DEA Programmation : Sémantique, Preuves et Langages
Université Paris VII

Rapport de stage

Une algèbre de filtrage pour le langage $\mathbb{C}$Duce

Kim Nguyễn

Stage de DEA effectué au LIENS dans l'équipe Langages, sous la direction de :

Véronique Benzaken
Équipe Base de Données, LRI, Université Paris-Sud

Giuseppe Castagna
Équipe Langages, Département d'Informatique, ENS Ulm

Alain Frisch
Équipe Langages, Département d'Informatique, ENS Ulm
Projet Cristal, INRIA Rocquencourt

Remerciements

Je tiens à remercier mes directeurs de stages, Giuseppe Castagna, Véronique Benzaken et Alain Frisch. Giuseppe pour sa confiance et son grand investissement personnel, Véronique pour sa grande disponibilité et Alain, pour la qualité de ses conseils et de son encadrement. Tous les trois ont permis que ce stage de recherche soit un réel plaisir. Je remercie aussi les membres des équipes Base de Données et Langages pour leur accueil des plus sympathiques.

Résumé

Le but initial du stage était d'étendre l'algèbre de filtrage de ℂDuce pour permettre la capture «en profondeur» de portions de documents XML. L'intégration de chemins XPATH aux motifs était une des directions envisagées. En essayant d'établir une sémantique claire pour ces chemins, l'idée nous est venue de les exprimer par un petit calcul de combinateurs très simples. Ces combinateurs se sont révélés être d'un grand intérêt puisque permettant d'exprimer une capture en profondeur, mais aussi d'unifier et d'exprimer plusieurs autres traits de ℂDuce : filtrage par motifs, itérateurs polymorphes sur les listes et les arbres,.... Notre travail s'est donc recentré sur ce calcul de combinateurs. Nous en présentons dans ce rapport les fondements théoriques, y établissons une discipline de typage précis et sûr et donnons un algorithme d'inférence de types correct et complet vis à vis de cette discipline de typage.

Table des matières

Chapitre 1

Introduction

1.1 État de l'art et motivations

La manipulation de données semi-structurées a pris ces dernières années un essor nouveau, avec l'arrivée du standard XML. XML est un format de documents ouvert, spécifié par le W3C (cf. [1] et [2]). Il est maintenant largement répandu et utilisé comme format d'échange commun de nombreuses applications et protocoles. Informellement, un document XML, représente des données sous la forme d'un arbre d'arité non fixée. Un exemple bien connu de document XML est donné à la figure 1.1.

Les opérations que l'on souhaite effectuer sur de tels documents sont :

- en assurer la validité ;
- en extraire des données ;
- les transformer en d'autres documents XML.

Le format XML en lui-même ne permet pas de donner une sémantique aux documents, les seules contraintes qu'il impose étant des contraintes de bon parenthésage des balises (aussi appelées *tag*) ainsi que des contraintes sur le jeu de caractères utilisés. L'ordre, la disposition et la signification des balises *n'est pas propre* au document. Il existe cependant plusieurs outils pour spécifier le contenu d'un document XML, les plus connus étant les DTD (Document Type Definition) et XMLSchema (celui-ci, plus riche que les DTD permet d'imposer des contraintes de cardinalités sur les éléments d'un document, d'en fixer l'ordre, ...). De même, pour l'extraction de données et la transformation de documents, il existe des spécifications parmis lesquelles XPATH, XQuery ou XSLT (cf [3], [4] et [5]). Il existe plusieurs implémentations de ces spécifications du W3C (Galax, libxslt,...). Ces spécifications étant très riches, les implémentations existantes sont soit inefficaces en pratique, soit optimisées pour des sous-ensembles des spécifications.

```
<contacts>
   <person>
      <name>Kim Nguyen</name>
      <email>kn@di.ens.fr</email>
   </person>
   <person>
      <name>Giuseppe Castagna</name>
      <phone>0144322082</phone>
   </person>
</contacts>
```

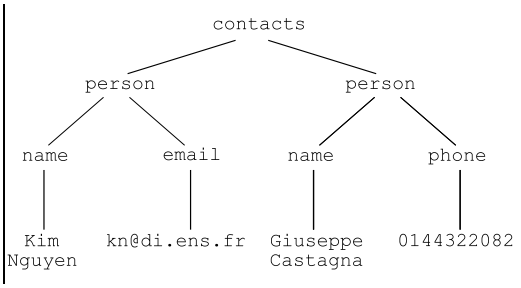

FIG. 1.1 – Un document XML stockant un carnet d'adresses

Une approche novatrice est celle qui a consisté à utiliser les résultats de la théorie des langages afin de créer un *langage de manipulation* de documents. Celà fut le cas de XDuce puis de ℂDuce (cf. [6] et [7] respectivement). Dans ces langages, les documents sont naturellement des valeurs de première classe et les contraintes sur ces documents ne sont tout simplement que *leur type*. De même l'extraction et la transformation de documents s'écrivent simplement comme des fonctions dans ces langages. Le système de type de ces langages est donc l'un des points clés pour une manipulation efficace. En effet :

- Un système de type riche permet d'exprimer aisement des documents complexes
- Un typage *statique* permet de typer les programmes à la compilation et donc de *garantir* que tous les documents engendrés par ces programmes sont valides. Le typage statique permet en outre d'éviter les nombreux tests dynamiques qui sont courants dans les autres approches
- L'inférence de type, lorsqu'elle est réalisable, est une aide précieuse pour le programmeur qui lui permet de détecter très tôt les erreurs
- Au moment de l'exécution, des informations déterminées au moment du typage peuvent être réutilisées à des fins d'optimisation.

1.2 Le langage ℂDuce

ℂDuce est un langage adapté à la manipulation de documents XML. Certaines de ses caractéristiques seront expliquées en détail au chapitre 2, cependant on peut d'ores et déjà noter que ℂDuce :

- Possède une algèbre de type riche : types de bases, type flèche, enregistrements et produits, types recursifs, combinaisons booléennes de types.
- Une relation de sous-typage ensembliste (sous-typage sémantique).
- Un polymorphisme *ad-hoc* par le biais de fonctions surchargées.
- Un opérateur de *pattern-matching* très précis permettant une extraction efficace des données.

De plus, il utilise un schéma de compilation très performant, utilisant des informations de typage à des fins d'optimisations.

Malgré toute la richesse de ce système de type, il existe des types d'opérations courantes pour le traitement de documents XML que l'on ne peut exprimer précisément en ℂDuce [1]. Les documents XML étant des arbres $n-aires$, les diverses opérations que l'on veut effectuer (regroupant l'extraction d'éléments, la transformation d'arbres,...) sont naturellement des itérateurs d'arbres. Ce sont, par nature, des opérateurs *polymorphes* (on peut penser à `map` par exemple, mais ce n'est pas le seul). Cependant, la nature de ce polymorphisme est assez différente de celle plus connue du polymorphisme à la ML. En effet, les collections en ℂDuce (arbres, listes,...) sont hétérogènes, ce que ne permet pas d'exprimer le système de type de ML. De plus, on souhaite que le typage de ces opérations soit extrèmement *précis*[2], car celà est nécéssaire si l'on souhaite *composer* des transformations.

1.3 But du stage et travail réalisé

Le but de notre stage a été de faire cohabiter cette notion de polymorphisme bien particulière avec un typage précis. La solution que nous avons retenue à été de définir un petit calcul de combinateurs, assez riche pour permettre d'exprimer

[1] On entend ici opérations pour lesquelles on n'arrive pas à avoir un typage précis. ℂDuce étant évidemment un langage Turing-complet, on peut y exprimer lesdites opérations et les typer... Tout le problème réside dans la précision du type obtenu.

[2] Nous expliciterons plus loin cette notion de précision.

ces opérateurs polymorphes mais cependant non Turing-complet. En effet, certaines restrictions sur le pouvoir expressif de ses opérateurs sont nécéssaires pour garantir la terminaison de certains algorithmes (inférence de type en particulier).

Après un bref rappel sur certains aspects essentiels de $\mathbb{C}$Duce nécéssaire à la compréhension de notre travail, nous présentons ce calcul de combinateurs, appelés *filtres*.

Chapitre 2

Cadre de travail

2.1 Rappels sur le langage ℂDuce

Dans le cadre de ce rapport, on se place dans une version réduite du langage ℂDuce, assez riche toutefois pour y faire apparaître les problèmes décrits précédement. La présentation de ℂDuce faite ici est très rapide et l'on se réferrera à [8] et [9] pour une présentation plus formelle du langage.

2.1.1 Système de types et sous-typage sémantique

Types

Les types en ℂDuce sont les termes clos et finis engendrés par :

$$\begin{array}{rcl} T & = & \alpha \mid C \mid \mu\alpha.C \\ C & = & A \mid C \wedge C \mid C \vee C \mid \neg C \\ A & = & T \to T \mid T \times T \mid b \mid \mathsf{Any} \mid \mathsf{Empty} \end{array}$$

b représente un ensemble de types de base (Int, Bool,...).

De manière un peu plus formelle, les types sont définis par une fonction d'interprétation ensembliste $[\![\_]\!] : \mathscr{I} \to \mathscr{P}(\mathscr{D})$, où $\mathscr{I}$ est l'ensemble des types et $\mathscr{D}$ un domaine (qu'on ne précise pas ici). $[\![\_]\!]$ est défini par :

$$\begin{array}{l} [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!] \\ [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \\ [\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \\ [\![\neg t]\!] = \mathscr{D} \setminus [\![t]\!] \\ [\![\mathsf{Any}]\!] = \mathscr{D} \\ [\![\mathsf{Empty}]\!] = \varnothing \\ \ldots \end{array}$$

La définition exacte est donnée très précisément dans [9], l'interprétation du type flèche en particulier est non-triviale.

