# Approximation statique de propriétés d'applications *web*

Mehdi Bouaziz
École normale supérieure, Paris

*Stage d'initiation à la recherche en informatique*
Juin-Août 2008

Avertissement : *la société MLstate souhaite conserver la confidentialité du présent document. Seules les personnes concernées par le stage d'initiation à la recherche en informatique sont autorisées à le consulter, et sont priées de ne pas le diffuser au-delà de ce cadre sans autorisation de la société MLstate.*

## Table des matières

# 1 Introduction

## 1.1 Présentation de la société MLstate

Les SSII, les Web-Agencies, les éditeurs de logiciels qui développent des produits SaaS (Software-as-a-Service) ou des sites Web 2.0 trouvent inadaptés les langages actuels, de Java à PHP. MLstate propose une solution innovante de développement d'applications SaaS à la fois simple, rapide et sûre.

Contrairement à ses concurrents qui ont recours à un empilement de couches technologiques, MLstate a développé, en partant de zéro, une solution d'envergure qui unit un langage, une base de données et un serveur Web, pour supprimer tout travail d'intégration.

Lauréat du Concours du Ministère de la Recherche 2008, Finaliste du Grand Prix d'Innovation de la ville de Paris 2007, MLstate est hébergée depuis sa création en août 2007 au sein de l'incubateur Paris Innovation Bourse — financé par la Mairie de Paris, la Chambre de Commerce et la région Île-de-France. La société emploie aujourd'hui 9 personnes, et une dizaine de postes sont à pourvoir en 2009.

## 1.2 Présentation du sujet du stage

Après des années de recherche sur les machines virtuelles et la distribution d'applications, un standard de fait est apparu : (X)HTML, CSS, JavaScript, dialoguant avec des serveurs selon le protocole HTTP(S). Ces langages ni typés ni compilés contiennent souvent des erreurs, qu'ils soient écrits manuellement ou générés. Pourtant, des méthodes d'analyse statique permettent de vérifier la correction de ces sources.

Les résultats attendus de cette analyse concernent autant la sûreté (vérification de l'existence de propriétés CSS, etc.) que la sécurité (vérification de l'absence de failles XSS, etc.). En théorie, une analyse de la sémantique du programme permet d'atteindre ces résultats. Mais ce calcul est bien sûr intractable et nous en cherchons des approximations.

Le but du stage est d'implémenter une approximation statique de constantes (entiers et surtout chaînes de caractères) pour le langage fonctionnel QML développé par MLstate pour le développement de serveurs d'applications *web*.

### 1.2.1 État de l'art

Le cadre formel de l'analyse statique d'approximations de la sémantique des programmes est dû à Patrick Cousot à la fin des années 1970. On trouvera une présentation récente des nombreux travaux dans [Cousot, 2000]. Le langage QML, faisant partie de la famille ML, est fortement typé. Ces types constituent une première forme d'interprétation abstraite, comme le montre [Cousot, 1997].

La vérification statique des propriétés est un compromis entre leur faisabilité et leur couverture. Si les différentes formes de typage des langages fonctionnels sont rapides, elles sont insuffisantes pour vérifier plus en détail les propriétés des entiers ou des chaînes de caractères.

Concernant les entiers, la détermination d'intervalles est une technique usuelle et couverte par les approches d'interprétation abstraite.

Pour les chaînes de caractères, les approximations ont été abordées dans le cadre de la vérification statique de dépassement de tampon pour le langage C. [Dor et al., 2001] s'intéressent donc à la longueur des chaînes de caractères et leurs travaux sont repris, de manière plus formelle par [Simon et King, 2002]. Une application intéressante est l'annotation de types par les résultats de cette analyse.

L'étude des chaînes au délà de leur longueur nécessite l'introduction de modèles — ou de grammaires spécifiques. [Christensen et al., 2003] introduisent par exemple des automates multi-niveaux pour représenter approximativement des CFG modélisant les chaînes en Java. [Minamide, 2005] applique une partie de ces travaux pour vérifier l'absence de failles sur des serveurs *web* écrits en PHP.

#### 1.2.2 Objectifs

Le but du stage est d'écrire un algorithme d'approximation statique des entiers et des chaînes de caractères pour QML. La classe de grammaire envisagée est celle des Parsing Expression Grammars (PEG), introduite par [Ford, 2004], ce qui est nouveau. Ce choix est motivé par l'adéquation des PEG à la description de chaînes générées par des programmes. En pratique, l'objectif est de vérifier la correction des éléments du Document Object Model, spécifié par [Hors et al., 2000], accédés par le code QML d'une application Web. Le plan prévu est donc le suivant :

- approximation par intervalles des entiers,
- approximation par grammaires PEG des chaînes de caractères,
- utilisation des approximations pour annoter les types constants d'entiers et de chaînes de caractères,
- application à la vérification de propriétés CSS.

# 2 Le langage QML

## 2.1 Présentation générale

QML est un langage fonctionnel dont la syntaxe est très proche de celle de OCaml [INRIA, 2008]. Il intègre une unique partie impérative : l'accès aux données, que l'on appellera base de données par la suite.

## 2.2 Types de données

### 2.2.1 Types de base

Ces types sont les mêmes qu'en OCaml :

- Unité **unit**
- Entiers **int**
- Nombres à virgule flottante **float**
- Caractères **char**
- Chaînes de caractères **string**

Remarquez l'absence du type booléen.

### 2.2.2 Types enregistrement

Comme en OCaml.

### 2.2.3 Types union

Comme les types somme de OCaml, à la différence qu'il n'y a pas de constructeurs. On se limitera aux unions d'enregistrements.

Pour déclarer un type union, la syntaxe est la suivante :

```
type foo = { a : int } / { b : int } / { a : int ; c : int }
```

La construction d'une valeur d'un type union nécessite une cœrcition.
Exemple : ({ a = 42 } : foo)
La déconstruction d'une valeur d'un type union se fait à l'aide d'un **match**. Exemple :

```
match (x : foo) with
| { a = y } → y
| { b = z } → z
```

Notez que si x vaut ({ a = 1 ; c = 2 } : foo) alors le code précédent renvoie 1.

Il n'y a pas de type booléen dans les types de base, on utilise donc le type

```
type bool = { true : unit } / { false : unit }
```

Le code **if** condition **then** $\text{valeur}_1$ **else** $\text{valeur}_2$ est un sucre pour

```
match (condition : bool) with
| { true = () } → valeur1
| _ →  valeur2
```

### 2.2.4 Types lambda

Les *types lambda* (ou «types flèche») sont les types des fonctions (ou clotûres), c'est-à-dire les types $\mathsf{T}_1 \rightarrow \mathsf{T}_2$, avec $\mathsf{T}_1$ et $\mathsf{T}_2$ deux types quelconques.

## 2.3 Algèbre du langage

Un code QML est composé :
– de définitions de types

**type** *typeident* = *typedef*

avec

| | | | |
|---|---|---|---|
| *typedef* | ::= | \| *typeconst* | (type constant) |
| | | \| *typeident* | (type nommé) |
| | | \| { *fieldident* : *typedef* [ ; *fieldident* : *typedef* [ ... ] ] } | (type enregistrement) |
| | | \| *typedef* / *typedef* [ / *typedef* [ ... ] ] | (type union) |
| *typeconst* | ::= | \| **unit** | |
| | | \| **int** | |
| | | \| **float** | |
| | | \| **char** | |
| | | \| **string** | |

