Interprétation par SK-traduction et syntaxe abstraite d'ordre supérieur

Julien Cohen

Laboratoire de Recherche en Informatique (LRI)
Université Paris-Sud

Laboratoire de Méthodes Informatiques (LaMI)
Université d'Évry Val d'Essonne
`jcohen@lami.univ-evry.fr`

Dans cet article nous exposons certaines des techniques utilisées pour l'implémentation de l'interprète MGS. De manière générale, ces techniques peuvent être utilisées pour le développement rapide d'interprètes pour des langages fonctionnels avec traits impératifs. Nous nous concentrons sur la réalisation de l'*évaluateur* qui est le module dédié à l'interprétation du noyau fonctionnel du langage. L'originalité de cette implémentation est qu'elle repose sur la réduction des applications par le langage hôte au lieu de la gérer elle-même. Des objectifs de simplicité, d'orthogonalité et d'efficacité ont motivé ce choix. Ce type d'approche est connu sous le nom de syntaxe abstraite d'ordre supérieur [PE88] et n'a jamais été utilisée, à notre connaissance, pour le développement d'un interprète. Nous utilisons une technique de *combinatorisation* afin de transformer les fonctions définies par l'utilisateur en fonctions du langage hôte. Les fonctions non-strictes et les traits impératifs sont intégrés à ce schéma grâce à l'utilisation de *glaçons*. Nous serons donc amenés à étudier la SK-traduction en présence de traits impératifs et de fonctions non-strictes.

Organisation de l'article. La section 1 détaille les raisons qui nous ont poussés à choisir cette approche. Nous introduisons dans la section suivante une implémentation *classique* d'un interprète puis nous la comparons à l'approche d'ordre supérieur en section 3. La réalisation de notre approche par syntaxe d'ordre supérieur et combinatorisation est présentée en section 4. Elle est ensuite étendue aux fonctions non-strictes (section 5) et aux traits impératifs (section 6). Nous évoquons des optimisations possibles en section 7. La section 8 conclue cet article en dressant un bref bilan.

1. Objectifs et motivations

Dans cet article, nous étudions la manière de représenter une valeur fonctionnelle du langage interprété (ici MGS, voir [Coh04] pour une présentation complète du langage) par une valeur fonctionnelle du langage qui sert à implémenter notre interprète (ici OCaml [LDGV02]). Ce type de représentation simplifie l'écriture de l'interprète et relève de plusieurs domaines de recherche dont la logique combinatoire et la syntaxe abstraite d'ordre supérieur.

Trois problèmes se posent avec cette approche :

1. La seule manière de construire une valeur fonctionnelle comme *résultat d'un calcul* en Caml est par application de fonctions[1]. Or, les fonctions du langage interprété peuvent être définies

[1] À moins de disposer de techniques spéciales relevant de la programmation multi-niveaux, on ne peut produire

par des abstractions. Il faut donc pouvoir construire la fonction définie par cette abstraction uniquement par des applications. Ce point est résolu grâce à la traduction en combinateurs.

2. Certaines expressions du langage que nous interprétons correspondent à des fonctions non-strictes alors que l'on ne peut définir dans le langage d'implémentation que des fonctions strictes et à appel par valeur. La conditionnelle *if-then-else*, la conjonction logique et la disjonction sont des exemples de fonctions non-strictes. La solution est l'utilisation de glaçons.
3. Le langage à interpréter comporte des traits impératifs. Or, il est délicat de mélanger certains traits impératifs avec le séquencement des calculs induit uniquement par les appels fonctionnels (par exemple, l'opérateur de séquencement ne peut s'exprimer par une fonction lorsque l'ordre d'évaluation des arguments n'est pas spécifié). Là aussi, la solution passe par l'utilisation de glaçons.

Si l'évaluation d'un noyau fonctionnel dans un interprète est un problème classique, l'approche adoptée ici est originale. Elle répond aux six contraintes suivantes qui ont été posées *a priori* dans le cadre du projet MGS :

1. Nous voulons un schéma d'évaluation interprété.
2. L'évaluateur doit pouvoir s'implémenter indifféremment dans tout langage « raisonnable » (qui permet par exemple de manipuler des fonctions), doit pouvoir être porté (et a été porté dans notre cas) sur des environnement différents, et enfin, nous ne voulons pas que l'implémentation dépende de spécificités du langage d'implémentation. La stratégie d'évaluation développée dans cet article ne requière que la possibilité de passer des fonctions en paramètre à une autre fonction et de retourner des fonctions comme résultat d'une application de fonction. Nous avons choisi OCaml comme langage d'implémentation, mais le schéma d'évaluation proposé reste valable pour d'autres langages.
3. L'évaluateur doit rester simple (tout au plus quelques centaines de lignes de code) et extensible, tout en offrant des structures de contrôle utiles comme les exceptions et des outils de trace et de débogage. Bien que nous ne le détaillerons pas, l'introduction dans notre interprète d'un mécanisme d'exception à la OCaml et de possibilités de tracer des appels de fonctions est réalisée par moins d'une cinquantaine de lignes de code à partir du schéma d'évaluation proposé ici.
4. La performance n'est pas le souci premier, mais l'évaluateur doit néanmoins être le plus efficace possible sans sacrifier l'extensibilité et la simplicité du schéma d'évaluation adopté. Notre objectif est d'être plus efficace que des langages interprétés comme Mathematica ou Python sur l'aspect fonctionnel.
5. Le schéma d'évaluation adopté doit permettre le développement et *l'intégration immédiate* de bibliothèques développées dans le langage d'implémentation et l'intégration de bibliothèques développées dans un autre langage (comme C et C++). Ces bibliothèques doivent pouvoir se concevoir et se développer de manière complètement indépendante de l'implémentation de l'évaluateur proprement dit.
6. La notion de fonction offerte par le langage doit être uniforme. En particulier, un programmeur ne doit pas pouvoir distinguer une fonction qu'il a écrite lui-même dans le langage source d'une fonction « primitive » implémentée par une des bibliothèques évoquées dans le point précédent. Ainsi, une opération primitive d'ordre supérieur doit pouvoir accepter en argument aussi bien une fonction écrite par l'utilisateur qu'une autre primitive prédéfinie, et cela de manière transparente aussi bien pour le concepteur de la bibliothèque que pour le programmeur. C'est une propriété forte mais qui permet d'écrire simplement des bibliothèques puissantes, facilement paramétrables et extensibles.

Les points 2 et 5 nous font renoncer aux approches de type « traduction au vol vers un langage interprété » comme celle permise par Camlp4 [dR02]. La contrainte 3 exclue les approches de type

automatiquement (et facilement) la fonction OCaml associée à une expression arbitraire en syntaxe abstraite (même en supposant que l'on produit toujours une fonction d'un type donné).

« compilation vers une machine virtuelle » , du moins si l'on veut que le résultat soit raisonnablement efficace[2]. La contrainte 6 ne peut pas être assurée par les approches classiques par évaluation des termes comme décrit à la section 2.

En fait, prise au sens littéral, cette dernière contrainte impose que les fonctions écrites par le programmeur soient traduites au vol en des fonctions du langage d'implémentation. Cependant, puisque le langage d'implémentation est compilé, cela n'est possible que si les fonctions sont des valeurs puisque les seules entités qui peuvent se construire lors de l'exécution sont des valeurs. Les opérations permettant de construire des valeurs fonctionnelles dans un langage de programmation ne sont pas nombreuses. Nous ne voulons pas bien sûr embarquer de manière plus ou moins explicite un interprète du langage d'implémentation dans l'interprète construit. La *logique combinatoire* offre alors une solution : toute fonction peut se construire comme résultat de l'application d'un jeu réduit de fonctions primitives, les *combinateurs*.