Sous-typage

Après cette présentation ensembliste des types, la relation de sous-typage devient alors évidente :

$$t \leq s \text{ si et seulement si } [\![t]\!] \subseteq [\![s]\!]$$

Le fait que le sous-typage soit défini comme l'inclusion ensembliste donne gratuitement des propriétes sur le sous-typage (la transitivité en particulier, qui est nécessaire à la préservation du typage).

Types de bases et exemples

Dans la suite, nous illustrerons nos propos par plusieurs exemples. Ces derniers font intervenir plusieurs types de base, nous en donnons ici un apperçu :

- les types bien connus Int, String, Bool, ...
- les *tags* : `'Nil`, `'A`, `'Foo`, ...
- Les valeurs (à l'exception des fonctions) sont aussi des types : `1` est le type de l'entier `1` et $\texttt{1} \leq \mathsf{Int}$. De même, $\mathsf{Bool} \equiv \texttt{'True} \vee \texttt{'False}$ et on a donc que $\texttt{'True} \leq \mathsf{Bool}$ et $\texttt{'False} \leq \mathsf{Bool}$

2.1.2 Opérateur de *pattern-matching*

ℂDuce est muni d'un opérateur de *pattern-matching* très puissant, possédant un typage très précis et de l'inférence de type. Nous nous bornons ici à donner une intuition de son fonctionnement et les notions essentielles en terme de typage qui y sont associées.

Motif

Un motif (*pattern*) est un terme infini engendré par la grammaire suivante :

p	$::=$	x	capture
	$\mid$	t	contrainte de type
	$\mid$	$p\&p$	conjonction
	$\mid$	$p\|p$	alternative
	$\mid$	(p, p)	produit
	$\mid$	$(x := c)$	constante

où $x \in \mathbb{V}$ l'ensemble des variables et $c \in \mathbb{C}$ l'ensemble des constantes. On considère l'ensemble $\mathbb{P}$ des motifs bien formés comme l'ensemble des productions de la grammaire ci-dessus qui vérifient en plus certaines conditions de bonne formation que nous ne détaillons pas ici (régularité, conditions sur les variables capturées...).

On note v/p le filtrage d'une valeur par un motif. v/p est une substitution des variables de captures de p vers les valeurs, ou de manière équivalente, un ensemble de couples (variable,valeur). L'application v/p peut échouer (par exemple, si on applique un motif produit à une valeur qui n'est pas un couple, ou si une contrainte de type n'est pas satisfaite). On note t/p le filtrage d'un type par un motif : c'est une substitution des variables de captures de p vers les types.

Type des motifs

Le type $\lfloor p \rfloor$ d'un motif p est exactement l'ensemble des valeurs v telles que v/p n'échoue pas. Ce type est inféré lors du typage des motifs et permet de detecter :

- les motifs superflus (qui ne seront jamais utilisées)
- les motifs incomplets (le motif peut échouer sur certaines valeurs).

Exemples

Les motifs sont utilisés en ℂDuce avec la construction `match with` rappelant celle d'OCaml. Exemple :

```
match x with
|    Int -> x+1
|    (y&Int,x&Int) -> y+z
|    String -> to_int(x)+1
```

Pour chaque motif p à gauche, on calcule x/p et s'il n'y a pas d'échec, on calcule l'expression de droite dans l'environnement augmenté de x/p. Notons que si x n'est pas d'un type t, tel que $t \leq \mathsf{Int} \vee (\mathsf{Int} \times \mathsf{Int}) \vee \mathsf{String}$, alors il se produit une erreur de typage.

2.2 Notations

Dans la suite, nous utilisons les notations suivantes que nous rappelons sans redéfinir formellement :

- nous noterons Γ les environnements de typage, *i.e.* des substitutions des variables vers les types.
- de même nous notons γ les environnements d'évaluation, *i.e.* des substitutions des variables vers les valeurs.
- $v : t$: la valeur v a le type t
- $\gamma : \Gamma$ pour dire que pour tout (x, v) dans γ, il existe t tel que (x, t) appartient à Γ et $v : t$.
- $\sqsubseteq$ est le pré-ordre sous-terme.
- nous utilisons l'ordre sous-terme strict $\sqsubset$ induit par $\sqsubseteq$ sur des termes infinis (réguliers) en le définissant comme : $f \sqsubset g$ si : $f \sqsubseteq g$ et $g \not\sqsubseteq f$

Chapitre 3

Les filtres

3.1 Présentation

Rappelons que notre but est de pouvoir définir des opérateurs polymorphes avec typage précis. Prenons l'exemple simple de l'opérateur `fst` qui renvoie la première composante d'un couple. Pour que cet opérateur soit le plus générique possible, il est nécessaire de lui donner le type $\mathsf{Any} \times \mathsf{Any} \to \mathsf{Any}$. Celà n'est pas satisfaisant car on obtient alors que `fst(1,2)` est de type Any : il y a une *perte d'information* au niveau du typage. Une approche « frontale » consite à faire apparaitre le polymorphisme au niveau du système de types ℂDuce, en l'étendant vers un système de types avec polymorphisme implicite (à la ML) ou explicite (système F), tout en conservant la relation de sous-typage. Cette extension a déjà été considéré dans [10] et plusieurs problèmes apparaissent, notamment la non-décidabilité de la relation de sous-typage ([11]).

L'approche que nous choisissons est différente et permet de laisser inchangé le système de types de ℂDuce. Nous définissons un petit calcul de combinateurs, appelés *filtres*[1]. Ces derniers ne sont pas considérés comme des valeurs de première classe, mais plutôt comme des «macros» ou «*template*». Ils ne sont donc pas typés lors de leur déclaration mais lors de leur application. Il permettent donc d'écrire un code générique tout en conservant un typage précis (la précision venant du fait que le type exact des arguments est connu au moment du typage). Par exemple, il est possible de définir `fst` comme un filtre, et l'on a alors que :

- `fst` n'est pas une valeur du langage
- `fst(1,2)` est bien de type `1` (on se souvient que `1` est le type de la valeur `1`, et est sous-type de Int)

Cette approche avait déjà été retenue pour ℂDuce de manière limitée. En effet, il existe dans le langage des opérateurs primitifs, qui ne sont pas de première classe, et disposant de règles de typage spécifiques : `@` (concat), `map`, `transform`, `xtransform`. Cette solution a cependant des limites, car il n'est pas concevable de devoir ajouter au langage de manière primitive un nouvel opérateur ainsi que les règles de typage associées, à chaque fois que le besoin s'en fait sentir. Notre solution permet d'unifier ce procédé en laissant à l'utilisateur la possibilité de définir ses propres opérateurs. Cette approche « méta-programmation » ne doit cependant pas faire perdre de vue que l'on définit pour ces filtres une discipline de typage et qu'ils possèdent des propriétés bien précises. Ce ne sont donc pas de simples « macros syntaxiques ».

Nous pouvons maintenant définir formellement ces filtres, en donner la sémantique puis traiter du typage des filtres et des propriétés de ces derniers.

[1] Le nom n'est pas très original et déjà abondant dans la littérature, il n'est donc pas définitif...

3.2 Définitions

Définition 1 (Filtres). Un filtre est un terme (possiblement infini) *régulier* défini par la grammaire suivante :

$$\begin{array}{lcll} f & ::= & e & \text{expression} \\ & | & p \to f & \text{pattern, } p \in \mathbb{P} \\ & | & f; f & \text{séquence} \\ & | & (f, f) & \text{paire} \\ & | & f \mid f & \text{union} \end{array}$$

Définition 2 (Variables capturées par un filtre). On définit $Var(f)$ l'ensemble des variables capturées par le filtre f comme :

$$\begin{array}{lcl} Var(e) & = & \varnothing \\ Var(f_1; f_2) & = & Var(f_1) \cup Var(f_2) \\ Var(f_1 | f_2) & = & Var(f_1) \cup Var(f_2) \\ Var((f_1, f_2)) & = & Var(f_1) \cup Var(f_2) \\ Var(p \to f) & = & Var(p) \cup Var(f) \end{array}$$

Définition 3 (Portée des variables capturées). On défini $FreeVar(f)$, l'ensemble des variables libres d'un filtre f :

$FreeVar(e)$ est défini pour les expressions que l'on considère.

$$\begin{array}{lcl} FreeVar(f_1; f_2) & = & FreeVar(f_1) \cup FreeVar(f_2) \\ FreeVar(f_1 | f_2) & = & FreeVar(f_1) \cup FreeVar(f_2) \\ FreeVar((f_1, f_2)) & = & FreeVar(f_1) \cup FreeVar(f_2) \\ FreeVar(p \to f) & = & FreeVar(f) \setminus Var(p) \end{array}$$

Un filtre f doit vérifier les propriétés suivantes :

Variables : f est clos, *i.e.* $FreeVar(f) = \varnothing$.

Régularité : le nombre de sous-termes distincts de f est fini.

Déconstruction : sur chaque branche infinie de f il y a un nombre infini d'occurences du constructeur $(\_, \_)$.

et tout sous-terme f' de f vérifie :

Composition limitée : si f' est de la forme $f_1; f_2$, alors f' n'est pas un sous-terme de f_2.

Capture de variables : si f' est de la forme $p \to f''$ alors :

$$Var(p) \cap Var(f'') = \varnothing$$

Remarque : ces critères de bonne formation garantissent la terminaison de l'évaluation et du typage d'un filtre (on donne des contre-exemples en section 3.4.1).
Notation : nous notons $\mathbb{F}$ l'ensemble des filtres ainsi définis.

