Calculs avec Motifs Dynamiques

Thibaut Balabonski

▶ **To cite this version:**

Thibaut Balabonski. Calculs avec Motifs Dynamiques. 2008. ⟨hal-00476940⟩

HAL Id: hal-00476940
https://hal.science/hal-00476940v1

Preprint submitted on 27 Apr 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Calculs avec motifs dynamiques

Rapport technique
Laboratoire Preuves, Programmes, Systèmes
CNRS UMR 7126
Université Paris Diderot

Thibaut Balabonski
thibaut.balabonski@pps.jussieu.fr

Septembre 2008

Les mécanismes de filtrage présents dans les langages de programmation fonctionnelle usuels peuvent être enrichis de deux nouveaux genres de polymorphismes, dits polymorphismes de chemin et de motif, ainsi qu'il est fait dans le cadre du Pure Pattern Calculus (PPC) de B. Jay et D. Kesner. Ce rapport contribue à l'implémentation de ce formalisme en y étudiant différentes stratégies de réductions. L'évaluation paresseuse notamment demande une vision du mécanisme de filtrage lui-même, et motive la construction de l'Explicit Pattern Calculus (EPC), une variante de PPC avec mécanisme de filtrage explicite. La confluence de l'EPC ainsi que des théorèmes de simulation liant PPC et EPC sont prouvés.

Pattern matching mechanisms of usual functional programming languages can be enriched by two new kinds of polymorphisms (called path polymorphism and pattern polymorphism) as it is done in the Pure Pattern Calculus (PPC) of B. Jay and D. Kesner. This report contributes to the implementation of this framework by a study of some reduction strategies. Lazy evaluation in particular requires some insight in the patching mechanism itself, and motivates the construction of the Explicit Pattern Calculus (EPC) : a variant of PPC with an explicit pattern matching mechanism. The confluence of EPC as well as simulation theorems linking PPC and EPC are proven.

Table des matières

1 Problématique **2**
1.1 Programmer avec des structures de données . . . 2
1.2 Guide : du λ-calcul aux langages de programmation fonctionnels . . . 3
1.3 Le stage et sa place . . . 5

2 *Pure Pattern Calculus* : Le noyau des motifs dynamiques **7**
2.1 Idées directrices et choix de design . . . 7
2.2 Formalisme . . . 8

3 Évaluation stricte **10**
3.1 Stratégie en appel par valeur . . . 10
3.2 Une machine abstraite . . . 11

4 *Explicit Pattern Calculus* : Vers un contrôle plus fin **13**
4.1 L'infini et les limitations du noyau restreint . . . 13
4.2 Internalisation du mécanisme de filtrage . . . 14
4.2.1 Syntaxe . . . 14
4.2.2 Dynamique . . . 15
4.3 Résultats . . . 15
4.3.1 Confluence . . . 15
4.3.2 Cohérence avec le *Pure Pattern Calculus* . . . 16

5 Application I : La paresse **17**
5.1 Stratégie paresseuse et machine abstraite . . . 17
5.2 Un langage paresseux avec motifs dynamiques . . . 19

6 Application II : Structurer les données abstraites **19**
6.1 Dilemme de programmeur : élégance contre efficacité . . . 20
6.2 Une ouverture . . . 21

7 Bilan et perspectives **22**

Annexes **25**

A Preuves détaillées **25**

B Substitutions explicites **33**

C Implémentations **38**

1 Problématique

1.1 Programmer avec des structures de données

Les langages fonctionnels apportent un certain confort à l'activité de programmation en permettant de *calquer des programmes sur les définitions mathématiques* des fonctions à implémenter. Cela se traduit notamment dans la manipulation de données complexes par la définition de fonctions *par cas* : on distingue plusieurs *formes* possibles pour l'argument de la fonction, et on spécifie un traitement adapté à chacune. Ce mécanisme est nommé **filtrage**, et les formes caractérisant chaque cas sont appelées **motifs**.

Donnons par exemple la définition suivante pour les arbres binaires (en *Caml* [Cam]) :

```
type 'a binTree = Data of 'a                          (* Feuille       *)
                | Node of 'a binTree * 'a binTree    (* Noeud interne *)
```

On peut alors définir une fonction comptant le nombre de feuilles d'un arbre en distinguant deux cas :

- Sur une feuille, compter `1`. Ce cas correspond au motif `Data _`, où `_` désigne une information qu'on défausse.
- Sur un noeud interne, compter les feuilles dans les deux fils puis additionner les résultats. On a ici le motif `Node y z`, où `y` et `z` enregistrent les deux fils.

Concrètement, cette fonction s'écrit ainsi :

```
let rec countData = fun              (* countData : 'a binTree -> int *)
    | Data _      -> 1
    | Node y z    -> (countData y) + (countData z)
```

On a pris dans cet exemple la liberté d'écrire `Node y z` à la place du `Node(y,z)` de la véritable syntaxe *Caml*. On gardera cette habitude d'écrire les applications de constructeurs en forme curryfiée (comme dans l'assistant de preuves *Coq* [Coq07] par exemple) pour faciliter certaines illustrations et se rapprocher du *Pure Pattern Calculus* qui sera notre sujet.

Repousser les limites du filtrage.

Sous sa forme usuelle, le mécanisme de filtrage est assez restreint. D'abord les motifs que l'on peut utiliser pour caractériser chaque cas sont soumis à d'importantes contraintes : il est notamment interdit de les faire dépendre d'un *paramètre*. On les appellera pour cela **motifs statiques**. Ceci empêche notamment la définition d'un *éliminateur générique* qui correspondrait au code proposé ci-dessous :

```
let elim c = fun
    | c x  -> x      (* Si on trouve le constructeur, l'éliminer *)
    | y    -> y      (* Sinon, ne rien faire                     *)
```

Ici le motif `c x` est paramétré par le constructeur `c` (ou peut-être est-ce même une fonction ?). Ainsi, selon son premier argument la fonction `elim` pourrait donner naissance à plusieurs fonctions, définies par cas sur des structures différentes. Cette capacité correspond à ce qu'on appelle **polymorphisme de motif**, et est illustrée en figure 1 par deux exemples hypothétiques d'utilisation de l'éliminateur générique `elim`.

Notons que cette approche transformerait radicalement la nature des motifs : ils ne seraient plus des entités fixes, écrites dès l'origine dans leur forme définitive par le programmeur, mais des programmes devant être exécutés afin d'obtenir le motif proprement dit. C'est ainsi que l'on parlera de **motifs dynamiques**.

```
                                |       let f = fun z -> (Node z _)
                                |
      elim Data                 |       elim f
  =>  fun                       |   =>  fun
        | Data x  -> x          |         | Node x _  -> x
        | y       -> y          |         | y         -> y
                                |
      elim Data (Data 0)        |       elim f (Data 0)
  =>  0                         |   =>  Data 0
                                |
                                |       elim f (Node (Data 1) (Data 2))
                                |   =>  Data 1
```

Fig. 1 – Polymorphisme de motif.

Seconde restriction, il est également impossible d'accéder à l'intérieur d'une structure de données sans tester explicitement sa forme extérieure : il est usuellement obligatoire de faire commencer chaque motif par un constructeur fixé. Est donc encore exclue cette fonction qui mettrait à jour toutes les données d'une structure quelconque :

```
let rec mapData f = fun
    | Data x      -> Data (f x)                      (* Mettre à jour les données       *)
    | y z         -> (mapData f y) (mapData f z) (* Plonger dans les sous-structures *)
    | a           -> a                               (* Laisser les atomes inchangés    *)
```

L'exécution de cette fonction par cas aurait alors la forme suivante :

```
                      mapData f (Node (Data 1) (Data 2))
  =>  (      mapData f (Node (Data 1))      )    (mapData f (Data 2))
  =>  (mapData f Node)   (mapData f (Data 1))    (mapData f (Data 2))
  =>        Node             (Data (f 1))           (Data (f 2))
```

Et la fonction s'appliquerait à nos arbres binaires sans qu'il soit besoin d'y faire référence où que ce soit ! Elle s'appliquerait de même à toute structure contenant des objets de la forme `Data d`. Cette capacité correspond elle au **polymorphisme de chemin**.

Enfin, et ce problème n'est pas des moindres, les fonctions par cas ne peuvent traiter que des structures de données dont on connaît tout, au moins au moment de l'exécution : il est impossible de les appliquer à des *structures abstraites* dont la représentation précise nous serait cachée, structures qui sont pourtant souvent utilisées à des fins de robustesse et d'efficacité.

Les travaux présentés ici visent à affranchir le filtrage des restrictions précédentes. On partira du *Pure Pattern Calculus* [JK08] décrit en section 2, un mécanisme minimaliste réalisant les polymorphismes de motif et de chemin, et on avancera vers son implémentation en suivant les jalons posés par la section suivante. On proposera également à la fin une ouverture vers le filtrage des données abstraites.

1.2 Guide : du λ-calcul aux langages de programmation fonctionnels

Du simple concept de fonction à l'implémentation d'un langage fonctionnel, il y a un certain cheminement, dont le λ-calcul est une étape clé. On aura ici une démarche parallèle à ce chemin aujourd'hui bien connu. Mais les structures de données et le mécanisme de filtrage, qui souvent ne sont qu'une couche supplémentaire au-dessus d'un noyau théorique, auront dès l'origine un statut. Le rôle intermédiaire du λ-calcul sera alors tenu par le *Pure Pattern Calculus*. On présente dans

cette section la voie des langages fonctionnels purs afin d'illustrer la démarche et d'introduire les notions liées.

Du concept au formalisme.

Pour manipuler effectivement des fonctions, nous devrons avoir une vision *constructive* de ces objets. Dans cette optique il faudra répondre à deux questions essentielles :

- Comment fabrique-t-on une fonction ?
- Comment utilise-t-on une fonction ?

Pour répondre à la première, on identifiera une fonction à un objet *qui demande un argument* puis *qui retourne un résultat.* On a pour ceci en *Caml* la construction `fun x -> M`, où `x` désigne l'argument attendu et `M` les calculs à faire pour obtenir le résultat. On peut dans `M` faire référence à l'argument attendu par l'intermédiaire du nom `x` qui lui est lié. On parle d'**abstraction** par rapport à `x`, et en λ-calcul on écrit $\lambda x.M$ cette même construction. Il s'agit en logique d'une *règle d'introduction.*

Pour répondre à la seconde, on dira qu'on utilise une fonction en l'appliquant à un argument, ce qui se note simplement $M\ N$ pour l'**application** de M à N. On a cette fois en logique une *règle d'élimination.*

On termine de répondre à la question en décrivant l'interaction entre ces deux constructions. Il s'agit de la règle de **β-réduction**, ou plus généralement *élimination des coupures* en logique. L'idée est la suivante : l'application de l'abstraction $\lambda x.M$ à l'argument N a pour résultat M dans lequel on a remplacé toutes les occurences de x par N. On parle de **substitution** de x par N. Cette réduction se note :

$$(\lambda x.M)\ N \quad \to_\beta \quad M^{\{N/x\}}$$

On a ainsi par exemple :

$$(\lambda x.(x+x))\ 3 \quad \to_\beta \quad (x+x)^{\{3/x\}} \quad = \quad 3+3$$

Un point important à remarquer est que l'opération de substitution est définie à l'*extérieur* du formalisme : on a spécifié le résultat de l'opération, mais la syntaxe du λ-calcul ne rend pas compte de la manière dont cette opération est effectuée. On parle d'opération **implicite** ou *méta*-opération.

Dans un tel formalisme, dont nous avons décrit la dynamique pas-à-pas par une règle locale, il n'y a pas de réel **déterminisme** dans le sens où la règle β peut à un instant donné s'appliquer à plusieurs endroits d'un programme pour des résultats immédiats différents. Cependant le tout obtient une certaine cohérence grâce à une propriété *à long terme* appelée **confluence**. Elle assure que tous les enchaînements possibles de réductions locales aboutiront *tôt ou tard* au même résultat global, comme illustré par la figure 2 (où α^+ désigne "la lettre suivante" pour $\alpha = A, B, C \ldots$).

Déterminisation.

Mais si la confluence suffit à la théorie, c'est vraiment du déterminisme dont on aura besoin quand il faudra faire exécuter un programme par un ordinateur séquentiel. Pour avancer vers l'implémentation, il faut donc maintenant spécifier un **ordre d'évaluation** des programmes (qu'on appelle aussi une **stratégie**). On a pour le λ-calcul deux grands classiques :

- Appel par valeur, ou évaluation **stricte**. Chaque fonction évalue systématiquement tous ses arguments avant de commencer à s'exécuter elle-même. Cette stratégie est par exemple celle du langage *Caml.* Elle est adaptée aux langages possédant des *traits impératifs* et interagissant avec la mémoire de l'ordinateur.
- Appel par nom, ou évaluation **paresseuse**. Les fonctions s'exécutent uniquement elles-mêmes, et ne vont consulter leurs arguments qu'au moment où cela est nécessaire. On trouve cette stratégie dans le langage *Haskell.* Elle est une réponse naturelle pour traiter avec l'infini, ce qu'on va illustrer ci-dessous.

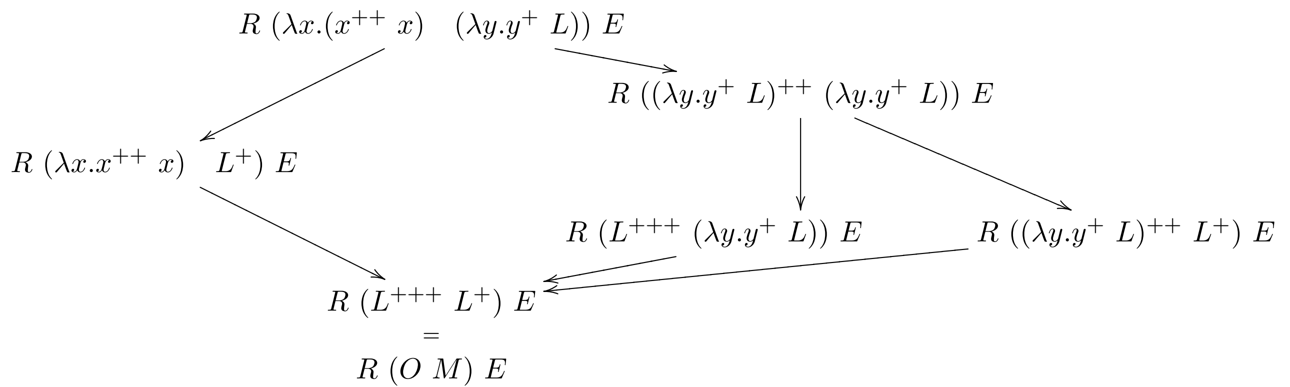


FIG. 2 – Plusieurs chemins mais une seule destination.

Exemple en syntaxe *Caml* :

```
let f b x y = if b
              then x
              else y
```

La fonction `f` teste d'abord son premier argument `b`, puis utilise ensuite l'un des suivants `x` ou `y` (mais on ne peut prédire lequel). Regardons l'application suivante :

```
f True 42 omega
```

où `omega` est un terme dont l'évaluation boucle (on a donc un calcul infini qui ne retourne jamais de résultat). La stratégie stricte évalue d'abord ses arguments, dont `omega`, et boucle alors. La stratégie paresseuse commence par exécuter son code : elle fait appel au booléen `b`, en l'occurence `True`, puis regarde la branche correspondante et répond `42`, négligeant son dernier argument et évitant le piège qu'il recèle.

Implémentation.
Chaque stratégie donne naissance à une fonction déterministe d'évaluation. On peut l'implémenter de différentes manières ([Jon87]) :

- Immédiatement comme une fonction d'**interprétation** écrite dans un autre langage. Cette fonction travaille directement sur les termes du calcul et effectue les réductions dans l'ordre prescrit.
- Par un **compilateur** transformant un programme en une suite d'instructions machine (pour une machine réelle ou virtuelle). Un programme compilé est donc exécuté directement par l'ordinateur, ce qui est susceptible d'être plus efficace. La compilation peut être une transformation compliquée faisant intervenir de nombreuses optimisations.
- Par une **machine abstraite** dédiée au langage que l'on étudie. Cette solution intermédiaire travaille directement sur les termes du calcul, mais est représentée par un système de transitions simple qui ne dépend plus d'un langage tiers.

On s'intéressera surtout aux machines abstraites, mais les interpréteurs seront une étape intermédiaire clé. On a pour le λ-calcul et ses dérivés de nombreuses machines : on citera par exemple la machine *CEK* [FF86] pour l'appel par valeur et la machine de Krivine [Kri05] pour l'appel par nom.

1.3 Le stage et sa place

Contexte.
En 1987, l'idée d'une extension du λ-calcul remplaçant l'abstraction usuelle d'un variable $\lambda x.M$

par l'abstraction d'un motif $\lambda P.M$ est évoquée par Simon PEYTON-JONES dans son livre [Jon87]. Il s'agit alors d'une piste de recherche proposée à la communauté. Trois ans plus tard, Vincent VAN OOSTROM publie le cadre général du λ-calcul avec motifs (*λ-calculus with patterns* [vO90]). Est alors notamment proposée une contrainte sur les motifs acceptables, nommée *Rigid Pattern Condition*, assurant la confluence des dérivés du λ-calcul avec motifs. Ce formalisme a été revisité très récemment par l'auteur et d'autres dans [KvOdV08].

Un autre cadre général pour les calculs avec motifs est celui du ρ-calcul, ou calcul de réécriture, d'Horatiu CIRSTEA et Claude KIRCHNER [CK98]. Ce formalisme permet notamment de décrire des systèmes de réécriture d'ordre supérieur, et permet certains filtrages pour l'instant hors de portée des autres calculs. Cependant, dans un cadre typé le ρ-calcul n'est pas une extension conservative du λ-calcul.

Mais la *Rigid Pattern Condition* assurant la confluence de ces deux cadres exclut la majorité des exemples de polymorphisme de chemin et de polymorphisme de motif qui nous intéressent. On orientera donc ce travail vers le *Pure Pattern Calculus* de Delia KESNER et Barry JAY [JK08] : un mécanisme minimaliste admettant les polymorphismes de chemin et de motif, bâti sur le slogan "tout terme peut être un motif". Ce cadre connaît une réalisation concrète expérimentale avec le langage *Bondi* [Bon05] : un langage de programmation fonctionnel similaire à *Caml*, avec des traits impératifs et orientés objets. Ce langage est implémenté par un interpréteur suivant une stratégie stricte qui n'a pas encore été formalisée.
Après avoir exploré les fondements de cette stratégie, on verra que le *Pure Pattern Calculus* ne permet pas de définir un ordre d'évaluation paresseux. Il faudra donc étendre le formalisme avant de pouvoir introduire une variante paresseuse de *Bondi* capable de traiter des données infinies, ainsi qu'il est fait dans le langage *Haskell* [Has] mais sur des motifs statiques uniquement.

Pour la construction des machines abstraites, on tirera profit d'un technologie récente due entre autres à Olivier DANVY [ABDM03], qui permet par une suite de transformations de programmes d'aller d'une fonction récursive, par exemple un interpréteur, à un système de transitions.

Réalisations.

Les apports de ce stage sont :

- La formalisation d'une stratégie stricte pour le *Pure Pattern Calculus* et la construction d'une machine abstraite l'implémentant correctement.
- Une extension du *Pure Pattern Calculus* internalisant le mécanisme de filtrage, nommée *Explicit Pattern Calculus*. Sont associés un théorème de confluence, ainsi que des résultats de simulation entre *Pure Pattern Calculus* et *Explicit Pattern Calculus*.
- L'extension du *Pure*- et de l'*Explicit- Pattern Calculus* par un mécanisme de substitution explicite, avec également des théorèmes de confluence et de simulation.
- La définition d'une stratégie paresseuse pour l'*Explicit Pattern Calculus*, ainsi que la construction d'une machine abstraite associée.
- L'implémentation d'une stratégie paresseuse dans l'interpréteur de *Bondi*.
- Une exploration d'un mécanisme de filtrage de données abstraites, les vues [Wad87], tirant parti de l'extension *Explicit Pattern Calculus*.

La section 2 décrira en détail les idées et le formalisme du *Pure Pattern Calculus*, et la suivante 3 décrira une stratégie stricte supportée par ce calcul, ainsi que la construction d'une machine abstraite associée. La section 4 motivera et définira l'*Explicit Pattern Calculus* avant de fournir les résultats validant cette extension. Les sections 5 et 6 travaillerons à partir de l'*Explicit Pattern Calculus*, la première pour définir et implémenter une stratégie d'évaluation paresseuse, et la seconde pour donner une ouverture vers le filtrage des données abstraites.