L'idée développée dans cet article est de faire reposer l'évaluation du noyau fonctionnel du langage source uniquement sur le mécanisme d'application de fonctions du langage d'implémentation et d'associer dynamiquement à chaque fonction écrite dans le langage source, une fonction du langage d'implémentation calculée au vol dans l'interprète.

Cette approche diffère de l'évaluation par réduction de combinateurs proposée par [PJ87] et développée notamment dans [PJS89, Hug82] (et qui relèvent de la catégorie des machines virtuelles) ou de la construction d'un réseau de fermetures dans [FL87], qui ne permet pas de faire abstraction des manipulations explicites d'environnements. L'approche originale que nous décrivons dans cet article mêle des idées issues des travaux sur la « syntaxe abstraite d'ordre supérieur » avec des notions issues de la logique combinatoire.

Dans la suite de cet article, nous étudions un schéma d'évaluation qui satisfait les contraintes décrite ci-dessus. Nous ne considérerons pas les « transformations » qui sont des fonctions définies par filtrage propres au langage MGS car les mécanismes de filtrage sont indépendants du problème posé et sont traitées à un autre niveau (voir par exemple [GMC02] pour l'implémentation du filtrage dans MGS). Notons toutefois que les gardes des motifs et les parties droites des règles de filtrage sont vues comme des fonctions des valeurs filtrées et sont donc traitées uniformément selon le schéma que nous présentons ici.

2. Architecture classique d'un interprète

2.1. Langage considéré

L'approche adoptée est assez peu habituelle. Nous allons donc l'introduire en partant de l'implémentation classique d'un interprète. Considérons le langage défini par le type `term` suivant :

[2] Une version de MGS a été développée avec un évaluateur correspondant à la machine virtuelle SECD [Lan65a]. Cette approche s'est révélée moins efficace que l'approche par interprétation directe des termes, comme décrit à la section 2, du moins tant que l'on en restait à une compilation directe sans optimisation particulière. Ce schéma d'évaluation ne répondait pas non plus aux contraintes 5 et 6.

```
type term =
  | Int of int
  | Var of ident
  | Abs of ident * term
  | App of term * term
  | ExtFun of value -> value
  | If of term * term * term
  | Get of memident
  | Set of memident * term
```

```
type value =
  | ValInt of int
  | ValFun of value -> value
  | ValAbs of ident * term * env

and env = (ident * term) list
```

Le type `term` définit la syntaxe abstraite du langage que nous interprétons. Ce type est exprimé dans le langage OCaml car notre interprète est implémenté dans ce langage. Toutefois nous souhaitons pouvoir adapter les idées présentées à tout langage pouvant manipuler des fonctions et ayant une stratégie d'évaluation d'appel par valeur (comme C ou C++ par exemple). Le langage choisi (ici OCaml) sera appelé le *langage hôte* ou langage d'implémentation.

Détaillons les constructions du langage. Le terme `Int 1` dénote l'entier 1. Le terme `Var "x"` dénote une occurrence de la variable x (ici, le type `ident`, comme le type `memident`, vaut `string`). Un terme `Abs ("x",`e`)` correspond à la fonction définie par l'abstraction de x dans e. Par exemple, `Abs ("x",Var "x")` correspond à la fonction identité que l'on écrirait `fun (x) = x` en syntaxe concrète (ici, la syntaxe MGS). Un terme `ExtFun` f dénote une *fonction prédéfinie* du langage. On parle aussi de *fonction externe* car on peut utiliser une fonction définie dans une bibliothèque. Cette fonction doit avoir le type `value -> value` où `value` est le type des valeurs renvoyées par la fonction d'évaluation. L'addition entre entiers est un exemple de fonction prédéfinie :

```
let plus = fun (ValInt i) -> ValFun (fun (ValInt j) -> ValInt (i+j) )
```

On écrira donc `ExtFun plus` pour la dénoter dans cette syntaxe abstraite. Notons que toutes les fonctions sont curryfiées dans cette syntaxe. La conditionnelle `If(`e_1`, `e_2`, `e_3`)` correspond au *if-then-else* et évalue sa branche *then* lorsque la condition e_1 est un entier strictement positif, et la branche *else* dans les autres cas. Un terme `Set(`m`,`e`)` dénote l'affectation à la mémoire m la valeur de l'expression e (notée `m := e` en syntaxe concrète). Enfin, un terme `Get` m dénote la valeur du contenu de la mémoire m (simplement notée `m` en syntaxe concrète, on suppose que la distinction entre une variable et une mémoire se fait à l'analyse syntaxique). Exemple :

```
Abs("x",If(Var "x", App( App (ExtFun plus, Var "x"), Int 1), Get m))
```

correspond à l'expression suivante : `fun (x) = if x then x+1 else m fi`.

Le langage considéré suffit à illustrer les difficultés rencontrées. Nous nous limitons aux entiers comme structures de données car l'objet de cet article est l'implémentation des fonctions. Nous ne considérons pas le séquencement dans le langage car celui-ci peut être encodé avec un *if-then-else*. En effet, e_1 ; e_2 est équivalent à *if* e_1 *then* e_2 *else* e_2. En outre, les autres fonctions non-strictes, comme le *ou* logique paresseux, peuvent également être encodées par un *if-then-else* : $e_1 || e_2$ est équivalent à *if* e_1 *then true else* e_2. Enfin, les traits impératifs comme la sortie écran (*print*) ou la sortie fichier peuvent être vues comme des écritures en mémoire.

2.2. Évaluation

Une première approche possible pour construire la fonction d'évaluation d'un interprète est de se fonder sur les équations de la sémantique dénotationnelle du langage. La figure 1 donne une implémentation possible qui utilise un environnement pour gérer l'instanciation des variables.

Détaillons le fonctionnement de cette évaluation :

```
let store = ref []

let rec evalD env = function
 | Int i -> ValInt i
 | Var id -> assoc id env
 | Abs (id,e) -> ValAbs (id, e, env)
 | App (e1,e2) ->
       (match (evalD env e1,evalD env e2) with
          | (ValFun f,v) -> f v
          | (ValAbs (id,e,env'), v) -> evalD ((id,v) : :env') e )
 | ExtFun f -> ValFun f
 | If (e1,e2,e3) ->
    (match evalD env e1 with
          | ValInt i when i>0 -> evalD env e2
          | _ -> evalD env e3)
 | Get m -> assoc m !store
 | Set (m,e) ->
    let v= evalD env e
    in begin store := (m,v) : :!store; v end
```

FIG. 1 – Évaluateur à la façon classique (fonction `evalD`).

Applications et variables. L'évaluation de l'application `(fun (x) =` e`) (`v`)` dans un environnement *env* consiste à évaluer e sachant que x est liée à v (v est une valeur). Ici la liaison de x à v se fait par un environnement, implémenté par une liste d'associations. L'instanciation des variables associée à l'application de fonctions est donc gérée explicitement.

On remarque une différence de traitement entre les fonctions du langage écrites par l'utilisateur et les fonctions primitives : l'application des fonctions primitives est gérée par le langage hôte alors que l'application des abstractions utilisateur nécessite la manipulation explicite de l'environnement et des variables.

Abstractions. Une abstraction pouvant être utilisée en dehors du contexte où elle est définie, on doit mémoriser la liaison des variables apparaissant dans son corps au moment de sa définition. C'est pour cela que l'environnement est stocké avec l'abstraction dans `ValAbs` (il s'agit d'une *fermeture*).