3.3 Évaluation

Nous présentons ici là sémantique dynamique des filtres et démontrons la terminaison de l'évaluation de ces derniers.

Définition 4 (Sémantique dynamique des filtres). L'application d'un filtre f à une valeur v dans un environnement γ est donné par le jugement :

$$\gamma \vdash_e f(v) \rightsquigarrow r \text{ où } r \text{ est soit une valeur, soit } \Omega.$$

La sémantique dynamique des filtres est donnée par les règles d'inférence suivantes :

$$\frac{}{\gamma \vdash_e e(v) \rightsquigarrow r} \quad \text{avec } r = \texttt{eval}(\gamma, \texttt{e}) \qquad \textbf{(e-expr)}$$

$$\frac{\gamma \vdash_e f_1(v_1) \rightsquigarrow r_1 \ \gamma \vdash_e f_2(v_2) \rightsquigarrow r_2}{\gamma \vdash_e (f_1, f_2)((v_1, v_2)) \rightsquigarrow (r_1, r_2)} \quad \text{si } r_1 \neq \Omega \text{ et } r_2 \neq \Omega \qquad \textbf{(e-prod-ok)}$$

$$\frac{\gamma \vdash_e f_1(v_1) \rightsquigarrow r_1 \ \gamma \vdash_e f_2(v_2) \rightsquigarrow r_2}{\gamma \vdash_e (f_1, f_2)((v_1, v_2)) \rightsquigarrow \Omega} \quad \text{si } r_1 = \Omega \text{ ou } r_2 = \Omega \qquad \textbf{(e-prod-err1)}$$

$$\frac{}{\gamma \vdash_e (f_1, f_2)(v) \rightsquigarrow \Omega} \quad \text{si } v \not\equiv (v_1, v_2) \qquad \textbf{(e-prod-err2)}$$

$$\frac{\gamma \cup v/p \vdash_e f(v) \rightsquigarrow r}{\gamma \vdash_e (p \rightarrow f)(v) \rightsquigarrow r} \quad \text{si } v/p \neq \Omega \qquad \textbf{(e-patt-ok)}$$

$$\frac{}{\gamma \vdash_e (p \rightarrow f)(v) \rightsquigarrow \Omega} \quad \text{si } v/p = \Omega \qquad \textbf{(e-patt-err)}$$

$$\frac{\gamma \vdash_e f_1(v) \rightsquigarrow r_1 \ \gamma \vdash_e f_2(r_1) \rightsquigarrow r_2}{\gamma \vdash_e (f_1; f_2)(v) \rightsquigarrow r_2} \quad \text{si } r_1 \neq \Omega \qquad \textbf{(e-comp-ok)}$$

$$\frac{\gamma \vdash_e f_1(v_1) \rightsquigarrow \Omega}{\gamma \vdash_e (f_1; f_2)(v) \rightsquigarrow \Omega} \qquad \textbf{(e-comp-err)}$$

$$\frac{\gamma \vdash_e f_1(v) \rightsquigarrow r_1}{\gamma \vdash_e (f_1 | f_2)(v) \rightsquigarrow r_1} \quad \text{si } r_1 \neq \Omega \qquad \textbf{(e-union1)}$$

$$\frac{\gamma \vdash_e f_1(v) \rightsquigarrow \Omega \ \gamma \vdash_e f_2(v) \rightsquigarrow r_2}{\gamma \vdash_e (f_1 | f_2)(v) \rightsquigarrow r_2} \qquad \textbf{(e-union2)}$$

Ω représente une erreur à l'évaluation. Notons que la règle **(e-expr)**, fait intervenir l'évaluation des expressions. Nous supposons que leur évaluation est déterministe.

Nous montrons maintenant que le filtrage d'une valeur termine à la condition que tout les filtres *expression* contenu dans le filtre appliqué terminent. Notre but est en effet de montrer que la non-terminaison de l'évaluation d'un filtre est liée uniquement aux expressions qu'il contient et non pas à la structure du filtre lui-même. De manière pratique, celà signifie que la non-terminaison du filtre est entièrement controlée par le programmeur.

Définition 5. On définit la fonction $c : \mathbb{F} \rightarrow \mathbb{N} \cup \{\infty\}$ comme la profondeur du premier constructeur $(\_, \_)$ dans f. ($c(f) = \infty$ si $(\_, \_)$ n'apparait pas dans f).

Théorème 1 (Terminaison du filtrage). *Soit v une valeur du langage et f un filtre bien formé dont tous les sous-termes* expression *terminent. L'évaluation de $f(v)$ termine.*

> ***Preuve (Terminaison du filtrage) :***
> On raisonne par induction sur le triplet $(f, v, c(f))$ muni de l'ordre $(\sqsubset, \sqsubset, <_{\mathbb{N}})_{lex}$. L'idée de la preuve est de montrer qu'à chaque étape de réduction :
> – soit l'on évalue un sous-terme strict de f. Le nombre de sous-terme disctincts d'un arbre régulier étant fini, l'évaluation termine (rappelons que l'ordre sous-terme strict est un ordre bien fondé pour les arbres réguliers).
> – soit l'on décompose une paire en ses deux composantes (les valeurs étant des termes finis, l'évaluation termine).

– soit l'on se rapproche strictement d'un filtre (f_1, f_2) qui forcera soit à déstructurer la valeur évaluée, soit échouera si la valeur n'est pas un couple.

Par cas sur les règles d'évaluation :

(e-eval) : par hypothèse, toutes les expressions du filtre terminent.

(e-prod-ok) : ici, la mesure décroit strictement en v car on déstructure une la paire (v_1, v_2) (et bien évidement, $f_1 \sqsubseteq (f_1, f_2)$ et $f_2 \sqsubseteq (f_1, f_2)$, donc la première composante n'augmente pas). Par hypothèse d'induction, les deux premisses terminent, donc l'évaluation termine.

(e-prod-err1) : identique au précédent.

(e-prod-err2) : ici l'évaluation se termine directement par Ω (pas de prémisse).

(e-patt-ok) : Ici, soit f contient un constructeur $(\_, \_)$, auquel cas on a $c(f) < c(p \to f)$, soit il n'en contient pas et il est donc fini (car un filtre infini contient une infinité de constructeurs $(\_, \_)$), donc l'évaluation termine.

(e-patt-err) : l'évaluation termine directement avec Ω

(e-comp-ok) : pour la deuxième prémisse, f_2 est un sous-terme strict de $f_1; f_2$ (par définition). L'évaluation de $f_2(r_1)$ termine par hypothèse d'induction.

Pour la première prémisse :

– Soit le filtre f_1 contient un constructeur $(\_, \_)$. Dans ce cas, la troisième composante diminue strictement : $c(f_1) < c(f_1; f_2)$.

– Soit le filtre f_1 ne contient pas de constructeur $(\_, \_)$, il est donc forcément fini (car un filtre infini contient une infinité d'occurences du constructeur $(\_, \_)$). L'évaluation d'un filtre fini termine (car à chaque étape d'évaluation on est réduit à un sous-terme strict).

(e-comp-err) identique à la première prémisse de **(e-comp-ok)**

(e-union1) identique à la première prémisse de **(e-comp-ok)**

(e-union2) identique à la première prémisse de **(e-comp-ok)**

□

3.4 Exemples

Avant de présenter le typage des filtres, nous illustrons leur comportement par des exemples, et montrons comment plusieurs opérateurs de ℂDuce peuvent être vus comme des filtres. Dans ces exemples, nous notons les filtres récursifs par des termes de la forme $\mu X.f$. Cette notation est réutilisée dans la suite pour l'algorithme de typage et une définition précise est donnée à la section 3.5.2.

3.4.1 Filtres malformés

Ces quelques contre-exemples illustrent la nécécité des conditions de bonne formation que l'on a imposées aux filtres.

Condition du (_, _)

On voit que si l'on n'impose pas la condition qu'il y ait une infinité d'occurences de (_, _) sur une branche infinie alors on peut écrire des filtres trivialement divergents :

$$\texttt{illformed} \equiv \mu X.(X|X)$$

Condition du ;

Si l'on ne force pas le membre droit d'un ; à être un sous-terme strict du filtre, alors l'évaluation ne termine pas :

$$\texttt{loop} \equiv \mu X.((x \to x, x \to x); X).$$

Il y a bien une infinité d'occurences du constructeur (_, _) mais `loop( (`v_1, v_2`))` ne termine pas. Il applique l'identité à v_1 et à v_2, reconstruit le couple (v_1, v_2) et se rappelle récursivement...

3.4.2 Filtres et motifs

Les filtres pouvant contenir des motifs, on peut exprimer la construction `match ... with` de ℂDuce par un filtre :

$$\begin{array}{c}\texttt{match}\ v\ \texttt{with}\\ p_1 \to e_1\\ \vdots\\ |p_n \to e_n\end{array}$$

est équivalent à l'expression : $(p_1 \to e_1 | \ldots | p_n \to e_n)(v)$

3.4.3 Filtres et listes

Les listes en ℂDuce

Les listes en ℂDuce suivent le modèle des listes « à la Lisp », *i.e.* une paire (élément, suite) ou une liste vide, dénotée par une constante spéciale `'Nil`. Le type des listes d'entiers est donc : $\mu\alpha.(\texttt{'Nil} \vee (\mathsf{Int} \times \alpha))$, que l'on note plus simplement [Int*]. De même, une liste hétérogène plus complexe de type :

$$\mu\alpha.(((\mathsf{Int} \vee \mathsf{Bool}) \times \alpha) \vee (\mu\beta.(\mathsf{String} \times \beta) \vee (\mathsf{String} \times \texttt{'Nil})))$$

sera notée [(Int|Bool)* String+]. Nous montrons maintenant que les filtres permettent de définir et typer précisément les opérations classiques sur les listes.