– de définitions de bases de données

**val** *dbident* : typedecl

– de valeurs

**val** *ident* = *expr*

avec

| | | | |
|---|---|---|---|
| *expr* | ::= | \| *const* | (valeur constante) |
| | | \| *validident* | (identifiant de valeur) |
| | | \| ( *expr* : *typeident* ) | (cœrcition) |
| | | \| **fun** *ident* →*expr* | (lambda) |
| | | \| *expr expr* | (application) |
| | | \| **let** *validident* = *expr* **in** *expr* | |
| | | \| **let rec** *ident* = *expr* [ **and** *ident* = *expr* [ ... ] ] **in** *expr* | |
| | | \| { *fieldident* = *expr* } / *expr* | (extension d'enregistrement) |
| | | \| *expr*. *fieldident* | (champ d'un enregistrement) |
| | | \| **match** *expr* **with** [ \| *pattern* →*expr* [ ... ] ] | (filtrage) |
| | | \| %%*fonction*%% | (fonction de la bibliothèque) |
| | | \| **assert** *expr* **in** *expr* | (assertion) |

| | | | |
|---|---|---|---|
| *const* | ::= | \| l'unité | (type **unit**) |
| | | \| un entier | (type **int**) |
| | | \| un nombre à virgule flottante | (type **float**) |
| | | \| un caractère | (type **char**) |
| | | \| une chaîne | (type **string**) |
| | | \| l'enregistrement vide | (type enregistrement quelconque) |

| | | | |
|---|---|---|---|
| *validident* | ::= | \| *ident* | (variable) |
| | | \| [\| *path* \|] | (variable de base de données) |

| | | |
|---|---|---|
| *ident* | ::= | un identifiant |

| | | |
|---|---|---|
| *typeident* | ::= | un identifiant de type |

| | | | |
|---|---|---|---|
| *pattern* | ::= | \| *const* | (valeur constante) |
| | | \| *ident* | (liaison) |
| | | \| _ | (toute valeur) |
| | | \| { *ident* = *pattern* } / *pattern* | (champ d'enregistrement) |

| | | |
|---|---|---|
| *path* | ::= | /*dbident*/$\mathbf{val}_0$/$\mathbf{val}_1$/.../$\mathbf{val}_n$ |

| où $\mathbf{val}_i$ | dépend du | type de /*dbident*/$\mathbf{val}_0$/.../$\mathbf{val}_{i-1}$ |
|---|---|---|
| une expression de type chaîne | dans le cas d'une | stringmap |
| une expression de type entier | dans le cas d'une | intmap |
| le nom d'un champ | dans le cas d'un | type enregistrement |

Les opérateurs (comme +, −, ∗, /, **not**, &&, ||, ^, ...) ne sont que du sucre pour l'appel d'une fonction de la bibliothèque.

On supposera que le lecteur est familier avec OCaml. C'est pourquoi on se passera de décrire la sémantique concrète du langage.

# 3 Interprétation abstraite

## 3.1 Prérequis

On supposera que le lecteur est familier avec les domaines de la sémantique et de l'interprétation abstraite, notamment des notions de treillis, de méthodes d'approximation de points fixes, d'opérateur d'élargissement et de rétrécissement.

Au besoin, on pourra se référer à [Cousot et Cousot, 1977] et [Cousot, 2001], ainsi qu'aux cours [Mauborgne, 2008] à l'X ou [Cousot et al., 2008] au MPRI.

## 3.2 Modèle utilisé

Notre analyse se fera sur un domaine abstrait non relationnel, c'est-à-dire que les propriétés calculées ne feront pas intervenir plusieurs variables à la fois.

Les propriétés concrètes calculées seront par exemple de la forme

*à tout moment, la valeur de la variable* x *est dans l'ensemble* $X$

à la différence d'un domaine relationnel qui peut calculer certaines propriétés de la forme

*à tout moment, la valeur de la variable* x *est inférieure à la valeur de la variable* y

ou encore

*à tout moment, la valeur de* x $-$ y *est dans l'ensemble* $Z$

À chaque variable est associée une valeur abstraite représentant des propriétés des valeurs concrètes que peut prendre la variable au cours de l'exécution du programme. Généralement on choisira comme valeur abstraite une *sur-approximation* de l'ensemble des valeurs concrètes, c'est-à-dire un ensemble auquel appartient toute valeur concrète prise par la variable durant l'exécution du programme. Idéalement la valeur abstraite sera l'ensemble des valeurs concrètes, mais cela est rarement possible, et une telle précision n'est parfois pas nécessaire.

Nous définirons donc pour chaque type de donnée un domaine abstrait permettant de représenter les valeurs abstraites de ce type.

Autant que possible, nous noterons par un dièse ($\sharp$) les valeurs, propriétés et opérations abstraites.

# 4 Domaines abstraits

## 4.1 Minimum requis

Pour un certain type de données $T$, le domaine abstrait $D_T^\sharp$ est l'ensemble des valeurs abstraites $X^\sharp$ de ce type. Si $D_T$ est l'ensemble des valeurs concrètes alors les valeurs abstraites représenteront des parties de $D_T$, le domaine abstrait sera donc une approximation de $\mathcal{P}(D_T)$, auquel on ajoute un élément (valeur indéfinie) représentant l'absence de connaissance sur la valeur de la variable. Pour calculer les valeurs abstraites par approximation de point fixe, décrite précédemment, on demande au domaine abstrait d'être un treillis complet qui contiendra au minimum :

- la valeur indéfinie, infimum du treillis $\bot^\sharp$
- l'ensemble vide $\varnothing^\sharp$
- l'ensemble de toutes les valeurs, c'est-à-dire $\mathcal{P}(D_T)$, supremum du treillis $\top^\sharp$

On veut aussi pouvoir réaliser un certain nombre d'opérations sur les éléments du domaine abstrait. Dans certains cas, on se contentera d'une approximation par valeur supérieur car le résultat d'une opération peut ne pas être dans le domaine ou le calcul peut être beaucoup plus complexe ou coûteux en temps ou en mémoire que le calcul de l'approximation.

Pour chaque domaine abstrait, ces opérations seront au minimum :

– l'union, telle que

$$X_1^\sharp \cup^\sharp X_2^\sharp \quad = \quad \begin{array}{ll} X_1^\sharp & \text{si } X_2^\sharp = \bot^\sharp \text{ ou } X_2^\sharp = \varnothing^\sharp \\ X_2^\sharp & \text{si } X_1^\sharp = \bot^\sharp \text{ ou } X_1^\sharp = \varnothing^\sharp \\ \top & \text{si } X_1^\sharp = \top^\sharp \text{ ou } X_2^\sharp = \top^\sharp \\ \langle \textit{valeur selon le type} \rangle & \text{sinon} \end{array}$$

– l'intersection, telle que

$$X_1^\sharp \cap^\sharp X_2^\sharp \quad = \quad \begin{array}{ll} \varnothing^\sharp & \text{si } X_1^\sharp = \varnothing^\sharp \text{ ou } X_2^\sharp = \varnothing^\sharp \\ \bot^\sharp & \text{si } X_1^\sharp = \bot^\sharp \text{ ou } X_2^\sharp = \bot^\sharp \\ X_1^\sharp & \text{si } X_2^\sharp = \top^\sharp \\ X_2^\sharp & \text{si } X_1^\sharp = \top^\sharp \\ \langle \textit{valeur selon le type} \rangle & \text{sinon} \end{array}$$

– la *différence unitaire*, que l'on notera $\setminus!^\sharp$, c'est-à-dire

$$X_1^\sharp \setminus!^\sharp X_2^\sharp \quad = \quad \begin{array}{ll} X_1^\sharp \setminus^\sharp X_2^\sharp & \text{si } \mathrm{card}^\sharp(X_2^\sharp) = 1 \\ X_1^\sharp & \text{sinon} \end{array}$$

où $X_1^\sharp \setminus^\sharp X_2^\sharp$ est une sur-approximation de $X_1^\sharp \setminus X_2^\sharp$, et $\mathrm{card}^\sharp(X^\sharp) = 1$ est une sous-approximation du test $\mathrm{card}(X^\sharp) = 1$, avec $\mathrm{card}(\bot^\sharp) = 1$ étant faux ; c'est-à-dire $\mathrm{card}^\sharp(X^\sharp) = 1 \Rightarrow \mathrm{card}(X^\sharp) = 1$ — notez qu'il est inutile de pouvoir calculer le cardinal de $X^\sharp$ dans le cas général.