A la fin de ce document, des annexes regrouperont les données jugées moins utiles à la compréhension du stage et de ses enjeux. L'annexe A donnera le détail des preuves éludées dans le

rapport. L'annexe B définira ensuite des mécanismes de substitutions explicites pour le *Pure Pattern Calculus* et l'*Explicit Pattern Calculus* et fournira les preuves des théorèmes associés. Enfin l'annexe C fournira les évaluateurs des stratégies strictes et paresseuses et détaillera la construction des machines abstraites.

2 *Pure Pattern Calculus* : Le noyau des motifs dynamiques

Le *Pure Pattern Calculus*, pour lequel on utilisera parfois l'abbréviation *PPC*, est un calcul extrêment compact réalisant les idées de motifs dynamiques citées en introduction ainsi que les polymorphismes de chemin et de motif. Il a connu jusqu'à ce jour trois versions successives [JK06a], [JK06b] et [JK08]. On s'attachera d'abord à présenter leur base commune, ainsi que les raisons qui poussent à passer d'une version à la suivante. On donnera la syntaxe de la version la plus récente uniquement.

On vérifiera ensuite dans la section suivante qu'on peut aisément définir une stratégie d'évaluation stricte pour ce calcul, et on construira une machine abstraite l'implémentant. On verra cependant plus tard que la vision des mécanismes d'évaluation donnée par ce calcul n'est pas assez précise pour pouvoir y décrire syntaxiquement une stratégie d'évaluation paresseuse.

2.1 Idées directrices et choix de design

Pour réaliser le cahier des charges donné en introduction, de nombreux choix sont à faire. Ceux que nous ferons ici visent à donner une description la plus petite possible du calcul (en terme de constructions syntaxiques et règles de réduction notamment).

Slogan.
On a déjà évoqué le slogan central du *Pure Pattern Calculus* : "tout terme peut être un motif". Ceci fait écho à un slogan du λ-calcul : "tout terme peut être une fonction", et signifie que les motifs vont être des *citoyens de première classe* : ils pourront être passés comme paramètres, évalués, et retournés comme résultats.

Simplifier par le *méta*.
Comme on l'a vu, deux opérations sont nécessaires pour effectuer un filtrage :

- En premier lieu, le filtrage de l'argument contre le motif.
- Ensuite, l'application de la substitution générée par le filtrage.

On peut faire en sorte pour chacune qu'elle soit une opération explicite décrite à l'intérieur du langage, ou une *méta*-opération, implicite, spécifiée à l'extérieur, ainsi que la substitution cachée par la β-réduction du λ-calcul. Dans le *Pure Pattern Calculus* ces deux opérations seront implicites. Une seule règle de réduction effectuera de façon atomique les deux opérations de filtrage et de substitution l'une à la suite de l'autre.
On donnera pour se repérer le tableau suivant :

	Filtrage implicite	Filtrage explicite
Substitution implicite	*PPC*	
Substitution explicite		

Surcharger l'application.
On a jusqu'ici parlé de deux applications, qui sont moralement différentes :

- L'application d'une fonction (définie par cas) à un argument.
- L'application d'un constructeur à des données.

La première se réduit par la *méta*-opération évoquée au-dessus, alors que la seconde donne directement une nouvelle structure de données. Cependant, la nature d'une application étant définie par l'objet que l'on applique, on pourra confondre les deux applications en une même opération.

On peut alors voir, par une sorte de *sous-typage*, un constructeur comme une fonction construisant une structure de données.

Restreindre l'opération de filtrage.
On ne peut permettre (sous peine de perte de confluence) de filtrer ou d'utiliser comme motif n'importe quelle expression réductible. C'est pour cela que les approches précédentes se sont basées sur une *Rigid Pattern Condition* [KvOdV08] restreignant les motifs. Dans le *Pure Pattern Calculus* en revanche, aucune condition n'étant imposée aux motifs, c'est l'opération de filtrage elle-même qui devra contrôler ceci et différencier les motifs *stables* et prêts à l'emploi des motifs *instables* et dangereux.

C'est sur ce point que divergent les versions successives du calcul. La première version n'autorisait le déclenchement du filtrage que pour des termes irréductibles en un certain sens [JK06a]. Ceci induisait une définition mutuellement récursive entre l'opération de réduction et la méta-opération de filtrage. Les versions suivantes [JK06b] et [JK08], qui sont équivalentes, introduisent à la place une condition syntaxique pour le déclenchement du filtrage, avec la notion de *terme testable* (*matchable form*). On peut ainsi décrire d'abord la *méta*-opération de filtrage toute seule, puis l'utiliser pour définir la réduction, sans plus aucune dépendance cyclique.

2.2 Formalisme

On donne ici un condensé du formalisme du *Pure Pattern Calculus* dans sa dernière version [JK08].
Grammaire des termes :

t	$::=$	x	occurence normale de variable	*variable*
	$\mid$	$\hat{x}$	occurence liante ou constructeur	*matchable*
	$\mid$	$t\ t$	application de fonction ou de constructeur	
	$\mid$	$[\theta]\ t \rightarrow t$	abstraction d'un motif	*case*

Dans l'abstraction d'un motif, le terme à gauche de la flèche est le **motif**, et celui à droite est le **corps**, c'est-à-dire le traitement à effectuer en cas de filtrage réussi. θ est l'ensemble des noms liés de cette construction, la liaison s'effectuant comme suit :

$$[x]\ \hat{x} \rightarrow x$$

Remarque : ce terme représente la fonction identité. En effet le motif accepte n'importe quel argument, et le mémorise sous le nom x, et le corps restitue ensuite ce qui est associé au nom x.

On peut alors définir une relation d'α-équivalence entre termes, qui nous autorise à modifier les noms liés :

$$[\theta]\ p \rightarrow s \quad =_\alpha \quad [\theta^{\{y/x\}}]\ p^{\{\hat{y}/\hat{x}\}} \rightarrow s^{\{y/x\}}$$

Une occurence $\hat{x}$ liée correspond à une variable de filtrage, qui va générer une substitution. Une occurence $\hat{x}$ libre est en revanche un objet inerte. De plus, le renommage induit par l'α-conversion étant limité aux objets liés, une occurence $\hat{x}$ libre est donc caractérisée par le nom x. On a alors un objet *inerte* et *reconnaissable* : on peut s'en servir comme constructeur.

Dans la seconde version du *Pure Pattern Calculus* [JK06b], la notation $\hat{x}$ n'existait pas, et on ne pouvait savoir localement qui était constructeur ou variable. Les règles de réduction étaient donc paramétrées par un ensemble de constructeurs, et la notion de réduction devenait alors dépendante du contexte global. La distinction entre x et $\hat{x}$ a permis de réduire le *Pure Pattern Calculus* (version 3 [JK08]) à une unique règle de réduction indépendante du contexte :

$$([\theta]\ p \rightarrow s)\ u \rightarrow_{\beta_m} s^{\{u/[\theta]\ p\}}$$

où $\{u/[\theta]\ p\}$ est le résultat du filtrage de l'argument u contre le motif p avec les variables de filtrage θ. Il s'agit d'une substitution de domaine θ en cas de succès du filtrage. On a sinon deux possibilités : si le filtrage aboutit mais négativement ("u ne sera jamais une instance de p"), on renvoie une valeur d'échec ; si le test n'aboutit pas ("u et p ne sont pas encore sous une forme permettant la comparaison") $\{u/[\theta]\ p\}$ n'est pas défini. On appelle **filtrat** ce type d'objet dénoté par la *méta*-variable μ, qui correspond à la grammaire suivante :

$$\mu \quad ::= \quad \textit{échec} \mid \sigma$$

où σ est une substitution quelconque qu'on pourra noter explicitement $\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$.

Comme pour la réduction du λ-calcul, on peut appliquer la règle β_m dans n'importe quel contexte, c'est-à-dire n'importe où à l'intérieur des termes, et on note $\rightarrow_{ppc}$ la relation de réduction engendrée.

Pour définir l'opération de filtrage, on commence par caractériser les **termes testables** (*matchable forms*, *méta*-variable m) :

$$\begin{array}{rcl} m & ::= & d \mid [\theta]\ t \rightarrow t \\ d & ::= & \hat{x} \mid d\ t \end{array}$$

La méta-variable d désigne une **structure** : un constructeur appliqué à une liste de termes. Vient ensuite une première notion de filtrage (*compound matching*) :

$$\begin{array}{rcll} \{\!\{u/[\theta]\ \hat{x}\}\!\} & = & \{x \mapsto u\} & \text{si } x \in \theta \text{ (variable de filtrage)} \\ \{\!\{\hat{x}/[\theta]\ \hat{x}\}\!\} & = & \{\} & \text{si } x \notin \theta \text{ (constructeur)} \\ \{\!\{u\ v/[\theta]\ p\ q\}\!\} & = & \{\!\{u/[\theta]\ p\}\!\} \uplus \{\!\{v/[\theta]\ q\}\!\} & \text{si } u\ v \text{ et } p\ q \text{ testables (composé)} \\ \{\!\{u/[\theta]\ p\}\!\} & = & \textit{échec} & \text{autres cas avec } u \text{ et } p \text{ testables (erreur avérée)} \\ \{\!\{u/[\theta]\ p\}\!\} & & & \text{non défini sinon (besoin d'évaluer plus les termes)} \end{array}$$

où $\uplus$ est une opération d'union disjointe de filtrats définie comme suit :

$$\begin{array}{rcll} \textit{échec} \uplus \mu & = & \textit{échec} & \\ \mu \uplus \textit{échec} & = & \textit{échec} & \\ \sigma \uplus \sigma' & = & \sigma \cup \sigma' & \text{si } dom(\sigma) \cap dom(\sigma') = \emptyset \\ \sigma \uplus \sigma' & = & \textit{échec} & \text{sinon} \end{array}$$

où l'union deux substitutions disjointes est la suivante :

$$\begin{array}{c} \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\} \cup \{y_1 \mapsto v_1, \ldots, y_m \mapsto v_m\} \\ = \\ \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n, y_1 \mapsto v_1, \ldots, y_m \mapsto v_m\} \end{array}$$

L'opération $\uplus$ est associative et commutative. On prend de plus comme convention que l'union disjointe du filtrat *échec* avec un filtrat non défini est *échec*. Moralement, cela signifie que si on repère une inadéquation entre le motif et l'argument lors d'un filtrage, on peut interrompre de suite l'opération et renvoyer une erreur.

Ici, $\{\!\{u/[\theta]\ p\}\!\}$ génère la substitution correspondant au filtrage de u contre p. On passe à $\{u/[\theta]\ p\}$ en ajoutant un test : il faut que le domaine de la substitution soit égal à θ, sinon on renvoie *échec* quoi qu'il arrive.

En effet, dans le terme $([\theta]\ p \rightarrow s)\ u$, les noms de θ sont des variables liées de s, qui n'ont de sens que dans ce terme. Une fois la substitution $\{u/[\theta]\ p\}$ appliquée, ces variables ne doivent pas subsister. Elles doivent donc toutes participer au domaine de la substitution. Ce test au dernier moment est nécessaire à cause de l'aspect dynamique des motifs : en effet le domaine de la substitution engendrée correspond aux $\hat{x}$ liés du motif, mais certaines occurences peuvent disparaître au moment de l'exécution, rendant le résultat invalide.

Enfin, remarquons que l'alternative entre plusieurs motifs/traitements n'est pas présente dans cette syntaxe. Elle peut en effet être codée astucieusement en choisissant correctement ce que signifie l'application du résultat de filtrage *échec*.

Exemple de filtrage :

$$\begin{array}{rl} \multicolumn{2}{l}{([x]\ \underline{([z]\ \hat{z} \to z)\ (\hat{c}\ \hat{x})} \to s)\ (\hat{c}\ t)} \\ \to_{ppc} & ([x]\ \hat{c}\ \hat{x} \to s)\ (\hat{c}\ t) \\ \to_{ppc} & s^{\{x \mapsto t\}} \end{array}$$

Dans cet exemple, le filtrage de l'argument $\hat{c}\ t$ contre le motif $([z]\ \hat{z} \to z)\ (\hat{c}\ \hat{x})$ ne pouvait pas être effectué dès le début, car le terme $([z]\ \hat{z} \to z)\ (\hat{c}\ \hat{x})$ n'était pas testable, et donc le filtrage n'était pas défini.

3 Évaluation stricte

Avec ce calcul, Delia Kesner et Barry Jay ont donné une solution extrêmement concise aux motifs dynamiques. On va voir maintenant que ce niveau de détail permet de définir simplement un ordre d'évaluation en *appel par valeur*, puis en discuter l'implémentation. On en profitera pour introduire une approche due à Olivier Danvy pour construire des machines abstraites correctes [ABDM03].

3.1 Stratégie en appel par valeur

Notion de valeur.

La première étape est de définir la notion de *valeur*. Ici on voudra appeler valeur tout terme comparable à n'importe quel autre *suffisamment réduit*, ou autrement dit tout terme qui ne pourra jamais être *responsable* du non-aboutissement d'un filtrage.

Il faut donc que tous les sous-termes d'une valeur auxquel peut accéder l'opération de filtrage soient des termes testables. Une **valeur** sera alors d'une des formes :

- Un constructeur.
- Une fonction.
- Un constructeur appliqué à des valeurs.

On traduit ce cahier des charges par la grammaire suivante :

$$\begin{array}{rcl} v & ::= & dv \mid [\theta]\ t \to t \\ dv & ::= & \hat{x} \mid dv\ v \end{array}$$

Ceci est une restriction directe de la grammaire des termes testables : la différence est qu'on applique les constructeurs uniquement à des valeurs et non à des termes généraux.

Stratégie.

On peut maintenant adopter la stratégie suivante d'appel par valeur pour réduire le terme $([\theta]\ p \to s)\ u$. Littéralement, évaluer signifie "réduire à une valeur".

1. Évaluer p en p'.
2. Évaluer u en u'.
3. Calculer $\{u'/[\theta]\ p'\}$.

On encode cette stratégie dans la restriction suivante des **contextes de réduction**, c'est-à-dire les contextes dans lesquels on autorise l'application de la règle β_m.

$$\begin{array}{rcl} C & ::= & [] \\ & \mid & C\ t \\ & \mid & dv\ C \\ & \mid & ([\theta]\ C \to t)\ t \\ & \mid & ([\theta]\ v \to t)\ C \end{array}$$

A cette stratégie correspond une fonction d'évaluation, ou interpréteur. Son code *OCaml* est fourni en annexe C. À noter cependant que pour une implémentation raisonnable, on gère les substitutions par des environnements. Ceci impose d'expliciter le mécanisme de substitution. Comme le cadre du *Pure Pattern Calculus* n'apporte pas de difficultés particulières par rapport au cas connu du λ-calcul, et comme les substitutions explicites ne nous servent que pour ce point précis, la construction est reportée à l'annexe B.

3.2 Une machine abstraite

À partir de l'évaluateur écrit dans un langage fonctionnel, on va maintenant déduire une machine abstraite, c'est-à-dire un système de transitions *simple*, par une suite de transformations de programmes éprouvées. La correction de la machine obtenue sera immédiate d'après les théorèmes de correction de chaque transformation successive.

L'interpréteur que l'on possède déjà servira de point de départ à la construction. Plus généralement, ce plan permet de passer d'une fonction récursive définie sur des objets inductifs à un système de transitions effectuant le même calcul pas-à-pas. Les objectifs principaux seront :

- Expliciter le *contrôle* : on veut savoir de quelle façon exacte s'enchaînent les calculs.
- Tout aplatir : les objets trop complexes et les fonctions d'ordre supérieur ne peuvent être gérés par un système du premier ordre.

On procédera alors en appliquant dans l'ordre les opérations suivantes, qui seront décrites en détail dans la suite de cette section :

1. Aplatir les données. On ne veut que des valeurs du premier ordre : on va interdire notamment aux fonctions de contenir des paramètre extérieurs. La transformation à effectuer est connue sous le nom de *closure conversion* [Rey98].
2. Expliciter le contrôle. L'enchaînement des calculs doit être apparent. On passe pour cela en *style par passage de continuations*, voir [Plo75] pour une référence historique et [DF92] pour un point de vue plus général.
3. Aplatir le contrôle. On veut encore un contrôle du premier ordre : on *défonctionalise* l'espace des continuations [Rey98].

Notre interpréteur étant une fonction déjà relativement compliquée, on reporte la manipulation complète à l'annexe C. On va en revanche tout de suite illustrer les transformations sur une fonction récursive plus simple implémentant l'algorithme d'exponentiation rapide.

Supposons que les entiers naturels sont représentés par le type algébrique suivant :

```
type bin_int = ZERO
             | O of bin_int
             | I of bin_int
```

Ceci correspond à l'écriture binaire des entiers : `O n` correspond à la multiplication de `n` par deux, soit à l'ajout d'un `0` à droite de sa représentation binaire. Le constructeur `I` correspond lui à l'ajout d'un `1` à la notation binaire. Avec ceci on travaillera à partir de la fonction d'exponentiation suivante, qui retourne le nombre `x` à la puissance entière (positive ou nulle) `n`, avec un nombre de multiplications logarithmique en `n` :

```
let rec bin_exp x = fun
    | ZERO -> 1
    | O n -> let y = bin_exp x n
             in y*y
    | I n -> let y = bin_exp x n
             in x*y*y
```

Closure conversion.

Il s'agit de passer à un style où les fonctions ne peuvent plus contenir de variables libres. Elles ne pourront donc plus faire appel de façon anarchique à des objets de l'environnement. Pour ce faire, chaque fonction va maintenant prendre en argument supplémentaire l'environnement courant. Une fonction sera transformée en une **clôture fonctionnelle** contenant tous les paramètres dont elle a besoin. Ceci sera illustré lors de la défonctionalisation.

Continuations.

Ce qu'on appelle continuation est une fonction représentant le traitement futur que l'on veut appliquer à un objet. Le style de programmation **par passage de continuations** est l'utilisation systématique de ce principe : chaque fonction prend en argument d'abord les arguments normaux qu'on lui aurait donné en style direct, mais également en plus une continuation lui indiquant ce qu'elle doit faire de son résultat. Ainsi la dernière action d'une fonction n'est plus "retourner un résultat" (retourner où ? à qui ? pour en faire quoi ?), mais "appeler la continuation", c'est-à-dire donner son résultat à une fonction spéciale qui saura ce qu'il faut en faire. Les continuations matérialisent donc le cours du calcul, décident des choses à faire et de leur ordre. Elles explicitent ce qui lie entre eux les calculs élémentaires d'un programme.

La transformation de notre évaluateur en style direct en un autre en style par passage de continuations contient essentiellement les ingrédients suivant :

- Ajouter à la fonction un argument supplémentaire : la continuation à appliquer au résultat de l'évaluation, traditionnellement notée `k`.
- Ajouter aux cas de base l'application de la continuation courante.
- Traduire les résultats intermédiaires introduits par un `let ... in ...` par la construction d'une nouvelle continuation.

```
let rec bin_exp x = fun
    | (ZERO, k)  -> k 1
    | (O n, k)   -> bin_exp x (O n, fun y -> k (y*y))
    | (I n, k)   -> bin_exp x (I n, fun y -> k (x*y*y))
```

Défonctionalisation.

Avec la transformation précédente, on vient d'introduire dans le code de nombreuses fonctions potentiellement compliquées : les continuations sont construites avec des enchaînements de `fun ... -> ...`. La dernière étape sera donc d'aplatir ceci grâce à une défonctionalisation.

Cette transformation vise à exprimer toutes les fonctions que manipulera le programme par des constructions qui n'ont plus rien de fonctionnel. L'habituelle application sera donc remplacée par une fonction réalisant explicitement l'application.

L'idée ici est la suivante : toutes les continuations qui pourront voir le jour seront combinaison des `fun ... -> ...` apparaissant dans le texte du programme. On donnera donc un nom à chacune de ces fonctions et on codera les continuations comme des données construites avec ces noms. La fonction explicite d'application récupérera alors ces noms et appliquera le traitement correspondant.

Dans notre exemple, deux fonctions apparaissent pour construire les continuations :

- `fun y -> k (y*y)`, qui correspondra au constructeur `Y_EVEN(k)`
- `fun y -> k (x*y*y)`, qui correspondra au constructeur `Y_ODD(x,k)`

On a donné comme argument à chaque constructeur l'ensemble des variables libres de la fonction. On ajoute un troisième constructeur `STOP` pour la continuation initiale.

On définit donc maintenant conjointement une fonction d'exponentiation "défonctionalisée" et une fonction d'application de nos continuations codées.

```
let rec bin_exp x = fun
    | (ZERO, k)        ->  apply_cont(k, 1)
```

```
    | (O n, k)          ->  bin_exp x (O n, Y_EVEN(k) )
    | (I n, k)          ->  bin_exp x (I n, Y_ODD(x,k) )
and apply_cont = fun
    | (STOP, n)         ->  y
    | (Y_EVEN(k), y)    ->  apply_cont(k, y*y)
    | (Y_ODD(x,k), y)   ->  apply_cont(k, x*y*y)
```

La machine abstraite voulue se lit directement dans cette dernière écriture de notre fonction d'exponentiation : chaque ligne est une règle de transition.