Premier élement

Soit le filtre :

$$\texttt{head} \equiv (x, \mathsf{Any}) \to x$$

Ce filtre renvoie le premier élement d'une liste (plus exactement la première composante d'un couple...).

Reste d'une liste

$$\texttt{tail} \equiv (\mathsf{Any}, x) \rightarrow x$$

Le symétrique du précédent. On remarque que le typage garantit que ces opérateurs sont appliqués à des listes de taille non-nulle. En effet, si le type d'entrée contient `'Nil`, alors le typage de l'application du filtre à une valeur pouvant être `'Nil` échoue.

Concaténation de deux listes

$$@ \equiv (x, y) \rightarrow (x; \mu X.(\texttt{'Nil} \rightarrow y | (z \rightarrow z, X)))$$

Le filtre capture les deux arguments grâce au motif (x, y) puis applique un filtre récursif (X) sur la première liste (x). Ce filtre récursif ne fait qu'itérer la liste jusqu'à arriver à `'Nil`, qu'il remplace alors par la deuxième liste (y).

Applatissement d'une liste de listes

$$\texttt{flatten} \equiv \mu X.(\texttt{'Nil} \rightarrow \texttt{'Nil} | (z \rightarrow z, X); @)$$

Ce filtre montre que l'on peut réutiliser des filtres déja définis. Pour applatir une liste de listes, on itère jusqu'à trouver `'Nil`, puis on remonte en concaténant les listes. Notons que comme pour la concaténation d'une liste, les types sont entièrement préservés : si on a l de type [[Int* Bool Char] [Any+] [Bool] [Bool* (String|Int)]] alors le type de `flatten`(l) est : [Int* Bool Char Any+ Bool+ (String|Int)]

Map

$$\texttt{map} \equiv (f, l) \rightarrow (l; \mu X.(\texttt{'Nil} \rightarrow \texttt{'Nil} | (z \rightarrow f(z), X)))$$

Ici, on capture dans f une fonction et l'on itère la liste l en appliquant f tour à tour à chaque élément. Considérons l'exmple de la fonction f (surchargée) de type Int $\rightarrow$ Bool; String $\rightarrow$ Bool; Char $\rightarrow$ Int. La figure 3.1 donne le type de sortie de `map`(f,l) en fonction du type de l

type de l	type de `map`(f,l)
[Int*]	[Bool*]
[String Int*]	[Bool+]
[String* Int*]	[Bool*]
[String* Bool Int*]	[Bool* Int Bool]
[(String|Int|Char)*]	[(Bool|Int)*]

FIG. 3.1 – Types d'entrée et de sortie de `map`

3.4.4 Filtres et arbres

Nous donnons maintenant des exmples plus complexes faisant intervenir des arbres XML. On utilise l'encodage en arbre binaire : un arbre est un couple dont la première composante est le *tag* et la deuxième est soit une liste d'arbres soit une donnée.

Transformation

Voici un premier filtre simple sur les arbres qui transforme tous les *tags* `'A` en *tag* `'B` :

$$\texttt{trans\_ab} \equiv \mu X.((\texttt{`A} \rightarrow \texttt{`B} | z \rightarrow z, \mu Y.(\texttt{`Nil} \rightarrow \texttt{`Nil} | (X, Y)))$$

Ce filtre peut être appliqué à tout arbre contenant un *tag* `'A`. Le type de sortie est exactement le type d'entrée dans le quel les `'A` sont remplacés par des `'B`. Avec une fonction, à la place d'un filtre, le type de sortie aurait vu les *tags* autres que `'A` remplacés par Any.

Extraction

Le filtre suivant renvoit une séquence contenant le premier sous-arbre (*i.e.* le plus à gauche) de l'arbre de départ dont le *tag* est `'A`, il renvoie la séquence vide `'Nil` si un tel sous-arbre n'existe pas :

$$\begin{array}{l}
\texttt{extract} \equiv\ \mu\ X.(\\
\qquad\qquad (\texttt{`A},\ x)\ \rightarrow\ [x] \\
\qquad\quad |\ (z \rightarrow z, \mu Y(\\
\qquad\qquad\quad \texttt{`Nil} \rightarrow \texttt{`Nil} \\
\qquad\qquad\quad |\ (X, z \rightarrow z)); (x, \mathsf{Any}) \rightarrow\ x \\
\qquad\qquad\quad |\ (z \rightarrow\ z, Y); (\mathsf{Any}, x) \rightarrow\ x) \\
\qquad\quad) \\
\qquad) \\
\qquad)
\end{array}$$

Ce filtre peut être appliqué à tout arbre et il renverra un sous-arbre si un *tag* `'A` est présent dans ce dernier. Il est interressant de noter que si l'on supprime les deux lignes `'Nil` $\rightarrow$ `'Nil`, alors le typage de l'application échouera si le *tag* `'A` n'apparaît pas dans le type.

Transformation généralisée

On donne ici l'expression générale d'un itérateur d'arbre. Si $f1$, ..., fn sont n filtres transformant des arbres, alors le filtre :

$$\begin{array}{l}
\texttt{transform} \equiv\ \mu\ X.(\\
\qquad\qquad f1 | \ldots | fn \\
\qquad\quad |\ (z \rightarrow z, \mu Y(\\
\qquad\qquad\quad \texttt{`Nil} \rightarrow \texttt{`Nil} \\
\qquad\qquad\quad |\ (X, z \rightarrow z)) \\
\qquad\qquad\quad |\ (z \rightarrow\ z, Y) \\
\qquad\quad) \\
\qquad) \\
\qquad)
\end{array}$$

applique $f1|\ldots|fn$ à chaque noeud de l'arbre, transformant ceux qui correspondent et laissant les autres inchangés.

Vers des chemins XPath

Il est possible d'ajouter du sucre syntaxique aux filtres pour avoir des expressions ressemblant aux chemins XPath[2]. Le sous-ensemble des chemins XPath qui nous interesse est l'ensemble des productions finies non-vides de la grammaire :

$$C \quad ::= \quad \epsilon \mid /tag\ C \mid //tag\ C$$

[2]le but initial du stage était d'apporter un moyen de faire du filtrage en profondeur avec une syntaxe à la XPath, notre approche s'est donc révélée valide...

Leur interprétation est la suivante : `/<A>` renvoie le noeud courrant si le tag est `<A>` et `//<A>` renvoie tous les noeuds dont le tag est `<A>`. On peut bien sur composer les chemins de manière à avoir : `/<B>//<A>/<C>`. Ce chemin vérifie que l'arbre à bien un tag `<B>` et renvoie tous les sous-arbres `<C>` dont le père est `<A>`. Le but est d'exprimer des transformations du style *chemin* $\rightarrow e$, qui transforme tous les sous-arbres dénotés par le chemin donné avec l'expression e. On remarque que : `/<A>`$\rightarrow e$ est équivalent au filtre $(\texttt{<A>}, x) \rightarrow e$ et `//<A>`$\rightarrow e$ est exactement le filtre `transform` donné dans l'exemple précédent, où $f1$ vaut $(\texttt{<A>}, x) \rightarrow e$. On peut ainsi récursivement transformer tout chemin en une imbrication de filtres.

3.5 Typage

Nous présentons maintenant le typage des filtres. Le typage du filtre doit permettre de déterminer de manière statique les choses suivantes :

- Sûreté du typage : on souhaite bien sûr que le type des valeurs acceptés par le filtre soit soit un sous-ensemble de l'ensemble des valeurs qui ne provoquent pas d'erreur d'évaluation.
- type des valeurs renvoyées par le filtre : on souhaite qu'il soit le plus précis possible.
- Les informations récupérées lors du typages doivent pouvoir être réutilisées de manière à optimiser la compilation du filtre (comme celà est fait pour la compilation du *pattern-matching*).

Définition 6 (Type acceptable par un filtre). On associe à tout filtre f le type $\lfloor f \rfloor$ des avaleurs acceptables par f :

$$\begin{array}{lcl} \lfloor e \rfloor & \equiv & \mathsf{Any} \\ \lfloor (f_1, f_2) \rfloor & \equiv & \lfloor f_1 \rfloor \times \lfloor f_2 \rfloor \\ \lfloor f_1 | f_2 \rfloor & \equiv & \lfloor f_1 \rfloor \vee \lfloor f_2 \rfloor \\ \lfloor p \rightarrow f \rfloor & \equiv & \lfloor p \rfloor \wedge \lfloor f \rfloor \\ \lfloor f_1 ; f_2 \rfloor & \equiv & \lfloor f_1 \rfloor \end{array}$$

La condition que le type d'une valeur filtrée par f soit sous-type de $\lfloor f \rfloor$ est une condition nécessaire mais non suffisante :

$$\forall v \notin \lfloor f \rfloor, f(v) \rightsquigarrow \Omega$$

Celà est dû à l'opérateur de composition, $f_1; f_2$. Une condition suffisante serait que le type de *sortie* de f_1 soit sous-type de $\lfloor f_2 \rfloor$. Il nous faut donc déterminer le type de sortie d'un filtre pour pouvoir garantir la sûreté de la règle de composition.