– le *conditionnement*, que l'on notera $\gg^\sharp$

$$B^\sharp \gg^\sharp X^\sharp \quad = \quad \begin{array}{ll} X^\sharp & \text{si } B^\sharp \text{ représente une valeur qui peut être vraie} \\ \varnothing^\sharp & \textit{sinon} \end{array}$$

– le test d'égalité, que l'on notera $=^\sharp$

$$X_1^\sharp =^\sharp X_2^\sharp \quad = \quad \begin{array}{ll} \varnothing^\sharp & \text{si } X_1^\sharp = \varnothing^\sharp \text{ ou } X_2^\sharp = \varnothing^\sharp \\ \bot^\sharp & \text{si } X_1^\sharp = \bot^\sharp \text{ ou } X_2^\sharp = \bot^\sharp \\ \{vrai\} & \text{si } X_1^\sharp = X_2^\sharp \text{ et } \mathrm{card}^\sharp(X_1^\sharp) = 1 \\ \{faux\} & \text{si } X_1^\sharp \cap X_2^\sharp = \varnothing^\sharp \\ \{vrai, faux\} & \text{sinon} \end{array}$$

Ces opérations doivent être croissantes. La différence unitaire et le conditionnement ne le sont pas, mais la décroissance est bornée par le nombre de variables.

Lorsque cela est nécessaire, on demandera également un opérateur d'élargissement assurant la terminaison des itérations du calcul du point-fixe.

## 4.2 Booléens

### 4.2.1 Domaine abstrait

Comme le domaine concret $D_B = \{T, F\}$ est un ensemble fini et petit, on peut garder toutes les valeurs en prenant $D_B^\sharp = \{\bot^\sharp\} \cup \mathcal{P}(\mathbb{B})$, c'est-à-dire $\bot^\sharp$, $\varnothing^\sharp$, $\{T\}$, $\{F\}$, $\top^\sharp = \{T, F\}$

#### 4.2.2 Opérateurs

En plus de l'union, l'intersection, la différence unitaire, le conditionnement et le test d'égalité, on sera amené à utiliser les opérateurs binaires non ($\neg^\sharp$), et ($\wedge^\sharp$), et ou ($\vee^\sharp$), que l'on ne détaillera pas.

L'ensemble $D_B^\sharp$ est fini, cela assure la terminaison des itérations.

### 4.3 Entiers

#### 4.3.1 Domaine abstrait – intervalle

Ce domaine abstrait est décrit dans [Cousot et Cousot, 1977] avec ses opérateurs d'élargissement et de retrécissement.

Il permet d'approximer un ensemble d'entiers par un intervalle dans $\bar{\mathbb{Z}}$, idéalement le plus petit intervalle qui contient cet ensemble.

On notera ce domaine $D_{I_1}^\sharp = \{\bot^\sharp, \varnothing^\sharp\} \cup \{[a,b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\} \text{ et } a \leq b\}$

#### 4.3.2 Domaine abstrait – ensemble d'intervalles

Le domaine précédent se révèle inadapté dès que l'on a des valeurs isolées. Par exemple $\{0, 42, 10000\}$ serait grossièrement approximé par $[0, 10000]$, ce qui peut être insuffisamment précis dans certains cas.

De plus, comme nous le verrons dans la suite, le domaine abstrait des chaînes de caractères est plus précis. Nous avons donc décidé d'utiliser un autre domaine abstrait pour les entiers, en approximant un ensemble d'entiers par un ensemble d'intervalles disjoints de $\bar{\mathbb{Z}}$. Cependant, comme la taille d'un tel ensemble peut ne pas être borné, nous utiliserons ce que nous avons appelé un $_{\leq N}$-multi-intervalle, c'est-à-dire un ensemble d'au plus $N$ intervalles disjoints de $\bar{\mathbb{Z}}$, avec $N \geq 1$.

Pour $k \geq 0$, on dit qu'un ensemble $E \subseteq \bar{\mathbb{Z}}$ est un $_k$*-multi-intervalle* s'il existe $2k$ entiers $(a_i)_{0 \leq i \leq k-1}$ et $(b_i)_{0 \leq i \leq k-1}$ tels que

1. $a_0 \in \mathbb{Z} \cup \{-\infty\}$
2. $b_{k-1} \in \mathbb{Z} \cup \{+\infty\}$
3. $\forall i \in [\![0, k-2]\!], a_{i+1}, b_i \in \mathbb{Z}$ et $b_i < a_{i+1} + 1$
4. $\forall i \in [\![0, k-1]\!], a_i \leq b_i$
5. $E = \bigcup_{0 \leq i \leq k-1} [a_i, b_i]$

On notera $M_k$ l'ensemble des $_k$-multi-intervalles.

Pour $N \geq 1$, on dit qu'un ensemble $E \subseteq \bar{\mathbb{Z}}$ est un $_{\leq N}$-multi-intervalle si il existe un entier $k \leq N$ tel que $E$ soit un $_k$-multi-intervalle. On dit alors que $E$ est un $_{k \leq N}$-multi-intervalle. On notera $M_{\leq N}$ l'ensemble des $_{\leq N}$-multi-intervalles.

On fixera donc $N \geq 1$ et on choisira $D_I^\sharp = D_{I_N}^\sharp = \{\bot^\sharp\} \cup M_{\leq N}$.

### 4.3.3 Opérateurs

Comme certaines opérations naturelles sur des $_{\leq N}$-multi-intervalles peuvent ne pas donner des $_{\leq N}$-multi-intervalles, nous les approximerons avec un *opérateur de restriction* :

$$\begin{array}{rccc} \downarrow_{\leq N}^{\leq N'}: & M_{\leq N'} & \longrightarrow & M_{\leq N} \\ & E & \longmapsto & E' \supseteq E \end{array}$$

Dans un premier temps, on pourra prendre

$$\begin{array}{rccll} \downarrow_{\leq N}^{\leq N'}: & M_{\leq N'} & \longrightarrow & M_{\leq N} & \\ & E & \longmapsto & E & \text{si } E \in M_{\leq N} \\ & & & \downarrow_{N}^{N+1} \circ \ldots \circ \downarrow_{k-1}^{k} E & \text{si } E \in M_k \text{ avec } k > N \end{array}$$

avec pour $k \geq 1$

$$\begin{array}{rccc} \downarrow_{k}^{k+1}: & M_{k+1} & \longrightarrow & M_k \\ & E & \longmapsto & E' \supseteq E \end{array}$$

par exemple, l'opérateur qui fusionne les deux intervalles les plus proches.

Si on ne se préoccupe pas de la valeur $\bot^\sharp$ qui donne $\bot^\sharp$ dès qu'elle est un opérande, pour $X_1^\sharp, X_2^\sharp \in M_{\leq N}$, les opérateurs sont les suivants :

- l'union : $X_1^\sharp \cup^\sharp X_2^\sharp = \downarrow_{\leq N}^{\leq 2N} (X_1^\sharp \cup X_2^\sharp)$
- l'intersection : $X_1^\sharp \cap^\sharp X_2^\sharp = \downarrow_{\leq N}^{\leq 2N-1} (X_1^\sharp \cap X_2^\sharp)$
- la soustraction unitaire : $X_1^\sharp \setminus^\sharp X_2^\sharp = \downarrow_{\leq N}^{\leq N+1} (X_1^\sharp \setminus\!! X_2^\sharp)$
- les opérations arithmétiques : pour $\odot$ représentant $+$, $-$, $\times$ ou $\div$
  si $X_1^\sharp = \bigcup_{0 \leq i \leq k_1 - 1} [a_i, b_i]$ est un $_{k_1 \leq N}$-multi-intervalle
  et $X_2^\sharp = \bigcup_{0 \leq j \leq k_2 - 1} [c_j, d_j]$ est un $_{k_2 \leq N}$-multi-intervalle

$$X_1^\sharp \odot^\sharp X_2^\sharp \quad = \quad \downarrow_{\leq N}^{\leq \infty} \bigcup_{\substack{0 \leq i \leq k_1 - 1 \\ 0 \leq j \leq k_2 - 1}} [a_i, b_i] \odot^{[]} [c_j, d_j]$$

avec

$$[a_i, b_i] +^{[]} [c_j, d_j] \quad = \quad [a_i + c_j, b_i + d_j]$$

$$[a_i, b_i] -^{[]} [c_j, d_j] \quad = \quad [a_i - c_j, b_i - d_j]$$

$$\begin{array}{rcll} [a_i, b_i] \times^{[]} [c_j, d_j] & = & \bigcup_{\substack{a_i \leq x \leq b_i \\ c_j \leq y \leq d_j}} \{x \times y\} & \text{si } (b_i - a_i)(d_j - c_j) \leq N \\ & & [\inf P, \sup P], P = \{a_i c_j, a_i d_j, b_i c_j, b_i d_j\} & \text{sinon} \end{array}$$

$$[a_i, b_i] \div^{[]} [c_j, d_j] \quad = \quad \bigcup_{\substack{c_j \leq y \leq d_j \\ y \leq -1}} [b_i \div y, a_i \div y] \quad \bigcup \quad \bigcup_{\substack{c_j \leq y \leq d_j \\ y \geq 1}} [a_i \div y, b_i \div y]$$

