Décurryfication certifiée

Zaynah Dargaye

INRIA Rocquencourt,
78153 Le Chesnay CEDEX, France
`zaynah.dargaye@inria.fr`

Résumé

La décurryfication (transformation de fonctions curryfiées en fonctions n-aires) est une optimisation courante dans la compilation de langage fonctionnel. Nous présentons ici une preuve de préservation sémantique de cette optimisation menée dans l'assistant de preuve `Coq`.

1. Introduction

Il peut être nécessaire, dans le cadre de développements de logiciels critiques, de s'assurer que la sémantique du programme développé est préservée par le processus de compilation. Un compilateur ayant cette propriété est dit *certifié*. C'est dans cet esprit que le projet `Compcert` [7, 3] développe un compilateur d'un sous-ensemble du langage `C` vers l'assembleur `PowerPC`.

Nous développons ce compilateur à l'aide de l'assistant de preuve `Coq`[4, 2] : nous écrivons le compilateur dans le langage de spécification. Le compilateur est accompagné de sa preuve de préservation sémantique. On extrait de ce développement le programme `Caml` du compilateur, *via* le mécanisme d'extraction de `Coq` [9].

Le programme extrait doit être lui-même compilé afin d'être utilisable. Pour plus de confiance dans le code exécutable, on souhaite développer un compilateur certifié pour un sous-ensemble significatif de `ML` *assez complet pour exprimer l'ensemble des programmes issus de l'extraction.*

Il s'agit du fragment fonctionnel pur de `ML` avec types de bases, types fonctionnels et types concrets et qui est compilé vers le premier langage intermédiaire du compilateur `Compcert` : `Cminor`. La compilation d'un programme `mini-ML` en `Cminor` se décompose en quatre phases consécutives :
- La numérotation des constructeurs : on passe de constructeurs de types concrets nommés à des constructeurs représentés par leurs numéros d'apparition dans les définitions des types concrets.
- L'optimisation de l'appel de fonction, sur laquelle nous reviendrons.
- La globalisation des fonctions : on obtient un programme organisé en liste de fonctions et une fonction principale.
- La construction des fermetures en mémoire.

Entre chaque phase, nous avons un langage intermédiaire différent et pour chaque phase, nous avons une preuve de préservation sémantique entre le langage source et le langage cible. Ces quatre transformations sont développées en `Coq`, de même les preuves de préservation sémantique sont menées dans l'assistant de preuve `Coq`.

Nous nous focalisons ici sur une optimisation particulière de ce développement : l'optimisation de la compilation de l'appel de fonction.

Dans les langages fonctionnels, `Caml` [8], `Haskell` [13], `SML` [11] et le langage de spécification de `Coq`, il existe deux possibilités d'encodage des fonctions à plusieurs arguments : 1 – grouper les arguments de la fonction dans un n-uplet, 2 – *curryfier* la fonction. Une abstraction est dite *curryfiée* si elle s'écrit sous la forme suivante : $\lambda x_0.\ \ldots \lambda x_n.\ t$ (avec n maximal) c'est-à-dire qu'elle attend $n+1$ arguments.

Après n appels partiels, l'appel suivant correspond à l'évaluation du corps de la fonction avec tous ses arguments. Un appel partiel est une application dont le résultat est une fermeture intermédiaire : tous les arguments ne sont pas encore reçus.

Décurryfier une abstraction consiste à transformer une abstraction curryfiée en une fonction n-aire : $\lambda x_0.\ \ldots \lambda x_n.\ t$ devient $\lambda[x_0; \ldots; x_n].\ t$. Elle reçoit les $n+1$ arguments d'un coup lors de l'application. La décurryfication constitue une optimisation couramment utilisée dans la compilation de langages fonctionnels [1, 10, 6].

On tire avantage de la décurryfication lors de l'évaluation d'applications totales (applications imbriquées au nombre d'arguments attendus par une abstraction curryfiée). L'évaluation d'un terme de la forme $(\ldots(\lambda x_0.\ \ldots \lambda x_n.\ t)\ t_0)\ \ldots)\ t_n)$ se décompose en $n+1$ applications de la règle d'évaluation de l'application, tandis que l'évaluation du terme décurryfié $(\lambda[x_0; \ldots; x_n].\ t)\ [t_0; \ldots; t_n]$ n'invoque qu'une règle d'évaluation. L'avantage le plus concret de la décurryfication est que le code obtenu par compilation est plus économe en allocations.

La décurryfication peut s'effectuer ou bien *dynamiquement*, au vol pendant l'exécution, ou bien *statiquement*, par une transformation du programme pendant la compilation. La décurryfication dynamique, comme effectuée par les machines abstraites ZAM [6] et G-machine [14], reconnaît davantage d'applications curryfiées, mais ralentit l'exécution : il faut tester dynamiquement l'arité des fonctions, et les arguments doivent être passés dans une pile, l'utilisation de registres pour les passer étant impossible. Au contraire, la décurryfication statique peut échouer à reconnaître certaines applications curryfiées, mais se prête à la génération de code machine efficace, utilisant les registres du processeur pour le passage d'arguments. Pour ces raisons, le système OCaml utilise la décurryfication dynamique dans son compilateur de bytecode et la décurryfication statique dans son compilateur optimisant. Nous suivons l'approche statique de la décurryfication, qui s'intègre mieux dans une chaîne de compilation optimisante passant par le langage intermédiaire `Cminor`.

La décurryfication statique d'un programme écrit dans un style curryfié nécessite une analyse statique de ce programme. Cette analyse permet de repérer les abstractions curryfiées dont la transformation s'intéresse uniquement aux abstractions liées par un `let` et de convertir en conséquence les applications. On pourrait également décurryfier les fonctions anonymes. Cependant, leur fréquence ne justifie pas l'augmentation de la complexité de l'analyse.

Considérons le terme $\texttt{let}\ f = \lambda x.\ \lambda y.\ x+y\ \texttt{in}\ ((f\ 3)\ 4)$, sa transformation donnerait $\texttt{let}\ f = \lambda[x;y].\ x+y\ \ \texttt{in}\ f\ [3;4]$. Par contre, $\texttt{let}\ f = \lambda[x;y].\ x+y\ \texttt{in}\ f\ [3]$ n'est pas une traduction correcte : $f\ [3]$ n'a pas d'évaluation.

Pour toute fonction n-aire, il est possible de reconstruire la fonction curryfiée correspondante : $\texttt{curried}_n\ (f) = (\lambda z.\ \lambda x_0.\ \ldots \lambda x_n.\ z\ [x_0; \ldots; x_n])\ f$. Nous utiliserons ces combinateurs $\texttt{curried}_n$ pour pouvoir traduire les appels partiels. $(f\ 3)$ devient alors $((\texttt{curried}_1\ f)\ 3)$.

Dans cet article, nous nous intéressons à la formalisation de la décurryfication et à sa preuve de préservation sémantique afin de la certifier formellement. Cette optimisation s'intégrant dans le front-end de compilateur certifié pour `mini-ML`, la formalisation et la preuve sont menées en `Coq`. Une *fermeture* est la valeur associée à l'évaluation d'une abstraction. Elle est constituée du paramètre et du corps de l'abstraction et est accompagnée d'un environnement permettant, lors de l'évaluation, d'accéder aux valeurs des variables libres dans le corps. Le point clé de cette preuve réside en la caractérisation des fermetures, produites au cours des évaluations d'un terme et de sa transformation, en s'appuyant sur les informations récoltées par l'analyse. En caractérisant les fermetures d'origine et les fermetures après transformation selon l'analyse, on établit une relation entre ces fermetures. En étendant cette relation aux environnements d'exécution, nous obtenons une relation de correspondance entre environnements. Ces relations nous permettent de comparer les évaluations et donc d'établir la préservation sémantique de la transformation.

Afin d'illustrer notre étude, nous commençons par définir formellement deux langages fonctionnels minimaux. μML (prononcé *micro-ML*) : λ-calcul pur enrichi du lieur local `let`, à abstraction et application unaire ; et `nML` : λ-calcul pur enrichi du lieur local `let`, à abstraction et application $n-$aire (langage cible de la transformation). Puis, nous exposerons la traduction d'un terme μ`ML` en un terme `nML`, qui constitue une décurryfication du terme μ`ML` guidée par une analyse du terme visant à reconnaître les abstractions et applications curryfiées. Nous nous pencherons ensuite sur la caractérisation de certaines fermetures apparaissant lors des évaluations de termes μ`ML` et `nML`, en utilisant les informations fournies par l'analyse. Cette caractérisation sera la base de la relation de correspondance entre les environnements d'évaluation des deux langages, qui nous permettra alors de décrire la preuve de correction de cette transformation. Enfin, nous étendrons la décurryfication aux fonctions récursives.