Nous proposons dans un premier temps un système de types sous forme de règles d'inférence dans lequel nous montrons l'existence d'un tel type ainsi que la préservation du typage. Nous présenterons ensuite un algorithme pour effectivement calculer le type s.

3.5.1 Jugement de typage

Soit le jugement $\Gamma \vdash f(t) = s$ signifiant qu'un filtre appliqué à une valeur de type t renvoie une valeur de type s. On appelle $\mathscr{F}$ le système de déduction associé, définit par l'ensemble de règles de la figure 3.2. Nous choisissons de considérer les dérivations comme étant possiblement *infinies*.

Notons en premier lieu que chaque règle exige dans ses prémisses que le type fourni en entrée soit compatible avec le type du filtre considéré. On peut aussi remarquer que :

$$\frac{\mathtt{type}(\Gamma, \mathtt{e}) = s}{\Gamma \vdash e(t) = s} \qquad \textbf{(t-expr)}$$

$$\frac{\begin{array}{l} t \leq \lceil f_1 \rfloor \times \lceil f_2 \rfloor \\ t \simeq \bigvee_{(t_1^i, t_2^i) \in \pi(t)} (t_1^i \times t_2^i) \end{array} \quad \Gamma \vdash f_1(t_1^i) = s_1^i \quad \Gamma \vdash f_2(t_2^i) = s_2^i}{\Gamma \vdash (f_1, f_2)(t) = \bigvee_i (s_1^i \times s_2^i)} \qquad \textbf{(t-prod)}$$

$$\frac{\Gamma \cup t/p \vdash f(t) = s \quad t \leq \lceil p \rfloor \wedge \lceil f \rfloor}{\Gamma \vdash (p \to f)(t) = s} \qquad \textbf{(t-patt)}$$

$$\frac{\begin{array}{l} t \leq \lceil f_1 \rfloor \vee \lceil f_2 \rfloor \\ t_1 = t \wedge \lceil f_1 \rfloor \\ t_2 = t \wedge \neg \lceil f_1 \rfloor \end{array} \quad \Gamma \vdash f_1(t_1) = s_1 \quad \Gamma \vdash f_2(t_2) = s_2}{\Gamma \vdash (f_1 | f_2)(t) = \bigvee_{\{i | t_i \neq \mathsf{Empty}\}} s_i} \qquad \textbf{(t-union)}$$

$$\frac{\Gamma \vdash f_1(t) = s_1 \quad \Gamma \vdash f_2(s_1) = s_2 \quad t \leq \lceil f_1 \rfloor \quad s_1 \leq \lceil f_2 \rfloor}{\Gamma \vdash (f_1; f_2)(t) = s_2} \qquad \textbf{(t-comp)}$$

FIG. 3.2 – Système de déduction $\mathscr{F}$ du type d'un filtre

Régularité : ces dérivations parcourrent le filtre donné (qui est régulier). De plus, la condition de bonne formation sur les filtres de la forme $p \to f$ implique que le nombre de variables capturées le long d'une branche infinie est fini. On a donc la garantie que Γ ne grandit pas infiniment le long des dérivations. Notons finalement qu'on ne peut appliquer qu'au plus une règle à chaque étape, la dérivation pour un jugement donné (si elle existe) est donc unique. Les dérivations de typage sont donc régulières.

(t-expr) : `type` est la fonction de typage des expressions. Elle est mutuellement récursive avec la fonction de typage des filtres. Les mêmes remarques que pour l'évaluation du filtre **(e-expr)** s'appliquent ici.

(t-prod) : On note que le type le plus générique pour une combinaison booléenne de types produits est une réunion finie de types produits. Il faut cependant faire très attention car n'importe quelle décomposition finie ne convient pas. Nous choisissons ici la décomposition en produits maximaux. Nous justifions soigneusement ce choix dans la section suivante. Il assure la stabilité des filtres par rapport à la relation de sous-typage qui permet elle même de montrer la complétude de l'alogorithme d'inférence de type. Pour l'instant disons juste qu'il existe une fonction π qui à toute combinaison booléene de produit cartésiens associe la décomposition que nous avons choisie.

(t-patt) : il faut s'assurer que le calcul de t/p est possible. On pourrait croire que la condition $t \leq \lceil p \rfloor$ est suffisante mais ça n'est pas le cas. En effet, les filtres sont récursifs. On peut donc être ammené à utiliser le type de sortie d'un filtre *lors de son propre calcul*. L'exemple suivant illustre celà :

$$f = (x \to x, f); (x, (y, z)) \to ((y, x), z)$$

Si on appelle s le type de sortie de f, alors, on voit que le motif (y, z) essaye de déconstruire le type s qui n'est pas encore connu. Ce filtre, quoi que bien formé n'est donc pas typable. Ce problème de pouvoir effectivement calculer t/p, c'est à dire d'étendre le filtrage par motif aux types, n'est pas abordé ici. Le

formalisme exact est laissé pour des travaux futurs. Notons toutefois que des considérations pratiques montrent que t/p est calculable (la méthode consiste d'une part à «marquer» les types pour s'assurer que l'on ne déstructure pas un type en cours de définition – comme dans l'exemple précédent – et d'autre par à conserver un environnement pour les types et les motifs récursifs).

Typage des produits cartésiens

Le typage des types produits est le point délicat de ce système de type. En effet toute décomposition en réunion finie de produits cartésiens n'est pas stable par rapport à la relation de sous-typage. Illustrons ceci avec un exemple faisant intervenir les types interval d'entiers de $\mathbb{C}$Duce. Le type $\mathsf{0–4}$ est le type des entiers de 0 à 4 (et c'est un sous-type de Int)[3].

Considérons les filtres suivant :

$$f_1 = \mathsf{0–4} \to \texttt{`A} | \mathsf{5–8} \to \texttt{`B}$$
$$f_2 = \mathsf{0–3} \to \texttt{`C} | \mathsf{0–7} \to \texttt{`D}$$
$$f = (f_1, f_2)$$

et les types :

$$t = (\mathsf{0–4} \times \mathsf{0–3}) \vee (\mathsf{5–8} \times \mathsf{0–7})$$
$$s = \mathsf{4–6} \times \mathsf{1–2}$$

On a bien que $s \leq t$ (le «rectangle» s est bien inclu dans la «surface» t mais s est à cheval entre les deux rectangles formant t). Le système de type (si l'on suppose que t est décomposé de cette manière) nous permet de determiner :

$$\varnothing \vdash f(t) = \texttt{`A} \times \texttt{`C} \vee \texttt{`B} \times \texttt{`D}$$

Cependant, on a aussi :

$$\varnothing \vdash f(s) = (\texttt{`A} \vee \texttt{`B}) \times (\texttt{`C} \vee \texttt{`D})$$

On a donc $s \leq t$ mais $f(s) \not\leq f(t)$. Il nous faut donc choisir une décomposition qui permette d'obtenir la stabilité du sous-typage. Nous choisissons pour celà la décomposition en produit maximaux. Cette décomposition (que l'on note π) est telle que :

$$\text{si } t \simeq \bigvee_{(t_1^i, t_2^i) \in \pi(t)} (t_1^i \times t_2^i)$$

alors $\forall s$ tel que $s \leq t$,

$$s \simeq \bigvee_{(s_1^j, s_2^j) \in \pi(s)} (s_1^j \times s_2^j) \text{ et } \forall j, \exists i, \text{ tel que } (s_1^j \times s_2^j), \leq (t_1^i \times t_2^i)$$

Préservation du typage

Montrons maintenant la préservation du typage :

Théorème 2 (Préservation du typage). *Soient f, Γ, t et s tels que :*

$$\Gamma \vdash f(t) = s$$

alors :

$$\forall \gamma : \Gamma \ \textit{et}\ v : t \quad \textit{on a} \quad \gamma \vdash_e f(v) \rightsquigarrow v' \ \textit{et}\ v' : s$$

[3] Ce type est exprimable comme la combinaison booléenne $\mathsf{0} \vee \mathsf{1} \vee \mathsf{2} \vee \mathsf{3} \vee \mathsf{4}$, mais nous préférons garder la syntaxe $\mathbb{C}$Duce par soucis de concision et de clarté.

Preuve (Préservation du typage) :
Supposons :

$$\Gamma \vdash f(t) = s \qquad (\star)$$

Soit $\gamma : \Gamma$. Nous savons par le théorème 1 que l'évaluation d'un filtre termine toujours. Nous pouvons donc raisonner par induction sur la dérivation du jugement $\gamma \vdash_e f(v) \rightsquigarrow v'$ qui est finie :

- Cas de base :

  (e-expr) : par $(\star)$, nous savons que le filtre est bien typé, donc que le type de l'expression e est s. L'évalutaion de l'expression e donne donc bien une valeur v' qui est de type s (car on a la préservation du typage pour les expressions $\mathbb{C}$Duce standard).

  (e-patt-err) ne peut être appliquée dans ce cas là. En effet,si le filtre est de la forme $p \rightarrow f$, alors la dérivation de typage se termine par une règle **(t-patt)**. Les premisses de cette dernière sont vérifiées donc la règle **(e-patt-err)** ne s'applique pas.

  (e-prod-err2) de même que précédement, le filtre $f \equiv (f_1, f_2)$ est supposé bien typé donc v est forcément un couple. Cette règle ne s'applique donc pas ici.

  le résultat de l'évaluation est donc bien une valeur v' de type s.

