Unification des couleurs dans un lambda-calcul polychrome

Bernard Serpette, Pascal Manoury, Emmanuel Chailloux

Unification des couleurs dans un λ-calcul polychrome

Bernard Paul Serpette [1] & Pascal Manoury[2] & Emmanuel Chailloux[3]

1: Projet INDES,
Inria Sophia-Antipolis Méditerranée,
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
Bernard.Serpette@inria.fr

2: Laboratoire Preuves, Programmes et Systèmes (PPS - UMR 7126)
Université Paris Diderot (Paris 7)
Sorbonnes Paris Cité
Bâtiment Sophie-Germain
F-75205 PARIS Cedex 13
Pascal.Manoury@pps.univ-paris-diderot.fr

3: Laboratoire d'informatique de Paris 6 (LIP6 - UMR 7606),
Université Pierre et Marie Curie (Paris 6)
Sorbonnes Université
4, Place Jussieu, 75005 Paris, France
Emmanuel.Chailloux@lip6.fr

Résumé

Dans cet article nous étendons le λ-calcul bi-chrome présenté aux JFLA 2012 pour y introduire la polychromie[1]. On définit une nouvelle transformation, par β-expansion, qui regroupe les expressions de même couleur, chaque couleur pouvant représenter une unité de calcul. On ne se contente plus de pouvoir expliciter la localité d'un calcul dans un modèle à deux couleurs comme pour les clients-serveurs mais nous pouvons traiter les applications multi-tiers. Les propriétés de correction, de terminaison et de confluence de cette nouvelle transformation sont démontrées à l'aide de Coq.

1. Introduction

Nous avons présenté dans l'article " Séparation des couleurs dans un λ-calcul bichrome "[3] un λ-calcul à deux couleurs permettant d'expliciter une partie de l'évaluation d'un terme en précisant (via la couleur) la localité du calcul. Nous avons pu définir une transformation par β-expansion qui regroupe les expressions de même couleur. Un domaine d'application immédiat a été les clients-serveurs d'application Web, tout particulièrement les environnements de développement Hop [2] et OCsigen [1] qui permettent d'écrire dans un même programme la partie serveur et la partie client. L'une des couleurs représente le code serveur et l'autre le code client (code JavaScript exécuté par le navigateur). La transformation de programme que nous avions proposée permet de regrouper les ilôts de même couleur. La racine de l'arbre d'exécution étant dévolue aux serveurs, la transformation de programme permet d'extraire un code monolithique pour le serveur et, pour le client, une série d'expressions

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

indépendantes entre-elles. Les propriétés de correction, de terminaison et de confluence de cette transformation ont été 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.

En fait, les préliminaires de ce travail sur le λ-calcul coloré considéraient un nombre quelconque de couleurs, c'est en observant que la preuve de confluence ne passait pas que nous nous sommes restreints à deux couleurs. La conclusion de [3] montre un exemple avec trois couleurs où la confluence n'était pas établie. La restriction à deux couleurs n'empêche pas de rester fortement lié à Hop et Ocsigen avec le couple client/serveur et donc avec deux sites différents de calcul. Mais on peut aussi attribuer les couleurs à des langages spécifiques, la partie commune de ces langages contenant le minimum pour abstraire les communications. Ici nous choisissons le λ-calcul comme dénominateur commun. Cette vision, où l'on attribue un langage à une couleur, est aussi valide pour Hop et Ocsigen car la partie cliente doit être convertie en JavaScript. Par extension, d'autres langages peuvent intervenir, le cas le plus étudié est la présence d'un langage dédié à l'accès aux bases de données [4], mais on peut aussi envisager des langages utilisant les capacités des accélérateurs graphiques (GPGPU à la OpenCL), des langages dédiés à la musique (comme OpenMusic de l'Ircam)...

Pour permettre la coexistence de plusieurs langages métiers, nous proposons une nouvelle transformation permettant le regroupement polychrome d'expressions. Les définitions sont très proches de la version bichrome aux couleurs près. Ces définitions sont toujours formalisées en Coq[3]. La nouvelle transformation repose toujours sur une β-expansion guidée par le rapprochement chromatique qui nécessite maintenant de tenir compte d'un nombre quelconque de couleurs. Les preuves des propriétés de correction et confluence deviennent plus complexes et sont donc complètement à revoir.

L'étape élémentaire de la transformation va regrouper ensemble deux expressions de même couleur. A ce niveau la propriété la plus importante est la correction : le programme d'origine et le programme transformé doivent avoir le même comportement. Un compilateur répétera cette étape élémentaire jusqu'à trouver une forme normale. Ici, la propriété la plus importante est la confluence : le compilateur va choisir un parcours déterministe de l'arbre pour faire les transformations, la confluence permet d'assurer que le choix du parcours n'a pas d'influence sur le résultat final. Il faut aussi prouver que le compilateur termine. Ce sont les trois théorèmes que nous prouverons dans cet article. Nous mentionnerons en conclusion comment obtenir d'autres propriétés plus générales. La première de ces propriétés est l'optimimalité et correspond au nombre de couleurs dans le résultat final. Cette propriété est obtenue par un artefact sur la couleur de la racine de l'expression principale et dont la preuve semble sans difficulté. La seconde propriété concerne la conservation du critère de terminaison qui s'obtient en considérant une β-expansion compatible avec une stratégie de réduction par valeur. Là encore, la preuve semble sans difficulté.