2. Le langage μML

Pour plus de clarté et afin de nous focaliser sur l'étude de l'optimisation de la compilation de fermetures, nous travaillons sur un sous ensemble significatif de `mini-ML`. Ce sous ensemble forme le langage μ`ML`. Il est constitué de l'abstraction unaire, l'application unaire et de liaisons locales de variables *via* le lieur `let`.

Nous avons choisi la *formalisation de De Bruijn* [5] pour éviter tout problème lié à la génération de noms de variables lors de la transformation. Ce formalisme permet en outre de définir tout environnement relatif aux variables sous forme de listes simples qui sont plus faciles à manipuler que des tables associatives. Les variables sont représentées par des indices. La variable d'indice $n!$ est la variable liée au $(n+1)$-ième lieur (λ ou `let`) extérieur. Nous commençons par donner la syntaxe abstraite de μ`ML`, puis nous en donnerons la sémantique.

2.1. Syntaxe abstraite de μML

Nous donnons ici la syntaxe abstraite de μ`ML` sous forme de BNF.

Termes :	$t \;::= n!$	variable (indice de De Bruijn)
	$\mid i$	entier
	$\mid \lambda.\ t$	abstraction
	$\mid t_1\ t_2$	application
	$\mid \texttt{let}\ t_1\ \texttt{in}\ t_2$	définition locale

Un programme μ`ML` est un terme. Les variables sont désignées par leurs indices de De Bruijn. Une abstraction est constituée d'un λ désignant le paramètre et d'un terme qui est le corps de l'abstraction. L'application est unaire. Une définition locale, `let` t_1 `in` t_2 lie dans le terme t_2 la variable $0!$ à la valeur de t_1.

Le programme $t = \texttt{let}\ \lambda.\ \lambda.\ 1! + 0!\ \texttt{in}\ ((0!\ 3)4)$ présenté plus haut est un exemple de programme μ`ML`. Il s'agit d'une définition locale. Le sous terme gauche $(\lambda.\ \lambda.\ 1!+0!)$ lie la définition de l'abstraction curryfiée correspondant à une addition à la variable $0!$ dans le sous-terme droit. Dans le corps de l'abstraction la plus interne $(1!+0!)$, la variable $1!$ est liée au λ extérieur tandis que $0!$ est liée au λ le plus interne. Le sous-terme droit $((0!\ 3)4)$ est une application curryfiée. L'application $(0!\ 3)$ désigne l'application de la variable liée la plus proche (*i.e.* l'addition curryfiée) et lui applique comme premier argument l'entier 3. On applique alors à ce sous-terme l'entier 4.

2.2. Sémantique de μML

La sémantique de μ`ML` suit une stratégie d'évaluation faible (on ne réduit pas dans le corps des abstractions) en appel par valeur, et avec une évaluation des sous-termes de gauche à droite.

Les valeurs d'évaluation sont soit des valeurs entières, soit des fermetures formées d'un couple corps d'abstraction, environnement. Un environnement est une liste de valeurs. L'accès au $(n+1)$-ième élément d'un environnement e se note : $e(n)$. Les règles d'évaluations de μML sont similaires à celles du λ-calcul pur.

Valeurs :	$v ::= (i) \mid (t,e)$
Environnements :	$e ::= [] \mid v :: e$

$$e \vdash i \rightarrow (i)\ \text{(eval-int)} \qquad e \vdash (\lambda.\ t) \rightarrow (t,e)\ \text{(eval-lamb)} \qquad \frac{e(n) = v}{e \vdash n! \rightarrow v}\ \text{(eval-var)}$$

$$\frac{e \vdash t_1 \rightarrow v_1 \qquad v_1 :: e \vdash t_2 \rightarrow v}{e \vdash (\mathtt{let}\ t_1\ \mathtt{in}\ t_2) \rightarrow v}\ \text{(eval-let)}$$

$$\frac{e \vdash t_1 \rightarrow (t,e_1) \qquad e \vdash t_2 \rightarrow v_2 \qquad v_2 :: e_1 \vdash t \rightarrow v}{e \vdash (t_1\ t_2) \rightarrow v}\ \text{(eval-app)}$$

3. Le langage nML

nML diffère de μML par ses abstractions et ses applications qui sont n-aires.

3.1. Syntaxe de nML

Termes :	$t ::= n!$	variable (indice de De Bruijn)
	$\mid i$	entier
	$\mid \lambda^n.\ t$	abstraction d'arité $(n+1)$
	$\mid t\ [t_0; \ldots; t_n]$	application
	$\mid \mathtt{let}\ t_1\ \mathtt{in}\ t_2$	définition locale

Les λ des abstractions sont indicés par un entier naturel indiquant leur arité. Le terme $\lambda^n.\ t$ est une abstraction d'arité $n+1$ dont le corps est le terme t. L'application est formée d'un sous-terme gauche (fonction appelée) et d'une liste de termes constituant les arguments.

Le terme suivant est un exemple de programme nML : $\mathtt{let}\ \ \lambda^1.\ 1! + 0!\ \mathtt{in}\ 0![3;4]$. C'est la traduction du terme μML $t = \mathtt{let}\ \lambda.\ \lambda.\ 1! + 0!\ \mathtt{in}\ ((0!\ 3)4)$.

3.2. Sémantique de nML

La sémantique de nML est donnée dans le même style que celle de μML. Les fermetures de nML comportent une information supplémentaire : l'arité.

Valeurs :	$v ::= (i) \mid (n,t,e)$
Environnements :	$e ::= [] \mid v :: e$

$$e \vdash i \rightarrow (i)\ \text{(eval-int)} \qquad e \vdash (\lambda^n.\ t) \rightarrow (n,t,e)\ \text{(eval-lamb)} \qquad \frac{e(n) = v}{e \vdash n! \rightarrow v}\ \text{(eval-var)}$$

$$\frac{e \vdash t_1 \to v_1 \qquad v_1 :: e \vdash t_2 \to v}{e \vdash (\texttt{let}\ t_1\ \texttt{in}\ t_2) \to v} \text{ (eval-let)}$$

$$\frac{e \vdash t \to (n, t', e_1) \qquad e \vdash t_0 \to v_0 \qquad \ldots \qquad e \vdash t_n \to v_n \qquad v_n :: \ldots :: v_0 :: e_1 \vdash t' \to v}{e \vdash t[t_0; \ldots; t_n] \to v} \text{ (eval-app)}$$

Les règles d'évaluation de nML sont identiques à celles de μML, exceptées celles concernant l'abstraction et l'application.

Une abstraction $\lambda^n.\ t$ dans l'environnement e, s'évalue en une fermeture dont l'arité est $(n+1)$, le corps t et l'environnement e. La valeur sera alors (n, t, e).

Une application $t\ [t_0; \ldots; t_n]$ s'évalue en la valeur v dans l'environnement e, si t s'évalue dans e en une fermeture de la forme (n, t', e_1), que les arguments sont au nombre de $n+1$ et s'évaluent en $v_0, \ldots, v_n$ et que dans l'environnement $v_n :: \ldots :: v_0 :: e_1\ t'$ s'évalue en v.

4. Décurryfication

Après avoir introduit nos deux langages d'étude, nous pouvons nous intéresser à la transformation de termes μML en termes nML. Pour ce faire, nous commençons par expliciter les mécanismes d'analyse statique permettant de repérer dans un terme μML les abstractions et applications *"décurryfiables"*. Puis, nous pourrons exhiber l'algorithme de transformation.