4 *Explicit Pattern Calculus* : Vers un contrôle plus fin

Cette section va établir les limites de la description trop concise des motifs dynamiques qu'offre le *Pure Pattern Calculus*. On verra l'intérêt particulier qu'a la paresse dans le monde des structures de données, et pourquoi elle ne peut être décrite de façon raisonnable dans le précédent modèle.

On définira alors un nouveau calcul nommé *Explicit Pattern Calculus*, qui sera plus précis et permettra un contrôle beaucoup plus fin de l'évaluation. En effet, il rendra apparent tout le mécanisme de filtrage, de la comparaison des structures à la collecte des données.

4.1 L'infini et les limitations du noyau restreint

En λ-calcul, on a parfois tendance à considérer que les cas concrets où la paresse prend réellement le dessus sur le zèle sont suffisamment pathologiques pour être relativisés, ainsi que l'exemple donné en introduction avec une fonction divergente. C'est ainsi que des langages fonctionnels comme *Caml* ont choisi l'appel par valeur, qui apporte d'autres avantages pratiques comme une plus grande transparence dans l'utilisation de la mémoire.

Des structures infinies

En effet, le grand avantage de la paresse est d'apporter un comportement prudent face à l'infini, ce qu'on illustre en λ-calcul avec des programmes qui bouclent. Cependant utiliser de tels programmes n'est pas toujours un but en soi, ce qui rend plus modeste l'apport de cette stratégie.

Mais les structures de données apportent de nouvelles raisons de côtoyer volontairement l'infini. Prenons par exemple :

- Structures cycliques : certaines structures de données peuvent être finies mais contenir des cycles : réseaux pair-à-pair, structures avec pointeurs... Un parcours exhaustif d'une telle structure risque donc de durer très, très longtemps.
- Suites : on a également des structures qui sont toujours finies à un instant donné, mais qui peuvent être développées arbitrairement loin, pour lesquelles on peut toujours demander à consulter *l'élément suivant*.

Dans tous ces cas, on a une structure de données qui ne peut être évaluée/parcourue en un temps fini, la paresse est donc indispensable à sa manipulation. On notera d'ailleurs que dans *Caml* par exemple, des exceptions à la règle d'évaluation stricte sont faites dans certains de ces cas particuliers. Et ce genre d'exception est rare !

On va donc maintenant chercher à rendre le *Pure Pattern Calculus* plus prudent : il faudrait qu'une fonction par cas puisse prendre en argument une structure virtuellement infinie, et ne l'explore pas au-delà de ce qui est nécessaire à la décision du filtrage, ainsi qu'il est fait sur les motifs statiques dans le langage fonctionnel et paresseux Haskell [Has].

Inadéquation du filtrage implicite

Il faut faire des remarques sur l'opération de filtrage du *Pure Pattern Calculus*. Elle n'est définie à coup sûr que si à la fois :

- Le motif est réduit à une valeur, ce qui est facile à gérer.
- L'argument est *au moins aussi réduit que le motif.* Et cet énoncé assez vague est beaucoup plus problématique.

A ce stade, la réduction de l'argument devrait donc être paramétrée par le motif contre lequel on veut filtrer. Ce mécanisme forcerait alors la réintroduction d'une dépendance cyclique entre les définitions de la réduction et du filtrage.

On cherchera donc un mécanisme plus souple permettant d'imbriquer réduction et filtrage. Pour ceci, on ne peut les laisser à des niveaux distincts comme c'est le cas dans le *Pure Pattern Calculus* où la réduction est dans la syntaxe et le filtrage est *méta*. Il nous faudra donc remettre ces deux choses sur un pied d'égalité, en faisant apparaître de manière explicite le mécanisme de filtrage dans la syntaxe.

4.2 Internalisation du mécanisme de filtrage

On développe ici une nouvelle syntaxe, l'*Explicit Pattern Calculus*, qu'on pourra parfois abbrévier en *EPC*. La substitution restera une opération extérieure, mais le filtrage sera en revanche internalisé. On complète donc le tableau précédent :

	Filtrage implicite	Filtrage explicite
Substitution implicite	*PPC*	*EPC*
Substitution explicite		

Seront fournis des résultats de confluence et de simulation, signifiant respectivement “ce calcul a un sens” et “ce calcul est bien équivalent au précédent”.

4.2.1 Syntaxe

On va devoir introduire dans la syntaxe une nouvelle construction, représentant une opération de filtrage en cours. Les informations qu'on doit mémoriser pour réaliser un filtrage sont :

- L'ensemble θ des noms liés, pour différencier les constructeurs des variables de filtrage, puis pour effectuer la vérification finale sur le domaine de la substitution engendrée. On le gardera donc invariant jusqu'au bout.
- Un résultat partiel représentant ce qu'on a déjà calculé avec succès. Il s'agira d'un filtrat de domaine inclus dans θ, c'est-à-dire *échec* ou substitution de domaine inclus dans θ. Au cours du calcul, le domaine sera agrandi, ou le filtrat réduit à *échec*. On le note encore μ.
- Un ensemble de paires de termes restant à comparer, où à chaque fois la première composante vient du motif et la seconde de l'argument. Au cours du calcul, des paires pourront être modifiées, créées ou supprimées. On note l'ensemble c.

L'objet **filtrage** aura donc la forme :

$$\langle \theta | \mu | c \rangle$$

où θ est un lieur de la même manière que dans $[\theta]\ p \to s$. Le lien s'effectue selon le schéma suivant :

$$x\ \langle\ x\ |\ \mu\ |\ (\hat{x}, u) + c\ \rangle$$

Et voici la syntaxe étendue :

$$\begin{array}{rrcl}
\text{Termes} & t & ::= & x \mid \hat{x} \mid t\ t \mid [\theta]\ t \to t \mid t\ F \\
\text{Filtrages} & F & ::= & \langle \theta | \mu | c \rangle \\
\text{Fitrats} & \mu & ::= & \textit{échec} \mid \sigma \\
\text{Structures} & d & ::= & \hat{x} \mid d\ t \\
\text{Testables} & m & ::= & d \mid [\theta]\ t \to t
\end{array}$$

FIG. 3 – Règles locales de l'*Explicit Pattern Calculus*

Initialisation	$([\theta]\ p \to s)\ u$	$\to$	$s\,\langle\theta\vert\emptyset\vert\{(p,u)\}\rangle$	
Résolution	$s\,\langle\theta\vert\sigma\vert\emptyset\rangle$	$\to$	s^σ	si $dom(\sigma) = \theta$
	$s\,\langle\theta\vert\sigma\vert\emptyset\rangle$	$\to$	$s^\perp$	sinon
	$s\,\langle\theta\vert \acute{e}chec\vert c\rangle$	$\to$	$s^\perp$	
Filtrage	$\langle\theta\vert\mu\vert(\hat{x},u)+c\rangle$	$\to$	$\langle\theta\vert\mu\uplus\{x\mapsto u\}\vert c\rangle$	si $x\in\theta$
	$\langle\theta\vert\mu\vert(\hat{x},\hat{x})+c\rangle$	$\to$	$\langle\theta\vert\mu\vert c\rangle$	si $x\notin\theta$
	$\langle\theta\vert\mu\vert(p\ q,u\ v)+c\rangle$	$\to$	$\langle\theta\vert\mu\vert(p,u)+(q,v)+c\rangle$	si $(p\ q)$, $(u\ v)$ testables
	$\langle\theta\vert\mu\vert(p,u)+c\rangle$	$\to$	$\langle\theta\vert \acute{e}chec\vert c\rangle$	sinon si p et u testables

où s^σ dénote l'application de la substitution σ au terme s (*méta*-opération), $s^\perp$ le terme d'erreur, $+$ l'ajout d'un élément à un ensemble, et $\uplus$ est l'opération déjà vue d'union disjointe de filtrats. Pour assurer la confluence, on posera $t^\perp = \perp$ pour tout t, le terme $\perp$ pouvant être choisi arbitrairement parmi les termes purs. Par exemple $\perp = [x]\ \hat{x} \to x$ comme dans le *Pure Pattern Calculus* pour bénéficier de l'encodage de l'alternative entre plusieurs cas.

où c, pour "comparaisons", dénote un ensemble de paires de termes, et σ une substitution encore ainsi explicitée :

$$\sigma \equiv \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$$

On appelle **terme pur** un terme du *Pure Pattern Calculus*, ce qui correspond exactement aux termes qui ne contiennent aucun filtrage.

4.2.2 Dynamique

La dynamique de ce nouveau système, plus précise, va s'écrire en 8 règles données en figure 3, qu'on peut séparer en trois groupes :

- Règle d'initialisation : une règle unique, qui à partir d'une application introduit notre nouvel objet filtrage, déclenchant l'opération.
- Règles de résolution : trois règles achevant l'opération quand le filtrage a abouti, qui font systématiquement disparaître le terme de filtrage et appliquent au corps le traitement adapté. Elles correspondent dans le *Pure Pattern Calculus* au passage de $\{\!\{u/[\theta]\ p\}\!\}$ à $\{u/[\theta]\ p\}$, soit au test du domaine contre θ suivi de l'application de $\{u/[\theta]\ p\}$, maintenant encore atomique ou *méta*.
- Règles de filtrage : quatre règles contenant la mécanique interne du filtrage : décomposition de termes, association... On y retrouve les quatre équations définissant le filtrage implicite du *Pure Pattern Calculus*.

Ces règles peuvent encore s'appliquer dans n'importe quel contexte, jusque dans les filtrats en construction. On notera $\to_{epc}$ cette relation de réduction. De plus, on notera $\to_F$ la résolution par les seules quatre règles de filtrage et $\to_{FR}$ par les sept règles de filtrage et de résolution, encore une fois appliquées dans n'importe quel contexte.

On appelle filtrage soluble un filtrage qu'on peut résoudre par les seules règles de $\to_{FR}$, c'est-à-dire un $\langle\theta|\mu|c\rangle$ tel que $\langle\theta|\mu|c\rangle \to_{FR} \langle\theta|\mu'|\emptyset\rangle$ ou $\langle\theta|\mu|c\rangle \to_{FR} \langle\theta|\acute{e}chec|c'\rangle$

On doit pour finir étendre la spécification de la (*méta*)-substitution aux nouveaux termes, ce qu'on présente en figure 4.

4.3 Résultats

4.3.1 Confluence

On donne ici l'énoncé du résultat de confluence, ainsi que la méthode de preuve.

FIG. 4 – Spécification de l'opération de substitution.

$$
\begin{array}{rcll}
x_i^{\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}} & = & u_i & \\
x^\sigma & = & x & \text{si } x \notin dom(\sigma) \\
\hat{x}^\sigma & = & \hat{x} & \\
(t\ u)^\sigma & = & t^\sigma\ u^\sigma & \\
([\theta]\ p \to s)^\sigma & = & [\theta]\ p^\sigma \to s^\sigma & \text{si } \sigma \# \theta \\
t\,\langle \theta | \mu | c \rangle & = & t^\sigma\ \langle \theta | \mu[\sigma] | c^\sigma \rangle & \text{si } \sigma \# \theta
\end{array}
$$

où $\mu[\mu']$ dénote une composition syntaxique de filtrats ainsi définie :

$$
\begin{array}{rcl}
\textit{échec}[\mu] & = & \textit{échec} \\
\mu[\textit{échec}] & = & \textit{échec} \\
\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}[\sigma] & = & \{x_1 \mapsto u_1^\sigma, \ldots, x_n \mapsto u_n^\sigma\}
\end{array}
$$

et où la notation $\sigma \# \theta$ signifie traditionnellement "σ évite θ", autrement dit "σ et θ n'ont aucun nom de variable en commun".

Théorème 1. *Le calcul avec filtrage explicite est confluent.*

Autrement dit, pour tout terme t, si $t_1\ {}_{epc}^{*}\!\!\leftarrow t \to_{epc}^{*} t_2$, alors il existe un terme t_3 tel que $t_1 \to_{epc}^{*} t_3\ {}_{epc}^{*}\!\!\leftarrow t_2$.

On prouve ce théorème avec la méthode de réduction parallèle de TAIT et MARTIN-LÖF, présentée par exemple dans [Bar84]. Le plan est le suivant :

1. Définir une relation de réduction $\Rrightarrow$ telle que $\to_{epc} \subseteq \Rrightarrow \subseteq \to_{epc}^{*}$. On en déduit immédiatement que $\Rrightarrow^{*} = \to_{epc}^{*}$.
2. Montrer que $\Rrightarrow$ vérifie la *propriété du losange*.

Il convient de choisir $\Rrightarrow$ comme la contraction en parallèle de plusieurs radicaux présents au même instant dans un terme. La **propriété du losange** est une propriété bien plus forte que la confluence, mais généralement aussi bien plus simple à prouver quand elle est vérifiée. Elle s'énonce ainsi [BN98] :
Pour tout terme t, si $t_1 \Lleftarrow t \Rrightarrow t_2$, alors il existe un terme t_3 tel que $t_1 \Rrightarrow t_3 \Lleftarrow t_2$.

La différence avec la confluence est qu'à chaque fois (en hypothèse comme en conclusion) il n'y a plus qu'une étape de réduction.

Le détail de cette démonstration assez classique se trouve en annexe A.

4.3.2 Cohérence avec le *Pure Pattern Calculus*

On va énoncer ici deux résultats de simulation formalisant le lien entre le *Pure Pattern Calculus* et l'*Explicit Pattern Calculus*.

On aura besoin pour ceci d'un résultat sur le système de réduction $\to_{FR}$ (correspondant on le rappelle aux sept règles de filtrage et de résolution).

Lemme 1. $\to_{FR}$ *est une relation confluente et fortement normalisante.*

"Fortement normalisante" signifie qu'il n'existe pas de suite de réduction infinie.

Démonstration.
- On trouve facilement un ordre strict $\succ$ sur les termes, qui est bien fondé et vérifie : pour tous termes t et t', si $t \rightarrow_{FR} t'$ alors $t \succ t'$. Ceci assure la normalisation du système.
- On prouve la confluence par le lemme de NEWMAN [BN98]. Comme on possède déjà la propriété de normalisation, il suffit de vérifier que les paires critiques du système sont joignables. Le seul cas non trivial est le suivant : de $\langle\theta|\mu|(\hat{x}, u) + (\hat{x}, v) + c\rangle$ avec $x \in \theta$, on peut réduire en premier avec $(\hat{x}, u)$ et $(\hat{x}, v)$. Mais après réduction des deux le résultat sera $\langle\theta|\acute{e}chec|c\rangle$ par définition de l'union *disjointe* des filtrats.

□

Ce lemme assure immédiatement que tout terme t de l'*Explicit Pattern Calculus* admet une *unique forme normale* par $\rightarrow_{FR}$ [BN98]. On la notera $t\!\downarrow$.
On remarquera que $t\!\downarrow$ n'est pas forcément un terme pur : il se peut par exemple qu'un terme de filtrage ne soit pas testable, auquel cas le filtrage ne pourra pas être résolu par les seules règles $\rightarrow_{FR}$. On appellera terme **purifiable** un t tel que $t\!\downarrow$ est pur.

On énonce maintenant les deux résultats de simulation :

Théorème 2. *Si $t \rightarrow_{ppc} t'$, alors $t \rightarrow^*_{epc} t'$.*

Théorème 3. *Si $t \rightarrow_{epc} t'$ et $t\!\downarrow$ et $t'\!\downarrow$ sont des termes purs, alors $t\!\downarrow \rightarrow^*_{ppc} t'\!\downarrow$.*

Ces deux théorèmes sont prouvés en annexe A.
On peut voir que les hypothèses du théorème 3 sont relativement fortes : on demande à t et t' d'être purifiables. Ceci vient directement de la remarque suivante : on peut déclencher un filtrage avant qu'il ne soit soluble, et ainsi passer en une étape de réduction d'un terme pur à un terme non purifiable. Par exemple :

$$([y]\ x\ \hat{y} \rightarrow s)\ t \quad \rightarrow_{epc} \quad s\,\langle y|\emptyset|(x\ \hat{y}, t)\rangle$$

où la variable libre x dans le motif empêche l'évaluation immédiate du filtrage.

5 Application I : La paresse

La définition de stratégies d'évaluation paresseuses était la motivation originale de l'*Explicit Pattern Calculus*. On va donc récolter ici les premiers fruits en proposant une telle stratégie. On en décrira ensuite deux implémentations :
- Une nouvelle machine abstraite construite selon les mêmes principes que celle de l'évaluation stricte.
- La base d'un nouvel interpréteur pour le langage *Bondi*, qui rend ce dernier paresseux.

5.1 Stratégie paresseuse et machine abstraite

Dans le contexte purement fonctionnel du λ-calcul, la paresse se basait sur l'appel par nom : on ne fournissait à une fonction que les *noms* de ses arguments. Cette dernière exécutait alors uniquement son propre code, et n'allait chercher ce qui se cachait derrière les noms des arguments qu'en cas de nécessité.

Ici, nos fonctions étant des fonctions de filtrage, cette approche ne suffit pas : on est obligé de considérer *un peu* ce qui se cache derrière chaque nom pour pouvoir résoudre les filtrages. La paresse ne correspondra donc pas à un pur appel par nom mais plutôt à la maîtrise de l'exploration des arguments : il faudra procéder à leur évaluation uniquement à hauteur de ce qui est nécessaire. On se rapprochera ensuite du véritable appel par nom quand auront été associés des termes à chaque variable de filtrage.

Stratégie.
On va donc profiter de notre nouvelle maîtrise du mécanisme de filtrage pour intercaler des opérations *classiques* de réduction entre les pas successifs d'un filtrage. Pour évaluer un filtrage le plan suivra ces grandes lignes :

1. Déclencher le filtrage quels que soient le motif et l'argument.
2. Si le motif ou l'argument n'est pas testable, le réduire en tête jusqu'à le rendre testable. Rappelons que la notion de *testabilité* est une condition de *surface* : on ne regarde que la tête du terme.
3. Faire un pas de filtrage : déconstruction du constructeur de tête ou association d'un terme à une variable de filtrage par exemple.
4. Recommencer à l'étape 2 sur toutes les sous-structures engendrées.

On encode cette discipline dans la restriction suivantes des contextes de réduction, en rappelant que t désigne un terme général et d une structure, c'est-à-dire l'application d'un $\hat{x}$ à une suite de termes.

$$\begin{array}{rcl} C_p & ::= & [] \\ & | & C_p\ t \\ & | & t\ \langle \theta|\sigma|(C_p, t) + c\rangle \\ & | & t\ \langle \theta|\sigma|(d, C_p) + c\rangle \end{array}$$

On obtient une stratégie déterministe en ajoutant en plus une discipline pour gérer l'ensemble c, pour lequel on a pour l'instant simplement un opérateur de composition $+$ associatif et commutatif. Instancions par exemple c par une liste, et notons $x :: l$ l'ajout de x en tête de la liste l. On transforme les règles de filtrage en les suivantes :

$$\begin{array}{rcll} \langle \theta|\mu|(\hat{x}, u) :: c\rangle & \to & \langle \theta|\mu \uplus \{x \mapsto u\}|c\rangle & \text{si } x \in \theta \\ \langle \theta|\mu|(\hat{x}, \hat{x}) :: c\rangle & \to & \langle \theta|\mu|c\rangle & \text{si } x \notin \theta \\ \langle \theta|\mu|(p\ q, u\ v) :: c\rangle & \to & \langle \theta|\mu|(p, u) :: (q, v) :: c\rangle & \text{si } (p\ q) \text{ et } (u\ v) \text{ testables} \\ \langle \theta|\mu|(p, u) :: c\rangle & \to & \langle \theta|\textit{échec}|c\rangle & \text{autres cas avec } p \text{ et } u \text{ testables} \end{array}$$

On a ici une discipline de *pile* (*LIFO*), qui implémente une exploration *en profondeur d'abord* des motifs. On pourrait avoir une exploration *en largeur d'abord* avec une discipline de *file* (*FIFO*).

La stratégie ainsi proposée ne réduit les termes que jusqu'à la première forme testable rencontrée quoiqu'il arrive. Les formes testables sont en effet les **formes normales paresseuses**. Pour pousser l'évaluation jusqu'à son terme il faut combiner cette stratégie de réduction, pour le filtrage paresseux, avec une autre achevant l'évaluation.

On obtient alors finalement :

$$\begin{array}{rcl} C & ::= & [] \\ & | & C\ t \\ & | & dv\ C \\ & | & t\ \langle \theta|\sigma|(C_p, t) :: c\rangle \\ & | & t\ \langle \theta|\sigma|(d, C_p) :: c\rangle \end{array} \qquad \begin{array}{rcl} C_p & ::= & [] \\ & | & C_p\ t \\ & \times & \\ & | & t\ \langle \theta|\sigma|(C_p, t) :: c\rangle \\ & | & t\ \langle \theta|\sigma|(d, C_p) :: c\rangle \end{array}$$

Une nouvelle machine abstraite.
On applique successivement les transformations vues précédemment (*closure conversion*, transformation *CPS*, défonctionalisation) à l'évaluateur correspondant aux contextes de réduction précédents. La correction de la machine obtenue dérive encore une fois des résultats de correction de ces transformations, et le tout est présenté en annexe C.

