Séparation des couleurs dans un lambda-calcul bichrome

Emmanuel Chailloux, Bernard Serpette

Séparation des couleurs dans un λ-calcul bichrome

Emmanuel Chailloux[1] & Bernard Paul Serpette[2]

1: Laboratoire d'informatique de Paris 6 (UMR 7606),
Université Pierre et Marie Curie (Paris 6)
4, Place Jussieu, 75005 Paris
Emmanuel.Chailloux@lip6.fr
2: Inria Sophia-Antipolis Méditerranée,
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
Bernard.Serpette@inria.fr

Résumé

Dans cet article nous introduisons un λ-calcul bichrome pour expliciter une partie de l'évaluation d'un terme en précisant la localité du calcul[1]. L'intérêt est alors de pouvoir définir une transformation, par β-expansion, qui regroupe les expressions de même couleur. Les propriétés de correction, de terminaison et de confluence de cette transformation sont démontrées à l'aide de l'assistant de preuves Coq[2]. Cette transformation est indépendante de la sémantique de communication et de synchronisation de l'application. On s'intéresse alors aux applications utilisant deux unités de calcul comme les couples client-serveur de la programmation Web. Nous abordons le passage à un λ-calcul à plus de deux couleurs et montrons les difficultés que cela engendre.

1. Introduction

Bien que le λ-calcul ait été défini en monochrome lors de son invention par Alonzo Church [3], plusieurs extensions colorées ont déjà été proposées, soit pour en faire une présentation pédagogique, soit pour alléger des annotations sur les termes. Dans la première catégorie, le jeu de collage AlligatorEggs[3] cherche à expliquer le λ-calcul à des enfants. Pour cela il utilise la couleur pour relier des alligators affamés et des oeufs afin de constituer des familles. La couleur sert à expliciter les liaisons des variables dans les termes, des oeufs naissent de nouveaux alligators ou familles lors de la β-réduction. On retrouve l'utilisation de la couleur pour expliquer la programmation à des non-informaticiens comme dans l'article "Lambda-calculus and Music Calculi" [12], qui s'adresse à des musiciens et permet d'expliquer la programmation graphiquement. Dans une seconde catégorie, l'introduction de la couleur sert d'annotations sur les termes. Cette technique est utilisée pour ajouter des informations de typage sur les termes. Dans "Colored local type inference" [8], la coloration des types est utilisée pour préciser les directions de propagation. Dans la thèse "Types abstraits dans les systèmes répartis"[9], la notion de crochet coloré est reprise pour ajouter l'annotation de l'empreinte h d'un type abstrait où est produite une expression E, notée E_h^T. On retrouve dans "static typing for a faulty lambda calculus" [11] un λ-calcul coloré pour détecter les défauts intermittents et les récupérer par réplication

[1]Ce travail a bénéficié du soutien de l'ANR : projet **PWD** (Programmation du Web Diffus) ANR-09-EMER-009-01.

[2]les sources sont disponibles sur `ftp://ftp-sop.inria.fr/indes/rp/jfla2012.v`

[3]`http://worrydream.com/AlligatorEggs/` : Le jeu `AlligatorEggs` présente le λ-calcul à base d'alligators affamés (pour l'abstraction), d'oeufs (pour les variables), et de vieux alligators (pour les parenthèses). L'application est réalisée par la justaposition de familles d'alligators. Quand un alligator mange, il effectue une β-réduction. Les couleurs servent de liaison des variables.

de calcul avec coloriage RGB des types. Dans un autre type d'annotation, on retrouve une coloration de λ-terme pour guider des preuves en distinguant des éléments de syntaxe [6].

Dans notre travail, on ne cherche pas à éclairer un système de types mais plutôt à expliciter une partie de l'évaluation. Les annotations proposées dans cet article cherchent à alléger la lecture d'un calcul en précisant la localité du calcul. Chaque calculateur distinct pourra être représenté par une couleur différente. L'idée est de pouvoir représenter par une couleur le lieu où s'effectue le calcul. Cela est immédiatement utilisable pour les systèmes de programmation multi-tiers pour le web, où l'application est développée dans un seul langage et où apparaissent dans le code les échappements entre les serveurs et les clients. Ces échappements sont indiqués par $ et ~ en Hop [10], environnement Scheme où les valeurs transitant entre clients et serveurs utilisent de manière transparente le protocole HTTP. On retrouve un mécanisme proche dans les versions statiquement typées à la Ocsigen [1] avec les échappements : {{ et %).

Si nous reprenons la terminologie de Hop, l'échappement $ permet de spécifier que le code qui suit est exécuté sur le serveur, l'échappement ~ permet d'introduire le code client. Le code client peut faire appel de nouveau au serveur en réutilisant un échappement $. On peut imaginer, par exemple, que l'action associée à un bouton du client soit dépendante de certaines données du serveur (numéro de commande, proxy, base de données...). Un tel exemple est schématisé par un pseudo arbre de syntaxe abstraite (AST) en haut à gauche de la figure 1.