- la valeur absolue : $|X^\sharp|^\sharp = \downarrow_{\leq N}^{\leq N+1} ((X^\sharp)^- \cup (X^\sharp)^+)$
- le test d'infériorité :

$$\begin{array}{rcll} X_1^\sharp \leq^\sharp X_2^\sharp & = & \varnothing^\sharp & \text{si } X_1^\sharp = \varnothing^\sharp \text{ ou } X_2^\sharp = \varnothing^\sharp \\ & & \{T\} & \text{si } \sup X_1^\sharp \leq \inf X_2^\sharp \\ & & \{F\} & \text{si } \inf X_1^\sharp > \sup X_2^\sharp \\ & & \{T, F\} & \text{sinon} \end{array}$$

– la concaténation d'entiers considérés comme chaînes de caractères :

$$X_1^\sharp \mathbin{\hat{}^\sharp} X_2^\sharp = \downarrow_{\le N}^{\le N^2} \bigcup_{\substack{0\le i<k_1\\ 0\le j<k_2}} [a_i,b_i] \mathbin{\hat{}^\sharp} [c_j,d_j]$$

$$[a_i,b_i] \mathbin{\hat{}^\sharp} [c_j,d_j] = [(a_i \mathbin{\hat{}} c_j) \wedge (a_i \mathbin{\hat{}} d_j), (b_i \mathbin{\hat{}} c_j) \vee (b_i \mathbin{\hat{}} d_j)]$$

$$x \mathbin{\hat{}} y = \begin{cases} \mathit{erreur} & \text{si } y < 0 \\ -\infty & \text{si } x = -\infty \text{ ou } x < 0 \text{ et } y = +\infty \\ +\infty & \text{si } x = +\infty \text{ ou } x \ge 0 \text{ et } y = +\infty \\ \mathsf{int\_of\_string}\ (\mathsf{(string\_of\_int\ x)} \mathbin{\hat{}} \mathsf{(string\_of\_int\ y)}) & \text{sinon} \end{cases}$$

Précisons enfin les opérateurs d'élargissement — nécessaire à un calcul en temps fini et «petit» — et de rétrécissement :

$$X_1^\sharp \mathbin{\triangledown^\sharp} X_2^\sharp = \downarrow_{\le N}^{\le 2N+2} (X_1^\sharp \overset{\Leftarrow^\sharp}{\triangledown} X_2^\sharp \ \cup\ X_1^\sharp \overset{\Leftrightarrow^\sharp}{\triangledown} X_2^\sharp \ \cup\ X_1^\sharp \overset{\Rightarrow^\sharp}{\triangledown} X_2^\sharp)$$

$$X_1^\sharp \overset{\Leftarrow^\sharp}{\triangledown} X_2^\sharp = [(c_0 < a_0)\ ?\ -\infty : a_0, b_0 \wedge d_0]$$

$$X_1^\sharp \overset{\Leftrightarrow^\sharp}{\triangledown} X_2^\sharp = \bigcup_{0\le i<k} (\downarrow_1^{\le N} (X_1^\sharp \cap [e_i,f_i])) \overset{\Leftrightarrow^\sharp}{\triangledown_{[]}} (\downarrow_1^{\le N} (X_2^\sharp \cap [e_i,f_i]))$$

$$\text{où } X_1^\sharp \cup X_2^\sharp = \bigcup_{0\le i<k} [e_i,f_i] \in M_k$$

$$[a',b'] \overset{\Leftrightarrow^\sharp}{\triangledown_{[]}} [c',d'] = [(c' < a')\ ?\ c' - (d' - c' + 1) : a', (d' > b')\ ?\ d' + (d' - c' + 1) : b']$$

$$X_1^\sharp \overset{\Rightarrow^\sharp}{\triangledown} X_2^\sharp = [a_{k_1-1} \vee c_{k_2-1}, (d_{k_2-1} > c_{k_1-1})\ ?\ +\infty : c_{k_1-1}]$$

$$X_1^\sharp \mathbin{\triangle^\sharp} X_2^\sharp = \downarrow_{\le N}^{\le 2N+2} ([a_0,b_0] \overset{\Leftrightarrow^\sharp}{\triangle} [c_0,d_0] \ \cup\ [a_{k_1-1},b_{k_1-1}] \overset{\Leftrightarrow^\sharp}{\triangle} [c_{k_2-1},d_{k_2-1}] \ \cup \bigcup_{1\le i\le k_1-2} [a_i,b_i] \ \cup \bigcup_{1\le j\le k_2-2} [c_j,d_j])$$

$$[a',b'] \overset{\Leftrightarrow^\sharp}{\triangle} [c',d'] = [(a' = -\infty)\ ?\ c' : a' \wedge c', (b' = +\infty)\ ?\ d' : b' \vee d']$$

L'opérateur d'élargissement agit comme l'opérateur d'élargissement sur les intervalles sur les bords, et sur l'intérieur converge vers un point fixe en temps logarithmique en pire cas.

Notons que si $N = 1$ on retrouve le domaine abstrait des intervalles.

## 4.4 Nombres à virgule flottante

### 4.4.1 Domaine abstrait

Ce type n'a pas été traité avec détail durant le stage. Un simple système de propagation des constantes a été implémenté pour ce type.

Le domaine abstrait utilisé est $D_F^\sharp = \{\bot^\sharp, \top^\sharp\} \cup \{x_i \mid 0 \le i < n\}$ où les $x_i$ sont des **float**.

### 4.4.2 Opérateurs

Les opérations d'union, d'intersection et de différence unitaire sont exactes. On ajoute les opérations $\mathsf{int\_of\_float}^\sharp$ et $\mathsf{float\_of\_int}^\sharp$ :

$$
\begin{array}{lrcll}
\mathsf{int\_of\_float}^\sharp & : & D_F^\sharp & \longrightarrow & D_I^\sharp \\
 & & F^\sharp & \longmapsto & \bot^\sharp & \text{si } F^\sharp = \bot^\sharp \\
 & & & & \downarrow_{\leq N}^{\leq \infty} \cup_{0 \leq i < n} \{\mathsf{int\_of\_float}\ x_i\} & \text{si } F^\sharp = \cup_{0 \leq i < n} \{x_i\} \\
 & & & & [-\infty, +\infty] & \text{si } F^\sharp = \top^\sharp \\
 & & & & & \\
\mathsf{float\_of\_int}^\sharp & : & D_I^\sharp & \longrightarrow & D_F^\sharp \\
 & & I^\sharp & \longmapsto & \bot^\sharp & \text{si } I^\sharp = \bot^\sharp \\
 & & & & \{\mathsf{float\_of\_int}\ x\} & \text{si } I^\sharp = \{x\} \\
 & & & & \top^\sharp & \text{sinon}
\end{array}
$$

Ainsi que les opérateurs arithmétiques. Les autres opérations ($\mathsf{sqrt}$, $\mathsf{sin}$, ...) sur les nombres à virgule flottante pourront être approximées plus ou moins précisément avec un meilleur domaine abstrait. Pour $\odot$ représentant $+$, $-$, $\times$ ou $\div$ :

$$
\begin{array}{rcll}
F_1^\sharp \odot^\sharp F_2^\sharp & = & \bot^\sharp & \text{si } F_1^\sharp = \bot^\sharp \text{ ou } F_2^\sharp = \bot^\sharp \\
 & & \varnothing^\sharp & \text{si } F_1^\sharp = \varnothing^\sharp \text{ ou } F_2^\sharp = \varnothing^\sharp \\
 & & \{x_1 \odot x_2\} & \text{si } F_1^\sharp = \{x_1\} \text{ et } F_2^\sharp = \{x_2\} \\
 & & \top^\sharp & \text{sinon}
\end{array}
$$

On assure ainsi que, pour un programme $P$ donné, l'ensemble $F_P$ des **float** utilisés dans l'analyse est fini. Pour ce programme, on peut donc remplacer $D_F^\sharp$ par un ensemble $D_{F_P}^\sharp$ dans lequel les **float** utilisables sont ceux de $F_P$. $D_{F_P}^\sharp$ est donc un ensemble fini, ce qui suffit à assurer la terminaison des itérations.