Le programme suivant justifie le besoin de mémoriser l'environnement :

```
fun f (x) = (g := (fun (y) = x)) ; ;
f(1) ; ;
g(0) ; ;
```

Ici on veut que la fonction dans la mémoire g associe bien 1 à tout argument.

Lecture/écriture mémoire. La mémoire, elle aussi, est gérée par un environnement mais alors que l'environnement pour les variables est relatif à chaque expression évaluée, la mémoire est la même pour tout le programme (elle est d'ailleurs implémentée à travers une variable globale).

Fonctions prédéfinies. Comme les constantes entières, les fonctions prédéfinies ne sont pas modifiées par l'évaluation.

Conditionnelle. L'évaluation d'une conditionnelle n'évalue que la branche *then* ou *else* selon le résultat de l'évaluation de la condition.

3. Syntaxe abstraite d'ordre supérieur

Comme nous l'avons mentionné plus tôt, nous souhaitons déléguer totalement au langage d'implémentation la tâche de réduction des applications. Ce choix est justifié en raison de la simplicité de l'implémentation correspondante (le noyau de l'évaluateur sera très concis), de son efficacité potentielle (OCaml est connu pour gérer efficacement les applications de fonctions) et de sa grande abstraction (une unique notion de fonction est nécessaire). Ceci est mis en œuvre avec une syntaxe abstraite d'ordre supérieur.

La syntaxe abstraite d'ordre supérieur (HOAS d'après l'appellation en anglais) consiste à représenter des abstractions/liaisons de variables (λ, $\forall$, $\exists$ ou autres) dans un arbre de syntaxe par une fonction du langage hôte [PE88]. On peut transformer la syntaxe abstraite `term` donnée plus tôt en syntaxe d'ordre supérieur en supprimant les abstractions et les variables (et les mécanismes d'applications de fonctions reposeront sur ceux du langage hôte) :

```
type HOterm =
 | HOInt of int
 | HOApp of HOterm * HOterm
 | HOFun of value -> value
```

Un seul constructeur est nécessaire pour désigner à la fois les fonctions prédéfinies et les fonctions définies par l'utilisateur car elles ont toutes deux représentées par une fonction du type `value -> value` dans le langage hôte. Par exemple la fonction `fun (x) = x+1` peut être représentée par :

```
HOFun (fun x -> HOApp (HOApp (HOFun plus, x), HOInt 1))
```

Ceci est vrai également pour le type `value` : la construction `ValAbs` dénotant les fermetures n'est plus nécessaire. Nous n'incluons pas de construction pour la conditionnelle *if* dans cette syntaxe car elle peut être vue comme une fonction particulière. En effet le *if* peut être vu comme une fonction à trois arguments qui n'évalue pas tous ses arguments (une fonction non-stricte). Dans la section 5 nous verrons que l'on peut l'encoder par une fonction ordinaire. Les traits impératifs peuvent également être encodés dans cette syntaxe (section 6). Le choix de ne pas intégrer la conditionnelle ou les traits impératifs dans le type `HOterm` se justifie par l'objectif de ne pas complexifier le cœur de l'interprète et de plutôt recourir à des transformations syntaxiques en amont afin de se conformer à ce type.

L'évaluation des termes en syntaxe d'ordre supérieur se fait de la manière suivante :

```
let rec evalHO = function
 | HOInt i -> ValInt i
 | HOFun f -> ValFun f
 | HOApp (t1,t2) ->
   (match (evalHO t1, evalHO t2) with (HOFun f, v) -> f v )
```

On voit bien ici que c'est le langage hôte qui gère l'application `(f v)`. L'utilisation d'une syntaxe d'ordre supérieur a les avantages suivants :

- Les fonctions prédéfinies et les fonctions définies par l'utilisateur peuvent être traitées uniformément.
- Les variables ne sont plus gérées par l'interprète mais par le langage hôte. Ceci signifie notamment que l'on n'a pas besoin de gérer :
  - la portée des variables dans toutes les manipulations de l'arbre de syntaxe (les optimisations par exemple) ;
  - l'instanciation des variables lors de l'application ;
  - la propagation d'environnements (ou de substitutions).

Ces avantages permettent d'avoir un interprète plus concis et donc :

- plus simple à implémenter, comportant moins d'erreurs, plus maintenable ;
- plus simple à décrire, sur lequel on peut mieux raisonner ;

- plus propice à l'expérimentation.

La difficulté liée à cette approche est la production des valeurs fonctionnelles par l'analyse syntaxique ou comme résultat d'un calcul.

Problème P1 : production des fonctions du langage hôte. Dans Caml comme dans le λ-calcul, le seul moyen de renvoyer une fonction est de renvoyer une fonction prédéfinie ou d'en créer une par application d'autres fonctions. À moins d'utiliser des outils spécifiques pour la programmation multi-niveau, un calcul ne peut pas créer une fonction en utilisant l'abstraction comme on le fait en *écrivant* un programme.

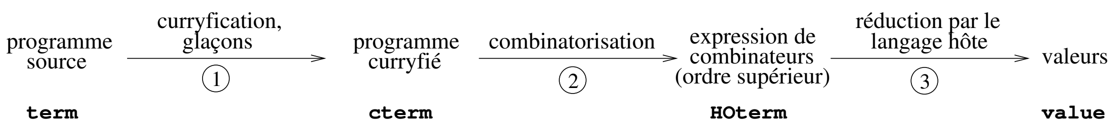
Problème P2 : intégration des fonctions non-strictes. Les fonctions non-strictes comme la conditionnelle *if-then-else*, le *ou* et le *et* paresseux ne peuvent être directement traduits en fonctions Caml. En effet, si on traduit le *if-then-else* par une fonction à trois arguments, les trois arguments seront évalués puisque Caml est un langage à appel par valeur.

Problème P3 : intégration des traits impératifs. Comme les fonctions non-strictes, les traits impératifs ne s'expriment pas tous comme des fonctions. C'est le cas du séquencement qui ne peut être vu comme une fonction à deux arguments (à moins d'avoir une garantie sur l'ordre d'évaluation des arguments). Par ailleurs même si un trait impératif peut être vu comme une fonction, comme c'est le cas du *print*, il faut veiller à ce que l'effet de bord ait lieu au « bon moment » . Par exemple, l'évaluation de l'expression

```
(fun (x) = print "0") (print "1")
```

doit imprimer 1 avant 0.

Notre réponse au problème P1 est la combinatorisation (section 4). Celle aux problèmes P2 et P3 est l'utilisation de glaçons (sections 5 et 6). Grâce à ces solutions techniques, l'évaluation des termes du langage considéré suit le schéma suivant :



① Transformation des fonctions non-strictes et des traits impératifs en fonctions en utilisant des glaçons. Nous appelons cette étape la *curryfication*, par extension du sens habituel.
② Transformation des abstractions en fonctions du langage hôte. La technique utilisée est la SK-traduction et cette étape est appelée la *combinatorisation*.
③ Évaluation en utilisant les mécanismes d'application du langage hôte (définie par `evalHO`).

4. Combinatorisation

L'utilisation de combinateurs est une manière de résoudre le problème P1. En effet, on peut transformer tout programme du λ-calcul pur en un programme équivalent[3] composé uniquement d'applications entre trois fonctions élémentaires (ou combinateurs) notées S, K et I [CF58]. Cette transformation est appelée SK-traduction ou SK-compilation. La SK-traduction est donc un moyen de créer de nouvelles fonctions par applications de fonctions.