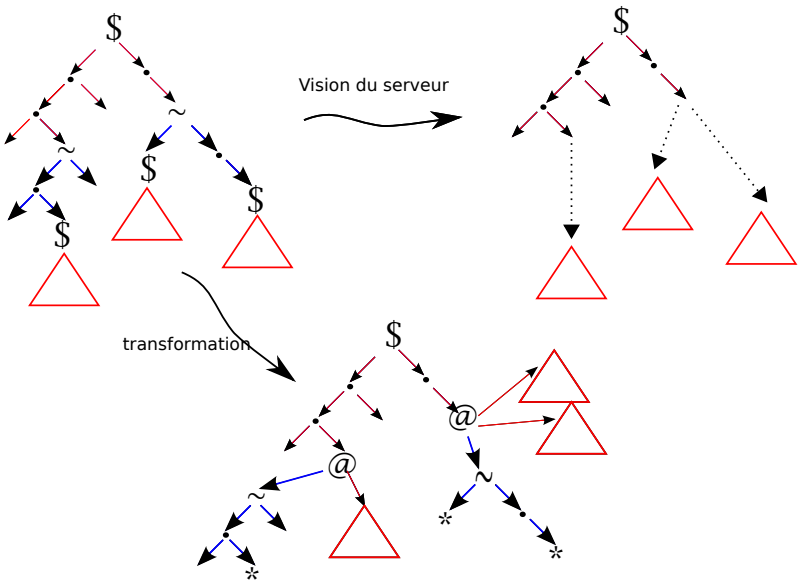

Figure 1: Exemple d'arbre Hop

Du point de vue du serveur, lorsqu'un échappement ~ apparait, le code client est *ignoré*, par contre l'environnement dans lequel ce code client est activé doit être préservé pour les $ contenus dans ce code. Il existe donc une *relation* entre l'apparition d'un noeud ~ et ses $ englobants. Ces *relations* sont décrites par les flèches en pointillés dans l'arbre en haut à droite de la figure 1. Le but de cet article est de formaliser une transformation permettant de mettre en évidence cette relation entre un échappement et les échappements qu'il contient. Cette transformation est schématisée en bas de la figure 1 où l'on notera que la transformation regroupe les codes de même couleur.

Cet article est découpé en quatre parties. La section 2 introduit les notations du λ-calcul bichrome afin de définir la transformation de regroupement des couleurs. La section 3 montre les propriétés de correction, de confluence et de terminaison de cette transformation. La section 4 montre des

exemples d'applications issues de la programmation Web. La conclusion discute du passage à un λ-calcul polychrome.

2. Définitions

2.1. λ-calcul bichrome

Le λ-calcul est classiquement défini avec des variables appartenant à un certain domaine *var*, avec des fonctions que nous noterons $\lambda x.B$ et une application que nous noterons (@ F A). Il se formalise en Coq comme suit :

Variable *var*:**Set.**

Inductive *expr* : **Set** :=
Var: *var* → *expr*
| Fun: *var* → *expr* → *expr*
| App: *expr* → *expr* → *expr*.

Nous utiliserons les notations $\lambda xy.B$ pour $\lambda x.\lambda y.B$, (@ F A B) pour (@ (@ F A) B) et **let** x=A **in** B pour (@ $\lambda x.B$ A). Les **let**s révèlent l'existence d'un *redex* (*Red*ucible *Ex*pression) qui est la structure sur laquelle s'appuie la β-réduction, le pas de calcul essentiel du λ-calcul.

Nous voulons permettre à l'utilisateur de spécifier, pour chaque expression, si l'évaluation de cette dernière devra se faire sur le serveur ou sur le client. Pour rester abstrait, les deux unités de calcul seront représentées par des couleurs et seront interchangeables. Toutes les expressions possèdent donc une annotation de couleur :

Inductive *color*:**Set.**
| Red: *color*
| Blue: *color*.

Inductive *cexpr* : **Set** :=
| CVar: *color* → *var* → *cexpr*
| CFun: *color* → *var* → *cexpr* → *cexpr*
| CApp: *color* → *cexpr* → *cexpr* → *cexpr*.

Pour la notation, les abstractions colorient[4] le constructeur λ ainsi que la variable liée : $\lambda^r x^r.B$, $\lambda^b x^b.B$. Les applications colorient les parenthèses englobantes ainsi que l'opérateur d'application : $(@^r\ F\ A\ )$, $(@^b\ F\ A)$. Les expressions seront coloriées de la couleur de leur racine, E^b=$\lambda^b x^b.(@^r\ x^b\ x^b\ )$, même si cette expression contient des sous-expressions d'une autre couleur. Les raccourcis $(@^r\ F\ A\ B\ )$ et $\lambda^r xy^r.B$ donnent $(@^r\ (@^r\ F\ A\ )\ B\ )$ et $\lambda^r x^r.\lambda^r y^r.B$. Pour les **let**s, les mots clés seront coloriés avec la couleur de l'application du redex et la variable avec la couleur de l'abstraction : $\mathbf{let}^r\ x^b$=A $\mathbf{in}^r\ B$ dénote $(@^r\ \lambda^b x^b.B\ A\ )$.

Nous n'essayerons pas de définir finement l'évaluateur du λ-calcul coloré, ce qui nécessiterait de clarifier la notion d'unité de calcul, d'expliciter les transferts de données, de formaliser la synchronisation entre ces unités de calcul, etc. Ce travail a été fait dans le cadre de Hop avec une sémantique dénotationnelle [10] et une sémantique opérationnelle [2]. Nous allons plutôt nous appuyer sur une sémantique du λ-calcul : l'interprétation d'une expression colorée *ce* sera l'interprétation d'une expression e où *ce* et e sont reliées par une certaine transformation T.

Si $[\![.]\!]$ représente la sémantique des expressions du λ-calcul alors la sémantique des expressions du λ-calcul coloré sera définie par $[\![ce]\!]_c = [\![T(ce)]\!]$.