## 4.5 Caractères

On utilisera le même domaine abstrait que pour les chaînes de caractères, ci-dessous.

## 4.6 Chaînes de caractères

### 4.6.1 État de l'art

L'analyse statique sur les chaînes de caractères est un domaine de recherche relativement récent. Les premiers travaux se sont intéressés à l'approximation de la longueur des chaînes. Dans des programmes C, cela consiste à connaître la position des 0 dans un tableau de caractères (donc des entiers). D'autres travaux ont cherché à vérifier la validité des pointeurs de chaînes de caractères ou au dépassement de tampon dans des programmes C.

Plus récemment, la question du contenu des chaînes est devenu un sujet d'intérêt, notamment avec l'avènement du *web*. En effet, que ce soit pour des requêtes SQL ou du contenu *web* généré dynamiquement, le contenu des chaînes s'avère être un point déterminant dans la sûreté et la sécurité des nouvelles applications. Preuve en est le nombre considérable de sites *web* touchés par des failles de type Injection SQL [Anley, 2002, Spett, 2003] ou Cross Site Script (XSS) [Klein, 2002].

En 2003, Christensen, Møller et Schwartzbach [Christensen et al., 2003] ont développé un analyseur de chaînes de caractères pour programmes Java (Java string snalyzer, JSA). Une grammaire hors-contexte (CFG) est extraite du programme, dans laquelle les symboles non-terminaux correspondent aux expressions de type chaîne de caractères du programme. Une variante de l'algorithme de Mohri-Nederhof [Mohri et Nederhof, 2001] est ensuite utilisé pour approximer la CFG par des expressions rationnelles.

En 2005, Minamide [Minamide, 2005] a développé un analyseur basé sur JSA pour vérifier la validité d'un document HTML généré par un programme PHP. La sortie du programme est approximée par une CFG, puis par des expressions rationnelles de profondeur bornée.

En 2006, Choi, Lee, Kim et Doh [Choi et al., 2006] ont choisi de ne pas augmenter la précision de l'approximation des chaînes de caractères et ont développé un opérateur d'élargissement permettant l'utilisation des techniques usuelles d'interprétation abstraite. Leur analyse, appliquée à la validation de requêtes SQL dans des programmes Java, utilise une sous-classe des expressions rationnelles, les *chaînes rationnelles*.

En 2007, Minamide [Minamide, 2007] a amélioré son analyseur en développant des tests d'inclusions (exponentiels) entre une CFG et un langage rationnel ou une grammaire spécifique comme celle des documents XML [Minamide et Tozawa, 2006].

### 4.6.2 Choix du domaine abstrait

La classe de grammaire initialement envisagée par le sujet de stage était celle des PEG [Ford, 2004]. Cependant la classe des PEG contient la classe des CFG, et les spécificités des PEG par rapport aux CFG ne se retrouvent pas dans les grammaires extraites de programmes.

De plus les opérations sur les CFG peuvent être plus complexes ou plus coûteuses que celles sur des expressions rationnelles, et certaines se révèlent même être indécidables dans le cas général (intersection, inclusion, test de vacuité, ...).

Enfin les propriétés à montrer dans le cadre de notre analyseur (propriétés CSS, attributs HTML) peuvent être exprimées avec des expressions rationnelles (et même des chaînes rationnelles) et ne nécessitent pas une classe de langage plus élaborée (ce qui serait le cas par exemple pour une structure HTML ou XML).

Nous avons donc choisi d'utiliser la classe des chaînes rationnelles [Choi et al., 2006] dont nous rappelons ci-dessous la définition et les principales propriétés.

### 4.6.3 Chaînes rationnelles

Une chaîne rationnelle est une séquence d'atomes qui sont soit un caractère abstrait, soit une répétition d'une chaîne rationnelle, avec la restriction que deux répétitions ne peuvent pas se succéder.

| | | |
|---|---|---|
| **Char** | $C$ | $\in \mathcal{P}(\{\mathsf{c} \text{ de type } \textsf{char}\}) \setminus \{\varnothing\}$ |
| **Atom** | $a$ | $::= C \mid r^*$ |
| **Reg** | $p, q, r$ | $\in \{a_1 a_2 \cdots a_n \mid n \geq 0, a_i \in \textbf{Atom}, \neg\exists i\ (a_i = p^* \wedge a_{i+1} = q^*)\}$ |
| chaîne abstraite | $P^\sharp, Q^\sharp$ | $\in \mathcal{P}(\textbf{Reg})$ |

On notera $\mathcal{L}(p)$ le langage défini par la chaîne rationnelle $p$ et $\mathcal{L}(P^\sharp) = \bigcup_{p \in P^\sharp} \mathcal{L}(p)$.

Remarquez que l'on peut utiliser une structure de $\leq_\infty$-multi-intervalle pour représenter **Char**.

### 4.6.4 Concaténation et élargissement

La sémantique abstraite de la concaténation (notée ^, comme en QML) est ainsi définie : deux chaînes régulières sont simplement concaténées, sauf dans le cas où le dernier atome de la

première et le premier atome de la dernière sont des répétitions. Si les deux répétitions sont les même, une des deux est supprimée ; sinon elles sont brutalement fusionnées.

$$P^\sharp \mathbin{\hat{}^\sharp} Q^\sharp = \{p \mathbin{\hat{}} q \mid p \in P^\sharp, q \in Q^\sharp\}$$

$$p \mathbin{\hat{}} q = \begin{cases} p'r^*q' & \text{si } p = p'r^*, q = r^*q' \\ p'\{c \mid c \text{ apparaît dans } r \text{ ou } r'\} & \text{si } p = p'r^*, q = r'^*q', r \neq r' \\ pq & \text{sinon} \end{cases}$$

L'opérateur d'élargissement de deux chaînes régulière est conçu de manière à garder un maximum de précision tout en permettant une preuve de terminaison. Deux ensembles de chaînes rationnelles peuvent être élargis simplement en élargissant chaque paire de chaînes de ces deux ensembles. Par exemple, supposons qu'après une itération $\{a, b\}$ devienne $\{aa, ba\}$. La solution la plus raisonnable serait $\{aa^*, ba^*\}$, donc nous voulons choisir $\{a \bigtriangledown aa, b \bigtriangledown bb\}$ plutôt que $\{a \bigtriangledown aa, a \bigtriangledown ba, b \bigtriangledown aa, b \bigtriangledown ba\}$. Ainsi, nous définissons $P^\sharp \bigtriangledown^\sharp Q^\sharp = \{p \bigtriangledown q \mid p \in P^\sharp, q \in Q^\sharp, p \mathcal{R} q\}$ pour une certaine relation totale $\mathcal{R}$ de $P^\sharp \times Q^\sharp$. La méthode pour trouver une telle relation est étudiée dans [Choi et al., 2006]. L'opérateur d'élargissement y est décrit en détail, nous en rappelons simplement la définition ici :

$$P^\sharp \bigtriangledown^\sharp Q^\sharp = \begin{cases} P^\sharp & \text{si } Q^\sharp = \bot^\sharp \\ Q^\sharp & \text{si } P^\sharp = \bot^\sharp \\ \{.p \bigtriangledown^k .q \mid p \in P^\sharp, q \in Q^\sharp, p \mathcal{R} q\} & \text{sinon} \\ \quad \text{pour une certaine relation totale } \mathcal{R} \text{ de } P^\sharp \times Q^\sharp & \\ \quad \text{et un certain } k \text{ fixé} & \end{cases}$$