Dans cette section nous considérons une version simplifiée du langage, sans les fonctions non-strictes et sans les traits impératifs. Les sections 5 et 6 s'attacheront à les intégrer dans le schéma d'évaluation que nous proposons ici.

[3]La notion d'équivalence n'est pas formalisée ici car ce n'est pas l'objet de cet article. Un point de vue formel des notions que nous exposons est disponible dans [Coh04].

SK-traduction classique

L'objet de la SK-traduction est de décomposer toute fonction en une combinaison de fonctions élémentaires [CF58]. Considérons la fonction $\lambda x.(e_1\ e_2)$. La variable x peut apparaître dans e_1 et dans e_2. Lorsqu'on abstrait x dans $(e_1\ e_2)$ on abstrait donc x dans e_1 et dans e_2. L'égalité suivante illustre bien ceci :

$$((\lambda x.e_1\ e_2)\ a) = ((\lambda x.e_1\ a)\ (\lambda x.e_2\ a))$$

C'est sur cette égalité qu'est fondée ce que l'on appelle la transformation S : pour abstraire x dans une application $(e_1\ e_2)$ il est suffisant de l'abstraire dans e_1 et dans e_2 et d'utiliser une fonction élémentaire chargée de distribuer l'argument reçu par cette fonction à chacune des nouvelles abstractions puis d'effectuer l'application. La fonction élémentaire distribuant l'argument reçu est le combinateur S. La règle de transformation S ainsi que la règle de réduction associée sont données dans la figure 2.

	transformation	réduction	implémentation
S	$\lambda x.(e_1\ e_2) \Rightarrow S\ (\lambda x.e_1)\ (\lambda x.e_2)$	$S\ f\ g\ e \rightarrow (f\ e)\ (g\ e)$	$\lambda x.\lambda y.\lambda z.((x\ z)\ (y\ z))$
K	$\lambda x.c \Rightarrow K\ c$	$K\ c\ e \rightarrow c$	$\lambda x.\lambda y.x$
I	$\lambda x.x \Rightarrow I$	$I\ e \rightarrow e$	$\lambda x.x$

Dans la transformation K, c désigne une constante (un entier, une fonction externe ou un combinateur) ou une variable autre que x. Le symbole $\Rightarrow$ dénote une étape de combinatorisation. Le symbole $\rightarrow$ dénote une étape de réduction.

FIG. 2 – SK-traduction pure.

Le combinateur S est en fait la fonction $\lambda f.\lambda g.\lambda x.((f\ x)\ (g\ x))$. Le résultat de la transformation S est bien composé de plusieurs fonctions simples : $\lambda x.e_1$ ainsi que $\lambda x.e_2$ sont bien plus simples que $\lambda x.(e_1\ e_2)$ et S est une constante du langage hôte. On dira que la transformation S fait *descendre les lambdas* car le lambda (l'abstraction) en tête du terme s'est vu déplacé dans les sous-termes de celui-ci. En itérant la transformation S pour faire descendre tous les lambdas, on arrive à des termes où les abstractions restantes sont soit de la forme $\lambda x.x$, soit $\lambda x.c$ où c est soit une constante, soit une variable autre que x. La première forme correspond au combinateur I. La seconde peut être décomposée en $((\lambda z.\lambda x.z)\ c)$ où $\lambda z.\lambda x.z$ correspond au combinateur K. La forme $\lambda x.c$ se transforma donc en $(K\ c)$. La forme $(K\ c)$ signifie que l'argument attendu par ce terme est inutile et sera donc « délaissé » , mais il est toutefois attendu. Si le terme à combinatoriser est clos, les abstractions mises sous la forme $(K\ x)$ auront disparu à la fin du processus car la variable x est liée par un lambda qui va « descendre » jusqu'à elle.

L'algorithme de SK-traduction consiste à appliquer les règles de transformation de la figure 2 tant qu'il reste des abstractions dans le terme. Les règles sont traditionnellement appliquées en commençant par les abstractions les plus internes du terme à transformer (stratégie *inner-most*). À la fin, il ne reste donc plus que des applications entre constantes (les combinateurs S, K et I, comme les autres fonctions externes, étant vus comme des constantes). Cet algorithme constitue l'étape ② du schéma d'évaluation mentionné en section précédente.

Implémentation des combinateurs

Les combinateurs S, K et I sont implémentés par des fonctions du langage hôte du type `value -> value` et peuvent donc être utilisés comme fonctions prédéfinies dans le type `HOterm` introduit en section précédente :

```
let app f x = match f with ValFun f' -> f' x

let comb_I = fun x -> x
let comb_K = fun c -> ValFun (fun x -> c)
let comb_S = fun f -> ValFun (fun g -> ValFun (fun x ->
       app (app f x) (app g x) ))
```

Exemple : la fonction définie par `fun (x) = 1` se SK-traduit en $(K\ 1)$ et est représentée par l'expression de combinateurs `HOApp(HOFun comb_K, HOInt 1)`.

L'évaluation des expressions de combinateurs (type `HOterm`) se fait par la fonction `evalHO` définie en section précédente (étape ③ du schéma d'évaluation).

5. Traitement des fonctions non-strictes

Considérons à présent la conditionnelle *if* dans le langage que nous voulons interpréter (soit le type `term` sans les constructions `Get` et `Set`).

Curryfication du *if*

Comme nous l'avons déjà mentionné, nous souhaitons transformer les programmes afin qu'ils soient puissent être représentés par une expression du type `HOterm` et évalués par la fonction `evalHO` plutôt que d'ajouter des cas dans la définition de ce type et de cette fonction. Par conséquent, nous allons transformer la construction `If (e1,e2,e3)` en une application (étape ① du schéma d'évaluation). Après cette transformation, nous dirons que le programme est en forme *curryfiée.* Un programme curryfié est représenté dans le type `cterm` suivant :

```
type cterm =
  | CInt of int
  | CVar of ident
  | CAbs of ident * cterm
  | CApp of cterm * cterm
  | CExtFun of value -> value
```

Dans cette syntaxe curryfiée, la conditionnelle *if-then-else* est vue comme une fonction *if* qui s'applique à trois arguments (ses trois branches). Toutefois on ne veut pas que la branche *then* et la branche *else* soient évaluées toutes les deux. Or, le langage cible est par appel par valeurs, *i.e.* les arguments passés à une fonction sont réduits avant de gérer l'application. Ce qu'il faudrait ici c'est avoir une stratégie d'appel par nom. La technique habituelle pour simuler un appel par nom lorsqu'on dispose d'un appel par valeur est l'utilisation de *glaçons.*

Notion de glaçon

Un *glaçon* est une construction du langage qui permet de geler l'évaluation d'un terme jusqu'au moment où celle-ci devient nécessaire [Plo75, HD97]. Le terme doit alors être *dégelé.* De manière usuelle, nous utiliserons une lambda abstraction comme glaçon. En effet nous considérerons qu'une abstraction est en forme réduite : lors de l'évaluation d'une abstraction, on n'évalue pas le corps de celle-ci tant qu'elle n'est pas appliquée. C'est donc l'application du terme gelé qui va provoquer le dégel. On dit que le corps d'une abstraction est *protégé par un lambda* pour signifier que ce corps ne sera pas évalué avant l'application de cette abstraction.