Une transformation naïve consisterait à *effacer* les couleurs. Ainsi $\lambda^r x^r.x^r$ se transforme en $\lambda x.x$. Malheureusement, des problèmes de conflits de nom apparaissent, $\lambda^r x^r.\lambda^b x^b.x^r$ se transformerait en

[4]Pour que le texte reste lisible avec une impression en noir et blanc, nous mettons également en exposant l'annotation de couleur, r pour rouge et b pour bleu. La version électronique et en couleur de l'article se trouve en `http://hal.inria.fr/JFLA2012/fr/`

$\lambda x.\lambda x.x$ alors que l'intention serait plutôt $\lambda x.\lambda y.x$, car deux variables de même nom mais de couleurs différentes ne sont pas identiques. Pour résoudre ce conflit, nous supposerons l'existence d'une fonction Ψ de $color \times var$ dans var, et l'effacement des couleurs se fera par la fonction $\downarrow$ définie comme suit :

```
Fixpoint  ↓ (ce:cexpr) : expr :=
          match ce with
          | CVar c v ⇒ Var Ψ(c,v)
          | CFun c v b ⇒ Fun Ψ(c,v) ↓ b
          | CApp c f a ⇒ App ↓ f ↓ a
          end.
```

Nous imposerons comme contrainte que Ψ soit injective : $\forall\ c_1,v_1,c_2,v_2,\ \Psi(c_1,v_1)=\Psi(c_2,v_2) \Rightarrow c_1,v_1=c_2,v_2$. Ainsi les conflits de nom disparaissent : $\forall\ c_1,c_2,v,\ c_1 \neq c_2 \Rightarrow \Psi(c_1,v) \neq \Psi(c_2,v)$. On peut imaginer, par exemple, que Ψ effectue la concaténation d'un numéro de couleur, d'un séparateur et du nom de la variable.

2.2. Contextes

La transformation que nous allons formaliser dans la section suivante s'appliquera à une sous-expression A d'une expression principale E. On peut préciser le *chemin* allant de E à A, mais cette notion de *chemin* est trop pauvre pour exprimer facilement le remplacement de A par une autre expression dans E. À la notion de *chemin* nous préférerons celle plus générale de *contexte*. Un contexte [7] est une expression dont une unique sous-expression est un trou.

```
Inductive context : Set :=
|  XHole: context
|  XFun: color → var → context → context
|  XRight: color → context → cexpr → context.
|  XLeft: color → cexpr → context → context.
```

Placer une expression dans un contexte consiste à remplacer le trou du contexte par l'expression.

```
Fixpoint  put (c:context) (e:cexpr) : cexpr :=
          match c with
          | XHole ⇒ e
          | XFun c v b ⇒ CFun c v (put b e)
          | XRight c f a ⇒ CApp c (put f e) a
          | XLeft c f a ⇒ CApp c f (put a e)
          end.
```

Si Δ est un contexte et e une expression, (put Δ e) sera simplement noté Δ(e).

La figure 2 montre graphiquement dans quel contexte se trouve l'application de droite de Ω=(@ $\lambda x.$(@ x x) $\lambda x.$(@ x x)). Ce contexte est Δ=(XLeft c $\lambda x.$(@ x x) (XFun c x (XHole)))

Un contexte Δ permet donc de repérer une sous-expression A de E, $E = \Delta(A)$, mais permet aussi d'opérer des substitutions. Le remplacement de A par B dans E s'exprime par $\Delta(B)$.

Un arc correspond à un contexte de longueur 1 et se définit par :

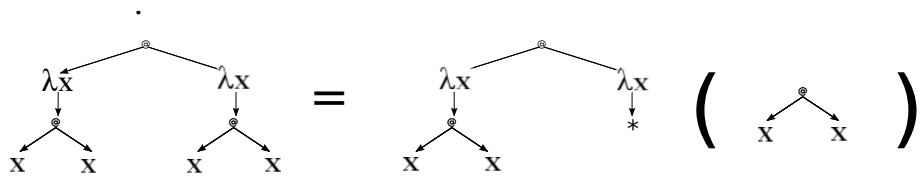

Figure 2: $\Omega = \Delta\ (@\ x\ x)$

Inductive *edge* : **Set** :=
| GFun: *color* → *var* → *edge*
| GRight: *color* → *cexpr* → *edge*.
| GLeft: *color* → *cexpr* → *edge*.

Rajouter une expression à un arc se définit par :

Fixpoint link (g:edge) (e:*cexpr*) : *cexpr* :=
match g **with**
| GFun c v⇒ CFun c v e
| GRight c a ⇒ CApp c e a
| GLeft c f ⇒ CApp c f e
end.

Si δ est un arc et e une expression, (link δ e) sera simplement noté δ(e). Comme les arcs et les contextes peuvent se comporter comme des fonctions internes sur les expressions colorées, nous utiliserons la notation $\bullet$ pour la composition, ainsi $\delta \bullet \Delta$ correspond à la fonction $x \mapsto$ (link δ (put Δ x)). Les arcs seront notés avec leur couleur : δ^r.