$$p.q \bigtriangledown^k p'.q' = \begin{cases} pq \odot^k p'q' & \text{si } q = \epsilon \text{ ou } q' = \epsilon \\ (p \odot^k p') \mathbin{\hat{}} a \mathbin{\hat{}} (.r \bigtriangledown^k .r') & \text{si } q = ar, q' = ar', \text{et } \textit{htr-d'étoile}(a) \leq k \\ pa.r \bigtriangledown^k p'a.r' & \text{si } q = ar, q' = ar', \text{et } \textit{htr-d'étoile}(a) > k \\ pa.r \bigtriangledown^k p'.a'r' & \text{si } q = ar, q' = a'r', a \neq a', \text{et } |q| > |q'| \\ p.ar \bigtriangledown^k p'a'.r' & \text{si } q = ar, q' = a'r', a \neq a', \text{et } |q| \leq |q'| \\ \text{où } |q| = n \text{ pour } q = a_1a_2 \cdots a_n & \\ \text{et } \textit{htr-d'étoile}(a) \text{ est la hauteur des répétitions dans } q & \end{cases}$$

$$p \odot^k q = \begin{cases} \epsilon & \text{si } p, q \in \{\epsilon\} \\ p^\circledast & \text{si } p \neq \epsilon, q = \epsilon, \text{et } \textit{htr-d'étoile}(p^*) \leq k \\ q^\circledast & \text{si } p = \epsilon, q \neq \epsilon, \text{et } \textit{htr-d'étoile}(q^*) \leq k \\ (.p' \bigtriangledown^{k-1} .q')^\circledast & \text{si } p = p'^*, q = q'^*, \text{et } k \geq 2 \\ (.p' \bigtriangledown^{k-1} .q)^\circledast & \text{si } p = p'^*, q \neq q'^*, \text{et } k \geq 2 \\ (.p \bigtriangledown^{k-1} .q')^\circledast & \text{si } p \neq p'^*, q = q'^*, \text{et } k \geq 2 \\ \{c \mid c \text{ apparaît dans } p \text{ ou } q\}^\circledast & \text{sinon} \\ \text{où } q^\circledast = q \text{ si } (q = q'^* \text{ ou } q = \epsilon) \text{ et } q^* \text{ sinon} & \end{cases}$$

Le contrôle de la hauteur d'étoile est nécessaire à la preuve de terminaison donnée dans [Choi et al., 2006], même si, en pratique, nous n'avons pas trouvé de cas ne terminant pas.

### 4.6.5 Chaînes rationnelles avec ensembles d'entiers

Pour faciliter les opérations de conversion entre chaînes et entiers, assez fréquentes dans des applications *web*, nous avons introduit un nouvel atome pour représenter des entiers au sein

d'une chaîne. Par exemple, $\mathsf{string\_of\_int}([42, 1984])$ donne l'ensemble de chaînes rationnelles $\{4[2-9], [5-9][0-9], [1-9][0-9][0-9], 1[0-8][0-9][0-9], 19[0-7][0-9], 198[0-4]\}$. Si plusieurs entiers sont utilisés ainsi dans des chaînes, on observe une explosion rapide de la taille des ensembles de chaînes rationnelles.

Nous redéfinissons donc **Atom** et **Reg** ainsi :

$$\begin{array}{lll} \textbf{Int} & I, J & \in D_I^\sharp \setminus \{\bot^\sharp\} \\ \textbf{Atom} & a & ::= C \mid I \mid r^* \\ \textbf{Reg} & p, q, r & \in \{a_1 a_2 \cdots a_n \mid n \geq 0, a_i \in \textbf{Atom}, \\ & & \quad \neg\exists i \ (a_i = p^* \wedge a_{i+1} = q^* \vee a_i = I \wedge a_{i+1} = J)\} \end{array}$$

L'opérateur de concaténation est redéfini de manière à fusionner deux ensembles d'entiers successifs ; et l'opérateur d'élargissement est redéfini en utilisant l'opérateur d'élargissement sur les entiers, lorsque cela est nécessaire.

On prendra pour domaine abstrait $D_S^\sharp = \{\bot^\sharp\} \cup \mathcal{P}(\textbf{Reg})$.

### 4.6.6 Opérateurs

Pour limiter la taille des ensembles de chaînes rationnelles et pouvoir les comparer plus facilement, on essayera autant que possible de ne pas avoir de chaînes rationnelles $p$ et $p'$ dans $P$ telles que $\mathcal{L}(p) \subseteq \mathcal{L}(p')$. On notera $\Downarrow$ un opérateur qui effectue une telle simplification

$$\begin{array}{rccl} \Downarrow : & D_S^\sharp & \longrightarrow & D_S^\sharp \\ & P^\sharp & \longmapsto & P'^\sharp \quad \text{tel que } \mathcal{L}(P'^\sharp) = \mathcal{L}(P^\sharp) \text{ et } \mathrm{card}(P'^\sharp) \leq \mathrm{card}(P^\sharp) \end{array}$$

Typiquement, lorsqu'on veut construire un certain $C^n$, on obtiendra l'ensemble $\{\epsilon, C\}$, puis $\{C, CC\}$, qui sera élargi en $\{C^*, CC^*\}$. Dans un tel cas, l'ensemble sera simplifié par $\Downarrow$ en $\{C^*\}$.

Les opérateurs sur les éléménts de $D_S^\sharp$ sont les suivants :

- l'union : $P_1^\sharp \cup^\sharp P_2^\sharp = \Downarrow P_1^\sharp \cup P_2^\sharp$
- l'intersection :

$$P_1^\sharp \cap^\sharp P_2^\sharp \quad = \quad \Downarrow \bigcup_{\substack{p_1 \in P_1^\sharp \\ p_2 \in P_2^\sharp}} p_1 \cap^\sharp p_2$$

  où $q = p_1 \cap^\sharp p_2$ est une sur-approximation de $p_1 \cap p_2$ de sorte que $\mathcal{L}(q) \supseteq \mathcal{L}(p_1) \cap \mathcal{L}(p_2)$. Une sur-approximation de cette opération peut être implémentée avec un algorithme exponentiel en la taille des chaînes — qui teste pour chaque répétition d'une chaîne toutes les césures dans l'autre chaîne.
- la différence unitaire, selon la définition donnée en 4.1. Avec

$$\begin{array}{rcll} (\mathrm{card}^\sharp(P^\sharp) = 1) & = & \bigwedge_{0 \leq i < n} (\mathrm{card}(a_i) = 1) & \text{si } P = \{p\} \text{ et } p = a_1 a_2 \cdots a_n \\ & & faux & \text{sinon} \end{array}$$

$$\begin{array}{rcll} (\mathrm{card}(a) = 1) & = & (\mathrm{card}(C) = 1) & \text{si } a = C \\ & & (\mathrm{card}(I) = 1) & \text{si } a = I \\ & & faux & \text{si } a = r^* \end{array}$$

– le test d'égalité :

$$P_1^\sharp =^\sharp P_2^\sharp = \begin{cases} \bot^\sharp & \text{si } P_1^\sharp = \bot^\sharp \text{ ou } P_2^\sharp = \bot^\sharp \\ \varnothing^\sharp & \text{si } P_1^\sharp = \varnothing^\sharp \text{ ou } P_2^\sharp = \varnothing^\sharp \\ \bigcup^\sharp_{\substack{p_1 \in P_1^\sharp \\ p_2 \in P_2^\sharp}} (p_1 =^\sharp p_2) & \text{sinon} \end{cases}$$

$$p_1 =^\sharp p_2 = \begin{cases} \{T\} & \text{si } p_1 = \epsilon \text{ et } p_2 = \epsilon \\ \{F\} & \text{si } p_1 = \epsilon, p_2 \neq \epsilon \text{ ou } p_1 \neq \epsilon, p_2 = \epsilon \\ \{T, F\} & \text{si } p_1 = q_1^* r_1 \text{ et } p_2 = q_2^* r_2 \\ (C_1 =^\sharp C_2) \wedge^\sharp (q_1 =^\sharp q_2) & \text{si } p_1 = C_1 q_1 \text{ et } p_2 = C_2 q_2 \\ (I_1 =^\sharp I_2) \wedge^\sharp (q_1 =^\sharp q_2) & \text{si } p_1 = I_1 q_1 \text{ et } p_2 = I_2 q_2 \\ \{T, F\} & \text{sinon} \end{cases}$$