Considérons l'exemple suivant comme illustration du fonctionnement des glaçons. Soit le terme $(\Omega\ \Omega)$ où Ω est $\lambda x.(x\ x)$ et dont l'évaluation ne termine pas. Si ce terme est passé à une fonction, il sera évalué. Son évaluation ne terminant pas, la réduction de l'application ne termine pas. On peut geler ce terme avec une abstraction : $\lambda x.(\Omega\ \Omega)$. Si cette abstraction est passée à une fonction, son corps ne sera évalué que lorsque nécessaire : si la fonction recevant ce terme a besoin de la valeur de cet argument elle devra le dégeler en l'appliquant à un argument choisi arbitrairement. Par exemple en l'appliquant à 0 : $(\lambda x.(\Omega\ \Omega)\ 0)$. Cette application se réduit en l'application $(\Omega\ \Omega)$ qui sera réduite sans fin. En revanche, si la fonction n'utilise pas son argument, elle ne le dégèlera pas et elle pourra ainsi renvoyer un résultat.

Insertion de glaçons

Nous allons donc geler les branches *then* et *else* de la conditionnelle afin que ne soit évaluée que la branche nécessaire. La constante *if*, après avoir évalué son premier argument dégèlera son deuxième *ou bien* son troisième argument selon la valeur du premier. Notons que cette méthode est très classique puisqu'elle apparaît déjà dans [Lan65b].

La fonction `curryfy` (figure 3) effectue la transformation du programme initial (type `term`) vers sa forme curryfiée et comportant les glaçons (type `cterm`) et constitue l'étape ① du schéma d'évaluation que nous proposons.

```
let rec curryfy = function
 | Int i -> CInt i
 | Var id -> CVar id
 | Abs (id,e) -> CAbs (id, curryfy e)
 | App (e1,e2) -> CApp (curryfy e1, curryfy e2)
 | ExtFun f -> CExtFun f
 | If (e1,e2,e3) ->
    let gel_e2 = Abs(dummy_id,e2)
    and gel_e3 = Abs(dummy_id,e3)
    in CApp ( CApp ( CApp (CExtFun ite, curryfy e1),curryfy gel_e2),
              curryfy gel_e3)

and dummy_id = " "
```

FIG. 3 – Fonction de curryfication.

La fonction `ite`, du type `value->value`, aura été définie auparavant (voir ci-dessous). Elle reçoit ses trois arguments *déjà évalués* (appel par valeur). Les deux derniers étant chacun protégés par un lambda, l'évaluation ne s'est pas propagée sous ceux-ci. Si le premier argument est un entier strictement positif, la fonction constante `ite` va *dégeler* son deuxième argument en l'appliquant à un argument quelconque, ici 0. Cette application va mener à l'évaluation de l'expression initialement dans la branche *then*. Dans le cas contraire, le troisième argument sera dégelé, menant à l'évaluation de la branche *else*.

```
let ite = fun e1 -> ValFun (fun e2 -> ValFun (fun e3 ->
    match e1 with
     | ValInt i when i>0 -> app e2 dummy_val
     | _ -> app e3 dummy_val ))

and dummy_val = ValInt 0
and app = function (ValFun f) -> fun v -> f v
```

Exemple de curryfication

L'expression `fun (x) = if x then x+1 else x+2 fi` sera curryfiée en :

```
fun (x) = (((ite x) (fun _ = x+1)) (fun _ = x+2))
```

Plus précisément, ceci correspond à la transformation de

```
Abs ("x", If(Var "x",
             App (App(ExtFun plus, Var "x"), Int 1) ,
             App (App(ExtFun plus, Var "x"), Int 2) ))
```

en :

```
CAbs ("x", CApp(CApp(CApp(CExtFun ite, CVar "x"),
                        CAbs (" ", CApp (CApp(CExtFun plus, CVar "x"), CInt 1) )),
                  CAbs (" ",CApp (CApp(CExtFun plus, CVar "x"), CInt 2) )))
```

SK-traduction

La SK-traduction pure (sans optimisations) garantit qu'aucune application dans une abstraction ne sera réduite tant que cette abstraction n'est pas elle même appliquée. Pour nous en convaincre, prenons l'expression $\lambda x.(c_1\ c_2)$ comme exemple, où c_1 et c_2 sont des constantes. Elle se transforme en $S\ (K\ c_1)\ (K\ c_2)$. Dans cette expression, il n'y a pas d'application de c_1 à c_2. D'après la règle de réduction de S, c_1 ne sera appliquée à c_2 que lorsque S aura reçu un troisième argument : l'argument de la fonction définie initialement.

Plus généralement, quelles que soient la forme de e_1 et la forme de e_2 dans l'expression $\lambda x.(e_1\ e_2)$, la transformation S gèle :

– l'application de e_1 à e_2 puisque e_1 n'est pas appliqué à e_2 dans $S\ \lambda x.e_1\ \lambda x.e_2$ et que l'application ne sera faite que lorsque S recevra un troisième argument ;
– les applications pouvant apparaître dans e_1 et e_2 en distribuant l'abstraction sur chacune d'elles.

Durant le processus de traduction, la transformation S *suspend* ainsi toute application se trouvant sous un lambda. Donc la SK-traduction pure garantit que rien n'est réduit sous un lambda[4]. Ceci signifie que les applications sous un glaçon ne sont réduites que lorsque le glaçon est dégelé, *i.e.* lorsque l'application gelée est appliquée à un argument factice pour la dégeler.

Remarque : le langage hôte n'évalue pas le corps des abstractions avant qu'elles ne soient appliquées. Toutefois, ceci n'est pas exploitable ici car il ne reste pas d'abstractions dans le code produit par SK-traduction. C'est pourquoi nous avons dû analyser ici le comportement après SK-traduction des applications protégées par une abstraction.

Dans cette section, nous avons vu que les fonctions non-strictes pouvaient s'intégrer au schéma d'évaluation que nous proposons sans complexifier la syntaxe d'ordre supérieur ou son évaluation, grâce à l'utilisation de glaçons. Il n'a pas non plus été nécessaire de modifier l'étape de SK-traduction pour les intégrer. À présent nous allons voir que l'on peut intégrer de manière similaire les traits impératifs à notre schéma.

6. Traitement des traits impératifs

La technique des glaçons que nous avons utilisée pour geler les branches d'un *if* peut être utilisée pour intégrer des traits impératifs au langage. Nous intégrons à présent au langage considéré l'écriture

[4] En réalité, dans l'implémentation, des applications partielles sont réduites, comme $(K\ c)$, mais aucune application issue de l'expression originale ne l'est.

et la lecture en mémoire (soit le type `term` tel que présenté en section 2.1). Comme nous l'avons vu, ces deux traits sont suffisant puisque les autres traits impératifs peuvent être simulés dans ce langage.

Exemple : l'expression `if m1 then m2 := m1 else x fi` s'encode par `If( Get "m1", Set("m2", Get "m1"), Var "x")`.

Comme nous l'avons fait pour le *if*, nous allons nous ramener à la syntaxe curryfiée afin d'entrer dans le cadre de la SK-traduction. Regardons comment les traits impératifs peuvent être ramenés à cette syntaxe simple (type `cterm`).

Écriture en mémoire

L'écriture mémoire `Set(m,e)` peut être vue comme une application d'une fonction constante set_m à l'expression e. La curryfication consistera donc à transformer `Set(m,e)` en `CApp(`set_m`,`e'`)` où e' est la forme curryfiée de e.

Ici il n'est pas nécessaire d'utiliser un glaçon car l'*effet* aura lieu nécessairement au bon moment. Trois cas peuvent se présenter :

- Si l'affectation a lieu dans le corps d'une abstraction, l'application de set_m à son argument est naturellement gelée jusqu'à l'application de celle-ci, ce qui convient.
- Si l'affectation est dans une branche *then* ou *else*, un glaçon sera inséré par la curryfication du *if* et l'application de set_m à son argument sera ainsi gelée, jusqu'au bon moment.
- Si l'affectation n'est ni dans une abstraction ni dans un *if*, elle sera évaluée comme toute application qui n'est pas protégée. Ceci convient. En effet, considérons l'expression `m := f(1)`, curryfié en `CApp(`set_m`, CApp(CExtFun `f`, CInt 1))`. L'application de f à 1 sera effectuée avant de passer le résultat à set_m (appel par valeur). Ceci correspond bien au résultat attendu.