5.2 Un langage paresseux avec motifs dynamiques

On va donner ici un aperçu de *Bondi* [Bon05], un langage de programmation expérimentant les motifs dynamiques, à l'évaluation stricte implémentée par un interpréteur. On décrira ensuite les transformations qui ont été apportées à cet interpréteur pour implémenter une évaluation paresseuse. Le nouveau code est situé en annexe C.

Bondi.
Ce langage est comparable à Caml [Cam] dans le sens où il est bâti sur une base fonctionnelle, mais augmenté de traits impératifs : séquences d'instructions, primitives de contrôle, références... et aussi de traits orientés objets : définitions de classes, appels de méthodes... Il admet également un typage statique fort avec un mécanisme de vérification/inférence de type. On rappelle que **typage statique** signifie que types sont vérifiés lors de la compilation d'un programme et non lors de son exécution, et que le **typage fort** est l'interdiction des conversions arbitraires d'un type vers un autre, appelées coercitions, qui correspondent à l'usage du mot clé `cast` dans le langage *C*. Son objectif est d'intégrer dans ce large contexte les polymorphismes de chemin et de motif, par l'intermédiaire des motifs dynamiques. Attention cependant, à cette heure la version officielle disponible en ligne ne supporte que les motifs statiques, il faut la dernière version en développement pour pouvoir observer les motifs dynamiques à l'œuvre.

Ce langage fonctionne en appel par valeur, et est implémenté par un interpréteur codé en *Caml*. Les étapes successives de l'interprétation sont les suivantes :

1. Analyse lexicale et syntaxique du code. La représentation intermédiaire produite est un type algébrique nommé `p_term` (`p` pour *parsing*).
2. Inférence de type. Pour bien cloisonner les différentes étapes, on a une nouvelle représentation, d'un autre type algébrique `i_term` (pour *inference*). Les différences entre `p_term` et `i_term` sont cependant extrêmement minimes à l'heure actuelle : différences sur les annotations de types.
3. Évaluation de l'`i_term` produit à l'étape précédente. Le résultat est d'un dernier type `value`.

Un point important sur les représentations des données est que l'évaluateur ne manipule pas directement les valeurs : on a un environnement global contenant des valeurs et représentant une grande mémoire, et on manipule essentiellement des adresses de cette mémoire. Autrement dit, on manipule en réalité des pointeurs vers les valeurs qu'on croit avoir entre les mains.

Modifications apportées.
Pour obtenir un interpréteur véritablement paresseux, deux choses sont à faire :

- Implémenter l'évaluateur de notre stratégie paresseuse. Cela peut se faire sans modification des notions d'`i_term` ou de `value` ni modification du type de l'évaluateur. En effet, pour représenter les fonctions le type `value` contient déjà des clôtures, c'est-à-dire des termes quelconques accompagnés d'un environnement. On se servira de ces clôtures pour représenter des termes seulement partiellement évalués comme ceux que l'on manipule au cours d'un filtrage.
- Implémenter un mécanisme de partage évitant qu'un terme dupliqué par l'appel par nom n'ait à être réduit plusieurs fois. Cela se fait facilement ici, grâce au choix fait par l'interpréteur original de manipuler des pointeurs sur les valeurs plutôt que les valeurs elles-mêmes.

6 Application II : Structurer les données abstraites

On va maintenant profiter du mécanisme de filtrage explicite développé dans la section 4 pour proposer un début de solution à un problème intrinsèque au filtrage : l'inadéquation entre les données *filtrables* dont on doit connaître la représentation exacte et les données des situations

réelles qui sont souvent *abstraites* : on on a des moyens limités pour les manipuler, et on ne les connaît que par leur comportement.

En quelque sorte, on doit pour filtrer avoir des structures de données *explicites*, mais on peut aussi avoir de bonnes raison d'utiliser des structures *implicites*. Le parallèle avec les opérations explicites ou implicites semble assez profond, et permet de mieux comprendre certains enjeux de part et d'autre.

6.1 Dilemme de programmeur : élégance contre efficacité

Ainsi, la notion de filtrage n'existe classiquement que quand on a des structures de données représentées explicitement par des constructeurs, et construites en suivant une règle fixée définitivement. On a par exemple les *types algébriques* en *Caml* [Cam] ou *inductifs* en Coq [Coq07] [PPM90]. Si les motifs dynamiques et en particulier le polymorphisme de motif ont permis de desserrer cette dernière condition, nous sommes toujours incapables d'effectuer un filtrage sur une structure dont les constructeurs nous sont inaccessibles.

Or, dans un contexte réel de programmation, des considérations d'efficacité nous font faire un usage intensif des *types abstraits* : des données dont nous ne connaissons pas la représentation, mais que nous pouvons manipuler à l'aide d'un certain nombre de primitives. Ces données abstraites apportent le grand avantage de la transparence à l'implémentation : nous utilisons des primitives dont les effets et résultats sont clairement spécifiés, mais la manière dont le tout est implémenté n'a aucune importance. En particulier, cette implémentation cachée peut être modifiée sans crainte d'altérer les programmes y faisant appel. En revanche, sur de telles données à la structure inconnue, la définition de fonctions par cas ne peut se faire que par une suite explicite et éventuellement laborieuse de tests divers et variés sur le contenu de la donnée.

Un exemple simple de ce conflit d'intérêts est celui des nombres entiers : ils sont dans les langages raisonnables des entités abstraites, servies avec quelques primitives arithmétiques. Ceci permet de les représenter par les entiers machine et d'utiliser les opérations arithmétiques câblées dans le processeur : l'efficacité est maximale.

L'utilisation de ces entiers dans des fonctions définies par cas ne peut donc que passer par des tests. L'algorithme d'exponentiation rapide, qu'on a montré précédemment avec une fonction définie par trois cas sur une représentation binaire explicite des entiers naturels, s'écrit donc dans un contexte plus commun :

```
let rec exp x n =
    if n=0
    then 1
    else if n mod 2 = 0
         then let y = exp x (n/2)
              in y*y
         else (* n = 2k+1 *)
              let y = exp x (n/2)
              in x*y*y
```

Dans cette seconde version, la fonction est bien plus efficace, notamment par son utilisation de l'opération de multiplication câblée dans le processeur. Dans la première version, on avait passé sous silence le coût caché de la multiplication (qu'il fallait d'ailleurs réimplémenter manuellement) mais il n'était hélàs pas négligeable.
En revanche, avec ses imbrications de `if ... then ... else ...`, même correctement indenté ce nouveau code est plus difficile à écrire ou lire, et bien sûr la différence entre les deux styles devient de plus en plus marquée à mesure que les structures et les cas deviennent moins triviaux.

Comment concilier alors l'efficacité d'un type de données abtrait avec la commodité d'un type explicitement contruit ? Phil Wadler propose, quand on a de façon générale plusieurs représentations possibles d'un type de données, d'introduire des fonctions bijectives de conversion entre

les différentes représentations [Wad87]. Chaque couple de bijections réciproques étant appelé une **vue**. Ainsi, on peut à loisir utiliser la représentation la plus adaptée à chaque situation.

Pour ne pas convertir complètement un entier binaire en un entier machine et inversement à la moindre occasion, ce qui serait une opération assez coûteuse qu'on préfèrerait éviter, on peut procéder alors en suivant cette idée :

- La représentation *normale* des entiers est abstraite : implémentée par les entiers machine par exemple.
- On donne une fonction de conversion partielle qui met en place les constructeurs *de surface*.

Ainsi, en notant `int` le type des entiers, on donne une nouvelle définition pour les entiers binaires :

```
type bin_int2 = ZERO
              | O of int
              | I of int
```

On a ainsi un constructeur en surface, et un vrai entier efficace à l'intérieur. La fonction de conversion, dont un sens suffira ici, est :

```
let int2bin n =
    if n = 0
    then ZERO
    else if n mod 2 = 0
         then O (n/2)
         else I (n/2)
```

On retrouve ici les conditionnelles imbriquées qu'on avait dans le filtrage précédent. Mais elles se retrouvent écrites ici une bonne fois pour toute et n'auront plus jamais à être recopiées.

L'exemple précédent peut donc se récrire de façon élégante *et* efficace :

```
let rec bin_exp2 x n =
    match int2bin n with
    | ZERO  -> 1
    | O m   -> let y = bin_exp2 x m
               in y*y
    | I m   -> let y = bin_exp2 x m
               in x*y*y
```

On a ici ajouté le coût d'une conversion de surface à chaque appel récursif, mais toutes les multiplications se font sur des entiers machine, ce qui est un gain considérable.

Le mécanisme illustré ici n'est cependant pas encore le plus pratique : il faut prévoir à l'avance la profondeur à laquelle on a besoin de convertir nos données (que ce serait-il passé si un motif avait été `I(O m)` ?). En l'état, cette solution est donc inapplicable dans un cadre où les motifs sont dynamiques, car on ne sait pas à l'avance la *quantité de structure* qui va être testée.

6.2 Une ouverture

L'enjeu est donc maintenant de permettre à une fonction de conversion partielle comme `int2bin` d'évoluer en fonction des motifs à côté desquels elle s'utilisera. Il faudra par exemple qu'`int2bin` s'applique une fois à l'argument quand on filtre contre le motif `O m`, mais trois fois si on filtre à la place contre `I (O ZERO)`. Notons qu'en réalité le système de types de *Caml* gêne l'application récursive naïve d'`int2bin`, mais on pourra oublier ceci pour notre exemple. En effet, seuls des formalismes non typés ont été présentés dans ce rapport.

On proposera comme piste de faire porter la conversion partielle par le motif. Ce groupement d'un motif `p` et d'une conversion `f` serait alors appelé une **vue**, pourrait recevoir une notation comme par exemple `<<f ;p>>`, et serait utilisé comme un motif. On écrirait alors l'exemple précédent de cette nouvelle manière, où chaque motif porte la conversion à appliquer à l'argument :

```
let rec bin_exp2 x       = fun
    | <<int2bin; ZERO>>  -> 1
    | <<int2bin; O m>>   -> let y = exp_bin2 x m
                            in y*y
    | <<int2bin; I m>>   -> let y = exp_bin2 x m
                            in x*y*y
```

Plus précisément, pour que la conversion suive exactement les évolutions du motif, on ne la présentera pas comme une grosse fonction intégralement écrite à côté du motif. On la décrira plutôt indirectement, en la répartissant dans tout le motif : par exemple, certains constucteurs porteront la petite conversion dont ils ont besoin et rien d'autre. La conversion associée à un motif sera donc en réalité la combinaison des petites conversions présentes à l'intérieur du motif. Cette conversion globale évoluera donc naturellement avec le motif.
Par exemple, pour filtrer un argument comme s'il s'agissait d'un arbre binaire représenté comme dans les exemples de l'introduction, le motif `Node (Data x) y` pourrait être remplacé par un autre de la forme :

```
<< f ; Node <<g;Data x>> <<h;y>> >>
```

où `f`, `g` et `h` sont trois conversions locales, qui ainsi combinées correspondent à une conversion qu'on pourrait écrire comme suit :

```
let conversion a = match (f a) with Node b c -> Node (g b) (h c)
```

Il faut maintenant pouvoir faire en sorte que la conversion globale à laquelle nous n'avons pas directement accès soit appliquée au moment opportun, si possible sans avoir à la calculer explicitement comme nous venons de le faire sur un exemple. L'idée serait donc, à chaque fois qu'on a une vue dans le motif, d'appliquer la conversion locale au morceau correspondant de l'argument. À nouveau, il va donc falloir introduire des calculs annexes entre les différentes étapes d'un filtrage, ce qui va faire encore une fois appel au filtrage explicite.

On pourrait donc ajouter à l'*Explicit Pattern Calculus* une règle de la forme :

$$\langle\theta|\mu|(<< f;p >>, u) + c\rangle \quad\rightarrow\quad \langle\theta|\mu|(p, f\ u) + c\rangle$$

Ainsi, quand à l'intérieur d'un filtrage on rencontre une vue, on s'interrompt le temps d'appliquer la conversion à l'argument local (ici u) et de calculer le résultat, puis on peut reprendre le filtrage proprement dit.

Cependant, cette intuition n'est pas applicable aussi brutalement. Quand on rencontre une construction $<< f;p >>$, p est bien *à sa place* mais f en revanche est un programme destiné à être *donné* à un interlocuteur inconnu et exécuté dans un *contexte différent*. Ceci génère naturellement des problèmes au niveau des variables, qu'on ne sait plus à quoi lier.
Une première approche a donc été explorée, dans laquelle les conversions sont extrêmement contraintes, avec l'interdiction des variables libres notamment. Cependant, dans le formalisme actuel les variables de filtrage libres sont les constructeurs ($\hat{x}$), et ne peuvent donc pas être grossièrement interdites. Un certain travail reste donc à faire avant que ce système ne soit viable et véritablement intégré, qui pourra demander de nouveaux aménagements du formalisme.

7 Bilan et perspectives

Partant du *Pure Pattern Calculus*, un formalisme concis offrant des possibilités avancées de filtrage, un premier langage de programmation utilisant ces possibilités a vu le jour, *Bondi*. Nous avons pu ici formaliser la base de la stratégie d'évaluation stricte mise en œuvre par ce langage. Au-delà, nous nous sommes avancés vers une variante paresseuse de ce langage, en partant de ses bases

théoriques mêmes. On aura vérifié que le formalisme actuel ne suffisait pas à décrire formellement une telle variante, et il aura donc fallu proposer une extension de la théorie : l'*Explicit Pattern Calculus*. Ce nouveau formalisme, qui est capable de décrire lui-même son mécanisme de filtrage et donc de le contrôler finement, a été prouvé confluent. Ont également été précisées les conditions dans lesquelles le *Pure-* et l'*Explicit- Pattern Calculus* étaient équivalents.

À partir de cette nouvelle fondation théorique, une stratégie paresseuse a pu être définie, puis implémentée. La variante désirée de *Bondi* capable de manipuler des structures de données infinies est donc en train de voir le jour ! Également, ont été construites des machines abstraites, permettant de faire un pas supplémentaire vers l'implémentation réelle de ces langages, en attendant d'arriver à de vrais compilateurs.

Cette approche transversale, à la fois théorique et pratique, a donc permis à la fois de définir un formalisme général et de vérifier sa validité de plusieurs manières : théoriquement par l'énoncé et la preuve de théorèmes bannissant les comportements indésirables, et en pratique par des implémentations expérimentales concrétisant la réalisation des objectifs.

Cette approche à plusieurs niveaux des mécanismes de filtrage peut encore être poussée plus loin. Un niveau supplémentaire notamment pourrait être ajouté avec l'étape ultime de l'implémentation : la définition et la création d'un compilateur pour un langage de programmation fonctionnel avec motifs dynamiques. Et aussi en poussant plus loin les questions posées ici : il faudra notamment clarifier les statuts des variables de filtrage avant de pouvoir, côté théorique trouver une réponse satisfaisante au mécanisme des vues, et côté pratique concrétiser de puissants outils de filtrages permettant de traiter des données abstraites.

Enfin, les aspects de typage ont été complètement éludés dans ce document. Deux approches récentes et non publiées, respectivement de Barry Jay et Alexandre Miquel existent. La première notamment, dans cette double approche théorique et pratique, tend vers le typage effectif du langage *Bondi*. Ces deux voies restent à approfondir et forment un troisième défi à relever dans la continuité de ce document.

Remerciements

Je tiens à remercier Delia Kesner pour son implication dans ce travail. Merci également à Alexandre Miquel, Barry Jay et Olivier Danvy pour ce qu'ils ont apporté d'ouverture et de suggestions.

Références

[ABDM03] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. March 2003.

[Bar84] Henk Barendregt. *The Lambda Calulus : Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bon05] Bondi programming language, 2005. `http ://www-staff.it.uts.edu.au/~cbj/bondi`.

[Cam] The objective caml language. `http ://caml.inria.fr/`.

[CK98] Horatiu Cirstea and Claude Kirchner. ρ-calculus, the rewriting calculus. *5th International Workshop on Constraints in Computational Logics (CCL)*, 1998.

[Coq07] The coq development team logical project. the coq proof assistant reference manual version 8.1, 2007.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control, a study of the cps transformation. *Mathematical Structures in Computer Science*, 4(2) :361–391, 1992.

[FF86] Matthias Felleisen and Matthew Flatt. Control operators, the secd machine, and the λ-calculus. *Formal Description of Programming Concepts III*, pages 193–217, 1986.

[Has] The haskell language. `http ://www.haskell.org/`.

[JK06a] Barry Jay and Delia Kesner. Pure pattern calculus. *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006*, pages 100–114, 2006. Available as `www-staff.it.uts.edu.au/~cbj/Publications/purepatterns.pdf`.

[JK06b] C. Barry Jay and Delia Kesner. Patterns as first-class citizens. 2006. Available as `http ://hal.archives-ouvertes.fr/hal-00229331/fr/`.

[JK08] Barry Jay and Delia Kesner. First-class patterns. *To appear*, 2008.

[Jon87] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Kes07] Delia Kesner. The theory of explicit substitutions revisited. *Proceedings of the 16th EACSL Annual Conference on Computer Science and Logic (CSL)*, pages 238–252, 2007.

[Kri05] Jean-Louis Krivine. Un interprète du λ-calcul. brouillon, 2005. Available online at `http ://www.logique.jussieu.fr/~krivine`.

[KvOdV08] Jan-Willem Klop, Vincent van Oostrom, and Roel de Vrijer. Lambda calculus with patterns. *Theoretical Computer Science*, (Coppo, Dezani, and Ronchi Festschrift (To appear)), 2008.

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

[PPM90] Franck Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. *Proceedings of the fifth international conference on Mathematical foundations of programming semantics*, pages 209–226, 1990.

[Rey98] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 4(11) :363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[vO90] Vincent van Oostrom. Lambda calculus with patterns. Technical report ir-228, Vrije Universiteit, Amsterdam, 1990.

[Wad87] Philip Wadler. Views : a way for pattern matching to cohabit with data abstraction. *14th ACM Symposium on Principles of Programming Languages*, January 1987.

Fig. 5 – Définition de $\rightarrow_{ppc}$.

$$([\theta]\ p \rightarrow s)\ u \rightarrow_{ppc} s^{\{u/[\theta]\ p\}} \qquad \frac{t \rightarrow_{ppc} t'}{t\ u \rightarrow_{ppc} t'\ u} \qquad \frac{p \rightarrow_{ppc} p'}{[\theta]\ p \rightarrow s \rightarrow_{ppc} [\theta]\ p' \rightarrow s}$$

$$\frac{u \rightarrow_{ppc} u'}{t\ u \rightarrow_{ppc} t\ u'} \qquad \frac{s \rightarrow_{ppc} s'}{[\theta]\ p \rightarrow s \rightarrow_{ppc} [\theta]\ p \rightarrow s'}$$

Fig. 6 – Définition de $\rightarrow_{epc}$.

$$([\theta]\ p \rightarrow s)\ u \rightarrow_{epc} s\ \langle \theta | \emptyset | (p,u) \rangle$$

$$\frac{dom(\sigma) = \theta}{s\ \langle \theta | \sigma | \emptyset \rangle \rightarrow_{epc} s^{\sigma}} \qquad \frac{dom(\sigma) \neq \theta}{s\ \langle \theta | \sigma | \emptyset \rangle \rightarrow_{epc} s^{\perp}} \qquad s\ \langle \theta | \acute{e}chec | \emptyset \rangle \rightarrow_{epc} s^{\perp}$$

$$\frac{x \in \theta}{\langle \theta | \mu | (\hat{x}, u) + c \rangle \rightarrow_{epc} \langle \theta | \mu \uplus \{x \mapsto u\} | c \rangle} \qquad \frac{x \notin \theta}{\langle \theta | \mu | (\hat{x}, \hat{x}) + c \rangle \rightarrow_{epc} \langle \theta | \mu | c \rangle}$$

$$\frac{(p\ q) \text{ et } (u\ v) \text{ testables}}{\langle \theta | \mu | (p\ q, u\ v) + c \rangle \rightarrow_{epc} \langle \theta | \mu | (p,u) + (q,v) + c \rangle} \qquad \frac{\text{autres cas avec } p \text{ et } u \text{ testables}}{\langle \theta | \mu | (p,u) + c \rangle \rightarrow_{epc} \langle \theta | \acute{e}chec | c \rangle}$$

$$\frac{t \rightarrow_{epc} t'}{t\ u \rightarrow_{epc} t'\ u} \qquad \frac{p \rightarrow_{epc} p'}{[\theta]\ p \rightarrow s \rightarrow_{epc} [\theta]\ p' \rightarrow s} \qquad \frac{s \rightarrow_{epc} s'}{s\ F \rightarrow_{epc} s'\ F}$$

$$\frac{u \rightarrow_{epc} u'}{t\ u \rightarrow_{epc} t\ u'} \qquad \frac{s \rightarrow_{epc} s'}{[\theta]\ p \rightarrow s \rightarrow_{epc} [\theta]\ p \rightarrow s'} \qquad \frac{F \rightarrow_{epc} F'}{s\ F \rightarrow_{epc} s\ F'}$$

$$\frac{u_i \rightarrow_{epc} u_i'}{\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\} \rightarrow_{epc} \{x_1 \mapsto u_1, \ldots, x_i \mapsto u_i', \ldots, x_n \mapsto u_n\}}$$

$$\frac{p \rightarrow_{epc} p'}{\langle \theta | \mu | (p,u) + c \rangle \rightarrow_{epc} \langle \theta | \mu | (p',u) + c \rangle} \qquad \frac{u \rightarrow_{epc} u'}{\langle \theta | \mu | (p,u) + c \rangle \rightarrow_{epc} \langle \theta | \mu | (p,u') + c \rangle}$$

A Preuves détaillées

Relations de réduction

Dans les preuves à venir, on voudra parfois raisonner *par récurrence sur la relation de réduction.* On donne donc plus formellement les définitions de $\rightarrow_{ppc}$ et $\rightarrow_{epc}$ dans les figures 5 et 6.