4.1. Reconnaissance des abstractions et applications *"décurryfiables"*

L'analyse statique qui détermine les possibilités de décurryfication fait partie de la famille générale des analyses de flot de contrôle (*control-flow analyses* ou *closure analyses* dans la littérature en anglais) [15, 12]. De manière générale, ces analyses associent à chaque variable et chaque sous-expression d'un programme fonctionnel une sur-approximation de ses valeurs possibles. En particulier, pour une application $t\ t'$, l'analyse de flot détermine un ensemble de fonctions $\{\ldots \lambda x_i.t_i \ldots\}$ en lesquelles t peut s'évaluer. Lorsque cet ensemble ne contient qu'un élément, la fonction appelée est *statiquement connue* et plusieurs optimisations deviennent possibles : expanser la fonction (*inlining*), décurryfier l'appel si la fonction est n-aire, spécialiser la représentation de sa fermeture (*lightweight closure conversion*), *etc.*

Les analyses de flot de contrôle classiques utilisent des itérations de point fixe, les rendant coûteuses en temps de compilation et difficiles à certifier en Coq. Nous allons donc utiliser une analyse très simplifiée, qui garde seulement trace des arités des fonctions liées par `let` (on notera qu'une analyse simplifiée similaire est utilisée dans le compilateur optimisant d'OCaml).

L'analyse statique vise dans un premier temps à reconnaître de potentielles abstractions n-aires associées à une variable liée en attribuant à de telles variables une indication d'arité le cas échéant. Dans un second temps, muni de ces informations, elle détecte de telles variables dans des applications curryfiées dont le nombre d'applications imbriquées correspond à l'arité. Cette analyse suppose la connaissance d'informations liées aux variables apparaissant dans un terme. Elle se fait donc au travers d'un environnement permettant de stocker et consulter des informations qui indiquent le caractère fonctionnel et l'arité potentielle, le cas échéant, associée à une variable.

Les informations manipulées par l'analyse sont définies comme suit :

$$\gamma ::= \texttt{Unknown} \mid \texttt{Known}\ m.$$

Nous associerons à une variable liée à une abstraction une information de la forme `Known` m où m précise que la décurryfication donnerait une fonction n-aire d'arité $m+1$. Si la variable est liée à autre chose qu'une abstraction, nous lui associerons l'information `Unknown`.

Considérons le terme suivant : $\texttt{let}\ \lambda.\ \lambda.\ t\ \texttt{in}\ t_2$. Nous associerons l'information $\texttt{Known}\ 1$ à la variable liée par le $\texttt{let}$. En effet, elle est liée à une abstraction dont la décurryfication serait d'arité 2. Par contre, dans le terme $\texttt{let}\ (t_1\ t_2)\ \texttt{in}\ t$, nous associerons à la variable liée par le $\texttt{let}$ l'information $\texttt{Unknown}$.

Nous avons besoin de propager ces informations le long de l'analyse, afin de les utiliser notamment dans le traitement des applications. Pour ce faire, nous les propageons au travers d'un environnement :

$$\Gamma ::= [] \mid \gamma :: \Gamma.$$

Donnons nous quelques abréviations pour plus de lisibilité :
- Les abstractions curryfiées : $\texttt{nabs}\ m\ t = \underbrace{\lambda.\ \ldots \lambda}_{m+1}.\ t.$
- Les applications imbriquées : $\texttt{napp}\ t\ (t_0, \ldots, t_m) = (\ldots (t\ t_0) \ldots\ t_m)$.

Repérer les abstractions curryfiées se fait tout simplement en repérant les définitions locales ($\texttt{let}\ t_1\ \texttt{in}\ t_2$) où t_1 est de la forme $\texttt{nabs}\ m\ t$, auquel cas, cette variable sera associée à l'information $\texttt{Known}\ m$ dans l'environnement d'analyse. Sinon, on lui associera l'information $\texttt{Unknown}$.

$$\texttt{is\_curried\_abs}\ (t) = \begin{cases} \texttt{Known}\ m & \text{si } t = \texttt{nabs}\ m\ t_1 \\ \texttt{Unknown} & \text{sinon} \end{cases}$$

La décurryfication des applications nécessite le repérage des applications imbriquées ($\texttt{napp}\ t\ (t_0, \ldots, t_m)$) où l'appelé le plus interne t est une variable $n!$ dont on sait qu'elle est associée à une abstraction curryfiée d'arité potentielle $m+1$: $\texttt{Known}\ m$ dans l'environnement d'analyse. Comme l'algorithme de transformation descend récursivement dans la structure du terme $\mu\texttt{ML}$, il s'agit de repérer les applications dont le sous-terme gauche est de la forme ($\texttt{napp}\ n!\ (t_0, \ldots, t_{m-1})$) avec $\Gamma(n) = \texttt{Known}\ m$ où Γ est l'environnement d'analyse et $\Gamma(n)$ son $n+1$-ième élément. En effet, en appliquant le sous-terme droit t_m, on obtient une application totale.

4.2. Traduction de terme μML en terme nML

Lors de l'analyse, l'environnement Γ se comporte comme une approximation de l'environnement d'évaluation.

Si on analyse le terme $\lambda.\ t$ dans l'environnement Γ alors, on analyse le sous terme t dans l'environnement $\texttt{Unknown} :: \Gamma$.

Si $\texttt{let}\ t_1\ \texttt{in}\ t_2$ s'analyse dans Γ alors t_2 s'analyse dans $\texttt{is\_curried\_abs}\ t_1 :: \Gamma$. En particulier, si $t_1 = \texttt{nabs}\ m\ t_0$ alors on considère t_0 dans $\underbrace{\texttt{Unknown} :: \ldots :: \texttt{Unknown}}_{m+1} :: \Gamma$ que nous notons $\texttt{nUnknown}\ (m+1)\ \Gamma$.

Nous effectuons l'analyse et la traduction d'un terme $\mu\texttt{ML}$ en un terme $\texttt{nML}$ en même temps, en utilisant l'environnement d'analyse Γ comme environnement de traduction. Les mécanismes décrits plus haut permettent alors de traduire convenablement les applications, les sous-termes gauches de liaisons locales, selon que ce soit des abstractions ou pas, et les variables. Nous traduirons les fonctions potentiellement n-aires par la curryfication de leurs traductions. Notons $\texttt{NCurry}(m)$ le combinateur de curryfication à $m+1$ arguments défini comme suit : $\lambda^0.\ \underbrace{\lambda^0 \ldots \lambda^0}_{m+1}.\ (m+1)!\ [m!, \ldots, 0!]$.

$[\![t]\!]_\Gamma$ désigne la traduction du terme $\mu\texttt{ML}$ t dans l'environnement d'analyse Γ et est définie comme suit :

$$\begin{array}{lcl} [\![i]\!]_\Gamma & = & i \\ [\![n!]\!]_\Gamma & = & \begin{cases} \texttt{NCurry}(m)\ n! & \text{si } \Gamma(n) = \texttt{Known}\ m\,; \\ n! & \text{sinon} \end{cases} \end{array}$$

$$
\begin{array}{rcl}
[\![\lambda.\ t]\!]_\Gamma & = & \lambda^0.\ [\![t]\!]_{(\mathtt{Unknown}::\Gamma)} \\
[\![\mathtt{let}\ t_1\ \mathtt{in}\ t_2]\!]_\Gamma & = & \begin{cases} \mathtt{let}\ \lambda^m.\ [\![t]\!]_{(\mathtt{nUnknown}\ (m+1)\ \Gamma)}\ \mathtt{in}\ [\![t_2]\!]_{(\mathtt{Known}\ m::\Gamma)} & \text{si } t_1 = \mathtt{nabs}\ m\ t\,; \\ \mathtt{let}\ [\![t_1]\!]_\Gamma\ \mathtt{in}\ [\![t_2]\!]_{\mathtt{Unknown}::\Gamma} & \text{sinon} \end{cases} \\
[\![t_a\ t_b]\!]_\Gamma & = & \begin{cases} n!\ [[\![t_0]\!]_\Gamma, \ldots, [\![t_m]\!]_\Gamma, [\![t_b]\!]_\Gamma] & \text{si } t_a = \mathtt{napp}\ n!\ (t_0, \ldots, t_m) \\ & \text{et } \Gamma(n) = \mathtt{Known}(m+1)\,; \\ [\![t_a]\!]_\Gamma\ [[\![t_b]\!]_\Gamma] & \text{sinon} \end{cases}
\end{array}
$$

En guise d'illustration, considérons quelques exemples de traduction. Bien que nous soyons dans un formalisme de De Bruijn, pour plus de clarté nous nommons ici les variables :

- **Une application curryfiée totale d'une abstraction connue** : le terme μML $\mathtt{let}\ f = \lambda x.\ \lambda y.\ x + y\ \mathtt{in}\ f\ 3\ 4$ sera traduit par le terme nML $\mathtt{let}\ f = \lambda^1[x; y].\ x + y\ \mathtt{in}\ f\ [3; 4]$.
- **Une application curryfiée partielle d'une abstraction connue** : le terme μML $\mathtt{let}\ f = \lambda x.\ \lambda y.\ x + y\ \mathtt{in}\ f\ 3$ sera traduit par le terme nML $\mathtt{let}\ f = \lambda^1[x; y].\ x + y\ \mathtt{in}\ (\mathtt{NCurry}(1)\ f)\ [3]$.
- **Une application curryfiée totale d'une abstraction inconnue** : Le terme μML $(\lambda f.\ f\ 3\ 4)\ (\lambda x.\ \lambda y.\ x + y)$ sera traduit par le terme nML $(\lambda^0 f.\ f\ 3\ 4)\ [(\lambda^0 x.\ \lambda^0 y.\ x + y)]$.