– la conversion de chaîne en entier :

$$\mathsf{int\_of\_string}^\sharp P^\sharp = \begin{cases} \bot^\sharp & \text{si } P^\sharp = \bot^\sharp \\ \downarrow_{\leq N}^{\leq \infty} \bigcup_{\substack{p \in P^\sharp \\ p \neq \epsilon}} \mathsf{int\_of\_string}^\sharp p & \text{sinon} \end{cases}$$

$$\mathsf{int\_of\_string}^\sharp p = \begin{cases} [0,0] & \text{si } p = \epsilon \\ \bigcup_{c \in C} (\mathsf{int\_of\_char\ c}) \mathbin{\hat{}}^\sharp (\mathsf{int\_of\_string}^\sharp q) & \text{si } p = Cq \\ I \mathbin{\hat{}}^\sharp (\mathsf{int\_of\_string}^\sharp q) & \text{si } p = Iq \\ J \cup (J +^\sharp [0, +\infty]), J = \mathsf{int\_of\_string}^\sharp q & \text{si } p = r^* q \end{cases}$$

– la conversion de chaîne en nombre à virgule flottante, très grossière :

$$\mathsf{float\_of\_string}^\sharp P^\sharp = \begin{cases} \bot^\sharp & \text{si } P^\sharp = \bot^\sharp \\ \bigcup_{p \in P^\sharp} \mathsf{float\_of\_string}^\sharp p & \text{sinon} \end{cases}$$

$$\mathsf{float\_of\_string}^\sharp p = \begin{cases} \{\mathsf{float\_of\_string\ s}\} & \text{si } \mathrm{card}^\sharp(p) = 1 \text{ et } \mathcal{L}(p) = \{s\} \\ \top^\sharp & \text{sinon} \end{cases}$$

– la conversion d'entier en chaîne : $\mathsf{string\_of\_int}^\sharp X^\sharp = \{X^\sharp \textit{ considéré comme un atome}\}$

– l'extraction d'une sous-chaîne : $\mathsf{sub}^\sharp(P^\sharp, I^\sharp, L^\sharp)$. Cette question est traitée dans [Choi et al., 2006] uniquement dans le cas où $I^\sharp$ et $L^\sharp$ sont en fait des constantes :

$$\mathsf{sub}^\sharp(P^\sharp, i, l) = \bigcup_{p \in P^\sharp} \mathsf{sub}^\sharp(p, i, l)$$

$$\mathsf{sub}^\sharp(p, i, l) = \begin{cases} \{\epsilon\} & \text{si } i = 0, l = 0 \\ \{C\} \mathbin{\hat{}}^\sharp \mathsf{sub}^\sharp(q, 0, l-1) & \text{si } i = 0, j > 0, p = Cq \\ \mathsf{sub}^\sharp(q, i-1, l) & \text{si } i > 0, l > 0, p = Cq \\ \mathsf{sub}^\sharp(r \mathbin{\hat{}}^\sharp r^* q, i, l) \cup \mathsf{sub}^\sharp(q, i, l) & \text{si } i \geq 0, j > 0, p = r^* q \\ \varnothing^\sharp & \text{sinon} \end{cases}$$

Nous avons étendu cet opérateur pour gérer les atomes ensemble d'entiers, il suffit de découper les intervalles d'entiers. Nous avons également étendu l'opérateur pour traiter les cas où $I^\sharp$ et $L^\sharp$ sont des éléments quelconques de $D_I^\sharp$. L'idée générale est de faire l'union du même calcul pour $i \in I^\sharp$ et $l \in L^\sharp$ mais en utilisant une autre approximation lorsque $i$ et $l$ deviennent «trop grands» – par exemple supérieurs au nombre d'atomes de la chaînes.

Cette autre approximation, $\mathsf{sub}^{\star\sharp}$, consiste à garder les répétitions intactes, on obtient alors des facteurs de la chaîne rationnelle initiale, qui sont en nombre fini (équivalent au carré du nombre d'atomes $at(p)$ de la chaîne). Toutefois ce nombre peut être grand. Une solution pour le limiter consiste à remplacer l'union par un élargissement :

$$\mathsf{sub}^{\sharp}(P^{\sharp}, I^{\sharp}, L^{\sharp}) \quad = \quad \Downarrow \bigcup_{p \in P^{\sharp}} \left( \begin{array}{l} \bigcup_{\substack{i \in I^{\sharp} \\ i \leq at(p)}} \bigtriangledown^{\sharp}_{\substack{l \in L^{\sharp} \\ l \leq at(p)}} \mathsf{sub}^{\sharp}(p,i,l) \\ \bigcup \bigcup_{\substack{0 \leq i \leq at(p) \\ begin(i, I^{\sharp}, p) \\ 0 \leq l \leq at(p)}} \mathsf{sub}^{\star\sharp}(p,i,l) \end{array} \right)$$

– le remplacement de caractères : $\mathsf{replace}^{\sharp}(P^{\sharp}, C_1^{\sharp}, C_2^{\sharp})$,
– la suppression de blancs à gauche : $\mathsf{ltrim}^{\sharp}(P^{\sharp})$. On se reportera pour ces deux opérateurs à la définition donnée dans [Choi et al., 2006].

Sur le même modèle, on peut ainsi ajouter d'autres opérateurs de façon relativement aisée.

## 4.7 Types enregistrements

On utilisera pour ces types le domaine abstrait produit des domaines abstraits des types de chacun des champs. Pour un type enregistrement

**type** T = { field$_1$ : T$_1$ ; field$_2$ : T$_2$ ; ... ; field$_n$ : T$_n$ }

le domaine abstrait sera $D_T^{\sharp} = D_{T_1}^{\sharp} \times D_{T_2}^{\sharp} \times \ldots \times D_{T_n}^{\sharp}$ dont on notera les valeurs $X^{\sharp} = \{field_1 = X_1^{\sharp}; field_2 = X_2^{\sharp}; \ldots; field_n = X_n^{\sharp}\}$.

## 4.8 Types unions

On utilisera également pour ces types un domaine abstrait produit des domaines abstraits de chacun des cas de l'union, auquel on associe également une valeur booléenne indiquant la présence du cas de l'union dans la valeur. Pour un type union

**type** T = T$_1$ / T$_2$ / ... / T$_n$

le domaine abstrait sera $D_T^{\sharp} = D_B^{\sharp} \times D_{T_1}^{\sharp} \times D_B^{\sharp} \times D_{T_2}^{\sharp} \ldots \times D_B^{\sharp} \times D_{T_n}^{\sharp}$ dont on notera les valeurs $X^{\sharp} = B_1^{\sharp}(X_1^{\sharp}) / B_2^{\sharp}(X_2^{\sharp}) / \ldots / B_n^{\sharp}(X_n^{\sharp})$.

## 4.9 Cas des types récursifs

Les domaines abstraits précédents ne peuvent pas être utilisés dans le cadre de types (enregistrements ou unions) récursifs. Il est donc nécessaire d'avoir d'autres domaines abstraits adaptés à ces types.

### 4.9.1 Cas des types listes

On appelle *type liste* tout type équivalent à

**type** (T$_1$,T$_2$) list = { hd : T$_1$ ; tl : (T$_1$,T$_2$) list } / { nil : T$_2$ }

c'est-à-dire une chaîne de valeurs de type T$_1$ terminée par une valeur de type T$_2$.

Pour les types listes, on pourrait utiliser un domaine abstrait proche de celui des chaînes rationnelles en remplaçant **Char** par le domaine abstrait du type T$_1$, $D_{T_1}^{\sharp}$.