Confluence

On prouve ici la confluence de la relation de réduction $\rightarrow_{epc}$. On rappelle que $\rightarrow_{epc}$ est définie par les huit règles suivantes appliquées dans des contextes quelconques :

Initialisation	$([\theta]\ p \rightarrow s)\ u$	$\rightarrow$	$s\ \langle \theta \mid \emptyset \mid \{(p,u)\} \rangle$	
Résolution	$s\ \langle \theta \mid \sigma \mid \emptyset \rangle$	$\rightarrow$	s^{σ}	si $dom(\sigma) = \theta$
	$s\ \langle \theta \mid \sigma \mid \emptyset \rangle$	$\rightarrow$	$s^{\perp}$	sinon
	$s\ \langle \theta \mid \acute{e}chec \mid c \rangle$	$\rightarrow$	$s^{\perp}$	
Filtrage	$\langle \theta \mid \mu \mid (\hat{x}, u) + c \rangle$	$\rightarrow$	$\langle \theta \mid \mu \uplus \{x \mapsto u\} \mid c \rangle$	si $x \in \theta$
	$\langle \theta \mid \mu \mid (\hat{x}, \hat{x}) + c \rangle$	$\rightarrow$	$\langle \theta \mid \mu \mid c \rangle$	si $x \notin \theta$
	$\langle \theta \mid \mu \mid (p\ q, u\ v) + c \rangle$	$\rightarrow$	$\langle \theta \mid \mu \mid (p,u) + (q,v) + c \rangle$	si $(p\ q)$, $(u\ v)$ testables
	$\langle \theta \mid \mu \mid (p,u) + c \rangle$	$\rightarrow$	$\langle \theta \mid \acute{e}chec \mid c \rangle$	sinon si p et u testables

On prouve ce théorème avec la méthode de réduction parallèle de Tait et Martin-Löf, présentée par exemple dans [Bar84]. Le plan est le suivant :

1. Définir une relation de réduction $\Rrightarrow$ telle que $\rightarrow_{epc} \subseteq \Rrightarrow \subseteq \rightarrow^*_{epc}$. On en déduit immédiatement que $\Rrightarrow^* = \rightarrow^*_{epc}$.
2. Montrer que $\Rrightarrow$ vérifie la *propriété du losange.*

Réduction parallèle

Pour définir $\Rrightarrow$, on regarde trois groupes de règles, *id*, *cgr* et *red*, qui correspondent respectivement au cas de base, au passage au contexte et à l'application d'une réduction de $\rightarrow_{epc}$. Ces règles sont détaillées par la table 7.

Lemme 2. $\rightarrow_{epc} \subset \Rrightarrow$.

Démonstration. Par induction sur $\rightarrow_{epc}$: chaque règle de $\rightarrow_{epc}$ a son pendant dans $\Rrightarrow$, de plus $\Rrightarrow$ s'étend à tout contexte. □

Lemme 3. *L'ensemble des termes testables est stable par* $\rightarrow_{epc}$.

Démonstration. Par induction sur la grammaire des termes testables. □

Lemme 4. $\Rrightarrow \subset \rightarrow^*_{epc}$

Démonstration. Par récurrence sur la dérivation de $\Rrightarrow$.

- Les cas *id* et *cgr* sont immédiats.
- Les règles *red* sont immédiates (avec le lemme 3 pour certaines), exceptée la suivante :

$$\frac{t \Rrightarrow t' \qquad \sigma \Rrightarrow \sigma'}{t\,\langle\theta|\sigma|\emptyset\rangle \Rrightarrow t'^{\sigma'}}\ dom(\sigma) = \theta$$

  Dans ce cas, l'hypothèse de récurrence donne $t \rightarrow^*_{epc} t'$ et $\sigma \rightarrow^*_{epc} \sigma'$, d'où :

$$\begin{array}{lll} t\,\langle\theta|\sigma|\emptyset\rangle & \rightarrow^*_{epc} & t'\,\langle\theta|\sigma|\emptyset\rangle \\ & \rightarrow^*_{epc} & t'\,\langle\theta|\sigma'|\emptyset\rangle \\ & \rightarrow_{epc} & t'^{\sigma'} \end{array}$$

□

Propriété du losange

Lemme 5. *Soient σ et τ deux substitutions vérifiant $\sigma\#dom(\tau)$. Alors pour tout terme t,*

$$(t^{\tau})^{\sigma} = (t^{\sigma})^{(\tau[\sigma])}$$

Démonstration. Par examen des variables libres de t et des substitutions. □

Lemme 6. *Pour tout terme t et toute substitution σ, si $t \Rrightarrow t'$ et $\sigma \Rrightarrow \sigma'$ alors $t^{\sigma} \Rrightarrow t'^{\sigma'}$.*

Démonstration. Par récurrence sur la dérivation de $t \Rrightarrow t'$.

- *id* : immédiat.
- *cgr* : par définition, la substitution se distribue, on applique alors l'hypothèse de récurrence sur les sous-termes et on refactorise. Par exemple : $(t\ u) \Rrightarrow (t'\ u')$,

$$(t\ u)^{\sigma} = t^{\sigma}\ u^{\sigma} \Rrightarrow t'^{\sigma'}\ u'^{\sigma'} = (t'\ u')^{\sigma'}$$

  car $t^{\sigma} \Rrightarrow t'^{\sigma'}$ et $u^{\sigma} \Rrightarrow u'^{\sigma'}$.

FIG. 7 – Réduction parallèle $\Rrightarrow$

id		$t \Rrightarrow t \qquad c \Rrightarrow c \qquad \mu \Rrightarrow \mu$
cgr	Termes purs	$\dfrac{t \Rrightarrow t' \qquad u \Rrightarrow u'}{t\ u \Rrightarrow t'\ u'} \qquad \dfrac{p \Rrightarrow p' \qquad s \Rrightarrow s'}{[\theta]\ p \to s \Rrightarrow [\theta]\ p' \to s'}$
	Filtrages	$\dfrac{t \Rrightarrow t' \qquad F \Rrightarrow F'}{t\ F \Rrightarrow t'\ F'} \qquad \dfrac{\mu \Rrightarrow \mu' \qquad c \Rrightarrow c'}{\langle\theta\|\mu\|c\rangle \Rrightarrow \langle\theta\|\mu'\|c'\rangle}$ $\dfrac{p \Rrightarrow p' \qquad u \Rrightarrow u' \qquad c \Rrightarrow c'}{(p,u) + c \Rrightarrow (p',u') + c'} \qquad \dfrac{\mu \Rrightarrow \mu' \qquad u \Rrightarrow u'}{\mu \uplus \{x \mapsto u\} \Rrightarrow \mu' \uplus \{x \mapsto u'\}}$
red	Initialisation	$\dfrac{p \Rrightarrow p' \qquad s \Rrightarrow s' \qquad u \Rrightarrow u'}{([\theta]\ p \to s)\ u \Rrightarrow s'\ \langle\theta\|\epsilon\|\{(p',u')\}\rangle}$
	Résolution	$\dfrac{t \Rrightarrow t' \qquad \sigma \Rrightarrow \sigma'}{t\ \langle\theta\|\sigma\|\emptyset\rangle \Rrightarrow t'^{\sigma'}}\ dom(\sigma) = \theta \qquad \dfrac{t \Rrightarrow t'}{t\ \langle\theta\|\sigma\|\emptyset\rangle \Rrightarrow t'^{\perp}}\ dom(\sigma) \neq \theta$ $\dfrac{t \Rrightarrow t'}{t\ \langle\theta\|\acute{e}chec\|c\rangle \Rrightarrow t'^{\perp}}$
	Filtrage	$\dfrac{\mu \Rrightarrow \mu' \qquad u \Rrightarrow u' \qquad c \Rrightarrow c'}{\langle\theta\|\mu\|(\hat{x},u) + c\rangle \Rrightarrow \langle\theta\|\mu' \uplus \{x \mapsto u'\}\|c'\rangle}\ x \in \theta$ $\dfrac{\mu \Rrightarrow \mu' \qquad c \Rrightarrow c'}{\langle\theta\|\mu\|(\hat{x},\hat{x}) + c\rangle \Rrightarrow \langle\theta\|\mu'\|c'\rangle}\ x \notin \theta$ $\dfrac{\mu \Rrightarrow \mu' \quad c \Rrightarrow c' \qquad p \Rrightarrow p'\ q' \qquad u \Rrightarrow u'\ v'}{\langle\theta\|\mu\|(p,u) + c\rangle \Rrightarrow \langle\theta\|\mu'\|(p',u') + (q',v') + c'\rangle}\ p \text{ et } u \text{ testables}$ $\dfrac{c \Rrightarrow c'}{\langle\theta\|\mu\|(p,u) + c\rangle \Rrightarrow \langle\theta\|\acute{e}chec\|c'\rangle}$ autres cas avec p et u testables

- *red* : seul un cas mérite un nouveau traitement, celui de la réduction $t\,\langle\theta|\tau|\emptyset\rangle \Rrightarrow t'^{\tau'}$. On veut donc $(t\,\langle\theta|\tau|\emptyset\rangle)^{\sigma} \Rrightarrow (t'^{\tau'})^{\sigma'}$. Or on a

$$(t\,\langle\theta|\tau|\emptyset\rangle)^{\sigma} = t^{\sigma}\,\langle\theta|\tau[\sigma]|\emptyset\rangle$$

  On écrit $\tau = \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$ et $\tau' = \{x_1 \mapsto u'_1, \ldots, x_n \mapsto u'_n\}$. Par hypothèse de récurrence, pour tout i, $u_i^{\sigma} \Rrightarrow u_i'^{\sigma'}$, donc $\tau[\sigma] \Rrightarrow \tau'[\sigma']$. De plus $t^{\sigma} \Rrightarrow t'^{\sigma'}$. On obtient alors avec une règle *red* :

$$t^{\sigma}\,\langle\theta|\tau[\sigma]|\emptyset\rangle \Rrightarrow (t'^{\sigma'})^{(\tau'[\sigma'])}$$

  Or, par le lemme de composition de substitutions 5,

$$(t'^{\sigma'})^{(\tau'[\sigma'])} = (t'^{\tau'})^{\sigma'}$$

□

Lemme 7. $\Rrightarrow$ *a la propriété du losange.*

Démonstration. Supposons $t_1 \Lleftarrow t \Rrightarrow t_2$. On raisonne encore sur les dérivations de $t_1 \Lleftarrow t$ et $t \Rrightarrow t_2$.

- si un côté se réduit par *id*, c'est évident.
- si les deux côtés se réduisent par *cgr*, la récurrence s'applique immédiatement.
- si un côté se réduit par *cgr* et l'autre par *red*, on discute le cas suivant :

$$t\,\langle\theta|\sigma|\emptyset\rangle$$

$$t_1\,\langle\theta|\sigma_1|\emptyset\rangle \Lleftarrow \qquad \Rrightarrow (t_2)^{\sigma_2}$$

  Par hypothèse de récurrence, on a $t_1 \Rrightarrow t_3 \Lleftarrow t_2$ et $\sigma_1 \Rrightarrow \sigma_3 \Lleftarrow \sigma_2$. On a alors le diagramme suivant :

$$t_1\,\langle\theta|\sigma_1|\emptyset\rangle \qquad\qquad (t_2)^{\sigma_2}$$

$$\Rrightarrow (t_3)^{\sigma_3} \Lleftarrow$$

  où la flèche de gauche est une règle *red* et la flèche de droite vient du lemme 6.
  Aucun rédex ne pouvant être masqué par $\Rrightarrow$ (avec le lemme 3), les autres cas ne posent aucun problème particulier.
- si les deux côtés se réduisent par *red*, on a une limitation sur la forme de t.
  Dans le cas $([\theta]\ p \to s)\ u$, les deux flèches utilisent la même règle et la conclusion est immédiate.
  Dans le cas $t\,\langle\theta|\mu|c\rangle$, de nombreuses combinaisons sont possibles, mais remarquons que deux réductions modifiant le domaine de μ soit sont indépendantes (concernent des variables de θ différentes) soit réduisent μ sur *échec*. De même, si un côté réduit $t\,\langle\theta|\mu|c\rangle$ sur $t^{\perp}$, cette opération est toujours possible depuis l'autre côté, le rédex incriminé ne pouvant pas disparaître. Le seul cas difficile est :

$$t\,\langle\theta|\sigma|\emptyset\rangle$$

$$(t_1)^{\sigma_1} \Lleftarrow \qquad \Rrightarrow (t_2)^{\sigma_2}$$

Mais par récurrence, $t_1 \Rrightarrow t_3 \Lleftarrow t_2$ et $\sigma_1 \Rrightarrow \sigma_3 \Lleftarrow \sigma_2$, et le lemme 6 permet de fermer le diagramme :

$$\begin{array}{ccc} (t_1)^{\sigma_1} & & (t_2)^{\sigma_2} \\ & \searrow\!\!\searrow\!\!\searrow \quad \swarrow\!\!\swarrow\!\!\swarrow & \\ & (t_3)^{\sigma_3} & \end{array}$$

□

On peut enfin poser la conclusion : comme $\rightarrow_{epc} \subset \Rrightarrow \subset \rightarrow_{epc}^*$, on a $\Rrightarrow^* = \rightarrow_{epc}^*$. Or $\Rrightarrow$ a la propriété du losange, donc $\Rrightarrow^*$ et $\rightarrow_{epc}^*$ également. Donc $\rightarrow_{epc}$ est confluente.

Simulations

Simulation par l'*Explicit Pattern Calculus*

On prouve ici le théorème 2 :

$$\text{Si } t \to_{ppc} t', \text{ alors } t \to^*_{epc} t'.$$

La preuve se fait par induction sur $t \to_{ppc} t'$ avec le lemme suivant :

Lemme 8. *Soient p et u deux termes purs.*

1. *Si $\{\!\{u/[\theta]\ p\}\!\} = \sigma$, alors pour tous μ et c, $\langle\theta|\mu|(p,u)+c\rangle \to^*_{FR} \langle\theta|\mu \uplus \sigma|c\rangle$.*
2. *Si $\{\!\{u/[\theta]\ p\}\!\} = échec$, alors pour tous μ et c, il existe c' tel que $\langle\theta|\mu|(p,u)+c\rangle \to^*_{FR} \langle\theta|échec|c'\rangle$.*

Démonstration. Point 1.
Par récurrence sur p.

- Si $p = x$, alors le filtrage n'est pas défini, ce n'est pas notre cas.
- Si $p = \hat{x}$, que $\hat{x}$ soit libre ou lié le résultat est immédiat.
- Si $p = [\theta]\ q \to s$, on a $\{\!\{u/[\theta]\ p\}\!\} = échec$, ce n'est pas notre cas.
- Si $p = p_1\ p_2$ (terme testable), comme le filtrage réussit, on a forcément $u = u_1\ u_2$ avec u testable. On a alors :

$$\sigma = \{\!\{u/[\theta]\ p\}\!\} = \{\!\{u_1/[\theta]\ p_1\}\!\} \uplus \{\!\{u_2/[\theta]\ p_2\}\!\} = \sigma_1 \uplus \sigma_2$$

avec $dom(\sigma_1) \cap dom(\sigma_2) = \emptyset$.
Par hypothèse de récurrence, on a pour tous μ et c :

$$\begin{array}{lcl} \langle\theta|\mu|(p_1,u_1)+c\rangle & \to^*_{epc} & \langle\theta|\mu \uplus \sigma_1|c\rangle \\ \langle\theta|\mu|(p_2,u_2)+c\rangle & \to^*_{epc} & \langle\theta|\mu \uplus \sigma_2|c\rangle \end{array}$$

Ainsi, pour tous μ et c :

$$\begin{array}{lcl} \langle\theta|\mu|(p,u)+c\rangle & \to_{epc} & \langle\theta|\mu|(p_1,u_1)+(p_2,u_2)+c\rangle \\ & \to^*_{epc} & \langle\theta|\mu \uplus \sigma_1|(p_2,u_2)+c\rangle \\ & \to^*_{epc} & \langle\theta|(\mu \uplus \sigma_1) \uplus \sigma_2|c\rangle \\ & = & \langle\theta|\mu \uplus (\sigma_1 \uplus \sigma_2)|c\rangle \\ & = & \langle\theta|\mu \uplus \sigma|c\rangle \end{array}$$

(on rappelle que $\uplus$ est associatif).

□

Démonstration. Point 2.
Par récurrence sur p encore.

- $p = x$: toujours pas défini.
- $p = \hat{x}$: dans notre cas d'erreur, forcément $x \notin \theta$ et $u \neq \hat{x}$. La réduction est alors immédiate.
- $p = [\theta]\ q \to s$: réduction immédiate (ou indéfini, ce qui n'est pas).
- $p = p_1\, p_2$ (terme testable) : u est forcément testable également. Si u n'est pas une application, la réduction à l'erreur est alors immédiate. Supposons $u = u_1\ u_2$. On a donc

$$\{\!\{u_1/[\theta]\ p_1\}\!\} \uplus \{\!\{u_2/[\theta]\ p_2\}\!\} = échec$$

On a trois causes possibles pour l'erreur (voir définition de $\uplus$) : le terme de gauche, le terme de droite, ou l'opération d'union disjointe. Dans tous les cas, on peut commencer la réduction par :

$$\langle\theta|\mu|(p,u)+c\rangle \to_{epc} \langle\theta|\mu|(p_1,u_1)+(p_2,u_2)+c\rangle$$

Si l'erreur venait de la gauche ou la droite, on conclut par hypothèse de récurrence. Si l'erreur vient de l'union disjointe, on applique deux fois le point 1. du lemme pour obtenir :

$$\langle\theta|\mu|(p,u)+c\rangle \to^*_{epc} \langle\theta|\mu \uplus (\sigma_1 \uplus \sigma_2)|c\rangle$$

On conclut par définition de $\uplus$.

□

Simulation de l'*Explicit Pattern Calculus*

On prouve maintenant le théorème 3 :

Si $t \to_{epc} t'$ et $t\downarrow$ et $t'\downarrow$ sont des termes purs, alors $purt \to^*_{ppc} t'\downarrow$.

Pour ce faire on énonce les deux lemmes suivants :

Lemme 9. *Pour tous μ, p_i, u_i, μ', p'_j, u'_j purs,*

si $\langle\theta|\mu|(p_1,u_1)+\ldots+(p_n,u_n)\rangle \to_{FR} \langle\theta|\mu'|(p'_1,u'_1)+\ldots+(p'_{n'},u'_{n'})\rangle$

alors $\mu \uplus \{\!\{u_1/[\theta]\ p_1\}\!\} \uplus \ldots \uplus \{\!\{u_n/[\theta]\ p_n\}\!\} = \mu' \uplus \{\!\{u'_1/[\theta]\ p'_1\}\!\} \uplus \ldots \uplus \{\!\{u'_{n'}/[\theta]\ p'_{n'}\}\!\}$

Démonstration. Par induction sur $\to_{FR}$ et par examen des quatre règles de filtrage. □

Ce lemme s'étend immédiatement à une réduction $\to_{FR}$ de longueur arbitraire.