Cet article est découpé en trois parties. La section 2 reprend les notations du λ-calcul bichrome pour les étendre au calcul polychrome afin de définir la transformation de regroupement des couleurs. La section 3 montre les propriétés de correction, de confluence et de terminaison de cette nouvelle transformation. La conclusion discute de la qualité du regroupement des couleurs et revient sur la nécessité d'employer une β-réduction non-déterministe.

2. Définitions

Nous allons reprendre dans cette section les définitions introduites dans la version bicolore de la transformation [3]. La seule différence notoire est le domaine des couleurs qui passe d'un ensemble à deux éléments aux entiers naturels.

[2] Les anciennes sources sont disponibles sur ftp://ftp-sop.inria.fr/indes/rp/jfla2012.v
[3] Les nouvelles sources sont disponibles sur ftp://ftp-sop.inria.fr/indes/rp/jfla2014.v

2.1. λ-calcul polychrome

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 :

Définition 1 (`expr`)

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, le processeur en charge de son évaluation. Pour rester abstrait, les unités de calcul seront représentées par des couleurs. Toutes les expressions possèdent donc une annotation de couleur :

Définition 2 (`color` **et** `cexpr`)

Definition *color*:= **nat.**

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 leurs racines, E^b=$\lambda^b x^b$.(@^{r} x^b x^b), même si cette expression contient des sous-expressions d'une autre couleur. 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 : **let**^{r} x^b=A **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 [7] et une sémantique opérationnelle [2]. Nous allons plutôt nous appuyer sur une sémantique du λ-calcul : l'interprétation d'une expression colorée E^c sera l'interprétation d'une expression E où E^c et E sont reliées par une transformation T.

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

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

[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, v pour violet...

Définition 3 (`clean`)

```
Fixpoint  ↓ (E^c:cexpr) : expr :=
          match E^c 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. Pour déterminer la position de A dans E, ainsi que son éventuel remplacement, nous utiliserons la notion de *contexte* [6]. Comme pour la notion de *chemin* dans un arbre, les contextes sont exprimés comme une liste d'arcs reliant deux sous-expressions. Les arcs des contextes sont plus riches car ils gardent en mémoire, dans leurs structures, les expressions voisines de celle qui est pointée par l'arc:

Définition 4 (`edge`)

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

Par exemple un arc de type *GLeft* est un arc partant d'un noeud d'application vers la fonction mais se souvenant de l'expression argument. Ainsi, à partir d'un arc et d'une expression, il est possible de reconstruire l'expression d'origine :

Définition 5 (`link`)

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

Finalement, un contexte se définit de la même manière qu'une liste d'arcs. On notera $*$ le contexte vide et, si δ est un arc et Δ un contexte, on note $\delta \bullet \Delta$ le contexte dont le premier arc est δ et la suite Δ, voire, plus simplement $\delta\Delta$. On formalise cette définition comme le type inductif:

Définition 6 (`context`)