- Cas inductif :

  (e-prod-ok) : $f \equiv (f_1, f_2)$ et $v \equiv (v_1, v_2)$. Remarquons que, par hypothèse, (v_1, v_2) est de type $t \simeq \bigvee_{(t_1^i, t_2^i,) \in \pi(t)} (t_1^i \times t_2^i)$. Il existe donc i, tel que $v_1 : t_1^i$ et $v_2 : t_2^i$. Par $(\star)$, on obtient $\Gamma \vdash f_1(t_1^i) = s_1^i$, et par hypothèse de récurrence, $\gamma \vdash_e f_1(v_1) \rightsquigarrow v_1'$ avec $v_1' : s_1^i$. Il en va de même pour $\gamma \vdash_e f_2(v_2) \rightsquigarrow v_2'$ avec $v_2' : s_2^i$. Ainsi, $v' \equiv (v_1', v_2') : s_1^i \times s_2^i$. Nous pouvons remarquer finalement que $s_1^i \times s_2^i \leq \bigvee_i (s_1^i \times s_2^i)$, donc $v' \equiv (v_1', v_2') : \bigvee_i (s_1^i \times s_2^i)$

  (e-prod-err1) : l'hypothèse $(\star)$ assure que si $f \equiv (f_1, f_2)$ alors la règle **(e-prod-ok)** est applicable, donc **(e-prod-err1)** ne l'est pas. (leurs conditions étant mutuellement exclusives).

  (e-patt-ok) : ici encore, par $(\star)$, $\Gamma \cup t/p \vdash f(t) = s$, et, par hypothèse d'induction : $\gamma \cup v/p \vdash_e f(v) \rightsquigarrow v'$ et $v' : s$ donc $\gamma \vdash_e (p \rightarrow f)(v) \rightsquigarrow v'$ et $v' : s$

  (e-comp-ok) : par $(\star)$, $\Gamma \vdash f_1(t) = s_1$, donc, par hypothèse d'induction, $\gamma \vdash_e f_1(v) \rightsquigarrow v'$ et $v' : s_1$. D'autre par, $\Gamma \vdash f_2(s_1) = s_2$, donc par hypothèse d'induction, $\gamma \vdash_e f_2(v') \rightsquigarrow v''$ et $v'' : s_2$. Ainsi, $\gamma \vdash_e (f_1; f_2)(v) \rightsquigarrow v''$ et $v'' : s_2$.

  (e-comp-err) : ce cas ne se produit jamais. En effet, si $f \equiv (f_1; f_2)$ alors $(\star)$ assure que l'on peut appliquer la règle **(e-comp-ok)**.

  (e-union1) et **(e-union2)** : ces deux règles ne peuvent pas être dissociées. En effet, nous savons que $v : t_1 \vee t_2$. Mais v étant une valeur, nous pouvons raffiner en disant que soit $v : t_1$, soit $v : t_2 \setminus t_1$. Si $v : t_1$ alors, la règle **(e-union1)** s'applique et $\gamma \vdash_e f_1(v) \rightsquigarrow v'$ avec $v' : s_1$ (et en particulier, $v' \neq \Omega$). Sinon la règle **(e-union2)** s'applique. En effet, si $v : t_2 \setminus t_1$, alors $v \notin \lbag t_1 \rbag$ (par définition), donc $\gamma \vdash_e f_1(v) \rightsquigarrow \Omega$, et d'autre part, $v : t_2$ donc $\gamma \vdash_e f_2(v) \rightsquigarrow v'$ avec $v' : s_2$ (par hypothèse d'induction). Donc soit $v' : s_1$, soit $v' : s_2$. Comme, $s_1 \leq s_1 \vee s_2$ et $s_2 \leq s_1 \vee s_2$, nous avons bien, $v' : s_1 \vee s_2 \equiv s$

□

Stabilité des filtres par rapport à la relation de sous-typage

Lemme 1. *Soient s, t et u des types tels que : $s \leq t$. Alors on a :*
- $s \wedge u \leq t \wedge u$
- $s \vee u \leq t \vee u$

> ***Preuve :***
> Résultat immédiat par l'interprétation ensembliste de $\leq$. □

Lemme 2 (Stabilité du filtrage). *Les filtres sont stables par rapport à la relation de sous-typage $\leq$:*

$$\forall f \in \mathbb{F},\ \forall t, s\ t.q.\ s \leq t,\ \textit{et}\ \forall\ \Gamma\ t.q.\ \Gamma \vdash f(s) = s'\ \textit{et}\ \Gamma \vdash f(t) = t',\ \textit{on a}\ : s' \leq t'$$

> ***Preuve :***
> Soit $s \leq t$, nous voulons montrer que $s' \leq t'$, ou, de manière équivalente $\forall \gamma : \Gamma$ et $v : s,\ \gamma \vdash_e f(v) \rightsquigarrow v',$ on a $v' \in [\![f(t)]\!]$
> Cette présentation peut sembler plus lourde, mais elle sert juste à faire apparaitre la valeur v qui était implicite. Nous pouvons donc réutilliser le triplet $(f, v, c(f))$ équipé de l'ordre $(\sqsubset, \sqsubset, <_{\mathbb{N}})_{lex}$, déjà vu lors de la terminaison de l'évaluation et raisonner par induction :
> - e : on sait que $\texttt{type(}\Gamma\texttt{,}e\texttt{)} = t'$ mais on a aussi que $\texttt{type(}\Gamma\texttt{,}e\texttt{)} = s'$ donc ici, $s' = t'$ d'où $s' \leq t'$.
> - (f_1, f_2) : remarquons qu'on a $s \leq t$, et
>
> $$t \simeq \bigvee_{(t_1^i, t_2^i) \in \pi(t)} (t_1^i \times t_2^i)$$
>
> donc :
>
> $$s \simeq \bigvee_{(s_1^j, s_2^j) \in \pi(s)} (s_1^j \times s_2^j)$$
>
> par définition de notre décomposition, nous savons que pour tout j,il existe i tel que : $s_1^j \times s_2^j \leq t_1^i \times t_2^i$, avec $s_1 \leq t_1^i$ et $s_2 \leq t_2^i$. On peut appliquer l'hypothèse d'induction sur f_1, s_1^j et t_1^i (resp. sur f_2, s_2^j et t_2^i) et on obtient bien que $f_1(s_1^j) \leq f_1(t_1^i)$ (resp. $f_2(s_2^j) \leq f_2(t_2^i)$). Il s'en suit que $f(s) \leq f_1(t)$
> - $f_1|f_2$: par le **lemme 1**, et là règle de typage **(union)** :
>
> $$\begin{array}{rcl} s \wedge \lfloor f_1 \rfloor & \leq & t \wedge \lfloor f_1 \rfloor \\ s \wedge \neg \lfloor f_1 \rfloor & \leq & t \wedge \neg \lfloor f_1 \rfloor \end{array}$$
>
> donc par hypothèse d'induction :
>
> $$\begin{array}{rcl} f_1(s \wedge \lfloor f_1 \rfloor) & \leq & f_1(t \wedge \lfloor f_1 \rfloor) \\ f_2(s \wedge \neg \lfloor f_1 \rfloor) & \leq & f_2(t \wedge \neg \lfloor f_1 \rfloor) \end{array}$$
>
> d'où :
>
> $$f_1(s \wedge \lfloor f_1 \rfloor) \vee f_2(s \wedge \neg \lfloor f_1 \rfloor) \leq f_1(t \wedge \lfloor f_1 \rfloor) \vee f_2(t \wedge \neg \lfloor f_1 \rfloor)$$
>
> - $p \rightarrow f'$: on applique directement l'hypothèse d'induction, d'où $f'(s) \leq f'(t)$, et donc $f(s) \leq f(t)$
> - $f_1; f_2$: Soit $f_1(s) = s_1$ et $f_1(t) = t_1$. Par hypothèse d'induction, $s_1 \leq t_1$. D'où, $f_2(s_1) \leq f_2(t_1)$, donc $f(s) \leq f(t)$.
>
> □

3.5.2 Algorithme de typage

Présentation

Nous présentons maintenant un algorithme pour calculer effectivement le type d'un filtre. Nous faisons quelques modifications d'ordre purement syntaxiques qui allègent la présentation de l'algorithme, les résultats montrés précédement restent toujours valables.

Dans un premier temps, plutôt que de considérer les filtres comme des arbres réguliers possiblement infinis, on choisit d'exprimer explicitement la récursion :

Définition 7 (Filtres avec variable de récursion).

$$\begin{array}{rcll} f & ::= & e & \text{expression} \\ & | & p \to f & \text{pattern, } p \in \mathbb{P} \\ & | & f ; f & \text{séquence} \\ & | & (f, f) & \text{paire} \\ & | & f \mid f & \text{union} \\ & | & X & \text{variable} \\ & | & \mu X.f & \text{récursion} \end{array}$$

Cette représentation est plus proche d'une implémentation machine, mais elle reste équivalente à la précédente[4], si l'on impose les bonnes conditions sur les variables de récursions. Ces restrictions sont :

- un filtre doit être clos.
- une variable de récursion ne peut pas être séparée de son lieur par un ;.
- une variable de récursion est forcément séparée de son lieur par au moins un constructeur (_,_).
- une variable de récursion ne peut pas être séparée de son lieur par un $p \to \ldots$ si $Var(p) \neq \varnothing$

La figure 3.3 présente l'algorithme sous forme d'un ensemble de règles d'inférences $\mathscr{F}_{\mathscr{A}}$, qui ressemble au système $\mathscr{F}$ donné à la section précédente mais avec des règles supplémentaires pour traiter les cas récursifs. Le jugement est maintenant de la forme :

$$\Gamma, \Delta \vdash_{\mathscr{A}} f(t) = s$$

où Γ est l'environnement associant les variable de capture des motifs à leur type et Δ celui associant les variables de récursion des filtres à leur type. Plus précisément, les éléments de Δ sont les triplets $(X : t, \alpha)$ où X est une variable de récursion de filtre, t un type d'entrée et α une variable de type.