2.3. Transformation

Dans cette section nous allons définir une transformation permettant de regrouper les expressions d'une même couleur. L'idée générale est de remplacer une expression par une variable et de lier cette variable et cette expression plus haut dans l'expression. Prenons par exemple une application rouge où l'expression correspondant à la fonction est en bleu : $(@^r\ F^b\ A\ )$. On peut prendre une variable rouge x^r et la mettre à la place de F^b : $(@^r\ x^r\ A\ )$. Puis on lie x^r à F^b en introduisant un redex : $\mathbf{let}^b\ x^r{=}F^b\ \mathbf{in}^b\ (@^r\ x^r\ A\ )$, soit, $(@^b\ \lambda^r x^r.(@^r\ x^r\ A\ )\ F^b)$. Cet exemple n'est pas concluant car la *remontée* de F^b n'a pas rejoint une autre expression bleue. Supposons que l'expression initiale soit $(@^b\ (@^r\ F^b\ A\ )\ B^b)$, les deux expressions F^b et B^b sont séparées par l'application rouge, mais en appliquant la précédente transformation on construit l'expression $(@^b\ (@^b\ \lambda^r x^r.(@^r\ x^r\ A\ )\ F^b)\ B^b)$ et F^b a rejoint B^b sous une application bleue.

Pour que cette transformation soit valide, la seule précaution à prendre est que x^r ne doit pas une variable libre de A. Un second cas est à envisager lorsque F^b *remonte* une abstraction $(@^b\ \lambda^r y^r.F^b\ B^b)$ et se transforme en : $(@^b\ (@^b\ \lambda^r x^r.\lambda^r y^r.x^r\ F^b)\ B^b)$, ici il suffit que les deux variables soient différentes et il est impératif que la variable y^r ne soit pas utilisée par F^b, i.e. y^r n'est pas libre dans F^b. Cette condition est la plus générale mais est dépendante du contexte, on lui préfèrera une condition plus dure stipulant que si l'expression bleue F^b veut *remonter* au dessus d'une expression rouge alors F^b ne doit pas avoir de sous-expression rouge (caci impliquant que y^r n'est pas libre dans F^b). Cette nouvelle condition n'est pas restrictive car si F^b est juste sous une expression rouge et contient une sous-expression rouge, il est possible de rassembler les expressions rouges avant de s'occuper de celles en bleue. Cette restriction permet d'imposer l'enchaînement des transformations des feuilles vers la racine (i.e. *bottom-up*) mais n'empêchera pas, *in fine*, une transformation de se faire.

Donc, pour que la transformation soit possible il faut avoir une expression E^b d'une couleur donnée ayant une sous-expression A^b de la même couleur ($E^b{=}\Delta(A^b)$) et que toutes les expressions sur le chemin, non nul, de E^b vers A^b soient de la couleur opposée. Il faut nécessairement avoir un arc sortant de E^b, vers une expression de la couleur opposée, et, à la fin du chemin, une expression de cette couleur opposée avec un arc vers A^b. E^b peut donc s'écrire ${\delta_1}^b({\Delta_2}^r({\delta_2}^r(A^b)))$ ou $({\delta_1}^b{\bullet}{\Delta_2}^r{\bullet}{\delta_2}^r)(A^b)$. Tous les arcs composant ${\Delta_2}^r$ doivent être de la même couleur.

Sous toutes ces conditions, et si v^r est une variable n'apparaissant pas dans E^b, alors $E^b{=}{\delta_1}^b({\Delta_2}^r({\delta_2}^r(A^b)))$ peut se réécrire en : $R^b{=}{\delta_1}^b(@^b\ \lambda^r v^r.{\Delta_2}^r({\delta_2}^r(v^r))\ A^b)$, soit ${\delta_1}^b(\mathbf{let}^b\ v^r{=}A^b\ \mathbf{in}^b\ \Delta^r({\delta_2}^r(v^r)))$. On notera cette transformation par $E^b \nearrow R^b$. En résumé :

$$\frac{\begin{array}{c}\text{Monochrome}(\Delta^r)\\ \wedge\ \text{Monochrome}(A^b)\\ \wedge\ v^r \text{ une variable fraîche}\end{array}}{{\delta_1}^b({\Delta_2}^r({\delta_2}^r(A^b))) \nearrow {\delta_1}^b(\mathbf{let}^b\ v^r{=}A^b\ \mathbf{in}^b\ \Delta^r({\delta_2}^r(v^r)))}$$

La définition Coq de cette relation nécessite d'expliciter toutes les couleurs implicites de la règle présentée ci-dessus, mais ne demande pas d'effort particulier; les définitions de *Monochrome* sur les expressions et les contextes sont immédiates.

Bien sûr, nous autoriserons la transformation à se faire sur n'importe quelle sous-expression d'une expression principale, ou plus formellement : $\forall E_1 E_2,\ E_1 \nearrow E_2 \Rightarrow \forall \Delta_p, \Delta_p(E_1) \nearrow \Delta_p(E_2)$

Graphiquement, la transformation générale se résume dans la figure 3 ci-dessous :

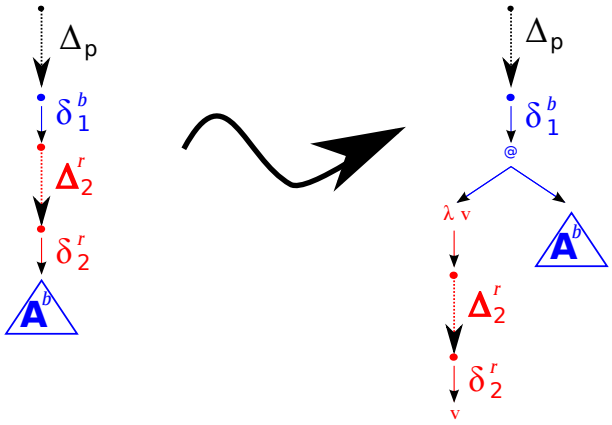

Figure 3: Remontée de A^b dans le contexte ${\Delta_2}^r{\bullet}{\delta_2}^r$

3. Propriétés

3.1. Correction de la transformation

La transformation est l'inverse de la β-réduction (une β-expansion) :

Théorème 1 (Correction) $E_1 \nearrow E_2 \Rightarrow\downarrow E_2 \rightarrow_\beta\downarrow E_1$.