```
Inductive context : Set :=
|  XHole : context
|  Edge : edge→ context → context.
```

Placer une expression dans un contexte consiste à remplacer le trou du contexte par l'expression reçue en argument. Cette opération se définit par induction simple sur la liste des arcs:

Définition 7 (put)

Fixpoint *put (c:context) (e:cexpr) : cexpr :=*
match *c* **with**
| *XHole* ⇒ *e*
| *Edge g h* ⇒ *(link g (put h e)*
end.

Si Δ est un contexte et e une expression, (put Δ e) sera simplement noté Δ(e). De la même manière, si δ est un arc (link δ e) sera noté δ(e).

La figure 1 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=(Edge (GRight c $\lambda x.$(@ x x)) (Edge (GFun c x) (XHole)))

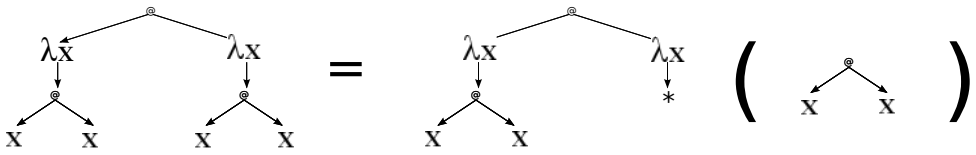

Figure 1: Ω = Δ (@ x x)

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 $\Delta(B)$.

Les arcs seront notés avec leur couleur : δ^r. De la même manière, les contextes non vides sont notés avec la couleur de leurs premier arc : Δ^r.

2.3. Transformation

Le but de la transformation est de réduire la fragmentation des couleurs d'une expression polychrome en conservant la dénotation de l'expression. Le moyen de cette transformation est simplement une β-expansion guidée par le rapprochement chromatique. Par exemple, dans l'expression,

$$(@^b\ (@^r\ F^b\ A\ )\ B^b),$$

les deux expressions F^b et B^b sont séparées par l'application rouge. En sortant F^b de l'application rouge $(@^r\ \ )$ par création d'un redex, on obtient

$$(@^b\ (@^b\ \lambda^r x^r.(@^r\ x^r\ A\ )\ F^b)\ B^b),$$

c'est-à-dire, $(@^b\ \lambda^r x^r.(@^r\ x^r\ A\ )\ F^b\ B^b)$, où F^b a rejoint B^b sous une application bleue.

Dans l'exemple ci-dessus, F^b a rejoint B^b en remontant une application. On veut pouvoir également remonter les abstractions. Ainsi, partant d'une expression de la forme

$$(@^b\ \lambda^r y^r.F^b\ B^b)$$

pour que F^b et B^b se rejoignent, on appliquera la transformation de cette expression vers

$$(@^b\ \lambda^r x^r.y^r.x^r\ F^b\ B^b)$$

Pour que ces transformations soient valides, il faut choisir convenablement la variable introduite par le redex: x^r ne doit pas être une variable libre de A, dans le premier cas; x^r doit être différente de y^r et y^r ne doit pas être libre dans F^b, dans le second.

Ces conditions seront réalisées si l'on pose comme condition plus générale que pour remonter une expression bleue F^b au dessus d'une application ou d'une abstraction rouge, il faut que F^b ne contienne *aucune sous-expression rouge*. Quoique plus restrictive, cette contrainte ne bloquera pas le processus de réunion des couleurs. En effet, si F^b contenait une sous-expression rouge, il suffirait de remonter d'abord celle-ci hors de F^b pour pouvoir ensuite faire remonter F^b elle-même hors de son contexte rouge. Par exemple:

$$(@^b\ (@^r\ (@^b\ {F_1}^r\ {F_2}^b)\ A\ )\ B^b)$$

devient

$$(@^b\ (@^r\ \lambda^b x^b.(@^b\ x^b\ {F_2}^b)\ {F_1}^r\ A\ )\ B^b)$$

qui devient à son tour

$$(@^b\ \lambda^r x^r.(@^r\ x^r\ {F_1}^r\ A\ )\ \lambda^b x^b.(@^b\ x^b\ {F_2}^b)\ B^b\ )$$

Si elle n'est pas bloquante, la contrainte de monochromie de l'expression à remonter imposera un certain ordonnancement des applications des transformations, des feuilles vers la racine.

Les chemins de la transformation Donc, pour que la transformation soit possible, il faut avoir une expression E^b d'une couleur donnée ayant une sous-expression A^b, monochrome, de la même couleur. C'est-à-dire que $E^b{=}\Delta(A^b)$. Pour être plus précis, puisque E est colorée en bleu, Δ doit aussi l'être. C'est-à-dire que la couleur du premier arc du chemin qui mène à A^b doit aussi être le bleu: on a donc $E^b{=}\delta^b{\bullet}\Delta'(A^b)$. Ceci suppose que $\Delta{=}\delta^b{\bullet}\Delta'$ *est non vide*. Nous exigerons de surcroît que cette condition soit *maximale*. C'est-à-dire que Δ' ne contienne pas d'arc de couleur bleue, toutefois, Δ' peut être polychrome.