En effet, dans les deux premiers exemples, le sous-terme gauche $f = \lambda x.\ \lambda y.\ x + y$ est traduit par la fonction d'arité 2 : $f = \lambda^1[x; y].\ x + y$ puisque $\lambda x.\ \lambda y.\ x + y = \mathtt{nabs}\ 1\ (x + y)$. Le sous-terme droit est alors traduit dans l'environnement $[(f; \mathtt{Known}\ 1)]$. Dans le premier exemple, $f\ 3\ 4$ constitue un appel total et est traduit par $f\ [3; 4]$ puisqu'on sait que f est une fonction attendant deux arguments. En effet, le sous-terme gauche de l'application $f\ 3$ est bien une application partielle n'attendant plus qu'un argument.

Par contre, dans le second exemple, $f\ 3$ est un appel partiel, f est donc traduit comme une variable dans l'environnement $[(f; \mathtt{Known}\ \ 1)]$ et donc par $(\mathtt{NCurry}(1)\ f)$.

Enfin, dans le troisième exemple, nous avons affaire à un appel curryfié qui pourrait être décurryfié. Cependant, notre algorithme ne se préoccupe pas des fonctions non liées par un lieur local, l'application est donc traduite par une application simple.

5. Préservation sémantique

La décurryfication étant formellement définie par la traduction d'un terme μML en un terme nML, nous pouvons entrer dans le vif du sujet : sa preuve de correction. Il nous faut montrer qu'il y a préservation du comportement entre un terme μML et sa traduction. Autrement dit que l'évaluation de l'un correspond à l'évaluation de l'autre. Il nous faut matérialiser cette notion de correspondance avant d'exhiber la preuve de correction. Il s'agit de mettre en évidence un lien entre valeurs μML et nML. Ce lien est donné sous forme d'une relation paramétrée par l'approximation d'évaluation donnée par l'analyse statique. En effet, l'analyse permet de donner une aproximation des formes des fermetures apparaissant dans un programme μML ainsi que dans sa décurryfication.

5.1. Caractérisation des fermetures

L'analyse statique nous permet lors de la traduction de différencier différents cas d'application. Elle nous permet de caractériser structurellement des fermetures (*i.e.* la structure de leurs composants) apparaissant lors de l'évaluation d'un terme μML tout comme pour le terme nML issu de la traduction.

5.1.1. Caractérisation des fermetures μML

Intéressons nous aux règles de sémantique de μML qui sont susceptibles de faire apparaître des fermetures :

$$\frac{e(n) = (t, e_1)}{e \vdash n! \rightarrow (t, e_1)}\ (1) \qquad \frac{e \vdash t_1 \rightarrow (\lambda.\ t, e_1) \quad e \vdash t_2 \rightarrow v_2 \quad v_2 :: e_1 \vdash t \rightarrow (t, v_2 :: e_1)}{e \vdash (t_1\ t_2) \rightarrow (t, v_2 :: e_1)}\ (2)$$

$$e \vdash (\lambda.\ t) \rightarrow (t, e)\ \ (3)$$

Utilisons maintenant les informations provenant de l'analyse pour caractériser ces fermetures.

Considérons la règle (1). Supposons que $\Gamma(n) = \texttt{Known}\ m$, alors il vient $t = \texttt{nabs}\ m\ t'$ et nous avons que dans e, $n!$ s'évalue en $(\texttt{nabs}\ m\ t', e_1)$.

Intéressons nous à l'évaluation d'un terme de la forme $\texttt{napp}\ n!\ (t_0, \ldots, t_k)$ dans l'environnement e. Supposons $\Gamma(n) = \texttt{Known}\ m$ et $k < m$. Sachant que t_0 dans e s'évalue en v_0, …, t_k en v_k, nous pouvons prévoir que $\texttt{napp}\ n!\ (t_0, \ldots, t_k)$ s'évaluera en $(\texttt{nabs}\ (m-k)\ b, v_k :: \ldots :: v_0 :: e_1)$ (k application de la règle (2)).

Dans les autres cas, nous n'avons pas d'information nous permettant de caractériser les fermetures introduites. Nous notons $e_1 @ e_2$ la concaténation de la liste e_1 à la liste e_2.

Notons $\mu\texttt{clos}\ m\ t\ e\ args = (\texttt{nabs}\ (m - |args|)\ t, args @ e)$.

Nous pouvons ainsi construire des règles d'évaluation hybrides : l'évaluation d'un terme t dans un environnement d'évaluation e dont les fermetures sont caractérisées grâce aux informations provenant d'un environnement d'analyse Γ.

$$\frac{e(n) = (t, e_1) \quad \Gamma(n) = \texttt{Known}\ m \quad t = \texttt{nabs}\ m\ t'}{e \vdash n! \rightarrow (\mu\texttt{clos}\ m\ t'\ e_1\ [])}\ \text{(eval-var-abs)}$$

$$\frac{e(n) = (\mu\texttt{clos}\ m\ t'\ e_1\ []) \quad e \vdash t_0 \rightarrow v_0 \quad \ldots \quad e \vdash t_k \rightarrow v_k \quad k < m}{e \vdash \texttt{napp}\ n!\ (t_0, \ldots, t_k) \rightarrow (\mu\texttt{clos}\ m\ t'\ e_1\ [v_k; \ldots; v_0])}\ \text{(eval-curried-app)}$$

Nous avons donc trois sortes de fermetures qui peuvent apparaître lors de l'évaluation d'un terme μML :

- les fermetures correspondant à une variable liée à une abstraction curryfiée : $(\mu\texttt{clos}\ m\ t'\ e_1\ [])$,
- les fermetures correspondant à une application imbriquée partielle, dont l'appelé le plus interne est une variable que l'on sait être liée à une abstraction curryfiée : $(\mu\texttt{clos}\ m\ t'\ e_1\ args)$,
- les fermetures quelconques pour les autres cas : (t, e).

5.1.2. Caractérisation des fermetures nML

De même nous pouvons caractériser les fermetures issues de la transformation. Commençons par nous intéresser à celle issue de l'évaluation d'une décurryfication d'une fonction n-aire.

$$\frac{e \vdash t \rightarrow v}{e \vdash \texttt{NCurry}(m)\ t \rightarrow (0,\ \underbrace{\lambda^0.\ldots.\lambda^0}_{m}.\ (m+1)![m!; \ldots; 0!], v :: e)}\ \text{(eval-}\texttt{NCurry}\text{)}$$

En effet, $\texttt{NCurry}(m)\ t = (\lambda^0.\ (\underbrace{\lambda^0.\ \ldots\lambda^0}_{m+1}.\ (m+1)![m!; \ldots; 0!]))\ [t]$.

Or, $e \vdash \texttt{NCurry}(m) \rightarrow (0,\ \underbrace{\lambda^0.\ \ldots\lambda^0}_{m+1}.\ (m+1)![m!; \ldots, 0!]; e)$. En appliquant la règle d'application, on obtient eval-$\texttt{NCurry}$.

Notons : $(\texttt{nclos}\ m\ clos\ e_0\ args) = (0,\ \underbrace{\lambda^0.\ \ldots\lambda^0}_{m-|args|}.\ (m+1)![m!; \ldots; 0!], args@(clos :: e_0))$. Grâce à l'analyse statique nous savons un peu plus de choses, notamment que $t = n!$ tel que $\Gamma(n) = \texttt{Known}\ m$.

En effet, le combinateur NCurry n'apparaît que dans la traduction des variables. De plus, nous pouvons caractériser l'évaluation d'une application curryfiée de la forme $(\ldots((\mathtt{NCurry}(m)\ n!)\ [t_0])\ldots[t_k])$ où $k < m$ et $\Gamma(n) = \mathtt{Known}\ m$.