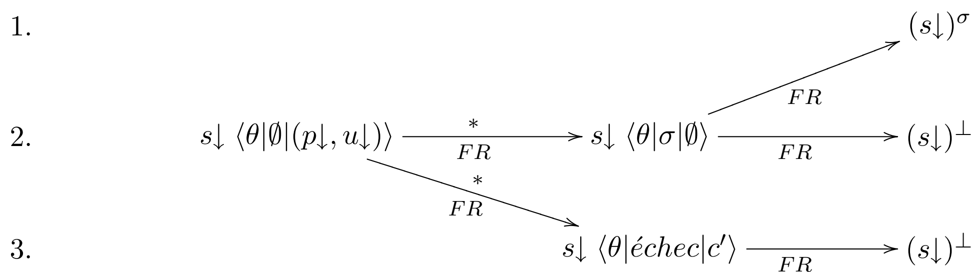
Lemme 10. *Pour tout terme t et filtrat μ, si $t \to^*_{ppc} t'$ et $\mu \to^*_{ppc} \mu'$, alors $t^\mu \to^*_{ppc} (t')^{\mu'}$.*

où on note $\{x_1 \mapsto u_1,\ldots,x_n \mapsto u_n\} \to^*_{ppc} \{x_1 \mapsto u'_1,\ldots,x_n \mapsto u'_n\}$ quand $u_i \to^*_{ppc} nu'_i$ pour tout i.

Démonstration. Voir [JK08]. □

On appellera filtrage soluble un $F = \langle\theta|\mu|c\rangle$ tel que l'un des deux cas suivants est vérifié :
- Il existe σ tel que : $\langle\theta|\mu|c\rangle \to^*_{FR} \langle\theta|\sigma|\emptyset\rangle$.
- Il exists c' tel que : $\langle\theta|\mu|c\rangle \to^*_{FR} \langle\theta|échec|c'\rangle$.

Démonstration. Théorème 3. Par récurrence sur $t \to_{epc} t'$.
- Les cas de base avec les sept règles de résolution et de filtrage sont immédiats : en effet si $t \to_{FR} t'$ alors $t\downarrow = t'\downarrow$.
- Règle d'initialisation : on a alors $t = ([\theta]\ p \to s)\ u$ et $t' = s\ \langle\theta|\emptyset|(p,u)\rangle$. Donc $t\downarrow = ([\theta]\ p\downarrow \to s\downarrow)\ u\downarrow$ et $t'\downarrow = (s\downarrow\ \langle\theta|\emptyset|(p\downarrow,u\downarrow)\rangle)\downarrow$, et ces termes sont purs. La pureté de $t'\downarrow$ en particulier nous indique que le filtrage $\langle\theta|\emptyset|(p\downarrow,u\downarrow)\rangle$ est soluble : on a l'un des chemins suivants :



On raisonne par cas sur ces chemins :

1. Comme $t\downarrow$ est pur, $s\downarrow$ et σ le sont aussi. Donc $(s\downarrow)^\sigma {}_{FR}^{*}\!\leftarrow t' \to^*_{FR} t'\downarrow$ avec $s\downarrow^\sigma$ et t' purs. Par confluence de $\to_{FR}$ on a donc $(s\downarrow)^\sigma = t'\downarrow$.
   Enfin, par lemme 9 on a $\sigma = \{\!\{u\downarrow\ /[\theta]\ p\downarrow\}\!\}$. On est dans le cas où $dom(\sigma) = \theta$, donc $([\theta]\ p\downarrow \to s\downarrow)\ u\downarrow \to_{ppc} (s\downarrow)^\sigma$ par définition du *Pure Pattern Calculus*. Donc $t\downarrow \to^*_{epc} t'\downarrow$.
2. Idem.

3. Idem.

- Si $t = u\ v \rightarrow_{epc} u'\ v = t'$ avec $u \rightarrow_{epc} u'$. On sait que $u{\downarrow}$ et $u'{\downarrow}$ sont purs, car $t{\downarrow} = u{\downarrow}\ v{\downarrow}$ et $t'{\downarrow} = u'{\downarrow}\ v{\downarrow}$ le sont par hypothèse. Par hypothèse de récurrence on a donc $u{\downarrow} \rightarrow^*_{ppc} u'{\downarrow}$, $t{\downarrow} = u{\downarrow}\ v{\downarrow} \rightarrow^*_{ppc} u'{\downarrow}\ v{\downarrow} = t'{\downarrow}$.
- Si $t = u\ v \rightarrow_{epc} u\ v' = t'$ avec $v \rightarrow_{epc} v'$, idem.
- Si $t = [\theta]\ p \rightarrow s \rightarrow_{epc} [\theta]\ p' \rightarrow s = t'$ avec $p \rightarrow_{epc} p'$, idem.
- Si $t = [\theta]\ p \rightarrow s \rightarrow_{epc} [\theta]\ p \rightarrow s' = t'$ avec $s \rightarrow_{epc} s'$, idem.
- Si $t = s\ F \rightarrow_{epc} s'\ F = t'$ avec $s \rightarrow_{epc} s'$, par hypothèse $s{\downarrow}$ et $s'{\downarrow}$ sont purs et par hypothèse de récurrence $s{\downarrow} \rightarrow^*_{ppc} s'{\downarrow}$. Comme précédemment, comme $t{\downarrow}$ et $t'{\downarrow}$ sont purs, on a $t{\downarrow} = (s{\downarrow})^\mu$ et $t'{\downarrow} = (s'{\downarrow})^\mu$ pour un certain filtrat μ résultat de F. On obtient le résultat $t{\downarrow} \rightarrow^*_{ppc} t'{\downarrow}$ avec le lemme 10.
- Si $t = s\ F \rightarrow_{epc} s\ F' = t'$ avec $F \rightarrow_{epc} F'$, idem.

□

B Substitutions explicites

Pour arriver formellement aux évaluateurs et machines abstraites présentés dans l'annexe suivante, nous devons changer notre vision des substitutions. En effet, les substitutions implicites que nous avons utilisées dans la théorie supposent que l'opération de subtitution s'applique immédiatement et de façon atomique à chaque fois qu'une substitution naît : cela suppose une transformation de l'ensemble du programme. Ceci est peu efficace en pratique, et on préfère en général implémenter les subsitutions par des environnements dans lesquels on va chercher les valeurs des variables quand cela s'avère nécessaire.
Pour formaliser ceci nous devons encore une fois avoir un contrôle plus fin sur la gestion des substitutions, et donc étendre les formalismes pour leur ajouter des substitutions explicites. Nous allons donc devoir nous aventurer dans les deux cases précédemment ignorées du tableau :

	Filtrage implicite	Filtrage explicite
Substitution implicite		
Substitution explicite	PPC_{ES}	EPC_{ES}

On s'inspirera pour ceci d'un plan maintenant assez connu, dont un état-de-l'art est présenté dans [Kes07]. Cette annexe présente les définitions et preuves de ces extensions, pour lesquelles le nouveau cadre des motifs dynamiques n'apporte en réalité aucune difficulté supplémentaire.

Extensions

Les substitutions sont toujours représentées par des ensembles d'associations :

$$\sigma \quad ::= \quad \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$$

pour $x_1, \ldots, x_n$ des noms et $u_1, \ldots, u_n$ des termes.
On étend les grammaires des termes du *Pure-* et de l'*Explicit- Pattern Calculus* respectivement à celles-ci :

$$\begin{array}{lcl} t & ::= & x \mid \hat{x} \mid t\ t \mid [\theta]\ t \to t \mid t\ \sigma \\ t & ::= & x \mid \hat{x} \mid t\ t \mid [\theta]\ t \to t \mid t\ F \mid t\ \sigma \end{array}$$

Il convient ensuite de changer les règles qui appliquent une *méta*-substitution en des règles générant une substitution explicite. Pour le *Pure Pattern Calculus*, la règle

$$([\theta]\ p \to s)\ u \quad \to \quad s^{\sigma} \quad \text{si } \{u/[\theta]\ p\} = \sigma$$

devient maintenant :

$$([\theta]\ p \to s)\ u \quad \to \quad s\ \sigma \quad \text{si } \{u/[\theta]\ p\} = \sigma$$

Pour l'*Explicit Pattern Calculus*, c'est la règle

$$s\ \langle\theta|\sigma|\emptyset\rangle \quad \to \quad s^{\sigma} \quad \text{si } dom(\sigma) = \theta$$

qui est remplacée par :

$$s\ \langle\theta|\sigma|\emptyset\rangle \quad \to \quad s\ \sigma \quad \text{si } dom(\sigma) = \theta$$

Il nous faut ensuite définir les règles gérant les substitutions explicites. Ces dernières sont essentiellement une orientation des égalités définissant nos précédentes substitutions explicites. On a d'abord la base commune aux *Pure-* et *Explicit- Pattern Calculi* :

FIG. 8 – Définition de $\rightarrow_{ppces}$

$$\frac{\{u/[\theta]\ p\} = \sigma}{([\theta]\ p \rightarrow s)\ u \rightarrow_{ppces} s\ \sigma} \qquad \frac{t \rightarrow_{ppces} t'}{t\ u \rightarrow_{ppces} t'\ u} \qquad \frac{p \rightarrow_{ppces} p'}{[\theta]\ p \rightarrow s \rightarrow_{ppces} [\theta]\ p' \rightarrow s}$$

$$\frac{\{u/[\theta]\ p\} = \textit{échec}}{([\theta]\ p \rightarrow s)\ u \rightarrow_{ppces} s^{\perp}} \qquad \frac{u \rightarrow_{ppces} u'}{t\ u \rightarrow_{ppces} t\ u'} \qquad \frac{s \rightarrow_{ppces} s'}{[\theta]\ p \rightarrow s \rightarrow_{ppces} [\theta]\ p \rightarrow s'}$$

$$\frac{u_i \rightarrow_{ppces} u_i'}{\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\} \rightarrow_{ppces} \{x_1 \mapsto u_1, \ldots, x_i \mapsto u_i', \ldots, x_n \mapsto u_n\}}$$

$$x_i\ \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\} \rightarrow_{ppces} u_i \qquad \frac{x \notin dom(\sigma)}{x\ \sigma \rightarrow_{ppces} x}$$

$$(t\ u)\ \sigma \rightarrow_{ppces} (t\ \sigma)\ (u\ \sigma) \qquad \hat{x}\ \sigma \rightarrow_{ppces} \hat{x}$$

$$\frac{\sigma \# \theta}{([\theta]\ p \rightarrow s)\ \sigma \rightarrow_{ppces} [\theta]\ (p\ \sigma) \rightarrow (s\ \sigma)}$$

$$\begin{array}{rcll} x_i\ \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\} & \rightarrow & u_i & \\ x\ \sigma & \rightarrow & x & \text{si } x \notin dom(\sigma) \\ \hat{x}\ \sigma & \rightarrow & \hat{x} & \\ (t\ u)\ \sigma & \rightarrow & (t\ \sigma)\ (u\ \sigma) & \\ ([\theta]\ p \rightarrow s)\ \sigma & \rightarrow & [\theta]\ (p\ \sigma) \rightarrow (u\ \sigma) & \text{si } \sigma \# \theta \end{array}$$

Il faut encore ajouter une règle pour l'*Explicit Pattern Calculus* :

$$(t\ \langle\theta|\mu|c\rangle)\ \sigma \quad \rightarrow \quad (t\ \sigma)\ \langle\theta|\mu[\sigma]|c\ \sigma\rangle \quad \text{si } \sigma\#\theta$$

où la composition de filtrats $\mu[\mu']$ est adaptée de la précédente :

$$\begin{array}{rcl} \mu[\textit{échec}] & = & \textit{échec} \\ \textit{échec}[\mu] & = & \textit{échec} \\ \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}[\sigma] & = & \{x_1 \mapsto u_1\ \sigma, \ldots, x_n \mapsto u_n\ \sigma\} \end{array}$$

Une fois encore, on peut appliquer toutes les règles dans tous les contextes, ce qu'on formalise dans les figures 8 et 9. Dans les deux cas on notera $\rightarrow_S$ la réduction obtenue avec uniquement les cinq (resp. six) règles de substitution. On note également respectivement $\rightarrow_{ppces}$ et $\rightarrow_{epces}$ les réductions du *Pure-* et de l'*Explicit- Pattern Calculus* avec substitutions explicites.

Simulations et confluences

Comme précédemment, on vérifie aisément que le système $\rightarrow_S$ est dans les deux cadres confluent et fortement normalisant. On en déduit donc une fonction de normalisation et on notera encore $t{\downarrow}$ l'unique forme normale du terme t.

Lemme 11. *Pour tout terme t, $t{\downarrow}$ ne contient plus aucune substitution.*

Ainsi, contrairement au cas précédent, la fonction de purification s nous ramène toujours sur un terme *pur*, où "pur" signifie maintenant un terme du *Pure-* ou de l'*Explicit- Pattern Calculus* selon le cas, c'est-à-dire un terme sans substitutions explicites. Les énoncés des lemmes de simulation seront plus simples et plus puissants grâce à cette propriété.

On commence par donner un autre lemme :

Lemme 12. *Si t et $u_1, \ldots, u_n$ sont purs, alors $(t\ \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}){\downarrow}$ est pur.*

FIG. 9 – Définition de $\rightarrow_{epces}$

$$([\theta]\ p \rightarrow s)\ u \rightarrow_{epces} s\ \langle\theta|\emptyset|(p,u)\rangle$$

$$\frac{dom(\sigma)=\theta}{s\ \langle\theta|\sigma|\emptyset\rangle \rightarrow_{epces} s\ \sigma} \qquad \frac{dom(\sigma)\neq\theta}{s\ \langle\theta|\sigma|\emptyset\rangle \rightarrow_{epces} s^{\perp}} \qquad s\ \langle\theta|\acute{e}chec|\emptyset\rangle \rightarrow_{epces} s^{\perp}$$

$$\frac{x\in\theta}{\langle\theta|\mu|(\hat{x},u)+c\rangle \rightarrow_{epces} \langle\theta|\mu \uplus \{x\mapsto u\}|c\rangle} \qquad \frac{s\notin\theta}{\langle\theta|\mu|(\hat{x},\hat{x})+c\rangle \rightarrow_{epces} \langle\theta|\mu|c\rangle}$$

$$\frac{(p\ q) \text{ et } (u\ v) \text{ testables}}{\langle\theta|\mu|(p\ q,u\ v)+c\rangle \rightarrow_{epces} \langle\theta|\mu|(p,u)+(q,v)+c\rangle} \qquad \frac{\text{autres cas avec } p \text{ et } u \text{ testables}}{\langle\theta|\mu|(p,u)+c\rangle \rightarrow_{epces} \langle\theta|\acute{e}chec|c\rangle}$$

$$\frac{t \rightarrow_{epces} t'}{t\ u \rightarrow_{epces} t'\ u} \qquad \frac{p \rightarrow_{epces} p'}{[\theta]\ p \rightarrow s \rightarrow_{epces} [\theta]\ p' \rightarrow s} \qquad \frac{s \rightarrow_{epces} s'}{s\ F \rightarrow_{epces} s'\ F}$$

$$\frac{u \rightarrow_{epces} u'}{t\ u \rightarrow_{epces} t\ u'} \qquad \frac{s \rightarrow_{epces} s'}{[\theta]\ p \rightarrow s \rightarrow_{epces} [\theta]\ p \rightarrow s'} \qquad \frac{F \rightarrow_{epces} F'}{s\ F \rightarrow_{epces} s\ F'}$$

$$\frac{u_i \rightarrow_{epces} u_i'}{\{x_1\mapsto u_1,\dots,x_n\mapsto u_n\} \rightarrow_{epces} \{x_1\mapsto u_1,\dots,x_i\mapsto u_i',\dots,x_n\mapsto u_n\}}$$

$$\frac{p \rightarrow_{epces} p'}{\langle\theta|\mu|(p,u)+c\rangle \rightarrow_{epces} \langle\theta|\mu|(p',u)+c\rangle} \qquad \frac{u \rightarrow_{epces} u'}{\langle\theta|\mu|(p,u)+c\rangle \rightarrow_{epces} \langle\theta|\mu|(p,u')+c\rangle}$$

$$x_i\ \{x_1\mapsto u_1,\dots,x_n\mapsto u_n\} \rightarrow_{epces} u_i \qquad \frac{x\notin dom(\sigma)}{x\ \sigma \rightarrow_{epces} x}$$

$$(t\ u)\ \sigma \rightarrow_{epces} (t\ \sigma)\ (u\ \sigma) \qquad \hat{x}\ \sigma \rightarrow_{epces} \hat{x}$$

$$\frac{\sigma\#\theta}{([\theta]\ p \rightarrow s)\ \sigma \rightarrow_{epces} [\theta]\ (p\ \sigma) \rightarrow (s\ \sigma)} \qquad \frac{\sigma\#\theta}{(s\ \langle\theta|\mu|c\rangle)\ \sigma \rightarrow_{epces} (t\ \sigma)\ \langle\theta|\mu[\sigma]|c\ \sigma\rangle}$$

Démonstration. Par récurrence immédiate sur t. □

Démonstration. Lemme 11, par récurrence sur t.

- Si $t = x$, c'est immédiat.
- Si $t = \hat{x}$, c'est immédiat.
- Si $t = u\ v$, alors $t{\downarrow} = u{\downarrow}\ v{\downarrow}$ et on conclut par récurrence.
- Si $t = [\theta]\ p \to s$, idem.
- Si $t = s\ F$, idem.
- Si $t = s\ \{x_1 \mapsto u_1, \dots, x_n \mapsto u_n\}$, alors par récurrence $s{\downarrow}$ et les $u_1{\downarrow}, \dots, u_n{\downarrow}$ sont purs. On conclut alors par lemme 12

□

On prouve maintenant un lemme exprimant le lien entre notre ancienne substitution implicite et notre nouvelle substitution explicite.

Lemme 13. *Pour $\sigma = \{x_1 \mapsto u_1, \dots, x_n \mapsto u_n\}$, on a :*

$$(s\ \sigma){\downarrow} \quad = \quad (s{\downarrow})^{\{x_1 \mapsto u_1{\downarrow}, \dots, x_n \mapsto u_n{\downarrow}\}}$$

Démonstration. On a d'abord $t\ \sigma \to_S^* t{\downarrow}\ \sigma{\downarrow}$. On montre ensuite $t{\downarrow}\ \sigma{\downarrow} \to_S^* (t{\downarrow})^{\sigma{\downarrow}}$ par récurrence sur $t{\downarrow}$. Cette récurrence est évidente car on a une correspondance exacte entre les cas de la définition de la *méta*-substitution et les règles définissant $\to_S$. □

Enfin, on énonce un lemme exprimant une commutation entre les opérations de filtrage et de purification :

Lemme 14. *Pour tous termes p et u, on a $\{u/[\theta]\ p\}{\downarrow} = \{u{\downarrow}\ /[\theta]\ p{\downarrow}\}$.*

Démonstration. On vérifie d'abord que $\{\!\{u/[\theta]\ p\}\!\}{\downarrow} = \{\!\{u{\downarrow}\ /[\theta]\ p{\downarrow}\}\!\}$ par récurrence sur la définition de $\{\!\{u/[\theta]\ p\}\!\}$. C'est alors immédiat. □

On peut maintenant énoncer et démontrer les résultats de simulation entre PPC et PPC_{ES} (resp. EPC et EPC_{ES}) :

Théorème 4.

1. *Si $t \to_{ppc} t'$, alors $t \to_{ppces}^*$.*
2. *Si $t \to_{epc} t'$, alors $t \to_{epces}^*$.*
3. *Si $t \to_{ppces} t'$, alors $t{\downarrow} \to_{ppc}^* t'{\downarrow}$.*
4. *Si $t \to_{epces} t'$, alors $t{\downarrow} \to_{epc}^* t'{\downarrow}$.*

Démonstration.

1. La seule règle de $\to_{ppc}$ qui n'est pas directement reprise dans $\to_{ppces}$ est :

$$([\theta]\ p \to t)\ u \quad \to \quad t^{\sigma} \quad \text{si } \{u/[\theta]\ p\} = \sigma$$

   On a en revanche à la place une règle générant le terme $t\ \sigma$, dans lequel il faut propager manuellement la substitution. Le lemme 13 permet de conclure.

2. La seule règle de $\to_{epc}$ qui n'est pas directement reprise dans $\to_{epces}$ est :

$$t\ \langle \theta | \sigma | \emptyset \rangle \quad \to \quad t^{\sigma} \quad \text{si } dom(\sigma) = \theta$$

   De même on commence par générer le terme $t\ \sigma$, puis on conclut avec le lemme 13.

3. On raisonne par récurrence sur $t \to_{ppces} t'$:
   - Si $t \to_S t'$, alors par confluence de $\to_S$ on a $t{\downarrow} = t'{\downarrow}$.
   - Si $t = ([\theta]\ p \to s)\ u \to_{ppces} s^{\perp} = t'$, c'est immédiat.