Dans la version bichrome, nous pouvions nous contenter d'une simple alternance de couleurs: dans $\Delta^r(F^b)$, pour remonter l'expression bleue F^b du contexte rouge Δ^r, on créait l'application bleue d'une abstraction rouge: $(@^b\ \lambda^r x^r.\Delta^r(x^r)\ F^b)$. C'est-à-dire, $\mathbf{let}^b\ x^r{=}\mathrm{F}^b\ \mathbf{in}^b\ \Delta^r(x^r)$. Cette solution simple n'est plus valable en présence de chemins polychromes. Si le chemin Δ' est polychrome, il y a de fortes chances que le début et la fin de ce chemin soient de couleurs différentes. Par exemple: $\delta^b{\bullet}\delta^v{\bullet}\delta^r(A^b)$. Pour procéder à la β-expansion de $\delta^v{\bullet}\delta^r(A^b)$, on a le choix entre une variable violette x^v ou une variable rouge x^r. Si l'on choisit la variable rouge x^r, on insère une abstraction rouge entre δ^b et δ^v: $\delta^b(\mathbf{let}^b\ x^r{=}A^b\ \mathbf{in}^b\ \delta^v{\bullet}\delta^r(x^r))$. Si l'on choisit la variable violette x^v, on obtient $\delta^b(\mathbf{let}^b\ x^v{=}A^b\ \mathbf{in}^b\ \delta^v\delta^r(x^v))$ qui remplace l'alternance rouge-bleu par une alternance rouge-violet.

Pour pallier ce problème de prolifération ou de stagnation de la polychromie, nous allons introduire autant de renommages de variables qu'il y a de changements de couleurs, le long de Δ'. Pour notre exemple la transformation sera: $\mathbf{let}^b\ x^v{=}\mathrm{A}^b\ \mathbf{in}^b\ \delta^v(\mathbf{let}^v\ x^r{=}x^v\ \mathbf{in}^v\ \delta^r(x^r))$.

On peut garder le même nom de variable et se contenter d'en changer la couleur. Cette série de renommages met à jour les transferts de contrôle entre les unités de calcul et ainsi, pour notre exemple, que le processeur à la source de δ^v doit servir d'intermédiaire.

Cette série de renommages le long d'un chemin Δ va être générée par la fonction $\Downarrow_{\Delta}^{v^c}$. La couleur c est celle du dernier arc menant à Δ. Implicitement la variable v^c contient la valeur de l'expression remontée. On donne ici la définition de cette fonction.

Définition 8 (`pushdown`)

$$
\begin{array}{l}
\textbf{Fixpoint } \Downarrow_{\Delta}^{v^c} \stackrel{\triangle}{=} \\
\quad \textbf{match } \Delta \textbf{ with} \\
\quad \mid * \Rightarrow v^c \\
\quad \mid \delta^g \bullet \Delta \Rightarrow \textbf{let } R{=}\Downarrow_{\Delta}^{v^g} \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{if } c{=}g \\
\qquad\qquad\qquad\qquad \textbf{then } \delta^g(R) \\
\qquad\qquad\qquad\qquad \textbf{else } \{\text{let } v^g{=}v^c \text{ in } \delta^g(R)\} \\
\quad \textbf{end.}
\end{array}
$$

Comme pour la version bichrome, la transformation va s'appliquer sur un chemin reliant deux expressions de même couleur g $\Delta_p(\delta^g(\Delta^c(A^g)))$ et va utiliser $\Downarrow$ après avoir engendré la première liaison pour A. On transformera donc $\Delta_p(\delta^g(\Delta^c(A^g)))$ en $\Delta_p(\delta^g(\{\ let\ v^c{=}A^g\ in\ \Downarrow_{\Delta^c}^{v^c}\ \}))$.

La variable v est choisie de telle sorte qu'elle n'apparaisse pas dans l'expression $\Delta^c(A^g)$, ni libre, ni liée, ni sous quelle couleur que ce soit. Pour satisfaire cette condition, nous supposons l'existence d'une fonction $gensym$: $cexpr \rightarrow var$ qui sait donner un nom de variable n'ayant pas d'occurrence dans l'expression passée en argument. Il est aisé de définir par induction sur l'expression E^c un prédicat $fresh$ tel que $(fresh\ x\ E^c)$ vérifie que x n'a pas d'occurrence dans E^c.

Ainsi, la variable v créée pour la transformation est générée par $gensym(\Delta^c(A^g))$

La précondition pour que la transformation sur $\Delta_p(\delta^g(\Delta^c(A^g)))$ puisse s'appliquer est que l'expression A^g soit monochrome, c'est-à-dire qu'elle ne contienne pas une sous-expression d'une autre couleur, que l'arc δ^g soit de la même couleur que A^g (implicitement donnée par les annotations de couleur sur δ et A), que le chemin Δ^c ne soit pas vide et que ce chemin ne contienne par la couleur g de A (ce que l'on note $g \notin \Delta^c$). Les prédicats correspondants à ces conditions sont facilement définissables par induction sur les expressions ou les contextes.

En résumé, la transformation est donnée par:

Définition 9 (`hstep`) *$E_1 \nearrow E_2$ si et seulement si:*

`hstep_step` *si $E_1 = \delta^g(\Delta^c(A^g))$ avec Δ^c non vide, si A^g est monochrome (de couleur g) et si $g \notin \Delta^c$ alors $E_1 \nearrow \delta^g(\{\text{let } v^c{=}A^g \text{ in } \Downarrow_{\Delta^c}^{v^c}\ \})$ où $v{=}gensym(\Delta^c(A^g))$.*

`hstep_link` *si $E_1 = \delta(E_1')$, alors, pour toute expression E_2', si $E_1' \nearrow E_2'$, alors , $E_1 \nearrow \delta(E_2')$.*

Cette définition est implémentée par le prédicat inductif `hstep`.

