour nommer les séquences de codes (tels *E* dans $\mathbf{push}_x\ E$). Ce schéma défini sur Λ_e peut être étendu pour tenir compte des optimisations ou d'autres combinateurs comme $\mathbf{appclos}_L$.

Dans le cas du schéma avec marques, **appclos** est dissimulé dans $\mathbf{grab}_e$, et ce sont les codes de la forme ($\mathbf{grab}_e\ F) \circ E$ qui nous intéressent. La transformation doit être étendue et une nouvelle version de $\mathbf{grab}_e$ définie.

5.2 Instanciation des composantes

Les paires de combinateurs (λ_s, $\mathbf{push}_s$), (λ_e, $\mathbf{push}_e$), et (λ_k, $\mathbf{push}_k$) n'ont pas encore de définitions. Chaque paire peut être vue comme le codage d'une composante de l'état d'une machine abstraite sous-jacente. Leurs règles de réduction spécifient un système de transition d'états. Nous pouvons maintenant choisir de garder les composantes séparées ou de les mélanger (toutes ou seulement certaines). Ces deux possibilités partagent la même définition de l'opérateur de séquencement: $\circ = \lambda xyz.\ x\ (y\ z)$.

Conserver les composantes séparées apporte de nouvelles propriétés, permettant des simplifications et des déplacements de code. Le séquencement de deux combinateurs manipulant des composantes différentes est alors commutatif et des combinateurs administratifs comme $\mathbf{swap}_{se}$ deviennent inutiles. Voici des définitions possibles (*c, s, e* étant des variables fraîche):

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

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

et de même pour (λ_k, $\mathbf{push}_k$). La réduction de nos expressions peut être vue comme un système de transition d'états d'une machine abstraite, e.g. :

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

Une seconde possibilité est de fusionner toutes les composantes. Ici, les combinateurs administratifs restent nécessaires.

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

Nous décrivons uniquement l'instanciation des composants pour la Cam. Une même pile contient les fermetures et des copies de l'environnement (i.e. du pointeur d'environnement). La composante "environnement courant" de la Cam est représentée dans notre cas par l'élément au sommet de l'unique pile. On obtient alors par exemple:

$\mathbf{push}_s\ V\ C\ S \rightarrow C\ (S,V)$ $\quad$ $\mathbf{mkclos}\ C\ ((S,E),F) \rightarrow C\ (S,(F,E))$

$\mathbf{dupl}_e\ C\ (S,E) \rightarrow C\ ((S,E),E)$ $\quad$ $\mathbf{swap}_{se}\ C\ ((S,E),X) \rightarrow C\ ((S,X),E)$

Nous récapitulons en Figure 10 les transformations modélisant les trois implantations étudiées.

Implantation	Transformations			Composantes
Cam	$\mathcal{V}a_L$	$\mathcal{A}s$	$\mathcal{S}$	s ≡ e ≡ k
Tabac	$\mathcal{V}a$	$\mathcal{A}c$	$\mathcal{S}$	(s ≡ e) k
Maks	$\mathcal{V}m$	$\mathcal{A}s$	$\mathcal{S}$	s (e ≡ k)

Figure 10 Différent schémas de compilation et instanciation de composantes

6 Conclusion

Dans cet article, nous avons présenté un cadre pour décrire, prouver et comparer les techniques de mise en œuvre et d'optimisation des langage fonctionnels. Nous l'avons utilisé pour modéliser trois implantations: la Cam, le compilateur Tabac et la machine de Krivine stricte. Nous avons ainsi mis en évidence leurs différences et similitudes fondamentales.

Notre premier langage intermédiaire Λ_s montre de grande similarités avec les expressions en CPS. En effet, il existe une définition des combinateurs (λ_s, $\mathbf{push}_s$) pour laquelle on obtient naturellement la transformation CPS de Fischer [12] (ce point est détaillé dans [10]). D'un autre coté, nos combinateurs ne sont pas complètement définis ; ils doivent juste respecter quelques propriétés). De ce point de vue, Λ_s apparaît comme un cadre puissant, plus abstrait que la CPS, pour spécifier les stratégies de réduction.

Les travaux connexes incluent la dérivation de machines abstraites à partir d'une sémantique dénotationnelle [30] ou opérationnelle [14] [27]. Leur but est de fournir une méthodologie à la dérivation formelle de mises en œuvre pour une classe (potentiellement large) de langages de programmation. Quel-

ques travaux explorent les relations entre deux machines abstraites comme la Tim et la G-machine [4][25] ou la CMCM et la Tim [21]. Le but est de montrer l'équivalence entre des machines apparemment très différentes. Mentionnons également Asperti [2] qui fournit une compréhension catégorique de la machine de Krivine et d'une Cam étendue. Enfin, différentes machines abstraites ont été prouvées correctes et leur stratégie de réduction a été mise en évidence par bi-simulation dans un λ-calcul avec substitutions explicites (λσ-calcul)[23]. Certaines transitions de la machine abstraite ne sont cependant pas observables dans la simulation. Aussi cette modélisation n'a pas encore permis de comparer précisément les machines.

Notre approche se concentre sur la description et la comparaison de choix fondamentaux. L'utilisation de transformations de programme est apparue comme bien adaptée pour modéliser précisément et complètement le processus de compilation. Ce cadre unifié simplifie les preuves de correction et rend possible le raisonnement à propos de l'efficacité du code produit ainsi qu'à propos de la complexité des transformations elles-mêmes. De nombreuses extensions (constantes, opérateurs primitifs, structures de données, appel par nom, machines hybrides) et optimisations standards (décurryfication, déboîtage ou "unboxing", hoisting, optimisations locales de code) peuvent aussi être décrites dans ce cadre [10]. Notre but à moyen terme est de fournir une taxonomie générale des implantations connues de langages fonctionnels. La dernière tâche difficile est l'expression des mises à jour destructives. Ceci est crucial afin de décrire complètement l'appel par nécessité et la réduction de graphe. Nous suspectons que cela compliquera les preuves de correction, mais ceci peut rester confiné dans une dernière étape. Toutes les transformations, preuves de correction, et optimisations des autres étapes resteraient valides. Nous travaillons actuellement sur ce sujet. Concernant l'appel par valeur et l'appel par nom, rien ne devrait nous empêcher de compléter notre étude des mises en œuvre existantes.

References

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

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

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

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

[5] G. Burn and D. Le Métayer. Proving the correctness of compiler optimisations based on a global analysis. *Journal of Functional Programming*, Jan. 1996.

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

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

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

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

[10] R. Douence and P. Fradet. A taxonomy of functional language implementations. Part I: Call-by-Value, *Rapport de recherche IRISA 972*, 1995.

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

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

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

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

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

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

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

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

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

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

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

[22] E. Meijer and R. Paterson. Down with lambda lifting. *copies disponibles par email* (erik@cs.kun.nl), 1991.

[23] B. Pagano. Bi-Simulation de Machines Abstraites en λσ-calcul. *Journées Francophones des Langages Applicatifs, INRIA Collection Didactique*, 1995.

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

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

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

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

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

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

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