Dans un environnement d'évaluation e et un environnement d'analyse Γ, considérons les deux règles hybrides suivantes :

$$\frac{e(n) = (m, t, e_0) \qquad \Gamma(n) = \mathtt{Known}\ m}{e \vdash \mathtt{NCurry}(m)\ n! \rightarrow (\mathtt{nclos}\ m\ (m, t, e_0)\ e\ [])} \ (\text{eval-}\mathtt{NCurry}\text{-var})$$

$$\frac{\begin{array}{c} e(n) = (m, t, e_0) \qquad \Gamma(n) = \mathtt{Known}\ m \\ e \vdash t_0 \rightarrow v_0 \qquad \ldots \qquad e \vdash t_k \rightarrow v_k \qquad k < m \end{array}}{e \vdash (\ldots((\mathtt{NCurry}(m)\ n!)\ [t_0])\ldots[t_k]) \rightarrow (\mathtt{nclos}\ m\ (m, t, e_0)\ e\ [v_k; \ldots; v_0])} \ (\text{eval-curried-app})$$

Tout comme pour les fermetures μML, nous avons trois sortes de fermetures pouvant apparaître lors de l'évaluation d'un terme nML issu de la transformation correspondant aux mêmes conditions :
- les fermetures correspondant à une variable liée à une abstraction curryfiée : $(\mathtt{nclos}\ m\ (m, t, e_0)\ e\ [])$,
- les fermetures correspondant à une application imbriquée partielle, dont l'appelé le plus interne est une variable que l'on sait être liée à une abstraction curryfiée : $(\mathtt{nclos}\ m\ (m, t, e_0)\ e_0\ args)$,
- les fermetures quelconques unaires pour les autres cas provenant de la transformation : $(0, t, e)$.

5.2. Correspondance entre les valeurs d'évaluation μML et nML

Les distinctions de fermetures en μML et nML proviennent des mêmes exploitations de l'analyse. Il paraît donc naturel d'exhiber un lien entre elles vis-à-vis de l'analyse. Ce lien se matérialise au travers d'une relation entre les fermetures μML et les fermetures nML paramétrée par les informations contenues dans l'environnement d'analyse.

Cette relation se stratifie en trois relations définies mutuellement :
- Correspondance entre les fermetures provenant de l'évaluation d'une variable. Elle définit dans quelles conditions une fermeture μML μv est en relation avec une fermeture nML nv selon l'information γ. Ce qui se note :$\mu v \sim_\gamma nv$.
- Propagation de cette relation aux environnements d'évaluation. Si $\forall n, \mu e(n) \sim_{\Gamma(n)} ne(n)$ alors μe et ne sont en correspondance par rapport à Γ, ce qui se note $\mu e \sim_\Gamma ne$.
- Correspondance entre valeurs issues d'évaluation : relation binaire entre une valeur μML μv et une valeur nML nv se notant $\mu v \sim nv$.

Ces relations traitent facilement les correspondances entre valeurs entières, elles se définissent comme suit :

$$\frac{\mu v \sim nv}{\mu v \sim_{\mathtt{Unknown}} nv} \ (\text{var-}\mathtt{Unknown}) \qquad \frac{[\![t]\!]_{\mathtt{nUnknown}\ (m+1)\ \Gamma} = u \qquad \mu e \sim_\Gamma ne}{(\mu\mathtt{clos}\ m\ t\ \mu e\ []) \sim_{(\mathtt{Known}\ m)} (m, u, ne)} \ (\text{var-}\mathtt{Known})$$

$$(i) \sim (i) \ (\text{match-int}) \qquad \frac{[\![t]\!]_{(\mathtt{Unknown}::\Gamma)} = u \qquad \mu e \sim_\Gamma ne}{(t, \mu e) \sim (0, u, ne)} \ (\text{clos-simpl})$$

$$\frac{[\![t]\!]_{(\mathtt{nUnknown}\ (m+1)\ \Gamma)} = u \qquad \mu e \sim_\Gamma ne \qquad |args| \leq m \qquad args \sim_{(\mathtt{nUnknown}\ (|args|)[])} args'}{(\mu\mathtt{clos}\ m\ t\ \mu e\ args) \sim (\mathtt{nclos}\ m\ (m, u, ne)\ e'\ args')} \ (\text{clos-curryfied})$$

Les règles (var-Unknown) et (var-Known) définissent la première de ces relations mutuelles : correspondance entre valeurs associées à une variable vis-à-vis d'une information d'analyse.

La première exprime le fait qu'une valeur μ`ML` μv et une valeur `nML` nv sont en correspondance vis-à-vis de l'information `Unknown` si elles sont en correspondance dans la relation entre valeurs d'évaluation.

La règle (var-`Known`) indique dans quelles conditions une fermeture curryfiée μ`ML` (μ`clos` $m\ t\ \mu e$ []) est en relation avec une fermeture n-aire (m, u, ne) `nML` vis-à-vis de l'information `Known` m. Il faut que la traduction de t dans l'environnement d'analyse Γ augmenté de $(m+1)$ approximations neutres (ie. `Unknown`) (`nUnknown` $(m+1)\ \Gamma$) pour les $(m+1)$ paramètres soit u et que les environnements des fermetures μ`ML` (μe) et `nML` (ne) soient en correspondance vis-à-vis de l'environnement d'analyse (Γ).

La relation de correspondance entre valeurs d'évaluation se définit *via* les trois règles suivantes : (match-int), (clos-simpl) et (clos-curryfied). La correspondance entre valeurs entières se réduit en l'égalité des entiers concernés. Une fermeture μ`ML` simple : $(t, \mu e)$ est en correspondance avec une fermeture `nML` $(0, u, ne)$ si dans l'environnement d'analyse (`Unknown` $::\ \Gamma$), u est la traduction de t et si μe et ne sont en correspondance vis-à-vis de Γ.

Enfin, une fermeture μ`ML` provenant d'un appel partiel d'une fermeture curryfiée (μ`clos` $m\ t\ \mu e\ args$) est en correspondance avec une fermeture curryfiée `nML` (`nclos` $m\ (m, u, ne)\ e'\ args'$) si dans l'environnement d'analyse (`nUnknown` $(m+1)\ \Gamma$), u est la traduction de t, si le nombre d'arguments déjà passés $|args|$ est inférieur à l'arité de la fermeture décurryfiée, si $args$ correspond à $args'$ vis-à-vis de (`nUnknown` $(|args|)$[]) et si les environnements de fermetures sont en correspondance vis-à-vis de Γ. On notera la présence d'un environnement quelconque dans la fermeture partielle `nML`. En effet, une fermeture de la forme `nclos` provient d'une évaluation d'une application partielle imbriquée dont l'appelant le plus interne est de la forme `NCurry`(m) $n!$. Or, il s'agit d'un terme clos (sans variable libre), l'environnement de fermeture est donc peu significatif.

5.3. Correction de la Décurryfication

Grâce aux relations définies ci-dessus, nous pouvons comparer une évaluation μ`ML` à une évaluation `nML`. Cette notion de correspondance de valeur d'évaluation nous permet de comparer l'évaluation d'un programme μ`ML` à l'évaluation de sa traduction en `nML`. Nous avons démontré le théorème de correction suivant :

Théorème 1 *(Préservation sémantique de la décurryfication)*
Si $[\![t]\!]_{[]} = u$ *et* $[] \vdash\ t \rightarrow \mu v$*, avec alors* $\exists\ nv, [] \vdash\ u \rightarrow nv$ *et* $\mu v \sim nv$.

La preuve se fait par induction sur les règles d'évaluation μ`ML`. On prouve, pour chaque règle d'évaluation μ`ML`, le diagramme de simulation suivant :

$$\begin{tikzcd} \mu e \vdash t \arrow[r, no head, "\mu e \sim_\Gamma ne"] \arrow[d] & ne \vdash [\![t]\!]_\Gamma \arrow[d, dotted] \\ \mu v \arrow[r, dotted, "\sim"] & nv \end{tikzcd}$$

Si le terme μ`ML` t s'évalue en une valeur μv dans l'environnement μe et que $\mu e \sim_\Gamma ne$ alors il existe une valeur nv, telle que dans l'environnement ne, le terme $[\![t]\!]_\Gamma$ s'évalue en nv et $\mu v \sim nv$.