3. Propriétés

La transformation $\nearrow$ a pour ambition d'être intégrée dans un compilateur. Ce dernier va enchaîner les étapes de transformation jusqu'à trouver un état stable (forme normale). D'une part, il faut assurer la correction de la transformation, c'est-à-dire que le programme transformé a le même comportement que le programme d'origine (correction: 3.1). Ensuite, pour soulager l'écriture du compilateur, il faut montrer que l'ordre dans lequel on enchaîne les transformations a peu ou pas de conséquences (confluence: 3.2). Enfin, il faut prouver que l'état stable est toujours atteignable (terminaison: 3.3)).

La polychromie nous a conduit à remanier les preuves de manière importante. Ne subsistent des scripts de preuves du calcul bichrome que 110 lignes qui correspondent aux définitions communes

de bases, sur les 2014 lignes de script. Nous présentons dans cette section les grandes lignes de ces preuves en mettant principalement l'accent sur la gestion des renommages par coloration des variables le long des chemins de transformation (fonction `pushdown`: 8). Pour le résultat de confluence, nous indiquons pourquoi et comment, pour mener à bien les preuves d'existence, il a été plus efficace de définir des fonctions explicites de construction des expressions réclamées par la confluence.

Pour les lemmes et théorèmes énoncés dans cette section, nous indiquons les noms correspondant dans le script Coq pour le lecteur intéressé par les détails des scripts de preuve.

3.1. Correction de la transformation

La transformation $E_1 \nearrow E_2$ procède à la β-expansion de l'expression à transformer, plus précisément à une série de β-expansions (pour toute relation R, nous noterons R^+ sa fermeture transitive et R^* sa fermeture réflexive et transitive). Il y a en effet deux catégories de β-expansion: l'une destinée au regroupement des sous-expressions colorées; l'autre, par effet induit de la multiplicité des couleurs[5], permet, par renommage, d'aller insérer une variable d'une couleur donnée à travers un contexte polychrome en conservant l'alternance des couleurs du contexte (fonction $\Downarrow$).

Théorème 1 (`hstep_sound`) $E_1 \nearrow E_2 \Rightarrow \downarrow E_2 \rightarrow_\beta^+ \downarrow E_1$.

Preuve par induction sur la définition de $E_1 \nearrow E_2$.

`hstep_link` il faut montrer que $\downarrow \delta(E_1) \rightarrow_\beta^+ \downarrow \delta(E_2)$ si $\downarrow E_1 \rightarrow_\beta^+ \downarrow E_2$ pour tout arc δ. Ce que l'on obtient par induction sur les cas de β-réduction pour chaque cas d'arc. C'est établi par le lemme `rt_beta_link`;

`hstep_step` il faut montrer que la transformation effective est bien une β-expansion, c'est-à-dire que $E_1(\{let\ v^c{=}A^g\ in\ \Downarrow_{\Delta^c}^{v^c}\ \}) \rightarrow_\beta^+ \downarrow\ \delta^g(\Delta^c(A^g))$ lorsque δ^g, Δ^c et A^g vérifient les prémisses de la transformation. C'est l'objet du lemme `step_sound`.

Il y a deux manières d'aborder l'induction pour s'occuper de Δ^c. La première, plus intuitive, suivrait l'ordre des évaluations d'un appel par valeur, le pas d'induction serait **let** v^{c_1}=A **in** **let** v^{c_2}= v^{c_1} **in** B $\rightarrow_\beta$ **let** v^{c_2}=A **in** B, mais des difficultés apparaissent dues à la présence potentielle de la variable v^{c_i} dans B. Il est plus facile de défaire les **let** du bas vers le haut avec un pas d'induction **let** v^{c_2}= v^{c_1} **in** B $\rightarrow_\beta$ subst(B,v^{c_2},v^{c_1}), ce qui correspond au renommage de la fonction $\Downarrow$. Ce pas d'induction est donné par le lemme qui suit.

Lemme 1 (`beta_pushdown_var`) $v \notin \Delta \Rightarrow \forall c, \downarrow \Downarrow_\Delta^{v^c} \rightarrow_\beta^* \downarrow \Delta(v_c)$