Ce système est très proche du système de type $\mathscr{F}$. En fait, l'intêret de ce système est de «cacher» la récursion dans les dérivations régulières, évitant ainsi la gestion de la substitution dans les preuves de préservation du typage et dans celle de stabilité.

Nous montrons maintenant que l'algorithme termine, et qu'il est correct et complet vis à vis du système de type $\mathscr{F}$ présenté précédemment.

Terminaison

Intéressons nous au traitement de la récursion, car c'est le point le plus sensible[5] pour la preuve de terminaison. Remarquons dans un premier temps qu'un filtre récursif peut être appliqué, au cours de son évaluation, à plusieur types d'entrée différents (on se souvient qu'en $\mathbb{C}$Duce, les collections (listes, arbres,...) sont hétérogènes, il est donc normal qu'un itérateur sur ces collections ait cette propriété). Nous sommes donc face à un cas (restreint) de récursion polymorphe. La

[4] Nous n'avons pas choisi ce formalisme comme définition de base des filtres car l'introduction d'un deuxième environnement et la gestion de la substitution en allourdit la présentation.

[5] Comme on pouvait s'y attendre !

$$\frac{\mathtt{type}(\Gamma, \mathtt{e}) = s}{\Gamma, \Delta \vdash_{\mathscr{A}} e(t) = s} \quad \textbf{(a-expr)}$$

$$\frac{\begin{array}{l} t \leq \lceil f_1 \rfloor \times \lceil f_2 \rfloor \\ t \simeq \bigvee_{(t_1^i, t_2^i) \in \pi(t)} (t_1^i \times t_2^i) \end{array} \quad \Gamma, \Delta \vdash_{\mathscr{A}} f_1(t_1^i) = s_1^i \;\; \Gamma, \Delta \vdash_{\mathscr{A}} f_2(t_2^i) = t_2^i}{\Gamma, \Delta \vdash_{\mathscr{A}} (f_1, f_2)(t) = \bigvee_i (s_1^i \times s_2^i)} \quad \textbf{(a-prod)}$$

$$\frac{\Gamma \cup t/p, \Delta \vdash_{\mathscr{A}} f(t) = s \;\; t \leq \lceil p \rfloor \wedge \lceil f \rfloor}{\Gamma, \Delta \vdash_{\mathscr{A}} (p \rightarrow f)(t) = s} \quad \textbf{(a-patt)}$$

$$\frac{\begin{array}{l} t \leq \lceil f_1 \rfloor \vee \lceil f_2 \rfloor \\ t_1 = t \wedge \lceil f_1 \rfloor \\ t_2 = t \wedge \neg \lceil f_1 \rfloor \end{array} \quad \Gamma, \Delta \vdash_{\mathscr{A}} f_1(t_1) = s_1 \;\; \Gamma, \Delta \vdash_{\mathscr{A}} f_2(t_2) = s_2}{\Gamma, \Delta \vdash_{\mathscr{A}} (f_1 | f_2)(t) = \bigvee_{\{i | t_i \neq \mathsf{Empty}\}} s_i} \quad \textbf{(a-union)}$$

$$\frac{\Gamma, \Delta \vdash_{\mathscr{A}} f_1(t) = s_1 \;\; \Gamma, \Delta \vdash_{\mathscr{A}} f_2(s_1) = s_2 \;\; t \leq \lceil f_1 \rfloor \;\; s_1 \leq \lceil f_2 \rfloor}{\Gamma, \Delta \vdash_{\mathscr{A}} (f_1; f_2)(t) = s_2} \quad \textbf{(a-comp)}$$

$$\frac{(X : t, \alpha) \in \Delta}{\Gamma, \Delta \vdash_{\mathscr{A}} (\mu X.f)(t) = \alpha} \quad \textbf{(a-rec1)}$$

$$\frac{(X : t, \alpha) \notin \Delta \;\; \Gamma, \Delta \cup \{(X : t, \alpha)\} \vdash_{\mathscr{A}} (f[X \leftarrow \mu X.f])(t) = s}{\Gamma, \Delta \vdash_{\mathscr{A}} (\mu X.f)(t) = \mu\alpha.s} \quad \textbf{(a-rec2)}$$
(α est une variable fraîche)

FIG. 3.3 – Système algorithmique $\mathscr{F}_{\mathscr{A}}$

règle **(rec-2)** reflète bien ce comportement. La règle **(rec-1)** constitue le cas de base et permet d'arêter le typage d'un filtre récursif si le type auquel on l'applique a déjà été visité. L'argument clé pour prouver la terminaison de l'algorithme apparait donc clairement : il nous faut nous assurer qu'un filtre (récursif) n'est appliqué qu'a un nombre *fini* de types différents, en d'autres termes que la règle **(rec-2)** n'est appliquée qu'un nombre fini de fois.

Définition 8 (Type visité). Soient f un filtre, t et s des types et Γ et Δ des environnements. On dit que le type t' est *visité* lors du typage de $f(t)$ par l'algorithme s'il existe : Γ', Δ', f' et s' tels que : $\Gamma', \Delta' \vdash_{\mathscr{A}} f'(t') = s'$ apparait dans la dérivation du jugement $\Gamma, \Delta \vdash_{\mathscr{A}} f(t) = s$.

Lemme 3. *Le nombre de constructeurs ; dans un filtre est fini.*

> ***Preuve :***
> Les filtres étant des arbres réguliers, ils ont un nombre fini de sous-termes distincts. Soit f un filtre. On sait par définition que pour chaque sous-terme f' de f tel que $f' \equiv f_1; f_2$, alors f' n'est pas un sous-terme de f_2. f_2 est donc un sous-terme distinct de f',donc de f. Chaque ; est donc associé à un sous-terme distinct de f qui sont en nombre fini. □

Lemme 4. *Le nombre de types distincts visités par l'algorithme de typage est fini.*

Preuve :
Remarquons premièrement qu'un type est un arbre régulier. Il a donc un nombre fini de sous-termes distincts. Remarquons aussi que chaque règle de l'algorithme, à l'exception de la règle **(a-comp)** applique ses premisses à des sous-termes du type d'entrée. La règle **(a-comp)** quand à elle applique sa deuxième premisse sur le type *de sortie* de la première, donc sur un type possiblement nouveau. Cependant, nous savons par le lemme 3 que le nombre de constructeurs ; dans un filtre bien formé est *fini*. Cette règle est donc appliquée un nombre fini de fois. L'algorithme visite donc un nombre fini de types distincts. □

Corollaire. *La règle* ***(rec-2)*** *est appliquée un nombre fini de fois.*

Théorème 3 (Terminaison de l'algorithme $\mathscr{F}_{\mathscr{A}}$). *L'algorithme de typage des filtres termine.*

Preuve :
On sait que pour un filtre donné f et un type donné t, la règle **(rec-2)** est appliquée un nombre fini de fois lors du typage. Appelons n ce nombre. Une induction sur le couple (n, f) avec l'ordre $(<_{\mathbb{N}}, \sqsubset)_{lex}$ prouve le théorème. □

Notons que dans notre approche, un même filtre peut être typé plusieurs fois (en nombre fini). Une autre approche est celle proposée dans [12]. H.HOSOYA y décrit les *regular expression filters*[6], comme des expressions régulières permettant d'appliquer des transformations aux parties d'un arbre XML «matchées» par les expressions régulières. Le choix qui y est fait pour le typage est de ne typer les expressions qu'une *seule fois*, en prenant comme type l'union de tous les types possibles en entrée. Il y a donc un typage moins précis que dans notre cas. Si l'on appliquait la même politique aux filtres alors le filtre suivant :

$$\mu X.(\texttt{`Nil} \rightarrow \texttt{`Nil} | (x \rightarrow x, X))$$

appliqué à la séquence [Int Bool String] renverrait le type :

[(Int|Bool|String) (Int|Bool|String) (Int|Bool|String)]

qui est moins précis. C'est cependant une façon simple d'assurer la terminaison de l'inférence de type et sa correction.

Correction du système algorithmique

Le système algorithmique $\mathscr{F}_{\mathscr{A}}$ est correct vis à vis du système de type $\mathscr{F}$:

Théorème 4 (Correction du système algorithmique $\mathscr{F}_{\mathscr{A}}$).

$$\forall\ \Gamma,\ \Delta,\ f,\ t,\ s \text{ tels que } \Gamma, \Delta \vdash_{\mathscr{A}} f(t) = s,\ \text{on a } \Gamma \vdash f(t) = s$$

Preuve :
Nous raisonnons par induction sur la dérivation du jugement $\Gamma, \Delta \vdash_{\mathscr{A}} f(t) = s$ (dérivation qui est finie, par le théorème 3). Par cas sur la dernière règle appliquée lors de la dérivation :

(a-expr) : si $f \equiv e$ et $\Gamma, \Delta \vdash_{\mathscr{A}} e(t) = s$, alors $\texttt{type}(\Gamma, e) = s$ et donc $\Gamma \vdash e(t) = s$

[6] C'est de là que vient le nom de nos combinateurs...