Donc la transformation de l'affectation en une application suffit à garantir que celle-ci aura lieu au bon moment. La fonction set_m peut être implémentée comme suit :

```
let set m = fun v -> begin store := (m,v) : :!store ; v end
```

(notons que `set "m1"` a bien le type `value->value`) et la curryfication est réalisée par :

```
let rec curryfy = function
 | ...
 | Set (m,e) -> CApp(CExtFun (set m), curryfy e)
```

Lecture en mémoire

L'affectation correspond canoniquement à une application, ce qui n'est pas le cas de l'accès mémoire. Si on considère `Get(m)` comme une constante (notée get_m), alors l'expression `fun (x) = m` sera compilée en $(K\ get_m)$. Mais l'évaluation de cette forme mène à l'évaluation de get_m. On a donc évalué get_m alors qu'elle était initialement protégée par une abstraction. On l'a évaluée *au mauvais moment* car l'accès mémoire aurait dû se faire à l'application de la fonction et non à sa définition. Ceci montre que l'on ne peut considérer comme *constantes* que des valeurs dont l'évaluation ne varie pas en fonction du moment où elles sont évaluées, d'où l'appellation. Le moyen de déclencher l'accès à la mémoire au bon moment est de considérer get_m comme une fonction qui attend d'être appliquée pour accéder à la mémoire. On va donc transformer `Get(m)` en une application afin de garantir que l'accès mémoire soit fait au bon moment, comme pour l'affectation. La forme `Get(m)` se curryfie donc en `CApp(`get_m`, `d`)` où get_m représente la fonction constante qui renvoie la valeur de la mémoire m lorsqu'elle est appliquée à n'importe quelle valeur, et où d représente un argument quelconque.

Exemple : l'expression `fun (x) = m1` représentée par `Abs("x", Get "m1")` est curryfiée en `CAbs("x", CApp(`get_{m1}`, CInt 0))` et combinatorisée en $S\ (K\ get_{m1})\ (K\ 0)$.

La fonction get_m peut être implémentée comme suit :

```
let get m = fun _ -> assoc m !store
```

(et `get "m1"` a bien le type `value->value`). Le *get* étant appliqué à une valeur factice, l'argument reçu peut être abandonné, d'où le « `fun _ -> ...` » . La curryfication est réalisée par :

```
let dummy_term = CInt 0

let rec curryfy = function
 | ...
 | Get m -> CApp(CExtFun (get m), dummy_term)
```

Différentes façons de suspendre l'évaluation d'un terme. Nous avons rencontré deux façons différentes de parvenir à un *gel*. Le gel repose sur le fait qu'une application n'est pas réduite sous une abstraction. Pour qu'il y ait *gel* il faut donc la réunion d'une abstraction (gelante) et d'une application (gelée).

Le *if* doit geler les applications dans ses branches, pour cela elles sont placées dans le corps d'une abstraction. L'accès mémoire, quand à lui, doit être gelé s'il est dans une abstraction (naturelle ou formée par un glaçon). L'accès mémoire doit donc prendre la forme d'une application.

Exemple

Les références mémoires peuvent servir à faire de la programmation récursive. Par exemple voici la fonction de Fibonacci :

```
fib := ( fun (i) = (if i-1 then 1 else fib(i-1) + fib(i-2) fi ))
```

Nous nous servirons de cette fonction comme témoin pour évaluer l'efficacité de notre combinatorisation dans la suite de cet article. Sa forme combinatorisée est :

```
fib :=
((S ((S ((S (K IF)) ((S ((S (K -)) I)) (K 1)))) ((S ((S (K S))
((S ((S(K S)) ((S (K K)) (K +)))) ((S ((S (K S)) ((S ((S (K S))
((S (K K)) (K!fib)))) ((S (K K)) (K 0))))) ((S ((S (K S)) ((S ((S (K S))
((S (K K)) (K -)))) ((S (K K)) I)))) ((S (K K)) (K 1)))))))
((S ((S (K S)) ((S ((S (K S)) ((S (K K)) (K!fib)))) ((S (K K)) (K 0)))))
((S ((S (K S)) ((S ((S (K S)) ((S (K K)) (K -)))) ((S (K K)) I))))
((S (K K)) (K 2))))))) ((S (K K)) (K 1)))
```

7. Performances

Dans l'exemple ci-dessus, la taille du terme en forme combinatorisée est de 108 applications. Il est bien connu que dans sa forme pure (sans optimisations), la SK-traduction fait exploser la taille des termes. Ceci engendre à la fois une grande consommation de mémoire et une interprétation inefficace, d'où la nécessité d'avoir recours à des optimisations.

L'implémentation sur laquelle nous avons effectué nos mesures est un interprète MGS complet et non une maquette dédiée uniquement à l'étude de la SK-traduction. L'évaluation de $fib(31)$ prend 9.3 secondes sur notre machine (Pentium IV 2.4GHz, OCaml 3.06, Linux Debian Woody, interprète compilé par `ocamlopt`), en suivant fidèlement le schéma d'évaluation proposé ici : curryfication, SK-traduction puis évaluation par `evalHO`. À titre de comparaison, l'interprète OCaml (`ocaml`) évalue $fib(31)$ en 0.23 secondes (même programme, avec des références). Il y a donc environ un facteur 40 entre les deux implémentations.

Dans l'interprète MGS que nous distribuons, nous avons implémenté trois familles d'optimisations qui nous ont permis d'atteindre notre objectif de performances avec le schéma d'évaluation proposé (colonne *SK-trad. opt.* du tableau ci-dessous, les durées sont en secondes). En effet, l'interprète ainsi optimisé est environ 7 fois plus rapide, sur l'exemple de la fonction de Fibonacci, que l'implémentation classique présentée en section 2.2 et est seulement 3 fois plus lent que l'interprète OCaml. Le tableau ci-dessous donne également des mesures sur un programme MGS complet, *Turing-torus* (voir [Coh04]), n'ayant pas d'équivalent direct dans d'autres langages. La faible différence de performances entre l'évaluation classique et l'évaluation que nous proposons vient du fait que ce programme fait une utilisation intensive du filtrage.

	`evalD`	SK-trad. pure	SK-trad. opt.	OCaml	Mathematica	Python
fib(31)	4.7	9.3	0.65	0.23	27	3.5
ackerman(3,8)	6.5	>120	1.5	0.27	22	3.2
turing-torus(100)	6.0	>120	5.1	/	/	/

Dans la suite de cette section, nous commençons par présenter la première sorte d'optimisation mise en œuvre : les optimisations classiques de la littérature sur la logique combinatoire. Nous verrons que les optimisations de la littérature ne sont pas correctes dans notre cadre et doivent être adaptées. Ensuite nous indiquerons brièvement la nature des autres optimisations implémentées.

7.1. Optimisations classiques

Le problème adressé par les optimisations classiques est la distribution systématique des lambdas : la règle de combinatorisation de $\lambda x.(e_1\ e_2)$ distribue l'abstraction de x sur e_1 et e_2 alors que x n'apparaît pas forcément dans e_1 et e_2. Nous nous penchons ici sur trois optimisations : l'η-réduction, l'utilisation des combinateurs B et C, et enfin la simplification d'un S en K.