Tous les problèmes liés à la correction de la transformation viennent de la substitution engendrée par la β-réduction : **let** x=A **in** B $\rightarrow_\beta$ subst(B,x,A). La définition générale de la substitution (par exemple page 146 de [8]) sur les abstractions fait apparaître une nouvelle variable : (subst λ v.B, x,A) = λ z.subst(subst(B,v,z),x,A) où z n'est libre ni dans A, ni dans B. Cette nouvelle variable provoque des casse-têtes dans la preuve. Pour contourner le problème, on introduit des cas particuliers pour la substitution. Par exemple le cas où x=v et le cas où x n'est pas libre dans B dispensent de faire la substitution dans le corps de l'abstraction. Plus généralement la convention de Barendregt [9] où v n'est pas libre dans A permet de se dispenser de l'introduction de la nouvelle variable. La difficulté de la preuve réside maintenant dans le fait de certifier que les substitutions restent dans ces cas particuliers. Lors des renommages, **let** v^{c_2}= v^{c_1} **in** B $\rightarrow_\beta$ subst(B,v^{c_2},v^{c_1}) la convention de Barendregt (x n'est pas libre dans v^{c_1} pour un λx se trouvant dans le chemin Δ) est respectée par le fait que $v \notin \Delta$ dans les prémisses du Lemme 1. Pour le redex principal *let* $v^c{=}A^g$ *in* B, cette

[5] Ce phénomène était absent de la version bicolore.

convention est respectée par le fait que A^g est monochrome et que le chemin Δ ne contient pas la couleur g, donc la couleur de x est différente de celle de A et donc ne peut pas apparaître libre dans celle-ci.

3.2. Confluence

La confluence permet de certifier que si deux transformations E_1 et E_2 sont possibles à partir d'une même expression E, alors faire a priori une des transformations n'empêchera pas de faire l'autre transformation a posteriori. Lorsque l'on n'obtient pas directement des expressions égales, on montre que E_1 et E_2 se transforment en deux expressions R_1 et R_2 pour lesquelles nous avons défini une α-équivalence *ad hoc*, notée $R_1 \equiv R_2$.

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

Dans la version monochrome, nous avions pris l'option d'un lemme permettant de comparer deux contextes Δ_1 et Δ_2 tels que E=$\Delta_1(A_1)$=$\Delta_2(A_2)$, les principaux cas sont donnés dans la figure 2. Cette étude de cas correspond à la méthode que l'on rencontre le plus souvent pour prouver la confluence du λ-calcul. Le cas central, où l'un des redex est inclus dans l'autre, est le cas *compliqué* du λ-calcul, alors que pour notre transformation cela correspond à un cas impossible. C'est pour cette raison que nous avons pris cette approche dans la version monochrome. Pour tous les cas différents de la comparaison, on donnait les expressions R_1 et R_2 et on montrait, cas par cas, la confluence. Sachant que $\Delta 1$ est de la forme $\Delta_p(\delta^g(\Delta^c(*)))$, beaucoup de cas particuliers apparaissent, sans compter ceux qu'engendrent l'introduction de $\Downarrow$.

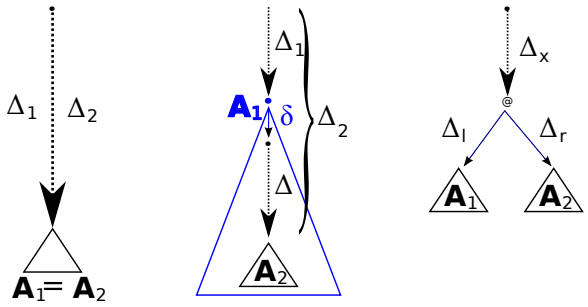

Figure 2: principaux cas de comparaison de deux contextes ayant même racine

Pour la version polychrome, nous avons fait le choix d'exhiber les fonctions qui construisent les expressions R_1 et R_2 réclamées par la confluence. On sait que : $E = \Delta_{p_1}(\delta_1(\Delta_1(A_1)) = \Delta_{p_2}(\delta_2(\Delta_2(A_2))$. Pour la première transformation on a : $E_1 = \Delta_{p_1}(\delta_1(\{\ let\ v{=}A_1\ in\ \Downarrow^v_{\Delta_1}\ \}))$. Il faut donc pouvoir réécrire cette expression E_1 sous une forme : $E_1 = \Delta_{p_{12}}(\delta_{12}(\Delta_{12}(A_2))$ pour que la transformation sur A_2 puisse avoir lieu.

Nous avons défini la fonction $\mathcal{C}$ qui, à partir des informations $\Delta_{p_1}, \delta_1, \Delta_1, A_1, \Delta_{p_2}, \delta_2, \Delta_2$, va produire le triplet : $\Delta_{p_{12}}, \delta_{12}, \Delta_{12}$. A_1 est nécessaire pour calculer le nom de la variable générée pour lier cette expression, i.e. $gensym(\Delta_1(A_1))$. Schématiquement, la fonction $\mathcal{C}$ va parcourir simultanément Δ_{p_1} et Δ_{p_2}, selon les cas du chemin le plus court entre les deux, $\mathcal{C}$ va parcourir ensemble Δ_{p_1} avec Δ_2 ou Δ_{p_2} avec Δ_1, ces trois parcours finissent par une comparaison entre Δ_1 et Δ_2.

Cette fonction $\mathcal{C}$ n'est pas forcément facile à écrire (du moins correcte du premier coup). De fait, nous avons défini trois fonctions principales chargées de construire le préfixe $\Delta_{p_{12}}$ (`build_prefix`), l'arc

δ_{12} (`build_edge`) et le contexte Δ_{12} (`build_path`). Elles font appel à 5 autres fonctions auxiliaires chargées chacune d'une configuration de mise en parallèle des chemins. Toutefois, cette complexité combinatoire reste sans comparaison avec la difficulté à expliciter, dans les lemmes ou via les scripts de preuve, les cas particuliers pour comparer les chemins menant à A_1 et à A_2.