Preuve E_1 peut s'écrire sous la forme $(\Delta_p{\bullet}{\delta_1}^b{\bullet}{\Delta_2}^r{\bullet}{\delta_2}^r)(A^b)$ et E_2 sous la forme $(\Delta_1{\bullet}{\delta_1}^b)(\mathbf{let}^b\ v^r{=}A^b\ \mathbf{in}^b\ (\Delta^r{\bullet}{\delta_2}^r)(v^r))$. La preuve se fait par induction sur Δ_p, puis par étude de cas sur ${\delta_1}^b$ pour faire

apparaître clairement le redex, et enfin par induction sur ${\Delta_2}^r \bullet {\delta_2}^r$ pour assurer que la substitution induite par la β-réduction se déroule correctement.

Par contre l'expression A^b peut remonter au-dessus d'abstractions rouges apparaissant dans le chemin ${\Delta_2}^r \bullet {\delta_2}^r$ et présuppose une stratégie de réduction forte. Ainsi la transformation ne conserve pas la terminaison pour une stratégie avec appel par valeur. Néanmoins, cette terminaison est respectée pour une stratégie de réduction par nécessité. Si cette restriction pour l'appel par valeur est trop forte, il est toujours possible de transformer A^b en $(@^r\ \lambda^b\_^b.A^b\ \_^r\ )$ et donc d'enfermer A^b dans un glaçon de la même couleur. Ce glaçon sera dégelé par une application de la couleur opposée.

3.2. Confluence

Il est important de savoir si un compilateur doit enchaîner les passes de transformation dans un ordre précis. La confluence permet de conclure que l'ordre n'a pas d'importance. Considérons une expression E ayant deux transformations différentes possibles sur A_1 et A_2 : $E = \Delta_1(A_1) = \Delta_2(A_2)$. La confluence permet de montrer qu'à partir des transformations faites sur A_1 d'une part, et sur A_2 d'autre part, il est possible d'appliquer des transformations sur ces deux expressions pour retrouver une expression commune. Pour prouver la confluence, il convient de situer les deux expressions A_1 et A_2 l'une par rapport à l'autre. On utilisera donc un premier lemme de séparation des cas :

Lemme 1 (Comparaison de deux contextes) [5]

$$
\begin{aligned}
&\forall \Delta_1, \Delta_2, A_1, A_2,\\
&\quad \Delta_1(A_1) = \Delta_2(A_2) \Rightarrow\\
&\qquad \Delta_1 = \Delta_2 \ \wedge\ A_1 = A_2\\
&\quad \vee\ \exists \delta, \Delta, \Delta_2 = \Delta_1 \bullet \delta \bullet \Delta\\
&\quad \vee\ \exists c, \Delta_x, \Delta_l, \Delta_r, (\Delta_1 = \Delta_x \bullet (GRight(c, \Delta_r(A_2))) \bullet \Delta_l\\
&\qquad\qquad \wedge\ \Delta_2 = \Delta_x \bullet (GLeft(c, \Delta_l(A_1))) \bullet \Delta_r)\\
&\quad \vee\ \exists \delta, \Delta, \Delta_1 = \Delta_2 \bullet \delta \bullet \Delta\\
&\quad \vee\ \exists c, \Delta_x, \Delta_l, \Delta_r, (\Delta_1 = \Delta_x \bullet (GLeft(c, \Delta_r(A_2))) \bullet \Delta_l\\
&\qquad\qquad \wedge\ \Delta_2 = \Delta_x \bullet (GRight(c, \Delta_l(A_1))) \bullet \Delta_r
\end{aligned}
$$

Preuve par induction sur Δ_1 et Δ_2.

Ce lemme permet de discriminer les deux chemins menant à A_1 et A_2. Les cas principaux sont montrés dans la figure 4. Les deux derniers cas sont symétriques des second et troisième. Selon la définition de la transformation, le contexte Δ_1 (resp. Δ_2) se décompose en $\Delta_p^1 \bullet \delta_1^1 \bullet \Delta_2^1 \bullet \delta_2^1$ (resp. $\Delta_p^2 \bullet \delta_1^2 \bullet \Delta_2^2 \bullet \delta_2^2$). Le premier cas, l'égalité, implique que les deux chemins se décomposent de la même manière en $\Delta_p \bullet \delta_1 \bullet \Delta_2 \bullet \delta_2$, donc les deux transformations sont identiques. Le deuxième cas où A_2 est une sous-expression de A_1 amène à une contradiction car A_1 et A_2 doivent être de même couleur par monochromie de A_1 et le chemin amenant à A_2 doit être d'une couleur opposée.

Pour le troisième cas[6], il faut énumérer tous les sous-cas où se trouvent les arcs δ_1^1 et δ_1^2, i.e. dans quels segments vont s'insérer les nouveaux redex. Pour ce faire nous utiliserons un second lemme de comparaison d'un contexte $\Delta_p \bullet \delta \bullet \Delta_s$ avec lui-même réécrit sous la forme $\Delta_x \bullet \Delta_d$. Le lemme précise que l'arc δ appartient soit à Δ_x soit à Δ_d.