Le cas intéressant de la preuve est celui de l'application (les propriétés présentées comme des résultats ont aussi été démontrées). Nous en détaillons ici la démonstration :

Lemme 1 *Si* $\mu e \vdash (t_a\ t_b) \rightarrow \mu v$ *et* $\mu e \sim_\Gamma ne$ *alors* $\exists nv, ne \vdash [\![(t_a\ t_b)]\!]_\Gamma \rightarrow nv$ *et* $\mu v \sim nv$.

Par induction, nous avons les trois propriétés suivantes :

– si $\mu e \vdash t_a \rightarrow (t, \mu e_a)$ et $\mu e \sim_\Gamma ne$ alors $\exists nv_a, ne \vdash [\![t_a]\!]_\Gamma \rightarrow nv_a$ et $(t, \mu e_a) \sim nv_a$
– si $\mu e \vdash t_b \rightarrow \mu v_b$ et $\mu e \sim_\Gamma ne$ alors $\exists nv_b, ne \vdash [\![t_b]\!]_\Gamma \rightarrow nv_b$ et $\mu v_b \sim nv_b$
– si $\mu v_b :: \mu e_a \vdash t \rightarrow \mu v$ et $\mu v_b :: \mu e_a \sim_{\Gamma'} ne'$ alors $\exists nv, ne' \vdash [\![t]\!] \rightarrow nv$ et $\mu v \sim nv$

De la transformation, nous distinguons deux cas à traiter :

– **I. Cas non optimisé** : $[\![(t_a\ t_b)]\!]_\Gamma = [\![t_a]\!]_\Gamma\ [[\![t_b]\!]_\Gamma]$ et t_a n'est pas une application partielle d'une fonction connue attendant un dernier argument.
– **II. Cas optimisé** : $[\![(t_a\ t_b)]\!]_\Gamma = n!\ [[\![t_0]\!]_\Gamma; \ldots ; [\![t_m]\!]_\Gamma; [\![t_b]\!]_\Gamma]$ et $t_a = \texttt{napp}\ n\ (t_0, \ldots, t_m)$ avec $\Gamma(n) = \texttt{Known}\ m + 1$.

I. Cas non optimisé :

En utilisant l'hypothèse d'induction sur l'évaluation de t_a et t_b, il vient les deux résultats :

– **(a)** : $ne\ \vdash [\![t_a]\!]_\Gamma \rightarrow nv_a$ et $(t, \mu e_a) \sim nv_a$
– **(b)** : $ne\ \vdash [\![t_b]\!]_\Gamma \rightarrow nv_b$ et $\mu v_b \sim nv_b$

Or, μv est le résultat de l'évaluation $\mu v_b :: \mu e_a \vdash t \rightarrow \mu v$. Pour appliquer le diagramme à cette évaluation, il nous faut identifier nv_a comme étant une fermeture et identifier le corps et l'environnement. Nous devons donc nous intéresser à la forme de la fermeture issue de l'évaluation de t_a. Pour ce faire, nous utilisons le résultat **(a)** et plus particulièrement $(t, \mu e_a) \sim nv_a$. Selon la règle utilisée pour obtenir cette correspondance, il s'offre à nous deux cas :

1) Il s'agit de la règle (clos-simpl). Il vient $nv_a = (0, u, ne_a)$ avec $[\![t]\!]_{(\texttt{Unknown}::e_{a1})} = u$ et $\mu e_a \sim_{e_{a1}} ne_a$. Nous pouvons appliquer l'hypothèse d'induction sur l'évaluation de t et nous obtenons ainsi le résultat.

2) Il s'agit de la règle (clos-curryfied). $(t, \mu e_a)$ provient d'une application partielle d'une fermeture curryfiée : $(t, \mu e_a) = (\mu\texttt{clos}\ m\ b\ \mu e_{a_1}\ args)$. Il vient $nv_a = (\texttt{nclos}\ m\ (m, u, ne_{a_1})\ ne_c\ args')$ où $\mu e_{a_1} \sim_{\Gamma_{a_1}} ne_{a_1}, |args| \leq m$ et $args \sim_{\texttt{nUnknown}\ (m+1)\ []} args'$ et $\mu e_{a_1} \sim_{\Gamma_{a_1}} ne_{a_1}$.

Nous allons distinguer selon le nombre de paramètres restant à passer pour avoir une application totale :

a – il reste plus d'un paramètre

Il s'agit d'une application partielle : il vient $\mu v = (\mu\texttt{clos}\ m\ b\ \mu e_{a_1}\ (\mu v_b :: args))$ et $nv = (\texttt{nclos}\ m\ (m, u, ne_{a_1})\ ne_c\ (nv_b :: args'))$ et le résultat se démontre par une application de la règle d'évaluation de l'application `nML` et en montrant par construction
$(\mu\texttt{clos}\ m\ b\ \mu e_{a_1}\ \mu v_b :: args) \sim (\texttt{nclos}\ m\ (m, u, ne_{a_1})\ ne_c\ (nv_b :: args'))$.

b – il reste un paramètre

Nous avons donc affaire à une application totale. Le résultat de l'évaluation est équivalent à celui de u dans l'environnement $((nv_b :: args')@ne_{a_1})$. On obtient cette évaluation en exploitant convenablement l'hypothèse d'induction sur l'évaluation de t. Il vient $((nv_b :: args')@ne_{a_1}) \vdash u \rightarrow nv$ et $\mu v \sim nv$. Cependant la traduction de t_a est de la forme : $(\ldots((NCurry\ m\ n!)\ [\![t_0]\!]_\Gamma)\ldots)[\![t_m]\!]_\Gamma)$. Nous avons donc à montrer que $(m+1)!\ [m!; \ldots ; 0!]$ s'évalue en nv dans l'environnement $((nv_b :: args' :: (m, u, ne_{a_1}))@ne_c)$. Pour cela nous utilisons le résultat intermédiaire suivant :

Si $|args| = n$ et $(v :: args)@me \vdash m \rightarrow vres$ alors
$(v :: args')@((n, m, me) :: e') \vdash (n+1)![n!; \ldots ; 0!] \rightarrow vres$.

Nous obtenons ainsi le résultat.

II. Cas optimisé :

Par abus de notation, notons $[\![args]\!]_\Gamma$ pour $[[\![t_0]\!]_\Gamma; \ldots ; [\![t_m]\!]_\Gamma]$ avec $args = (t_0, \ldots, t_m)$. Nous avons $[\![t_a]\!]_\Gamma = n!\ [\![args]\!]_\Gamma$, $t_a = \texttt{napp}\ n!\ args$ et que $\Gamma(n) = \texttt{Known}\ |args|$. De plus, notons $[\![args]\!]_\Gamma = nargs$. Nous avons $\mu e \sim_\Gamma ne$, il vient $\mu e(n) = (\mu\texttt{clos}\ \ |args|\ b\ e'\ [])$.

De là, nous utilisons la correspondance entre environnements pour en déduire que :
$ne(n) = (|args|, u, ne')$ et $(\mu\texttt{clos}\ \ |args|\ t\ e'\ []) \sim_{(\texttt{Known}\ |args|)} (|args|, u, ne')$.

Nous avons en hypothèse $\mu e \vdash \texttt{napp}\ n!\ args \rightarrow (t', e)$.

Nous utilisons le résultat intermédiaire suivant : Si $e \vdash \texttt{napp}\ n!\ args \rightarrow v$ alors $\exists vn, \exists vargs, e \vdash n! \rightarrow vn$ et $e \vdash args \rightarrow vargs$.

Nous obtenons donc $vargs$. En appliquant la règle hybride d'évaluation d'application partielle (eval-curried-app), nous obtenons $(t, e) = (\mu\texttt{clos}\ |args|\ t\ e'\ (rev\ vargs))$ si
$\mu e(n) = (\mu\texttt{clos}\ \ |args|\ b\ e'\ [])$ et $\mu e \vdash args \rightarrow vargs$.

De plus, nous savons que la transformation de t_a est une imbrication de m applications curryfiées de $n!$ aux éléments de $nargs$: $[\![t_a]\!]_\Gamma = \texttt{Nnapp}\ (\texttt{NCurry}(|args|)\ n!)\ nargs$. En utilisant l'hypothèse d'induction associée à l'évaluation de t_a, nous obtenons : $ne \vdash [\![t_a]\!]_\Gamma \rightarrow nv_1$ et $(\mu\texttt{clos}\ |args|\ t\ e'\ (rev\ vargs)) \sim nv_a$.