- Si $t = ([\theta]\ p \rightarrow s)\ u \rightarrow_{ppces} s\ \sigma = t'$ avec $\sigma = \{u/[\theta]\ p\}$. Par lemme 14 on a $\sigma\downarrow = \{u\downarrow /[\theta]\ p\downarrow\}$. Donc $t\downarrow = ([\theta]\ p\downarrow \rightarrow s\downarrow)\ u\downarrow \rightarrow_{ppc} (s\downarrow)^{\sigma\downarrow}$. Or par lemme 13, $(s\downarrow)^{\sigma\downarrow} = (s\downarrow\ \sigma\downarrow)\downarrow = (s\ \sigma)\downarrow = t'\downarrow$.
- Si la réduction se fait dans un contexte, par exemple $t = u\ v \rightarrow_{ppces} u'\ v = t'$ avec $u \rightarrow_{ppces} u'$, alors par hypothèse de récurrence $u\downarrow \rightarrow_{ppc} u'\downarrow$, et on a donc : $t\downarrow = u\downarrow\ v\downarrow \rightarrow_{ppc} u'\downarrow\ v\downarrow = t'\downarrow$.

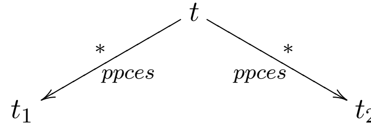
4. Idem.

□

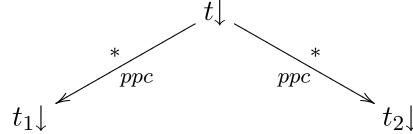
Maintenant, les résultats de simulation étant plus forts que dans le cas du filtrage explicite, nous allons pouvoir déduire plus simplement la confluence de nos calculs avec substitutions explicites PPC_{ES} et EPC_{ES}.

Théorème 5. *Les calculs PPC_{ES} et EPC_{ES} sont confluents.*

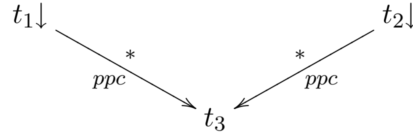
Démonstration. Soient trois termes t, t_1 et t_2. Supposons :



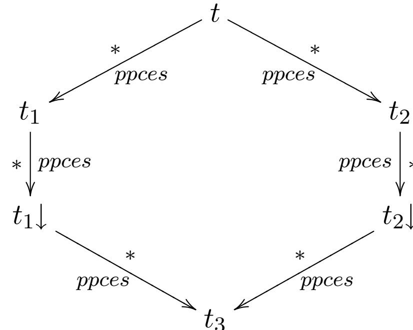
Alors par théorème de simulation 4 (point 3.), on a :



Or le *Pure Pattern Calculus* est confluent. On a donc :



ce qui est encore vrai avec $\rightarrow_{ppces}$ par simulation 4 (point 1.). On peut donc clore ainsi le diagramme :



La preuve pour EPC_{ES} est identique, avec les points 2. et 4. du théorème de simulation 4. □

C Implémentations

Stratégie stricte

Évaluateur

On donne ci-dessous le code Caml d'un interpréteur (ou évaluateur) du *Pure Pattern Calculus* en appel par valeur. On s'est octroyé les extensions suivantes :

- Des données de base (limitées pour l'exemple à des entiers) : `DAT of int`
- Une macro pour l'alternative d'un filtrage : `CHOICE`, c'est un `CASE` (abstraction normale d'un motif) augmenté d'un autre terme à appliquer à l'argument en cas d'échec de filtrage.
- L'erreur est représentée par un atome inerte `BOT`. Sa signification est laissée libre.

Enfin, on utilise les constructeurs `VAR` pour une occurence normale de variable, `CONS` pour un $\hat{x}$, et `APP` pour une application quelconque.

```
type ppc = DAT of int
         | VAR of string
         | CONS of string
         | CASE of string list * ppc * ppc
         | CHOICE of string list * ppc * ppc * ppc
         | APP of ppc * ppc
         | BOT
```

Pour gérer les substitutions on utilisera des environnements associant des termes à des noms de variables. On donne la signature d'un module `Env` implémentant ce mécanisme :

```
Module Env :
  sig
    type 'a t                                    (* type d'un environnement *)

    val empty : unit -> 'a t                     (* environnement vide *)
    val extend : string * 'a -> 'a t -> 'a t     (* ajout d'un élément *)
    val merge : 'a t -> 'a t -> 'a t             (* union disjointe *)
    val lookup : string * 'a t -> 'a             (* accès *)
```

On utilise une fonction `evalmatch : string list * ppc * ppc -> match` qui effectue le filtrage et génère le filtrat associé, le type `match` étant défini comme

```
type match = Fail | Subst of ppc Env.t
```

On a également une extension de la fonction `Env.extend` aux filtrats : `Match.extend : string * 'a -> match -> match`.

La fonction d'évaluation prend en argument un terme et un contexte, puis retourne une valeur (ou lève une exception). Les valeurs comprennent des objets de type `ppc`, plus des fonctions des valeurs dans les valeurs (`FUN of value -> value`).

```
(*  eval  :  ppc * ppc Env.t -> ppc  *)

let rec eval = fun
    | (DAT d, e) -> DAT d
    | (VAR x, e) -> Env.lookup (x,e)
    | (CONS x, e) -> CONS x
    | (CASE(tt,p,s), e) ->
        let vp = eval(p,e) in
        FUN (fun v -> match evalmatch(tt,vp,v) with
```

```
                    | Fail -> BOT
                    | Subst e' -> eval(s, Env.merge e e'))
    | (CHOICE(tt,p,s,r), e) ->
        let vp = eval(p,e) in
        FUN (fun v -> match evalmatch(tt,vp,v) with
                    | Fail -> eval(APP(r,v), e)
                    | Subst e' -> eval(s, Env.merge e e'))
    | (APP(t,u), e) ->
        let vt = eval(t,e) and
            vu = eval(u,e) in
        ( match vt with
          | FUN f -> f vu
          | CONS x as c -> APP(c,vu) )
    | (BOT, e) -> BOT
```

Dérivation d'une machine abstraite

Closure Conversion Dans le type des valeurs, on va faire disparaître le constructeur `FUN` et ajouter aux constructions fonctionnelles (`CASE` et `CHOICE`) un environnement pour en faire des clôtures. Les valeurs sont donc maintenant :

```
type value = VDAT of int
           | VCONS of string
           | VCASE of string list * ppc * ppc * value Env.t
           | VCHOICE of string list * ppc * ppc * ppc * value Env.t
           | VAPP of value * value
           | VBOT
```

La transformation va ensuite seulement remplacer les utilisation de `FUN` par un traitement explicite de chaque type de clôture. Voici le nouvel interpréteur :

```
(*  eval  :  ppc * value Env.t -> value  *)

let rec eval = fun
    | (DAT d, e) -> VDAT d
    | (VAR x, e) -> Env.lookup(x,e)
    | (CONS x, e) -> VCONS x
    | (CASE(tt,p,s), e) -> VCASE(tt,p,s,e)
    | (CHOICE(tt,p,s,r), e) -> VCHOICE(tt,p,s,r,e)
    | (APP(t,u), e) ->
        let vt = eval(t,e) and
            vu = eval(u,e) in
        ( match vt with
          | VCASE(tt,p,s,e') ->
              let vp = eval(p,e') in
              ( match evalmatch(tt,vp,vu,e) with
                | Fail -> VBOT
                | Subst e'' -> eval(s, Env.merge e' e'') )
          | VCHOICE(tt,p,s,r,e') ->
              let vp = eval(p,e') in
              ( match evalmatch(tt,vp,vu,e) with
                | Fail -> eval(APP(r, vu), e')
                | Subst e'' -> eval(s, Env.merge e' e'') )
```

```
        | VCONS x as c -> VAPP(c, vu) )
    | (BOT, e) -> VBOT
```

Passage de continuations Voici le code transformé :

```
(*  eval  :  ppc * value Env.t * (value -> value) -> value  *)

let rec eval = fun
    | (DAT d, e, k) -> k (VDat d)
    | (VAR x, e, k) -> k (Env.lookup (x,e))
    | (CONS x, e, k) -> k (VCONS x)
    | (CASE(tt,p,s), e, k) -> k VCASE(tt,p,s,e)
    | (CHOICE(tt,p,s,r), e, k) -> k VCHOICE(tt,p,s,r,e)
    | (APP(t,u), e, k) ->
        eval(t, e, fun vt ->
            eval(u, e, fun vu ->
                ( match vt with
                  | VCASE(tt,p,s,e') ->
                      eval(p, e', fun vp ->
                          evalmatch(tt,vp,vu,e, fun mu ->
                              ( match mu with
                                | Fail -> VBOT
                                | Subst e'' -> eval(s, Env.merge e' e'', k) )))
                  | VCHOICE(tt,p,s,r,e') ->
                      eval(p, e', fun vp ->
                          evalmatch(tt,vp,vu,e, fun mu ->
                              ( match mu with
                                | Fail -> eval(APP(r,vu), e', k)
                                | Subst e'' -> eval(s, Env.merge e' e'', k) )))
                  | VCONS x as c -> k VAPP(c, vu) )))
    | (BOT, e, k) -> VBOT
```

Défonctionalisation Les continuations vont donc ici être bâties sur six constructeurs, plus un constructeur d'arrêt :

```
type cont =
     | STOP
     | VT_ARG of ppc * value Env.t * cont
     | VU_FUN of value * value Env.t * cont
     | VP_CASE of string list * value * value Env.t * value Env.t * ppc * cont
     | MU_CASE of ppc * value Env.t * cont
     | VP_CHOICE of string list * value * value Env.t * value Env.t * ppc * ppc * cont
     | MU_CHOICE of value * ppc * ppc * value Env.t * cont
```

On définit à partir de là la fonction `eval` parallèlement avec la fonction `apply_cont` d'application des continuations codées.

```
(*  eval        :  ppc * value Env.t * cont -> value  *)
(*  apply_cont  :  cont * value -> value        *)

let rec eval = fun
    | (DAT d, e, k)            -> apply_cont(k, VDat d)
    | (VAR x, e, k)            -> apply_cont(k, Env.lookup(x,e))
```

```
    | (CONS x, e, k)           -> apply_cont(k, VCONS x)
    | (CASE(tt,p,s), e, k)     -> apply_cont(k, VCASE(tt,p,s,e))
    | (CHOICE(tt,p,s,r), e, k)  -> apply_cont(k, VCHOICE(tt,p,s,r,e))
    | (APP(t,u), e, k)         -> eval(t, e, VT_ARG(u,e,k))
    | (BOT, e, k)              -> VBOT
and apply_cont = fun
    | (STOP, v)                -> v
    | (VT_ARG(u,e,k), vt)      -> eval(u, e, VU_FUN(vt,e,k))
    | (VU_FUN(VCASE(tt,p,s,e'), e, k), vu)
                               -> eval(p, e', VP_CASE(tt,vu,e,e',s,k))
    | (VU_FUN(VCHOICE(tt,p,s,r,e'), e, k), vu)
                               -> eval(p, e', VP_CHOICE(tt,vu,e,e',s,r,k))
    | (VU_FUN(VCONS x, e, k), vu)
                               -> apply_cont(k, VAPP(VCONS x, vu))
    | (VP_CASE(tt,vu,e,e',s,k), vp)
                               -> evalmatch(tt,vp,vu,e, MU_CASE(s,e',k))
    | (MU_CASE(s,e',k), Fail)   -> VBOT
    | (MU_CASE(s,e',k), Subst e'')
                               -> eval(s, Env.merge e' e'', k)
    | (VP_CHOICE(tt,vu,e,e',s,r,k), vp)
                               -> evalmatch(tt,vp,vu,e, MU_CHOICE(s,e',k))
    | (MU_CHOICE(vu,s,r,e',k), Fail)
                               -> eval(APP(r,vu), e', k)
    | (MU_CHOICE(vu,s,r,e',k), Subst e'')
                               -> eval(s, Env.merge e' e'', k)
```

Stratégie paresseuse

Évaluateur

On agrandit le type des termes :

```
type ppc = DAT of int
         | VAR of string
         | CONS of string
         | CASE of string list * ppc * ppc
         | CHOICE of string list * ppc * ppc * ppc
         | APP of ppc * ppc
         | MATCH of ppc * string list * match * (ppc * ppc) list
         | CMATCH of ppc * ppc * string list * match * (ppc * ppc) list
```

Évaluateur paresseux du *Pure Pattern Calculus*.

```
(*  eval       :  ppc * ppc Env.t -> ppc  *)
(*  lazy_eval  :  ppc * ppc Env.t -> ppc  *)

let rec eval = fun
    | (DAT n, e) -> DAT d
    | (VAR x, e) -> eval(Env.lookup (x,e), e)
    | (CONS x, e) -> CONS x
    | (CASE(tt,p,s), e) ->
        FUN (fun u -> eval( MATCH(s,tt,Env.empty(),[(p,u)]), e))
    | (CHOICE(tt,p,s,r), e) ->
        FUN (fun u -> eval( CMATCH(s,APP(r,u),tt,Env.empty(),[(p,u)]), e))
    | (APP(t,u), e) ->
        let lt = lazy_eval(t,e) in
        ( match lt with
          | FUN f -> f u
          | _ -> let vt = eval(lt,e) and
                     vu = eval(u,e)  in
                 APP(vt,vu) )
    | (MATCH(_,_,Fail,_), _) -> BOT
    | (MATCH(s,tt,Subst e',[]), e) when check(tt,e') ->
        eval( s, Env.merge e e')
    | (MATCH(_,_,_,[]), _) -> BOT
    | (MATCH(s,tt,mu,(p,u)::c), e) ->
        let lp = lazy_eval(p,e) and
            lu = lazy_eval(u,e) in
        ( match (lp,lu) with
          | (CONS x,u) when List.mem x tt ->
              eval( MATCH(s,tt,Match.extend (x,u,mu),c), e)
          | (CONS x,CONS y) when x = y ->
              eval( MATCH(s,tt,mu,c), e)
          | (CONS x,CONS y) -> BOT
          | (CASE(_,_,_),_) -> BOT
          | (APP(p1,p2),APP(u1,u2)) ->
              eval( MATCH(s,tt,mu,(p1,u1)::(p2,u2)::c), e)
          | _ -> BOT )
    | (CMATCH(_,r,_,Fail,_), e) ->
        eval( r, e)
```

```
    | (CMATCH(s,_,tt,Subst e',[]), e) when check(tt,e') ->
        eval( s, Env.merge e e')
    | (CMATCH(_,r,_,_,[]), e) ->
        eval( r, e)
    | (CMATCH(s,r,tt,mu,(p,u)::c), e) ->
        let lp = lazy_eval(p,e) and
            lu = lazy_eval(u,e) in
        ( match (lp,lu) with
          | (CONS x,u) when List.mem x tt ->
              eval( CMATCH(s,r,tt,Match.extend (x,u,mu),c), e)
          | (CONS x,CONS y) when x = y ->
              eval( CMATCH(s,r,tt,mu,c), e)
          | (CONS x,CONS y) -> eval(r,e)
          | (CASE(_,_,_),_) -> eval(r,e)
          | (APP(p1,p2),APP(u1,u2)) ->
              eval( CMATCH(s,r,tt,mu,(p1,u1)::(p2,u2)::c), e)
          | _ -> eval(r,e) )
    | (BOT, _) -> BOT
and lazy_eval = fun
    | (DAT n, e) -> DAT d
    | (VAR x, e) -> lazy_eval(Env.lookup (x,e), e)
    | (CONS x, e) -> CONS x
    | (CASE(tt,p,s), e) ->
        FUN (fun u -> lazy_eval( MATCH(s,tt,Env.empty(),[(p,u)]), e))
    | (CHOICE(tt,p,s,r), e) ->
        FUN (fun u -> lazy_eval( CMATCH(s,APP(r,u),tt,Env.empty(),[(p,u)]), e))
    | (APP(t,u), e) ->
        let lt = lazy_eval(t,e) in
        ( match lt with
          | FUN f -> f u
          | _ -> let vt = lazy_eval(lt,e) and
                     vu = lazy_eval(u,e)  in
                 APP(vt,vu) )
    | (MATCH(_,_,Fail,_), _) -> BOT
    | (MATCH(s,tt,Subst e',[]), e) when check(tt,e') ->
        lazy_eval( s, Env.merge e e')
    | (MATCH(_,_,_,[]), _) -> BOT
    | (MATCH(s,tt,mu,(p,u)::c), e) ->
        let lp = lazy_eval(p,e) and
            lu = lazy_eval(u,e) in
        ( match (lp,lu) with
          | (CONS x,u) when List.mem x tt ->
              lazy_eval( MATCH(s,tt,Match.extend (x,u,mu),c), e)
          | (CONS x,CONS y) when x = y ->
              lazy_eval( MATCH(s,tt,mu,c), e)
          | (CONS x,CONS y) -> BOT
          | (CASE(_,_,_),_) -> BOT
          | (APP(p1,p2),APP(u1,u2)) ->
              lazy_eval( MATCH(s,tt,mu,(p1,u1)::(p2,u2)::c), e)
          | _ -> BOT )
    | (CMATCH(_,r,_,Fail,_), e) ->
```

```
        lazy_eval( r, e)
    | (CMATCH(s,_,tt,Subst e',[]), e) when check(tt,e') ->
        lazy_eval( s, Env.merge e e')
    | (CMATCH(_,r,_,_,[]), e) ->
        lazy_eval( r, e)
    | (CMATCH(s,r,tt,mu,(p,u)::c), e) ->
        let lp = lazy_eval(p,e) and
            lu = lazy_eval(u,e) in
        ( match (lp,lu) with
          | (CONS x,u) when List.mem x tt ->
              lazy_eval( CMATCH(s,r,tt,Match.extend (x,u,mu),c), e)
          | (CONS x,CONS y) when x = y ->
              lazy_eval( CMATCH(s,r,tt,mu,c), e)
          | (CONS x,CONS y) -> lazy_eval(r,e)
          | (CASE(_,_,_),_) -> lazy_eval(r,e)
          | (APP(p1,p2),APP(u1,u2)) ->
              lazy_eval( CMATCH(s,r,tt,mu,(p1,u1)::(p2,u2)::c), e)
          | _ -> eval(r,e) )
    | (BOT, _) -> BOT
```