On remarque que $\mathcal{C}$ permet de calculer autant R_1 que R_2, les deux expressions réclamées par la confluence.

Reste maintenant à prouver la correction de $\mathcal{C}$, c'est-à-dire que les valeurs qu'elle calcule permettent bien de retrouver l'expression résultant de la première transformation :

Lemme 2 (Correction de C : `build_premisse_correct`)
$\mathcal{C}(\Delta_{p_1}, \delta_1, \Delta_1, A_1, \Delta_{p_2}, \delta_2, \Delta_2) = (\Delta_{p_{12}}, \delta_{12}, \Delta_{12})$

$\Rightarrow \Delta_{p_1}(\delta_1(\{\text{ let } v{=}A_1 \text{ in } \Downarrow^{v}_{\Delta_1} \})) = \Delta_{p_{12}}(\delta_{12}(\Delta_{12}(A_2))$

Preuve *Sans problème particulier, il faut suivre la fonction $\mathcal{C}$, par induction lorsqu'elle est récursive, et vérifier l'égalité au fur et à mesure. Ce lemme permet surtout de mettre au point la fonction C.*

Il faut aussi prouver que l'arc et le deuxième contexte produit par la fonction $\mathcal{C}$ satisfont les prémisses exigées par la transformation (lemme `build_premisse_is_premisse`) : δ_{12} a la même couleur que A_2, que Δ_{12} est non vide et ne contient aucun arc de la couleur de A_2. Comme pour le lemme précédent, cela ne pose pas de problème particulier.

L'équivalence *ad hoc* Reste à prouver l'équivalence entre les deux résultats R_1 et R_2. A partir de $\mathcal{C}(\Delta_{p_1}, \delta_1, \Delta_1, A_1, \Delta_{p_2}, \delta_2, \Delta_2)$ on trouve $(\Delta_{p_{12}}, \delta_{12}, \Delta_{12})$ qui permet de construire R_1; à partir de $\mathcal{C}(\Delta_{p_2}, \delta_2, \Delta_2, A_2, \Delta_{p_1}, \delta_1, \Delta_1)$ on trouve $(\Delta_{p_{21}}, \delta_{21}, \Delta_{21})$ qui permet de construire R_2. L'équivalence va être établie en parcourant simultanément les deux triplets. Le cas le plus particulier est lorsque $\Delta_{p_{12}}\delta_{12} = \Delta_{p_{21}}\delta_{21}$. On est alors dans la situation où A_2 est *dans* Δ_{12} et A_1, *dans* Δ_{21}, sans que A_1 et A_2 soient sous-expressions l'une de l'autre; ce peut, par exemple, être les deux termes d'une application. En remontant A_1, puis A_2 on obtient une expression de la forme **let** $v_{11} = A_1$ **in let** $v_{12} = A_2$ **in** B_1; en remontant A_2 puis A_1, on obtient **let** $v_{22} = A_2$ **in let** $v_{21} = A_1$ **in** B_2. Cette situation se retrouve à chaque commutation de couleur par la fonction $\Downarrow$ dans la partie commune de Δ_{12} et Δ_{21}. Pour vérifier que les renommages sont correctement effectués, on pose que ces deux expressions doivent être équivalentes.

De manière générale, l'équivalence que nous posons ressemble à un *renommage explicite* des variables qui nous donnera les cas *ad hoc* d'α-équivalence pour les transformations opérées. En fait, nous avons spécifié trois équivalences

`let_equiv` la première, est utilisée pour R_1 et R_2 et ne va pas prendre en compte de renommage;

`let_equiv_rn1` la deuxième, va être utilisée dès que l'on va rencontrer δ_1 ou δ_2 et va prendre en compte un des deux renommages, *en dur* dans l'équivalence, $E_1 \equiv_{v_{11}=v_{21}} E_2$, pour exprimer que la variable v_{11} de E_1 est renommée v_{21} dans E_2;

`let_equiv_rn2` la dernière équivalence sera utilisée lorsque les deux arcs δ_1 et δ_2 auront été trouvés, cette équivalence va prendre en compte les deux renommages.

Certes, cette technique est particulièrement *ad hoc* mais elle permet d'alléger les cauchemars liés à l'α-équivalence.

3.3. Terminaison

Comme pour la version bichrome, la terminaison est prouvée par la décroissance du nombre d'arcs reliant deux expressions de couleurs différentes. Ce nombre d'arcs est calculé par la fonction B suivante :

Fixpoint B(e) $\stackrel{\triangle}{=}$
  match e **with**
  | CVar c v $\Rightarrow$ 0
  | CFun c v b^{c_b} $\Rightarrow$ B(b) + (c $\neq$ c_b)
  | CApp c f^{c_f} a^{c_a} $\Rightarrow$ B(f) + (c $\neq$ c_f) + B(a) + (c $\neq$ c_a)

Où l'opérateur $\neq$ est défini par : $c_1 \neq c_2 \stackrel{\triangle}{=}$ **if** c_1==c_2 **then** 0 **else** 1.

Théorème 3 (Fonction décroissante) *$E_1 \nearrow E_2 \Rightarrow B(E_1) = B(E_2) + 1$.*

Pour la version polychrome, il faut montrer que $\Downarrow$ n'introduit pas de rupture de couleur. On montre que lors des renommages, **let** $v^{c_2} = v^{c_1}$ **in** B, la couleur c_1 est celle de l'arc entrant et la couleur c_2 est celle de B.