Nous essayons donc d'en savoir un peu plus sur nv_a. On sait que
$ne \vdash (\texttt{NCurry}(|args|)\ n!) \rightarrow (\texttt{nclos}\ |args|\ (|args|, u, ne')\ ne\ [])$ *via* la règle (eval-NCurry).

De plus, de $ne \vdash \texttt{Nnapp}\ (\texttt{NCurry}(|args|)\ n!)\ nargs \rightarrow nv_1$, on déduit :
$\exists nvargs, ne \vdash nargs \rightarrow nvargs$ et $nv_a = (\texttt{nclos}\ |args|\ (|args|, u, ne')\ ne\ (rev\ nvargs))$.

Nous avons donc la correspondance entre fermetures μML et nML suivante :
$(\mu\texttt{clos}\ |args|\ t\ \mu e'\ (rev\ vargs)) \sim (\texttt{nclos}\ |args|\ (|args|, u, ne')\ ne\ (rev\ nvargs))$.

Bien que nous ayons caractérisé les fermetures, nous devons traiter les deux règles conduisant à une correspondance entre fermetures : (clos-simpl) et (clos-curryfied). En effet, $(\mu\texttt{clos}\ \ldots)$ est de la forme (t, e) et $(\texttt{nclos}\ \ldots)$ est de la forme $(0, u, e)$. Nous avons donc deux cas à traiter.

1) Correspondance provenant de (clos-simpl) :

Nous avons donc en hypothèse :

1 – $[\![t]\!]_\Gamma = ((m+1)!\ [m!; \ldots; 0!]$ et 2 – $((rev\ vargs)@\mu e) \sim_\Gamma ((rev\ nvargs)@(|args|, u, ne') :: e')$.

En utilisant l'hypothèse d'induction sur l'évaluation de t, on obtient :
$(nv2 :: (rev\ nvargs)@(|args|, u, ne') :: e') \vdash ((m+1)![m!, \ldots, 0!] \rightarrow nv$. Or nous avons à montrer : $\exists nv, (nv2 :: (rev\ nvargs)@ne) \vdash u \rightarrow nv$.

Nous avons le résultat suivant :
Si $|args| = n$ et $(nv2 :: (rev\ nvargs)@(n, u, ne') :: e') \vdash (m+1)!\ [m!; \ldots; 0!] \rightarrow nv$ alors
$(nv2 :: (rev\ nvargs)@ne) \vdash u \rightarrow nv$.

Appliqué ici , nous obtenons le résultat.

2) Correspondance provenant de (clos-curryfied) :

Nous avons en hypothèses :

1 – $[\![t]\!]_{(\texttt{nUnknown}\ |args|\ \Gamma)} = u$, 2 – $e' \sim_\Gamma ne'$ et 3 – $vargs \sim_{(\texttt{nUnknown}\ |args|\ [])} nvargs$.

Nous en déduisons $(v_2 :: (rev\ vargs)@e') \sim_{(\texttt{nUnknown}\ (|args|+1)\ \Gamma)} (nv_2 :: (rev\ nvargs)@ne')$. Enfin, en utilisant l'hypothèse d'induction sur l'évaluation de t, nous obtenons le résultat.

6. Extension aux fonctions récursives

Extension de μML aux fonctions récursives (μMLrec) : Sa syntaxe est la même que μML à l'ajout près des définitions récursives destinées à la définition des fonctions récursives. Ainsi, un terme de la forme $\texttt{letrec}\ t_1\ \texttt{in}\ t_2$ définit une fonction récursive de corps t_1 dans le terme t_2. Au niveau de l'évaluation, il nous faut ajouter des valeurs de fermetures récursives $(t, e)_{rec}$ ainsi que deux règles d'évaluations semblables à celle du λ-calcul.

Termes : $t ::= \ldots \mid \texttt{letrec}\ t_1\ \texttt{in}\ t_2$

Valeurs : $v ::= \ldots \mid (t, e)_{rec}$

$$\frac{(t_1, \mu e)_{rec} :: \mu e \vdash t_2 \rightarrow v}{\mu e \vdash \texttt{letrec}\ t_1\ \texttt{in}\ t_2 \rightarrow v} \qquad \frac{\mu e \vdash t_1 \rightarrow (t, e)_{rec} \quad \mu e \vdash t_2 \rightarrow v_2 \quad v_2 :: (t, e)_{rec} :: e \vdash t \rightarrow v}{\mu e \vdash t_1\ t_2 \rightarrow v}$$

Le terme $\texttt{letrec}\ f\ x = x + (f\ x)\ \texttt{in}\ (f\ 3)$ s'écrit en μMLrec $\texttt{letrec}\ 0! + (1!\ 0!)\ \texttt{in}\ (0!\ 3)$. Dans le sous-terme gauche, le paramètre est traduit comme la variable 0! tandis que la variable de récursion est traduite par la variable 1!.

Extension de nML aux fonctions récursives (nMLrec) : Nous y ajoutons des termes de la forme $\texttt{letrec}^n\ t_1\ \texttt{in}\ t_2$ où n indique l'arité de la fonction récursive définie. Du point de vue de l'évaluation, nous avons besoin de fermetures récursives : $(n, t, e)_{rec}$ où n indique l'arité de la fermeture. Nous avons besoin de deux nouvelles règles d'évaluation pour traiter les fonctions récursives. (eval-letrec) : un terme de la forme $\texttt{letrec}^n\ t_1\ \texttt{in}\ t_2$ s'évalue en la valeur v dans l'environnement ne si le terme t_2 s'évalue en v dans l'environnement $(n, t_1, ne)_{rec} :: ne$. (eval-appr) : le terme $t_f\ [t_0; \dots; t_n]$ s'évalue en la valeur v dans l'environnement ne, si t_f s'y évalue en la fermeture récursive $(n, t, e)_{rec}$ et les $t_1, \dots t_n$ en $v0; \dots; v_n$ alors t s'évalue en v dans l'environnement $v_n :: \dots :: v_0 :: (n, t, e)_{rec} :: e$.

Termes : $t ::= \dots \mid \texttt{letrec}^n\ t_1\ \texttt{in}\ t_2$
Valeurs : $v ::= \dots \mid (n, t, e)_{rec}$

$$\frac{(n, t_1, ne)_{rec} :: ne \vdash t_2 \to v}{ne \vdash \texttt{letrec}^n\ t_1\ \texttt{in}\ t_2 \to v}\ \text{(eval-letrec)}$$

$$\frac{\begin{array}{c} ne \vdash t_f \to (n, t, e)_{rec} \\ ne \vdash t_0 \to v_0 \quad \dots \quad ne \vdash t_n \to v_n \quad v_n :: \dots :: v_0 :: (n, t, e)_{rec} :: e \vdash t \to v \end{array}}{ne \vdash t_f\ [t_0; \dots; t_n] \to v}\ \text{(eval-appr)}$$

Décurryfication des fonctions récursives : Du point de vue de l'analyse statique, nous tentons de repérer les définitions récursives telles que le premier sous terme soit de la forme $\texttt{nabs}\ n\ t$. En effet, dans une telle configuration, nous savons que la fonction récursive ainsi définie est d'arité $n + 2$.

La traduction d'un terme de la forme $\texttt{letrec}\ t_1\ \texttt{in}\ t_2$ s'énonce donc comme suit :

$$[\![\texttt{letrec}\ t_1\ \texttt{in}\ t_2]\!]_\Gamma \quad = \quad \begin{cases} \texttt{letrec}^{(n+1)}\ [\![t]\!]_{(\texttt{nUnknown}\ (n+2)\ (\texttt{Known}\ (n+1)::\Gamma))}\ \texttt{in}\ [\![t_2]\!]_{(\texttt{Known}\ (n+1)::\Gamma)} \\ \quad \text{si } t_1 = \texttt{nabs}\ n\ t; \\ \texttt{letrec}^0\ [\![t_1]\!]_{(\texttt{Unknown}::\texttt{Unknown}::\Gamma)}\ \texttt{in}\ [\![t_2]\!]_{(\texttt{Unknown}::\Gamma)} \\ \quad \text{sinon} \end{cases}$$

Nous remarquons que dans les deux cas, la traduction de t_1, nécessite la connaissance d'informations sur la variable de récursion.