L'η-réduction

L'η-réduction (*eta*-réduction) intervient sur les termes avant la SK-traduction. Elle consiste à remplacer $\lambda x.(e\,x)$ par e lorsque x n'apparaît pas libre dans e. Il est simple de se rendre compte que ceci n'est pas correct dans notre cadre en considérant l'exemple suivant : $\lambda x.(+\ (get_a\ 0)\ x)$. Ici, l'accès mémoire à a doit se faire *au moment où la fonction est utilisée* et non pas au moment où elle est définie. L'η-réduction viole ce principe puisque la fonction résultante ne serait plus définie par une abstraction mais par une application : $(+\ (get_a\ 0))$. Nous parlerons d'*évaporation des lambdas* lorsqu'un lambda disparaît, ne retardant plus l'évaluation d'un terme.

Le problème qu'il nous faut éviter est la réduction d'une application située dans une abstraction due à l'évaporation d'un lambda. Il existe des cas où aucune application n'est dégelée en enlevant un lambda. Par exemple on peut transformer $\lambda x.(+\ x)$ en $+$ ou bien $\lambda x.(y\ x)$ en y sans modifier la sémantique du programme. En fait, si c est une expression qui ne contient pas d'applications, on peut η-réduire $\lambda x.(c\ x)$.

Nous retiendrons ce critère puisqu'il est suffisant pour pouvoir η-réduire (lorsque x n'apparaît pas libre dans c). Toutefois, dans l'implémentation actuelle de l'interprète MGS, l'analyse est plus fine. Nous η-réduisons par exemple $\lambda x.((+\ 1)\ x)$ bien que l'application $(+\ 1)$ soit dégelée car :

- ce dégel ne viole pas la sémantique des traits impératifs ou des fonctions non-strictes,
- le calcul dégelé prend un temps réduit et constant.

Une heuristique détermine quelles applications peuvent être dégelées ou non[5].

[5]Dans l'interprète actuel, un terme est qualifié de *pur* et peut être dégelé arbitrairement s'il ne contient ni traits impératifs, ni fonctions utilisateur. Sinon il sera qualifié d'*impur* et on ne pourra le dégeler par évaporation de lambdas.

Remplacer S par B ou C

Les optimisations utilisant les combinateurs B et C de [CF58] sont fondées sur la remarque que lorsqu'on transforme $\lambda x.(e_1\, e_2)$ en $S\,(\lambda x.e_1)\,(\lambda x.e_2)$, il peut être inutile de distribuer l'abstraction sur e_1 ou sur e_2 si x n'y apparaît pas libre. Par exemple, si x n'apparaît pas libre dans e_1, on transformera plutôt $\lambda x.(e_1\, e_2)$ en $B\, e_1\,(\lambda x.e_2)$ où le combinateur B agit de manière similaire au combinateur S : alors que S distribue son troisième argument aux deux premiers et effectue leur application, le combinateur B ne distribue son argument que sur son deuxième argument (le terme sur lequel l'abstraction a été distribuée) et effectue l'application. Le tableau ci-dessous rend ceci explicite :

	réduction	transformation
S	$S\ f\ g\ x \to (f\ x)\ (g\ x)$	$\lambda x.(e_1\ e_2) \Rightarrow S\ (\lambda x.e_1)\ (\lambda x.e_2)$
B	$B\ e\ f\ x \to e\ (f\ x)$	$\lambda x.(e_1\ e_2) \Rightarrow B\ e_1\ (\lambda x.e_2)$ si x n'apparaît pas libre dans e_1
C	$C\ f\ e\ x \to (f\ x)\ e$	$\lambda x.(e_1\ e_2) \Rightarrow C\ (\lambda x.e_1)\ e_2$ si x n'apparaît pas libre dans e_2

Le combinateur C est le symétrique de B : son utilisation est possible lorsque x n'apparaît pas libre dans e_2.

L'utilisation des combinateurs B et C présentée ci-dessus provoque des évaporations de lambda. Par exemple le terme $\lambda x.(M\ x)$ se transforme en $S\ (\lambda x.M)\ I$ où M est toujours protégé, mais se traduit en $B\ M\ I$ si on utilise B, lorsque x n'apparaît pas libre dans M. Dans ce cas le lambda s'est évaporé et les applications dans M sont dégelées. Les règles d'introduction de B et C ci-dessus ne sont donc pas correctes dans notre cadre.

Nous devons donc introduire une condition nouvelle à l'application des optimisations *à la* B et C : on peut *ne pas* distribuer un lambda sur une expression uniquement si elle ne contient pas d'application.

Ceci se concrétise par l'ajout de la condition « e_1 ne contient pas d'applications » pour l'introduction du combinateur B dans le tableau ci-dessus et de la condition « e_2 ne contient pas d'applications » pour l'introduction de C. Notons que ni e_1 ni e_2 ne peuvent être des abstraction car on applique ces transformations sur les abstractions les plus internes des termes (les combinateurs sont vus comme des constantes).

Comme la condition sur l'η-réduction, cette condition peut être affaiblie : dans l'interprète MGS nous utilisons l'heuristique qui qualifie un terme de pur ou d'impur pour décider de distribuer le lambda ou non (voir la note de bas de page 5).

Remplacer S par K

La dernière optimisation classique que nous examinons est le remplacement d'un combinateur S par un combinateur K. Cette optimisation consiste à transformer $S\ (K\ c_1)\ (K\ c_2)$ en $K\ (c_1\ c_2)$. On peut aussi la voir sous la forme de la transformation $\lambda x.(e_1\ e_2) \Rightarrow K\ (e_1\ e_2)$ applicable lorsque x n'apparaît libre ni dans e_1 ni dans e_2.

Ici encore, l'évaporation des lambdas pose un problème dans notre contexte puisque l'application $e_1\, e_2$ est dégelée.

Une variation autour des optimisations B et C va nous permettre de ne pas distribuer le lambda inutilement sur e_1 et e_2 tout en garantissant qu'aucune application n'est réduite trop tôt. Nous introduisons un nouveau combinateur noté N défini par :

	réduction	transformation
N	$N\ e_1\ e_2\ x \to e_1\ e_2$	$\lambda x.(e_1\ e_2) \Rightarrow N\ e_1\ e_2$

Ceci est une heuristique car on peut sans doute trouver un critère plus fin et on suppose arbitrairement que les fonctions prédéfinies coûtent peu à l'exécution.

La transformation N ci-dessus ne peut être appliquée que si x n'apparaît libre ni dans e_1 ni dans e_2 et si ces deux expressions ne contiennent pas d'application. Ce nouveau combinateur peut être vu comme une généralisation de B et C au cas où x n'apparaît ni dans e_1, ni dans e_2 et peut être utilisé à la place de l'optimisation $\lambda x.(e_1\ e_2) \Rightarrow K\ (e_1\ e_2)$ car dans $N\, e_1\, e2$, soit $(N\, e_1)\, e_2$, l'application de e_1 à e_2 est bien gelée.

Exemples

Le terme $\lambda x.\lambda y.((+\ x)\ y)$ se combinatorise en

$$C\ (B\ S\ (S\ (N\ N\ +)\ I))\ I$$

qui contient 8 applications au lieu de

$$((S\ ((S\ (K\ S))\ ((S\ ((S\ (K\ S))\ ((S\ (K\ K))\ (K\ +))))\ ((S\ (K\ K))\ I))))\ (K\ I))$$

qui contient 18 applications (cet exemple ne s' η-réduit pas).