Pour la confluence, ce lemme est appliqué sur Δ_1 et Δ_2 et le seul cas intéressant est quand les deux arcs δ_1^1 et δ_1^2 se trouvent dans la partie commune Δ_x (sinon les transformations sont indépendantes et commutent sans problème). On peut facilement déduire que le noeud d'application faisant la séparation est de la couleur opposée à A_1 et A_2, qui donc sont de la même couleur, et par voie de

[5] Lemme nommé `context_five_cases` en Coq

[6] $GRight$ et $GLeft$ sont les deux arcs possibles pouvant sortir d'un noeud d'application, voir la définition du type *edge*.

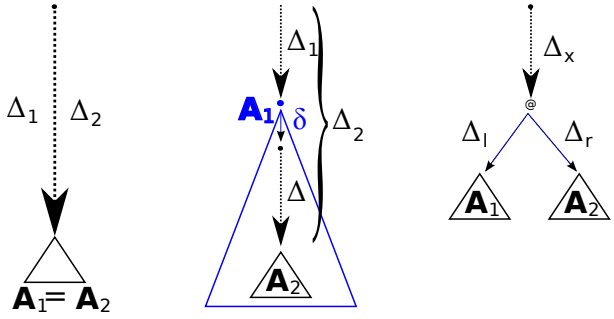

Figure 4: principaux cas de comparaison de deux contextes ayant même racine

conséquence que δ_1^1 et δ_1^2 sont identiques. Dans ce cas, la figure 5 montre la différence entre appliquer la transformation sur A_1 en premier ou A_2 en premier.

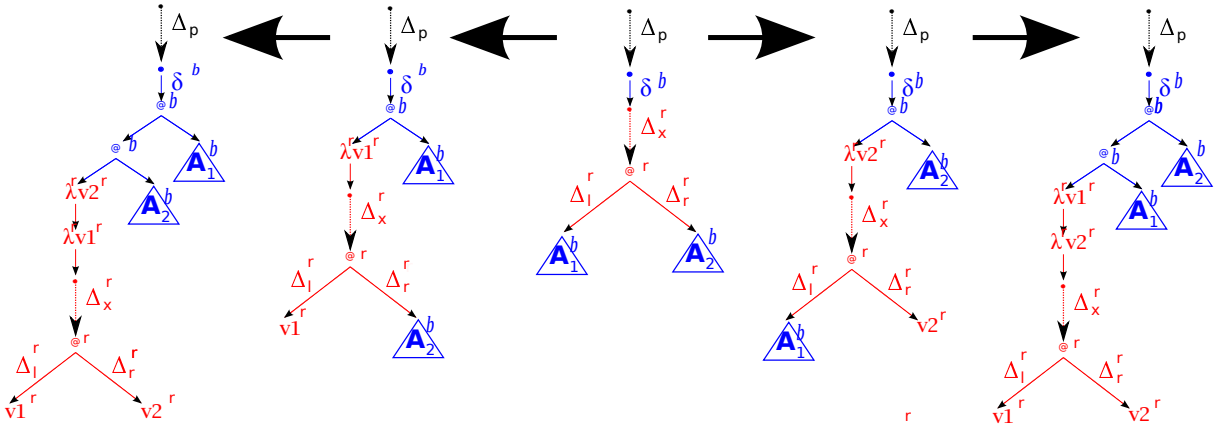

Figure 5: cas critique pour la confluence

Si l'on considère le raccourci où **let** $(v_1 = A_1)$ $(v_2 = A_2)$ **in** B se réécrit en $(@(@(\lambda v_1\ v_2.B)A_1)A_2)$, alors l'expression de gauche s'écrit Δ(**let** $(v_1 = A_1)$ $(v_2 = A_2)$ **in** B), tandis que celle de droite s'écrit Δ(**let** $(v_2 = A_2)$ $(v_1 = A_1)$ **in** B). Ainsi la relation $\nearrow$ est confluente modulo permutation dans la liaison des **let**s. Tout ceci nous permet d'annoncer le théorème de confluence où la relation d'équivalence $\equiv$ permet soit une permutation dans la liaison des lets (cf figure 5), soit un simple renommage d'une des variables v_1 et v_2 introduites par la transformation.

Théorème 2 (Confluence) $E \nearrow E_1 \ \wedge \ E \nearrow E_2 \Rightarrow$
$E_1 \equiv E_2$
$\vee\ \exists R_1, R_2,\ E_1 \nearrow R_1 \ \wedge \ E_2 \nearrow R_2 \ \wedge R_1 \equiv R_2.$

Preuve On utilise en premier le lemme de comparaison des deux contextes où se trouvent E_1 et E_2. L'égalité des contextes apporte le cas où $E_1 \equiv E_2$, l'inclusion amène à un contradiction. Dans le cas qui reste on applique le lemme de comparaison d'un contexte avec lui-même sur E_1 et E_2. Pour trois cas sur quatre on trouve R_1 et R_2 équivalents à un renommage près. Pour le dernier cas, exprimé dans la figure 5, les expressions R_1 et R_2 sont équivalents à une permutation dans la liaison d'un let. 🐓

3.3. Terminaison

Un compilateur va enchaîner les transformations tant que cela est possible. La terminaison de cet enchaînement de transformations est facilement prouvable. On peut compter le nombre d'arcs reliant deux expressions de couleurs différentes. Pour toute expression E, on considère le cardinal de l'ensemble : $Border(E) = \{\delta, E = (\Delta \bullet \delta)(A), color(\delta) \neq color(A)\}$. On remarque sur la figure 3 que la transformation diminue ce nombre exactement de 1 :

Lemme 2 (Fonction décroissante) $E_1 \nearrow E_2 \Rightarrow \mathbf{card}(Border(E_1)) = \mathbf{card}(Border(E_2)) + 1$.