Caractérisation des fermetures récursives : Tout comme pour μML, nous pouvons caractériser les fermetures apparaissant lors de l'évaluation d'un terme μMLrec. En dehors des trois formes caractérisées plus haut, la présence de fermetures récursives alliée aux informations provenant de l'environnement d'analyse, nous permet non seulement de caractériser les fermetures récursives mais aussi des fermetures partielles provenant de l'évaluation d'applications imbriquées.

Notons : $(\mu\texttt{closrec}\ m\ t\ e\ args) = (\texttt{nabs}\ \ (m - |args|)\ t, args@(\texttt{nabs}\ \ m\ t, e) :: e)$. Il vient les deux règles hybrides supplémentaires :

$$\frac{e(n) = (t, e_1)_{rec} \qquad \Gamma(n) = \texttt{Known}\ m \qquad t = \texttt{nabs}\ \ m\ b}{e \vdash n! \to (\texttt{nabs}\ \ m\ b, e_1)_{rec}}$$

$$\frac{e(n) = (\texttt{nabs}\ \ m\ b, e_1)_{rec} \qquad e \vdash t_0 \to v_0 \qquad \dots \qquad e \vdash t_k \to v_k \qquad k \le m}{e \vdash \texttt{napp}\ n!(t_0, \dots, t_k) \to (\mu\texttt{closrec}\ m\ t\ e_1\ [v_k; \dots; v_0])}$$

De même, nous obtenons des caractérisations supplémentaires pour les fermetures de `nMLrec`.

$$\frac{e(n) = (m, t, e_0)_{rec} \qquad \Gamma(n) = \texttt{Known}\ m}{e \vdash \texttt{NCurry}\ m\ n! \rightarrow (\texttt{nclos}\ m\ (m, t, e_0)_{rec}\ e\ [])}$$

$$\frac{\begin{array}{c} e(n) = (m, t, e_0)_{rec} \qquad \Gamma(n)\texttt{Known}\ m \\ e \vdash t_0 \rightarrow v_0 \qquad \ldots \qquad e \vdash t_k \rightarrow v_k \qquad k \leq m \end{array}}{e \vdash (\ldots((\texttt{NCurry}\ m\ n!)\ [t_0])\ldots[t_k]) \rightarrow (\texttt{nclos}\ m\ (m, t, e_0)_{rec}\ e\ [v_k; \ldots; v_0])}$$

Extension de la correspondance des valeurs : L'extension de caractérisation des fermetures nous amène à de nouvelles correspondances de fermetures :

$$\frac{[\![u]\!]_{(\texttt{Unknown::Unknown::}\Gamma)} = m \qquad e \sim_\Gamma me}{(u, e)_{rec} \sim_{(\texttt{Known}\ 0)} (0, m, me)_{rec}}\ (1) \qquad \frac{[\![u]\!]_{(\texttt{nUnknown}\ (n+2)\ (\texttt{Known}\ (n+1)\texttt{::}\Gamma)} = m \qquad e \sim_\Gamma me}{(\texttt{nabs}\ \ n\ u, e)_{rec} \sim_{(\texttt{Known}\ n+1)} (n+1, m, me)_{rec}}\ (2)$$

$$\frac{[\![u]\!]_{(\texttt{Unknown::Unknown::}\Gamma)} = m \qquad e \sim_\Gamma me}{(u, e)_{rec} \sim (0, m, me)_{rec}}\ (3)$$

$$\frac{[\![u]\!]_{(\texttt{nUnknown}\ (n+2)\ (\texttt{Known}\ (n+1)\texttt{::}\Gamma)} = m \qquad e \sim_\Gamma me}{(\texttt{nabs}\ \ n\ u, e)_{rec} \sim (\texttt{nclos}\ (n+1)\ (n+1, m, me)_{rec}\ e'\ [])}\ (4)$$

$$\frac{\begin{array}{c} [\![u]\!]_{(\texttt{nUnknown}\ (n+1)\ (\texttt{Known}\ n\texttt{::}\Gamma)} = m \\ e \sim_\Gamma me \qquad |args| \leq n \qquad args \sim_{\texttt{nUnknown}\ |args|\ []} args' \end{array}}{(\mu\texttt{closrec}\ n\ u\ e\ args) \sim (\texttt{nclos}\ (n)\ (n, m, me)_{rec}\ e'\ args')}\ (5)$$

Au niveau de la relation de correspondance entre valeurs provenant de l'évaluation de variable, nous ajoutons deux règles : (1) et (2). La règle(2) décrit les conditions de correspondance quand l'arité est supérieur à 1 (`Known (n+1)`) : la fermeture récursive $(\texttt{nabs}\ n\ u, e)_{rec}$ est en correspondance avec la fermeture récursive `nMLrec` $(n+1, m, me)_{rec}$ si les environnements e et me sont en correspondance vis-à-vis de Γ et et que m est la traduction de u dans l'environnement d'analyse $(\texttt{nUnknown}\ (n+2)\ (\texttt{Known}\ \ (n+1) :: \Gamma$.

La relation de correspondance entre valeurs issues d'évaluation est aussi enrichie. La règle (3) définit les conditions de correspondance entre fermetures récursives provenant d'évaluation simple (pas d'une règle hybride).

En ce qui concerne les fermetures partielles provenant d'applications imbriquées telles que l'appelant le plus interne s'évalue en une fermeture récursive, il nous faut distinguer deux cas. D'une part, lorsqu'il s'agit de l'évaluation d'une variable et de sa traduction *via* le combinateur `NCurry` (4) ; et d'autre part, lorsqu'on a affaire à une correspondance entre fermetures partielles (5).

Correction : Tout comme pour la traduction de terme μ`ML` en terme `nML`, nous avons démontré de la même manière le théorème de préservation sémantique par la transformation de terme μ`MLrec` en terme `nMLrec`. La preuve est similaire à celle présentée plus haut, elle traite plus de cas induits par les règles supplémentaire de la relation de correspondance de valeurs.

7. Conclusions

Dans cet article, nous avons présenté un algorithme de décurryfication statique et une preuve Coq qu'il préserve la sémantique. Il s'agit, à notre connaissance, de la première preuve vérifiée par machine de la correction d'une telle optimisation.

La transformation est formalisée un peu différemment en `Coq`. En effet, l'induction utilisée sur les termes n'est pas structurelle et donc elle ne peut être encodée par un point fixe `Coq`. Dans le développement, nous utilisons une formalisation relationnelle de l'algorithme pour les preuves et une formalisation un peu plus compliquée pour le développement. Nous avons démontré un théorème de correction entre la formalisation du développement et la relation. La preuve suit sensiblement celle exposée ici. Les relations et les sémantiques sont représentées par des types inductifs de `Coq`.

Ce travail peut se poursuivre dans plusieurs directions. L'une est d'enrichir ce développement avec d'autres optimisations nécessitant de l'analyse de code. L'expansion de code de fonction s'envisagerait au même niveau que la décurryfication. Tandis que la reconnaissance de fonction s'effectuerait dans une phase plus tardive, après le nommage des fonctions. Tout comme la décurryfication, ces optimisations peuvent se contenter d'analyses statiques plus simples que les analyses de flot de contrôle classiques.

Une autre direction est la formalisation et la preuve de correction en Coq d'analyses de flot de contrôle plus précises : k-CFA, *polymorphic splitting*, ...

Références

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

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development – Coq'Art : The Calculus of Inductive Constructions.* EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[3] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM 2006 : Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer-Verlag, 2006.

[4] Coq. `http://coq.inria.fr`.

[5] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indag. Math*, 34,5 :381–392, 1972.

[6] X. Leroy. The ZINC experiment : an economical implementation of the ML language. Rapport technique 117, INRIA, 1990.

[7] X. Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[8] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Logiciel et documentation disponibles sur le Web, `http ://caml.inria.fr/`, 1996–2006.

[9] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq.* PhD thesis, Université Paris-Sud, July 2004.

[10] S. Marlow and S. Peyton Jones. Making a fast curry : push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4–5) :375–414, 2006.

[11] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML (revised).* The MIT Press, 1997.

[12] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis.* Springer-Verlag, 1999.

[13] S. Peyton Jones. *Haskell 98 Language and Libraries : The Revised Report.* Cambridge University Press, 2003.

[14] S. L. Peyton Jones. *The implementation of functional programming languages.* Prentice-Hall, 1987.

[15] O. Shivers. Control-flow analysis in Scheme. In *Programming Language Design and Implementation 1988*, pages 164–174. ACM Press, 1988.