(a-prod) : $\Gamma, \Delta \vdash_{\mathscr{A}} (f_1, f_2)(\bigvee_{i\in\{1,\ldots,n\}} t_1^i \times t_2^i) = s$ où :

$$t \simeq \bigvee_{(t_1^i, t_2^i)\in\pi(t)} (t_1^i \times t_2^i)$$

et :

$$s \simeq \bigvee_i (s_1^i \times s_2^i)$$

Pour tout i, on a par hypothèse d'induction que $\Gamma, \Delta \vdash_{\mathscr{A}} f_1(t_1^i) = s_1^i$, donc $\Gamma \vdash f_1(t_1^i) = s_1^i$, de même pour f_2. Ainsi, par application de la règle **(t-prod)** on obtient finalement : $\Gamma \vdash (f_1, f_2)(t) = s$.

(a-patt) : on a $\Gamma, \Delta \vdash_{\mathscr{A}} (p \to f)(t) = s$. Par hypothèse d'induction, $\Gamma \cup t/p, \Delta \vdash_{\mathscr{A}} f(t) = s$ est correct et $t \leq \lfloor p \rfloor$, donc $\Gamma \cup t/p \vdash f(t) = s$ est dérivable, donc $\Gamma \vdash (p \to f)(t) = s$ (par application de **(t-patt)**).

(a-union) $\Gamma, \Delta \vdash_{\mathscr{A}} (f_1|f_2)(t) = s_1 \vee s_2$. La condition $t \leq \lfloor f_1 \rfloor \vee \lfloor f_2 \rfloor$ est donc vérifiée ainsi que $\Gamma, \Delta \vdash_{\mathscr{A}} f_1(t \wedge \lfloor f_1 \rfloor) = s_1$ et $\Gamma, \Delta \vdash_{\mathscr{A}} f_2(t \wedge \neg \lfloor f_1 \rfloor) = s_2$. Par hypothèse d'induction, on a donc $\Gamma \vdash f_1(t \wedge \lfloor f_1 \rfloor) = s_1$ et $\Gamma \vdash f_2(t \wedge \neg \lfloor f_1 \rfloor) = s_2$, et par la règle **(t-union)**, $\Gamma \vdash (f_1|f_2)(t) = s_1 \vee s_2$

(a-comp) On a : $\Gamma, \Delta \vdash_{\mathscr{A}} (f_1; f_2)(t) = s_2$, donc aussi $\Gamma, \Delta \vdash_{\mathscr{A}} f_1(t) = s_1$ et $\Gamma, \Delta \vdash_{\mathscr{A}} f_2(s_1) = s_2$ avec $t \leq \lfloor f_1 \rfloor$ et $s_1 \leq \lfloor f_2 \rfloor$. Par hypothèse d'induction, ces prémisses sont correctes d'où : $\Gamma \vdash f_1(t) = s_1$ et $\Gamma \vdash f_2(s_1) = s_2$ et ainsi, $\Gamma \vdash (f_1; f_2)(t) = s_2$

(a-rec1) Cette règle ne peut pas être la dernière règle appliquée, car elle nécessite que $(X : t, \alpha) \in \Delta$, il y a donc une application de la règle **(a-rec-2)** plus bas dans la dérivation.

(a-rec2) Rappelons qu'il n'y a pas de différence entre un filtre avec variables de récursion et un filtre vu comme un arbre régulier et de même pour les types. Pour éviter la confusion, on note $\bar{f}$ l'arbre régulier associé au terme $\mu X.f$, et de même, $\bar{s}$ le type vu comme un arbre régulier associé au type $\mu\alpha.s$ On a : $\Gamma, \Delta \vdash_{\mathscr{A}} (\mu X.f)(t) = \mu\alpha.s$, donc $\Gamma, \Delta \cup \{(X : t, \alpha)\} \vdash_{\mathscr{A}} (f[X \leftarrow \mu X.f])(t) = s$. Notons que s contient α qui représente le type de $\mu X.f(t)$, soit $\mu\alpha.s$. On peut donc en déduire par hypothèse d'induction que $\Gamma \vdash \bar{f}(t) = \bar{s}$.

□

Complétude du système algorithmique

Le système algorithmique $\mathscr{F}_{\mathscr{A}}$ est complet vis à vis du système de type $\mathscr{F}$:

Théorème 5 (Complétude du système algorithmique $\mathscr{F}_{\mathscr{A}}$ (non-prouvé)).

$$\forall\ \Gamma,\ f,\ t,\ s \text{ tels que } \Gamma \vdash f(t) = s,\ \text{on a } \Gamma, \varnothing \vdash_{\mathscr{A}} f(t) = s$$

Donner une preuve formelle de ce théorème est un peu ardu à cause des dérivations régulières. Nous les avions introduite pour ne pas avoir à gérer explicitement la récursion et la substitution dans les preuves de préservation du typage et de stabilité. Ce formalisme n'est cependant pas le plus approprié pour la preuve de complétude, que l'on voudrait faire naturellement par induction sur la dérivation du jugement $\Gamma \vdash f(t) = s$. De manière informelle, on voit bien que les règle de $\mathscr{F}$ et $\mathscr{F}_{\mathscr{A}}$ correspondent deux à deux et que les règles **(a-rec1)** et **(a-rec2)** ne sont qu'une représentation des dérivations régulières sous forme de dérivation finie.

Une autre approche est actuellement envisagée quand au formalisme des filtres pour pouvoir méler ces deux aspects de représentation finie et abscence de substitution. On considère le filtre comme un automate. Nous n'utilisons pas ce formalisme dans ce rapport mais il sera le sujet d'un travail futur.

Cependant, il est important d'ajouter que ne pas avoir la complétude de l'algorithme n'est pas génant. En effet, nous savons déjà que l'algorithme est correct vis à vis du système de type, donc si t est le type donné par le système et t_A celui donné par l'algorithme (ils sont tous deux uniques) alors nous avons $t_A \leq t$ (tout type de renvoyé par l'algorithme est un type dérivable par le système de type). La complétude nous assure l'égalité. Si nous n'avions pas la complétude, nous aurions calculé un type strictement plus petit et donc plus précis, ce qui serait même mieux. C'est donc par soucis d'exhaustivité que nous ennonçons le théorème de complétude.

Chapitre 4

Conclusion

4.1 Résumé du travail effectué

Nous avons proposé une algèbre de filtrage pour ℂDuce, étendant celle déjà existante basée sur les motifs. L'approche retenue s'est révélé s'intégrer parfaitement au modèle théorique de ℂDuce. En effet, nous pouvons exprimer de manière unifiée tous les opérateurs primitifs du langages (qui ont un rapport avec le filtrage), tels que le filtrage par motif, les itérateurs de séquences ou d'arbres, et de plus ajoutons la notion de filtrage en profondeur, associée à un typage très précis. Le point fort de notre travail consiste en l'obtention d'un algorithme de typage précis, tout en assurant la terminaison de ce dernier, sa correction et sa complétude vis à vis d'un système de type dont on a montré la sûreté.

Un reproche que l'on peut cependant faire à notre approche est de nécessiter la connaissance du corps du filtre pour pouvoir le typer, celà le rend donc difficilement compatible avec un système de modules avec interfaces et compilation séparée. ℂDuce ne possédant pas pour l'instant tel système de module, celà n'est pas génant, mais l'étude d'avoir un tel système tout en concervant les filtres reste un problème interessant.

4.2 Prolongements

Les bases théoriques des filtres étant posées, nous pouvons maintenant nous consacrer à leur implantation efficace et leur intégration au sein du langage ℂDuce. Le choix des automates d'arbres comme cible de compilation semble judicieux et les travaux déjà réalisé dans le cadre de la compilation efficace du filtrage par motif pourra être réutilisé.

Au niveau théorique, une étude précise du pouvoir expressif de ces combinateurs reste aussi à faire. Les extensions de notre calculs sont multiples, on pense par exemple à de la capture en profondeur avec des contraintes de cardinalité. On peut aussi penser utiliser les filtres comme langage cible pour la compilation de langages de requêtes de plus haut niveau (ℂQL par exemple, est une tentative d'intégrer des requête SQL à ℂDuce). Une troisième direction envisageable est d'étudier les propriétés des filtres dans le but de pouvoir implémenter un traitement en *streaming* (ou à la volée) des documents XML. Les documents XML étant de plus en plus important en terme d'espace de stockage, un chargement partiel des fichiers est un atout non négligeable pour le traîtement efficace des données. Les filtres pourraient être un point de départ prometteur pou cette étude, car ils forment un langage réduit et cependant assez expressif pour exprimer de nombreuses transformations de documents.

Bibliographie

[1] XML Version 1.0 (Third Edition). http ://www.w3.org/TR/2004/REC-xml-20040204/.

[2] http ://www.w3c.org.

[3] XML Path Language (XPath) Version 1.0. http ://www.w3.org/TR/xpath.

[4] XML Query (XQuery). http ://www.w3.org/XML/Query.

[5] XSL Transformations (XSLT) Version 1.0. http ://www.w3.org/TR/xslt.

[6] Haruo Hosoya and Benjamin C. Pierce. XDuce : A typed XML processing language. In *ACM Transactions on Internet Technology*, pages 117–148, 2003.

[7] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce : an XML-centric general-purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.

[8] Alain Frisch and Giuseppe Castagna. A Gentle Introduction to Semantic Subtyping. In *Second workshop on Programmable Structured Documents*, 2004.

[9] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *LICS '02, Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[10] Alain Frisch. Types récursifs, combinaisons booléennes et fonctions surchargées : une application au typage de XML. Mémoire de DEA, septembre 2001.

[11] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 1994.

[12] Haruo Hosoya. Regular expression filters for XML. In *Programming Languages Technologies for XML (PLAN-X)*, pages 13–27, 2004.

[13] Amélie Marian and Jérôme Siméon. Projecting XML Documents. Technical report, Columbia University, February 2003.

[14] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.