Preuve

En effet, avant la transformation, les arcs δ_1 et δ_2 sont dans $Border(E_1)$. Après la transformation, les arcs δ_1 et δ_2 sont enlevés de l'ensemble mais l'arc reliant l'application à la fonction du nouveau redex est ajouté. Tous les autres arcs de E_1 ont un statut inchangé, soit ils étaient dans $Border(E_1)$ et ils restent dans $Border(E_2)$, soit ils n'y étaient pas et ne seront pas introduits dans $Border(E_2)$. 🐓

Ainsi, si une expression E est telle que $\mathbf{card}(Border(E)) = n$, alors il faudra au plus n étapes de transformation pour atteindre une forme normale.

4. Application du modèle aux langages Web

Cette colorisation s'applique immédiatement à des applications faisant intervenir deux types d'unités de calcul. On présente ici deux exemples de services Web. Le premier, en Hop et Ocsigen, affiche un texte sur le serveur. Le second, en Links, pour un service d'écho.

Dans les deux systèmes multi-tiers pour le Web, Hop et Ocsigen, on écrit l'application dans un seul langage fonctionnel, à base de Scheme pour Hop et à base d'OCaml pour Ocsigen, et où apparaissent des échappements vers le serveur ou vers le client (respectivement $ et ~ pour Hop, et {{, }} et % pour Ocsigen). Ce programme construit, à partir du serveur, une page Web qui sera affichée sur le client et où les échappements sont explicites. On peut alors colorier ce type de programmes en deux couleurs comme le montre la figure 6.

	Hop	Ocsigen

code commun	`(define i 0)`
	`(define-service (f) (begin`
	`(print "SERVEUR") (incr i))`
		`let i = ref 0;;`
		`let f () = begin`
		`print_string "SERVEUR";`
		`incr i; !i end;;`

application sur le serveur à la construction de la page	`(define-service (main) (`
	`<HTML> ..."CLICK" :onclick`
	`~(alert $(f) ) ))`
		`let page () = ... html ...`
		`a ~onclick = {{ alert %(f())`
		`}} [ cdata "Click" ] ;;`

application sur le serveur à chaque clic	`(define-service (main) (`
	`<HTML> ..."CLICK" :onclick`
	`~(alert (with-hop $f ) ) ))`
		`let page () = ... html ...`
		`a ~onclick = {{ alert (call_service`
		`%f () ) }} [ cdata "Click" ] ;;`

Figure 6: colorisation de pseudo-code Hop et Ocsigen

L'application (f) ou f() de la deuxième ligne de la figure 6 s'effectue sur le serveur à la construction de la page envoyée sur le client Web, et donc ne sera calculée qu'une seule fois. A l'inverse à la troisième ligne, le `with-hop` ou `call_service`, dont la localisation n'est pas précisée, appelle le service `f` sur le serveur à chaque clic sur le composant graphique de la page Web. La variable globale `i` est donc

incrémentée à chaque appel de `f`.

Dans le système Links [4][5], on peut, en spécialisant les fonctions du côté serveur ou client dès leur définition, avoir un code source coloré. L'exemple de la figure 7 montre un serveur d'écho.

```
fun cprint(texte) client { appendChild (monChat, texte) }

fun echo(texte) server { cprint( texte ) }

page <#> {
<Html> ... <a onclick= echo( "bob" ) > ... </a> </Html>
} <\#>
```

Figure 7: colorisation de pseudo-code Links

La page construite par le serveur avec la fonction page réagit à un clic du client en appliquant la fonction echo du serveur sur la chaîne "bob"; le résultat est alors l'appel de cprint du côté client qui ajoutera au document le texte transmis par la fonction cprint vers le client. On obtient ainsi l'embryon d'un service d'écho ou de chat.

En Links l'indication de l'unité de calcul où la fonction s'appliquera permet de construire des applications symétriques qui n'apparaissaient pas naturellement en Hop ou Ocsigen.

5. Conclusion

Nous avons montré comment une β-expansion pouvait séparer les couleurs dans un λ-calcul bichrome. Cette transformation est-elle adaptable à un λ-calcul polychrome? La figure 8 pourrait être une transformation pour un exemple en 3 couleurs.

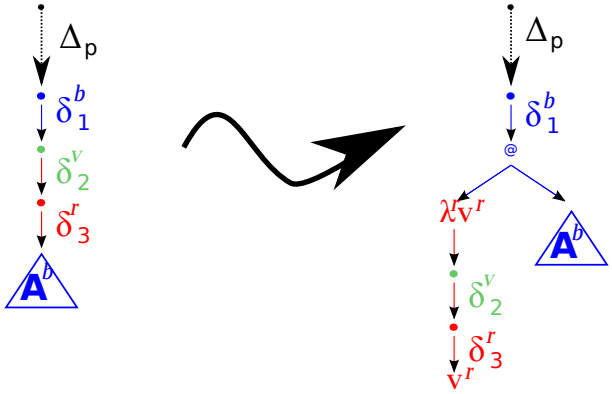

Figure 8: Transformation pour 3 couleurs

On voit qu'une expression bleue pourrait à la fois remonter des expressions rouges ou vertes et ainsi se rapprocher d'une autre expression bleue. Cette expression que l'on *remonte* se trouvant dans une expression rouge, il serait naturel d'introduire un redex dont la variable serait de cette couleur.