Dérivation d'une machine abstraite

Closure Conversion

```
(*  eval       :  ppc * ppc Env.t -> value  *)
(*  lazy_eval  :  ppc * ppc Env.t -> value  *)

let rec eval = fun
    | (DAT n, e) -> VDAT d
    | (VAR x, e) -> eval(Env.lookup (x,e), e)
    | (CONS x, e) -> VCONS x
    | (CASE(tt,p,s), e) -> VCASE(tt,p,s,e)
    | (CHOICE(tt,p,s,r), e) -> VCHOICE(tt,p,s,r,e)
    | (APP(t,u), e) ->
        let lt = lazy_eval(t,e) in
        ( match lt with
          | VCASE(tt,p,s,e') ->
                eval( MATCH(s,tt,Env.empty(),[(p,u)],e'), e)
          | VCHOICE(tt,p,s,r,e') ->
                eval( CHOICE(s,r,tt,Env.empty(),[(p,u)],e'), e)
          | _ -> let vt = eval(lt,e) and
                     vu = eval(u,e)  in
                 APP(vt,vu) )
    | (MATCH(_,_,Fail,_), _) -> VBOT
    | (MATCH(s,tt,Subst e',[]), e) when check(tt,e') ->
        eval( s, Env.merge e e')
    | (MATCH(_,_,_,[]), _) -> VBOT
    | (MATCH(s,tt,mu,(p,u)::c), e) ->
        let lp = lazy_eval(p,e) and
            lu = lazy_eval(u,e) in
        ( match (lp,lu) with
          | (CONS x,u) when List.mem x tt ->
```

```
                eval( MATCH(s,tt,Match.extend (x,u,mu),c), e)
           | (CONS x,CONS y) when x = y ->
                eval( MATCH(s,tt,mu,c), e)
           | (CONS x,CONS y) -> VBOT
           | (CASE(_,_,_),_) -> VBOT
           | (APP(p1,p2),APP(u1,u2)) ->
                eval( MATCH(s,tt,mu,(p1,u1)::(p2,u2)::c), e)
           | _ -> VBOT )
    | (CMATCH(_,r,_,Fail,_), e) ->
        eval( r, e)
    | (CMATCH(s,_,tt,Subst e',[]), e) when check(tt,e') ->
        eval( s, Env.merge e e')
    | (CMATCH(_,r,_,_,[]), e) ->
        eval( r, e)
    | (CMATCH(s,r,tt,mu,(p,u)::c), e) ->
        let lp = lazy_eval(p,e) and
            lu = lazy_eval(u,e) in
        ( match (lp,lu) with
           | (CONS x,u) when List.mem x tt ->
                eval( CMATCH(s,r,tt,Match.extend (x,u,mu),c), e)
           | (CONS x,CONS y) when x = y ->
                eval( CMATCH(s,r,tt,mu,c), e)
           | (CONS x,CONS y) -> eval(r,e)
           | (CASE(_,_,_),_) -> eval(r,e)
           | (APP(p1,p2),APP(u1,u2)) ->
                eval( CMATCH(s,r,tt,mu,(p1,u1)::(p2,u2)::c), e)
           | _ -> eval(r,e) )
    | (BOT, _) -> VBOT
and lazy_eval = fun
    | (DAT n, e) -> VDAT d
    | (VAR x, e) -> lazy_eval(Env.lookup (x,e), e)
    | (CONS x, e) -> VCONS x
    | (CASE(tt,p,s), e) -> VCASE(tt,p,s,e)
    | (CHOICE(tt,p,s,r), e) -> VCHOICE(tt,p,s,r,e)
    | (APP(t,u), e) ->
        let lt = lazy_eval(t,e) in
        ( match lt with
          | VCASE(tt,p,s,e') ->
                lazy_eval( MATCH(s,tt,Env.empty(),[(p,u)],e'), e)
          | VCHOICE(tt,p,s,r,e') ->
                lazy_eval( CHOICE(s,r,tt,Env.empty(),[(p,u)],e'), e)
          | _ -> let vt = lazy_eval(lt,e) and
                     vu = lazy_eval(u,e)  in
                 APP(vt,vu) )
    | (MATCH(_,_,Fail,_), _) -> VBOT
    | (MATCH(s,tt,Subst e',[]), e) when check(tt,e') ->
        lazy_eval( s, Env.merge e e')
    | (MATCH(_,_,_,[]), _) -> VBOT
    | (MATCH(s,tt,mu,(p,u)::c), e) ->
        let lp = lazy_eval(p,e) and
            lu = lazy_eval(u,e) in
```

```
        ( match (lp,lu) with
          | (CONS x,u) when List.mem x tt ->
               lazy_eval( MATCH(s,tt,Match.extend (x,u,mu),c), e)
          | (CONS x,CONS y) when x = y ->
               lazy_eval( MATCH(s,tt,mu,c), e)
          | (CONS x,CONS y) -> VBOT
          | (CASE(_,_,_),_) -> VBOT
          | (APP(p1,p2),APP(u1,u2)) ->
               lazy_eval( MATCH(s,tt,mu,(p1,u1)::(p2,u2)::c), e)
          | _ -> VBOT )
    | (CMATCH(_,r,_,Fail,_), e) ->
        lazy_eval( r, e)
    | (CMATCH(s,_,tt,Subst e',[]), e) when check(tt,e') ->
        lazy_eval( s, Env.merge e e')
    | (CMATCH(_,r,_,_,[]), e) ->
        lazy_eval( r, e)
    | (CMATCH(s,r,tt,mu,(p,u)::c), e) ->
        let lp = lazy_eval(p,e) and
            lu = lazy_eval(u,e) in
        ( match (lp,lu) with
          | (CONS x,u) when List.mem x tt ->
               lazy_eval( CMATCH(s,r,tt,Match.extend (x,u,mu),c), e)
          | (CONS x,CONS y) when x = y ->
               lazy_eval( CMATCH(s,r,tt,mu,c), e)
          | (CONS x,CONS y) -> lazy_eval(r,e)
          | (CASE(_,_,_),_) -> lazy_eval(r,e)
          | (APP(p1,p2),APP(u1,u2)) ->
               lazy_eval( CMATCH(s,r,tt,mu,(p1,u1)::(p2,u2)::c), e)
          | _ -> lazy_eval(r,e) )
    | (BOT, _) -> VBOT
```

CPS

```
(*  eval       :  ppc * ppc Env.t * (value -> value) -> value  *)
(*  lazy_eval  :  ppc * ppc Env.t * (value -> value) -> value  *)

let rec eval = fun
    | (DAT n, e, k) -> k (VDAT d)
    | (VAR x, e, k) -> eval(Env.lookup (x,e), e, k)
    | (CONS x, e, k) -> k (VCONS x)
    | (CASE(tt,p,s), e, k) -> k VCASE(tt,p,s,e)
    | (CHOICE(tt,p,s,r), e, k) -> k VCHOICE(tt,p,s,r,e)
    | (APP(t,u), e, k) ->
        lazy_eval(t, e, fun lt ->
            ( match lt with
              | VCASE(tt,p,s,e') ->
                    eval( MATCH(s,tt,Env.empty(),[(p,u)],e'), e, k)
              | VCHOICE(tt,p,s,r,e') ->
                    eval( CHOICE(s,r,tt,Env.empty(),[(p,u)],e'), e, k)
              | _ -> eval( lt, e, fun vt ->
                        eval( u, e, fun vu ->
                            k APP(vt,vu) ))))
    | (MATCH(_,_,Fail,_), _, _) -> VBOT
```

```
    | (MATCH(s,tt,Subst e',[]), e, k) when check(tt,e') ->
        eval( s, Env.merge e e', k)
    | (MATCH(_,_,_,[]), _, _) -> VBOT
    | (MATCH(s,tt,mu,(p,u)::c), e, k) ->
          lazy_eval(p, e, fun lp ->
              lazy_eval(u, e, fun lu ->
                  ( match (lp,lu) with
                      | (CONS x,u) when List.mem x tt ->
                          eval( MATCH(s,tt,Match.extend (x,u,mu),c), e, k)
                      | (CONS x,CONS y) when x = y ->
                          eval( MATCH(s,tt,mu,c), e, k)
                      | (CONS x,CONS y) -> VBOT
                      | (CASE(_,_,_),_) -> VBOT
                      | (APP(p1,p2),APP(u1,u2)) ->
                          eval( MATCH(s,tt,mu,(p1,u1)::(p2,u2)::c), e, k)
                      | _ -> VBOT ) ))
    | (CMATCH(_,r,_,Fail,_), e, k) ->
        eval( r, e, k)
    | (CMATCH(s,_,tt,Subst e',[]), e, k) when check(tt,e') ->
        eval( s, Env.merge e e', k)
    | (CMATCH(_,r,_,_,[]), e, k) ->
        eval( r, e, k)
    | (CMATCH(s,r,tt,mu,(p,u)::c), e, k) ->
          lazy_eval(p, e, fun lp ->
              lazy_eval(u, e, fun lu ->
                  ( match (lp,lu) with
                      | (CONS x,u) when List.mem x tt ->
                          eval( CMATCH(s,r,tt,Match.extend (x,u,mu),c), e, k)
                      | (CONS x,CONS y) when x = y ->
                          eval( CMATCH(s,r,tt,mu,c), e, k)
                      | (CONS x,CONS y) -> eval(r,e,k)
                      | (CASE(_,_,_),_) -> eval(r,e,k)
                      | (APP(p1,p2),APP(u1,u2)) ->
                          eval( CMATCH(s,r,tt,mu,(p1,u1)::(p2,u2)::c), e, k)
                      | _ -> eval(r,e,k) )))
    | (BOT, _, _) -> VBOT
and lazy_eval = fun
    | (DAT n, e, k) -> k (VDAT d)
    | (VAR x, e, k) -> lazy_eval(Env.lookup (x,e), e, k)
    | (CONS x, e, k) -> k (VCONS x)
    | (CASE(tt,p,s), e, k) -> k VCASE(tt,p,s,e)
    | (CHOICE(tt,p,s,r), e, k) -> k VCHOICE(tt,p,s,r,e)
    | (APP(t,u), e, k) ->
        lazy_eval(t, e, fun lt ->
            ( match lt with
              | VCASE(tt,p,s,e') ->
                     lazy_eval( MATCH(s,tt,Env.empty(),[(p,u)],e'), e, k)
              | VCHOICE(tt,p,s,r,e') ->
                     lazy_eval( CHOICE(s,r,tt,Env.empty(),[(p,u)],e'), e, k)
              | _ -> lazy_eval( lt, e, fun vt ->
                         lazy_eval( u, e, fun vu ->
```

```
                        k APP(vt,vu) ))))
  | (MATCH(_,_,Fail,_), _, _) -> VBOT
  | (MATCH(s,tt,Subst e',[]), e, k) when check(tt,e') ->
      lazy_eval( s, Env.merge e e', k)
  | (MATCH(_,_,_,[]), _, _) -> VBOT
  | (MATCH(s,tt,mu,(p,u)::c), e, k) ->
        lazy_eval(p, e, fun lp ->
            lazy_eval(u, e, fun lu ->
                ( match (lp,lu) with
                    | (CONS x,u) when List.mem x tt ->
                        lazy_eval( MATCH(s,tt,Match.extend (x,u,mu),c), e, k)
                    | (CONS x,CONS y) when x = y ->
                        lazy_eval( MATCH(s,tt,mu,c), e, k)
                    | (CONS x,CONS y) -> VBOT
                    | (CASE(_,_,_),_) -> VBOT
                    | (APP(p1,p2),APP(u1,u2)) ->
                        lazy_eval( MATCH(s,tt,mu,(p1,u1)::(p2,u2)::c), e, k)
                    | _ -> VBOT ) ))
  | (CMATCH(_,r,_,Fail,_), e, k) ->
      lazy_eval( r, e, k)
  | (CMATCH(s,_,tt,Subst e',[]), e, k) when check(tt,e') ->
      lazy_eval( s, Env.merge e e', k)
  | (CMATCH(_,r,_,_,[]), e, k) ->
      lazy_eval( r, e, k)
  | (CMATCH(s,r,tt,mu,(p,u)::c), e, k) ->
        lazy_eval(p, e, fun lp ->
            lazy_eval(u, e, fun lu ->
                ( match (lp,lu) with
                    | (CONS x,u) when List.mem x tt ->
                        lazy_eval( CMATCH(s,r,tt,Match.extend (x,u,mu),c), e, k)
                    | (CONS x,CONS y) when x = y ->
                        lazy_eval( CMATCH(s,r,tt,mu,c), e, k)
                    | (CONS x,CONS y) -> lazy_eval(r,e,k)
                    | (CASE(_,_,_),_) -> lazy_eval(r,e,k)
                    | (APP(p1,p2),APP(u1,u2)) ->
                        lazy_eval( CMATCH(s,r,tt,mu,(p1,u1)::(p2,u2)::c), e, k)
                    | _ -> lazy_eval(r,e,k) )))
  | (BOT, _, _) -> VBOT
```

Defunctionalization.
Type des continuations.

```
type cont =
     | STOP
     | LT_ARG of ppc * ppc Env.t * cont
     | VT_ARG of ppc * ppc Env.t * cont
     | VU_FUN of ppc * cont
     | LP_MATCH of ppc * string list * Match.t * ppc * (ppc * ppc) list *
                                                               ppc Env.t * cont
     | LU_MATCH of ppc * ppc * string list * Match.t * (ppc * ppc) list *
                                                               ppc Env.t * cont
     | LP_CMATCH of ppc * ppc * string list * Match.t * ppc * (ppc * ppc) list *
                                                               ppc Env.t * cont
```

```
     | LU_CMATCH of ppc * ppc * ppc * string list * Match.t * (ppc * ppc) list *
                                                                ppc Env.t * cont
     | LT_ARG_L of ppc * ppc Env.t * cont
     | VT_ARG_L of ppc * ppc Env.t * cont
     | VU_FUN_L of ppc * cont
     | LP_MATCH_L of ppc * string list * Match.t * ppc * (ppc * ppc) list *
                                                                ppc Env.t * cont
     | LU_MATCH_L of ppc * ppc * string list * Match.t * (ppc * ppc) list *
                                                                ppc Env.t * cont
     | LP_CMATCH_L of ppc * ppc * string list * Match.t * ppc * (ppc * ppc) list *
                                                                ppc Env.t * cont
     | LU_CMATCH_L of ppc * ppc * ppc * string list * Match.t * (ppc * ppc) list *
                                                                ppc Env.t * cont

(*  eval        :  ppc * ppc Env.t * cont -> value  *)
(*  lazy_eval   :  ppc * ppc Env.t * cont -> value  *)
(*  apply_cont  :  cont * value -> value            *)

let rec eval = fun
| (DAT n, e, k)              -> apply_cont(k, VDAT d)
| (VAR x, e, k)              -> eval(Env.lookup (x,e), e, k)
| (CONS x, e, k)             -> apply_cont(k, VCONS x)
| (CASE(tt,p,s), e, k)       -> apply_cont(k, VCASE(tt,p,s,e))
| (CHOICE(tt,p,s,r), e, k)   -> apply_cont(k, VCHOICE(tt,p,s,r,e))
| (APP(t,u), e, k)           -> lazy_eval(t, e, LT_ARG(u, e, k))
| (MATCH(_,_,Fail,_), _, _)  -> VBOT
| (MATCH(s,tt,Subst e',[]), e, k) when check(tt,e')
                             -> eval( s, Env.merge e e', k)
| (MATCH(_,_,_,[]), _, _)    -> VBOT
| (MATCH(s,tt,mu,(p,u)::c), e, k)
                             -> lazy_eval(p, e, LP_MATCH(s,tt,mu,u,c,e,k))
| (CMATCH(_,r,_,Fail,_), e, k)
                             -> eval( r, e, k)
| (CMATCH(s,_,tt,Subst e',[]), e, k) when check(tt,e')
                             -> eval( s, Env.merge e e', k)
| (CMATCH(_,r,_,_,[]), e, k) -> eval( r, e, k)
| (CMATCH(s,r,tt,mu,(p,u)::c), e, k)
                             -> lazy_eval(p, e, LP_CMATCH(s,r,tt,mu,u,c,e,k))
| (BOT, _, _)                -> VBOT
and lazy_eval = fun
| (DAT n, e, k)              -> apply_cont(k, VDAT d)
| (VAR x, e, k)              -> eval(Env.lookup (x,e), e, k)
| (CONS x, e, k)             -> apply_cont(k, VCONS x)
| (CASE(tt,p,s), e, k)       -> apply_cont(k, VCASE(tt,p,s,e))
| (CHOICE(tt,p,s,r), e, k)   -> apply_cont(k, VCHOICE(tt,p,s,r,e))
| (APP(t,u), e, k)           -> lazy_eval(t, e, LT_ARG_L(u, e, k))
| (MATCH(_,_,Fail,_), _, _)  -> VBOT
| (MATCH(s,tt,Subst e',[]), e, k) when check(tt,e')
                             -> lazy_eval( s, Env.merge e e', k)
| (MATCH(_,_,_,[]), _, _)    -> VBOT
| (MATCH(s,tt,mu,(p,u)::c), e, k)
                             -> lazy_eval(p, e, LP_MATCH_L(s,tt,mu,u,c,e,k))
```

```
| (CMATCH(_,r,_,Fail,_), e, k)
                            -> lazy_eval( r, e, k)
| (CMATCH(s,_,tt,Subst e',[]), e, k) when check(tt,e')
                            -> lazy_eval( s, Env.merge e e', k)
| (CMATCH(_,r,_,_,[]), e, k) -> lazy_eval( r, e, k)
| (CMATCH(s,r,tt,mu,(p,u)::c), e, k)
                            -> lazy_eval(p, e, LP_CMATCH_L(s,r,tt,mu,u,c,e,k))
| (BOT, _, _)               -> VBOT
and apply_cont = fun
| (STOP, v) -> v
| (LT_ARG(u,e,k), VCASE(tt,p,s,e'))
                            -> eval(MATCH(s,tt,Env.empty(),[(p,u)],e', e, k)
| (LT_ARG(u,e,k), VCHOICE(tt,p,s,r,e')
                            -> eval(CHOICE(s,r,tt,Env.empty(),[(p,u)],e'), e, k)
| (LT_ARG(u,e,k), lt)       -> eval(lt,e,VT_ARG(u,e,k))
| (VT_ARG(u,e,k), vt)       -> eval(u,e,VU_FUN(vt,k))
| (VU_FUN(vt,k), vu)        -> apply_cont(k, APP(vt,vu))
| (LP_MATCH(s,tt,mu,u,c,e,k), lp)
                            -> lazy_eval(u,e,LU_MATCH(s,tt,mu,u,c,e,k))
| (LU_MATCH(CONS x,s,tt,mu,c,e,k), lu) when List.mem x tt
                            -> eval( MATCH(s,tt,Match.extend (x,u,mu),c), e, k)
| (LU_MATCH(CONS x,s,tt,mu,c,e,k), CONS y) when x=y
                            -> eval( MATCH(s,tt,mu,c), e, k)
| (LU_MATCH(CONS x,_,_,_,_,_,_), CONS y)
                            -> VBOT
| (LU_MATCH(CASE(_,_,_),_,_,_,_,_,_), _)
                            -> VBOT
| (LU_MATCH(APP(p1,p2),s,tt,mu,c,e,k), APP(u1,u2))
                            -> eval( MATCH(s,tt,mu,(p1,u1)::(p2,u2)::c), e, k)
| (LU_MATCH(_,_,_,_,_,_,_), _)
                            -> VBOT
| (LP_CMATCH(s,r,tt,mu,u,c,e,k), lp)
                            -> lazy_eval(u,e,LU_CMATCH(s,r,tt,mu,u,c,e,k))
| (LU_CMATCH(CONS x,s,r,tt,mu,c,e,k), lu) when List.mem x tt
                            -> eval( CMATCH(s,r,tt,Match.extend (x,u,mu),c), e, k)
| (LU_CMATCH(CONS x,s,r,tt,mu,c,e,k), CONS y) when x=y
                            -> eval( CMATCH(s,r,tt,mu,c), e, k)
| (LU_CMATCH(CONS x,_,_,_,_,_,_,_), CONS y)
                            -> VBOT
| (LU_CMATCH(CASE(_,_,_),_,_,_,_,_,_,_), _)
                            -> VBOT
| (LU_CMATCH(APP(p1,p2),s,r,tt,mu,c,e,k), APP(u1,u2))
                            -> eval( CMATCH(s,r,tt,mu,(p1,u1)::(p2,u2)::c), e, k)
| (LU_CMATCH(_,_,_,_,_,_,_,_), _)
                            -> VBOT
| (LT_ARG_L(u,e,k), VCASE(tt,p,s,e'))
                            -> lazy_eval(MATCH(s,tt,Env.empty(),[(p,u)],e', e, k)
| (LT_ARG_L(u,e,k), VCHOICE(tt,p,s,r,e')
                            -> lazy_eval(CHOICE(s,r,tt,Env.empty(),[(p,u)],e'), e, k)
| (LT_ARG_L(u,e,k), lt)     -> lazy_eval(lt,e,VT_ARG_L(u,e,k))
| (VT_ARG_L(u,e,k), vt)     -> lazy_eval(u,e,VU_FUN_L(vt,k))
```

```
| (VU_FUN_L(vt,k), vu)       -> apply_cont(k, APP(vt,vu))
| (LP_MATCH_L(s,tt,mu,u,c,e,k), lp)
                             -> lazy_eval(u,e,LU_MATCH_L(s,tt,mu,u,c,e,k))
| (LU_MATCH_L(CONS x,s,tt,mu,c,e,k), lu) when List.mem x tt
                             -> lazy_eval(MATCH(s,tt,Match.extend (x,u,mu),c), e, k)
| (LU_MATCH_L(CONS x,s,tt,mu,c,e,k), CONS y) when x=y
                             -> lazy_eval(MATCH(s,tt,mu,c), e, k)
| (LU_MATCH_L(CONS x,_,_,_,_,_,_), CONS y)
                             -> VBOT
| (LU_MATCH_L(CASE(_,_,_),_,_,_,_,_,_), _)
                             -> VBOT
| (LU_MATCH_L(APP(p1,p2),s,tt,mu,c,e,k), APP(u1,u2))
                             -> lazy_eval(MATCH(s,tt,mu,(p1,u1)::(p2,u2)::c), e, k)
| (LU_MATCH_L(_,_,_,_,_,_,_), _)
                             -> VBOT
| (LP_CMATCH_L(s,r,tt,mu,u,c,e,k), lp)
                             -> lazy_eval(u,e,LU_CMATCH_L(s,r,tt,mu,u,c,e,k))
| (LU_CMATCH_L(CONS x,s,r,tt,mu,c,e,k), lu) when List.mem x tt
                             -> lazy_eval(CMATCH(s,r,tt,Match.extend (x,u,mu),c), e, k)
| (LU_CMATCH_L(CONS x,s,r,tt,mu,c,e,k), CONS y) when x=y
                             -> lazy_eval(CMATCH(s,r,tt,mu,c), e, k)
| (LU_CMATCH_L(CONS x,_,_,_,_,_,_,_), CONS y)
                             -> VBOT
| (LU_CMATCH_L(CASE(_,_,_),_,_,_,_,_,_,_), _)
                             -> VBOT
| (LU_CMATCH_L(APP(p1,p2),s,r,tt,mu,c,e,k), APP(u1,u2))
                             -> lazy_eval(CMATCH(s,r,tt,mu,(p1,u1)::(p2,u2)::c), e, k)
| (LU_CMATCH_L(_,_,_,_,_,_,_,_), _)
                             -> VBOT
```