4. Conclusion

Nous avons montré comment une série de β-expansions pouvait regrouper les couleurs dans un λ-calcul polychrome. Nous n'avons pas abordé le problème de l'optimalité, en présence de n couleurs, l'optimal serait d'obtenir n composantes de couleurs différentes. Cet optimal n'est en général pas atteint. En premier lieu, parce que deux sous-expressions de même couleur n'ayant pas un lien commun (par exemple deux expressions rouges sous une racine bleue) ne peuvent pas être regroupées. En second lieu, parce qu'une composante orpheline va empêcher toutes les expressions l'incluant d'être candidates à une transformation (car ne pouvant satisfaire le critère de monochromie). Par exemple un chemin comportant en alternance des arcs de couleurs bleue et rouge et finissant par un arc de couleur violette sera irréductible.

Une première solution pour résoudre ce problème serait d'alléger le critère de monochromie de l'expression à remonter. Le seul critère important est que l'ensemble des couleurs de l'expression à remonter et l'ensemble des couleurs sur le chemin de remontée de l'expression aient une intersection vide, autrement dit, si $E = \Delta_p(\delta^r(\Delta^b(A^r)))$, alors Δ^b, et A^r n'ont pas de couleur en commun.

Une seconde solution, qui semble meilleure, serait de considérer la racine de l'expression comme potentiellement de toutes les couleurs. Cette racine représenterait l'ensemble des expressions *top-level* du programme. Il n'y aurait ainsi plus d'expression orpheline en tant que sous-expression et le cas bloquant évoqué précédemment n'existerait plus. On pourrait voir ainsi la transformation comme une traduction de Hop ou Ocsigen vers Links [5] qui est aussi un langage où l'on précise les lieux d'exécution, mais les annotations ne sont valides qu'au niveau des formes top-level.

D'un autre côté, nous avons vu que le problème principal des preuves vient de la substitution. La substitution n'intervient que par le fait que l'on considère la β-réduction non déterministe où l'on ne précise pas comment on choisit la sous-expression à réduire. La β-réduction non déterministe est indispensable pour la preuve de correction, car la transformation nécessite une réduction forte, c'est-à-dire qu'il faut accepter de réduire sous les abstractions pour retrouver l'expression originale. En d'autres termes, la transformation ne préserve pas la terminaison pour un interprète en appel par valeur, il est possible d'avoir une expression dont le calcul termine mais dont l'expression transformée boucle. En suivant le principe de simulation d'un évaluateur paresseux par un évaluateur par valeur, il est possible de remonter une encapsulation de l'expression par une fonction sans argument, (un *thunk*), ainsi au lieu de remonter A, on remonte λ ().A, et symétriquement, au lieu de remplacer A par une variable, on applique directement cette variable (activation du *thunk*). Dans ce cadre, la preuve de correction se ferait sur un évaluateur standard en appel par valeur, sans référence à la problématique substitution.

Remerciements. Les auteurs remercient Christine Huet et Jérémie Salvucci 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, *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming (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] Emmanuel Chailloux et Bernard Serpette. Séparation des couleurs dans un lambda-calcul bichrome. In *Journées Francophones des Langages Applicatifs*, January 2012.

[4] James Cheney, Sam Lindley, et Philip Wadler. A practical theory of language-integrated query. In Greg Morrisett et Tarmo Uustalu, editeurs, *ICFP*, pp. 403–416. ACM, 2013.

[5] Ezra Cooper, Sam Lindley, Philip Wadler, et Jeremy Yallop. Links: Web Programming Without Tiers. In *FMCO*, pp. 266–296, 2006.

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

[7] Manuel Serrano et Christian Queinnec. A Multi-tier Semantics for Hop. *Higher-Order and Symbolic Computation*, pp. 1–23, 2010.

[8] Kenneth Slonneger et Barry L. Kurtz. *Formal syntax and semantics of programming languages - a laboratory based approach*. Addison-Wesley, 1995.

[9] Christian Urban, Stefan Berghofer, et Michael Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, CADE-21, pp. 35–50, Berlin, Heidelberg, 2007. Springer-Verlag.