Par contre, du fait d'une tierce couleur, l'abstraction liant la variable rouge peut être séparée de l'expression englobant l'expression bleue remontée. Donc nous avons rapproché deux expressions bleues, mais, dans le même temps, nous avons créé une expression rouge isolée des autres. La bonne nouvelle, pour la convergence, est qu'à l'origine les deux expressions bleues étaient séparées par deux couleurs, tandis qu'après la transformation, les deux nouveaux îlots rouges ne sont séparés que par une seule couleur. On peut donc réitérer la transformation pour éliminer définitivement ces deux îlots rouges. On remarque ici que le critère de terminaison, utilisé pour la version bichrome, n'est plus applicable.

De plus, comme le montre la figure 9, la confluence n'est plus aussi directe.

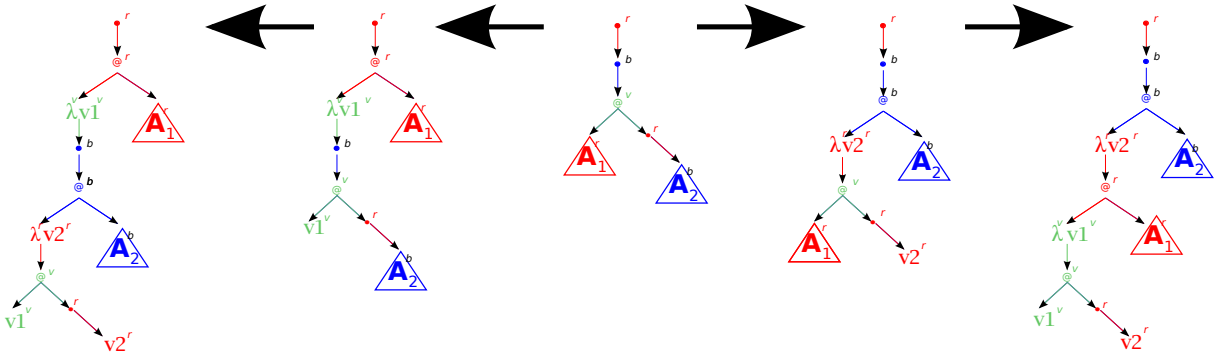

Figure 9: exemple de non confluence directe en 3 couleurs

Lorsque deux expressions sont susceptibles de remonter au-dessus d'un prédécesseur commun, ce prédécesseur peut prendre la tierce couleur, les deux expressions n'ont donc plus forcément la même couleur. Dans l'exemple, l'expression à gauche est rouge, celle de droite est bleue. Ainsi il est possible de mettre un arc rouge entre le prédécesseur commun et l'expression bleue. Si l'on transforme l'expression rouge en premier, réduction vers la gauche dans la figure 9, elle rejoint le plus haut niveau en une seule étape. Par contre, si l'on transforme l'expression bleue en premier, réduction vers la droite, la transformation va introduire une abstraction rouge, due à la présence de l'arc rouge aboutissant à l'expression $A_2{}^b$. Cette abstraction va empêcher l'expression rouge de remonter à son plus haut niveau. Il faudra donc considérer une confluence où un enchaînement de plusieurs transformations est nécessaire avant de retrouver deux expressions équivalentes.

Remerciements. Les auteurs remercient Ilaria Castellani, Benjamin Canou, Pascal Manoury et Bruno Pagano pour leur relecture attentive ainsi que les rapporteurs de l'article pour leurs remarques constructives.

Bibliographie

[1] Vincent Balat, Jérôme Vouillon, et Boris Yakobowski. Experience report: ocsigen, a web programming framework. In Graham Hutton et Andrew P. Tolmach, editeurs, *ICFP*, pp. 311–316. ACM, 2009.

[2] Gérard Boudol, Zhengqin Luo, Tamara Rezk, et Manuel Serrano. Towards reasoning for web applications: an operational semantics for hop. In *Proceedings of the 2010 Workshop on Analysis*

and Programming Languages for Web Applications and Cloud Applications, APLWACA '10, pp. 3–14, New York, NY, USA, 2010. ACM.

[3] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1985. (première édition en 1941).

[4] Ezra Cooper, Sam Lindley, Philip Wadler, et Jeremy Yallop. Links: Web programming without tiers. In *FMCO*, pp. 266–296, 2006.

[5] Ezra Cooper et Philip Wadler. The RPC calculus. In *Principles and Practice of Declarative Programming (PPDP)*, Coimbra, Portugal, 2009.

[6] Dieter Hutter et Michael Kohlhase. A colored version of the lambda-calculus. In *Proceedings of the 14th International Conference on Automated Deduction*, CADE-14, pp. 291–305, London, UK, 1997. Springer-Verlag.

[7] James H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[8] Martin Odersky, Christoph Zenger, et Matthias Zenger. Colored local type inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pp. 41–53, New York, NY, USA, 2001. ACM.

[9] Gilles Peskine. *Types abstraits dans les systèmes répartis*. PhD thesis, Université Paris Diderot (Paris 7), 2008.

[10] Manuel Serrano et Christian Queinnec. A multi-tier semantics for hop. *Higher-Order and Symbolic Computation*, pp. 1–23, 2010. 10.1007/s10990-010-9061-9.

[11] David Walker, Lester Mackey, Jay Ligatti, George A. Reis, et David I. August. Static typing for a faulty lambda calculus. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pp. 38–49, New York, NY, USA, 2006. ACM.

[12] D. Fober Y. Orlarey et D. Letz. Lambda calculus and music calculi. In *Proceedings of the International Computer Music Conference ICMA*, pp. 243–250, 1994.