Nous donnons ci-dessous le résultat de la transformation de la fonction de Fibonacci en utilisant les quatre combinateurs proposés.

```
fib :=
((C ((S ((B IF) ((C ((B -) I)) 1))) ((S ((B S) ((B (B +))
((S ((B S) ((N (N!fib)) 0))) ((C ((B C) ((B (N -)) I))) 1)))))
((S ((B S) ((N (N!fib)) 0))) ((C ((B C) ((B (N -)) I))) 2)))))
(K 1))
```

Le terme combinatorisé est composé de 46 applications contre 108 pour la SK-compilation pure.

7.2. Autres optimisations

D'autres optimisations ont été mises en œuvre dans la combinatorisation des termes afin d'améliorer les performance de l'interprète MGS. Nous avons notamment introduits des combinateurs n-aires s'inspirant des combinateurs matriciels de Diller [Dil02]. Nos combinateurs n-aires opèrent sur des applications et des abstractions multiples alors que les combinateurs que nous avons rencontrés jusqu'à présent sont introduits en considérant une abstraction et une application à la fois. Les combinateurs n-aires que nous avons utilisés généralisent le combinateur S : alors que S est introduit pour *une* abstraction sur *une* application, nous proposons d'utiliser un combinateur S_m^n pour traduire l'abstraction de n variable sur une application à $m+1$ opérandes comme $\lambda x_1 \ldots . \lambda x_n.(f\, e_1\, e_2\, \ldots\, e_m)$.

Les combinateurs de Diller sont une généralisation de S, B, C et N aux abstractions et applications multiples. Ils sont représentés par une matrice où le booléen en position (i, j) indique si le i-ème argument reçu par la fonction doit être distribué sur le j-ème opérande de l'application ou non.

Nous nous sommes limités à une généralisation de S pour deux raisons. La première est que les matrices de Diller ne répondent pas à notre objectif de simplicité : un combinateur de Diller est représenté par $n * m$ booléens là ou deux entiers suffisent pour nos combinateurs n-aires. La seconde raison concerne l'efficacité. En effet, nous avons vu que dans notre cadre, les cas où l'on peut se permettre de ne pas distribuer un lambda sont plus rare que dans le cadre du lambda calcul pur. Ainsi le coût d'application des combinateurs de Diller, qui est plus élevé que celui des combinateurs n-aires, n'est pas justifié face au faible nombre de fois où l'on peut ne pas distribuer un lambda.

Enfin, la troisième source d'optimisations que nous avons prise en compte concerne les calculs induits par la gestion des glaçons. Nous avons utilisé des combinateurs « paresseux » n'évaluant pas systématiquement leurs arguments (on peut en rencontrer dans [GD92]). Ainsi les termes à combinatoriser ne sont pas « gonflés » par l'introduction d'abstractions artificielles permettant ainsi de réduire la taille du terme renvoyé par la combinatorisation.

Nous avons réalisé ces trois familles d'optimisations afin de montrer qu'un évaluateur construit sur le schéma original que nous avons proposé pouvant avoir des performances raisonnables. Nous les avons décrites ici afin de montrer que les optimisations de la littérature peuvent être adaptées à un contexte avec traits impératifs et fonctions non-strictes. Une description plus détaillée de ces trois familles d'optimisations et de leur implémentation peut être trouvée dans [Coh04].

Notons que hormis les combinateurs paresseux mentionnés, aucune des optimisations proposées ne nécessite de modifier la fonction `evalHO` d'évaluation des termes en syntaxe d'ordre supérieur.

8. Conclusion

Correction. L'utilisation de combinateurs avec une sémantique d'appels par valeur a été étudiée dans [GD92]. Toutefois, l'intégration des traits impératifs dans la logique combinatoire n'a pas été étudiée auparavant à notre connaissance. Nous montrons dans [Coh04] que la combinatorisation en présence de traits impératifs et de fonctions partielles est correcte.

Bilan. Il existe peu ou pas d'implémentations d'interprètes de langages de programmation basés sur une syntaxe d'ordre supérieur. Ceci est dû à la difficulté de traduire au vol un arbre de syntaxe abstraite contenant des abstractions en un arbre de syntaxe d'ordre supérieur. Dans cet article nous avons vu comment interpréter un langage fonctionnel sans gérer nous-mêmes les mécanismes de réduction d'application. Pour cela nous avons utilisé la SK-traduction. Cette approche permet d'avoir un code concis pour le cœur de l'interprète, ce qui le rend simple à implémenter et à maintenir, et ce qui permet de pouvoir raisonner dessus et d'expérimenter avec. Elle permet également de ne pas avoir à distinguer les fonctions écrites dans le langage interprété de celles écrites dans le langage d'implémentation.

Nous avons vu comment cette approche, et notamment la SK-traduction, peut-être adaptée à un langage avec fonctions non-strictes et traits impératifs grâce à l'utilisation de glaçons.

Nous avons également montré que les optimisations connues dans la logique combinatoire peuvent être utilisées, sous condition de les adapter. Les optimisations simples que nous avons mises en œuvre, aussi bien classiques que récentes, nous ont permis d'obtenir un interprète efficace pour le langage MGS. Les techniques décrites dans cet article peuvent être utilisées pour le développement rapide d'interprètes de langages fonctionnels.

Remerciements. Je remercie vivement Jean-Louis Giavitto et Olivier Michel pour m'avoir guidé dans le développement des travaux présentés ici. Je remercie également Olivier Danvy pour m'avoir donné son point de vue éclairé sur ces travaux et pour m'avoir aidé à formaliser l'approche proposée, bien que cette formalisation ne soit pas exposée dans cet article. Je remercie enfin Antoine Spicher qui participe intensivement au développement de l'interprète MGS. Celui-ci peut-être librement obtenu à partir de la page web `http://mgs.lami.univ-evry.fr` .

Références

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North Holland, 1958.

[Coh04] Julien Cohen. *Intégration des collections topologiques et des transformations dans un langage de programmation fonctionnel.* PhD thesis, Université d'Évry Val d'Essonne, décembre 2004.

[Dil02] Antoni Diller. Efficient multi-variate abstraction using an array representation for combinators. *Inf. Process. Lett.*, 84(6) :311–317, 2002.

[dR02] Daniel de Rauglaudre. *CamlP4 - Reference Manual.* INRIA, 2002.

[FL87] Marc Feeley and Guy Lapalme. Using closures for code generation. *Journal of Computer Languages*, 12(1) :47–66, 1987.

[GD92] John Gateley and Bruce F. Duba. Call-by-value combinatory logic and the lambda-value calculus. In *Mathematical Foundations of Programming Semantics, 7th International Conference, 1991*, volume 598 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 1992.

[GMC02] J.-L. Giavitto, O. Michel, and J. Cohen. Pattern-matching and rewriting rules for group indexed data structures. *ACM SIGPLAN Notices*, 37(12) :76–87, December 2002.

[HD97] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3) :303–319, May 1997.

[Hug82] R. J. M. Hughes. Super Combinators—A New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, 1982.

[Lan65a] P. J. Landin. An abstract machine for designers of computing languages. In *Proc. IFIP Congress*, pages 438–439, 1965.

[Lan65b] P. J. Landin. Correspondence between algol 60 and Church's lambda-notation : part I. *Commun. ACM*, 8(2) :89–101, 1965.

[LDGV02] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. *The Objective Caml system release 3.06, Documentation and user's manual.* Projet Cristal, INRIA, 2002.

[PE88] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208. ACM Press, 1988.

[PJ87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[PJS89] Simon L. Peyton Jones and Jon Salkild. The spineless tagless G-machine. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 184–201. ACM Press, 1989.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1 :125–159, 1975.