### 4.9.2 Approximation par union

Tous les types récursifs ne sont pas aussi simples que les types listes. Une possibilité est d'unir toutes les valeurs d'un certain champ dans la structure récursive du type. Par exemple pour un type liste, la valeur abstraite pourrait être $\{hd = X_1^\sharp \in D_{T_1}^\sharp ; nil = X_2^\sharp \in D_{T_2}\}$ où la valeur $X_1^\sharp$ est l'union des valeurs du champ $hd$ prise sur toute la structure de la liste.

### 4.9.3 Approximation plus grossière

Enfin, on peut remplacer les champs de type récursif par une valeur $\top_T^\sharp$. Pour un type liste, la valeur abstraite serait $B_1^\sharp(\{hd = X_1^\sharp \in D_{T_1}^\sharp ; tl = \top_{(T_1,T_2)list}^\sharp\})/B_2^\sharp(\{nil = X_2^\sharp \in D_{T_2}^\sharp\})$. Faute de temps, c'est l'approximation que nous avons utilisé, à la différence toutefois que les niveaux de récursivité des valeurs abstraites d'un type récursif sont déroulés autant que le nécessite le programme et la valeur $\top_T^\sharp$ n'est utilisée que pour des valeurs de taille non bornée.

## 4.10 Bases de données

### 4.10.1 Domaine abstrait

Si on note $D_{Scal}^\sharp$ l'union des domaines abstraits scalaires $D_B^\sharp \cup D_I^\sharp \cup D_F^\sharp \cup D_S^\sharp$, une base de donnée QML peut être représentée comme une union de parallèlépipèdes rectangles disjoints de dimension finie dans l'union des espaces-produits ${D_{Scal}^\sharp}^\star = \bigcup_{k=1}^\infty {D_{Scal}^\sharp}^k$ auxquels sont associés des valeurs dans $D_{Scal}^\sharp$. Comme l'opération d'intersection est complexe et potentiellement génératrice d'un grand nombre de valeur, surtout pour les chaînes de caractères, on pourra se contenter de parallèlépipèdes rectangles que l'on ordonnera afin d'obtenir un ensemble similaire. C'est ce que nous utiliserons comme domaine abstrait :

$$D_{DB}^\sharp \quad = \quad \{\bot^\sharp\} \cup \{(path_i, val_i)_{0\leq i<n} \mid path_i \in {D_{Scal}^\sharp}^\star, val_i \in D_{Scal}^\sharp, n > 0\}$$

La valeur $(path_i, val_i)_{0\leq i<n}$ représente la base de données

$$\{[\![path_1]\!] \mapsto val_1, [\![path_2 \setminus path_1]\!] \mapsto val_2, \ldots, [\![path_{n-1} \setminus \bigcup_{0\leq i<n-1} path_i]\!] \mapsto val_{n-1}\}$$

### 4.10.2 Opérateurs

– l'union essaie de garder le résultat le plus précis entre l'ajout de valeurs de la première base de données à la seconde et l'ajout de valeurs dans l'autre sens :

$$\begin{array}{lcll} A_1^\sharp \cup^\sharp A_2^\sharp & = & A_1^\sharp & \text{si } A_2^\sharp = \bot^\sharp \\ & & A_2^\sharp & \text{si } A_1^\sharp = \bot^\sharp \\ & & (A_1^\sharp \overleftarrow{\cup}^\sharp A_2^\sharp) \vee^\sharp (A_2^\sharp \overleftarrow{\cup}^\sharp A_1^\sharp) & \end{array}$$

$$\begin{array}{lcll} A_1^\sharp \vee^\sharp A_2^\sharp & = & A_1^\sharp & \text{si } \mathrm{card}(A_1^\sharp) > \mathrm{card}(A_2^\sharp) \\ & & A_2^\sharp & \text{sinon} \end{array}$$

$$\begin{array}{lcll} A_1^\sharp \overleftarrow{\cup}^\sharp A_2^\sharp & = & A_1^\sharp & \text{si } A_2^\sharp = \varnothing^\sharp \\ & & (A_1^\sharp \overleftarrow{\cup}^\sharp (path_n, val_n)) \overleftarrow{\cup}^\sharp (path_i, val_i)_{0\leq i<n-1} & \text{si } A_2^\sharp = (path_i, val_i)_{0\leq i<n} \end{array}$$

$$A^\sharp \overset{\leftarrow^\sharp}{\cup} (path, val) = \begin{cases} A^\sharp & \text{si } path = \varnothing \\ (path, val \cup A^\sharp[\![path]\!]) ::^\sharp (A^\sharp \setminus^\sharp path) & \text{sinon} \end{cases}$$

$$A^\sharp[\![path]\!] = \begin{cases} \varnothing^\sharp & \text{si } A^\sharp = \varnothing^\sharp \\ (path_i, val_i)_{2\leq i\leq n}[\![path \setminus path_1]\!] \cup val_1 & \text{si } path \cap path_1 \neq \varnothing \\ (path_i, val_i)_{2\leq i\leq n}[\![path]\!] & \text{sinon} \end{cases}$$
$$\text{avec } A^\sharp = (path_i, val_i)_{1\leq i\leq n}$$

$$(path_0, val_0) ::^\sharp (path_i, val_i)_{1\leq i\leq n} = (path_i, val_i)_{0\leq i\leq n}$$

$$(path_i, val_i)_{1\leq i\leq n} \setminus^\sharp path = \begin{cases} \varnothing & \text{si } n = 0 \\ (path_i, val_i)_{2\leq i\leq n} \setminus^\sharp path & \text{si } path_1 \subseteq path \\ (path_1, val_1) ::^\sharp ((path_i, val_i)_{2\leq i\leq n} \setminus^\sharp path) & \text{sinon} \end{cases}$$

– on ne donne aucun opérateur d'intersection pour les bases de données
– la lecture d'une valeur : $A^\sharp[\![path]\!]$
– la modification d'une valeur :

$$(A^\sharp \leftarrow [\![path]\!] := val) = \begin{cases} A^\sharp & \text{si } path = \varnothing \text{ ou } val = \varnothing \\ (path, val) ::^\sharp (A^\sharp \setminus^\sharp path) & \text{si } \mathrm{card}^\sharp(path) = 1 \\ A^\sharp \overset{\leftarrow^\sharp}{\cup} (path, val) & \text{sinon} \end{cases}$$

# 5 Analyse statique de code QML

L'analyse se déroule en trois parties :

1. une étape de parcours des arbres d'expression QML pour construire le système de *contraintes* et la liste des *vérifications*
2. une étape de calcul des valeurs abstraites nécessaires aux vérifications, à partir des contraintes
3. une étape qui applique les vérifications et donne des messages d'alertes ou d'erreurs

Dans un premier temps, nous nous intéresserons à la partie purement fonctionnelle de QML en ignorant la base de données.

## 5.1 Interprétation abstraite de l'algèbre QML

Le *contexte* est une fonction de l'ensemble des variables $\mathcal{V}$ vers l'ensemble des identifiants abstraits $D^{0\sharp} = \bigcup_{\mathsf{T}\ \text{type}} \mathsf{T}(X^{0\sharp} \in D_T^{0\sharp})$, c'est-à-dire l'union des domaines abstraits de tous les types dans lesquels les valeurs abstraites scalaires sont remplacées par des identifiants uniques. L'identifiant $X^{0\sharp}$ fait référence à la valeur abstraite $X^\sharp$. Nous noterons $X^{0^+\sharp}$ un nouvel identifiant unique.

Les *contraintes* sont des relations de la forme $X^{0\sharp} := f^{0\sharp}(Y_1^{0\sharp}, Y_2^{0\sharp}, \ldots, Y_n^{0\sharp})$ où $f^\sharp$ est soit une constante abstraite du même type que $X$, soit un opérateur (cf. 4.1, 4.2.2, 4.3.3, 4.4.2, 4.6.6 et 4.10.2).

La fonction principale de l'analyseur est la fonction absint_expr, qui à une expression *expr* et un contexte $\sigma$ associe un identifiant abstrait tout en générant une liste de contraintes et de vérifications (cf. 5.3) :

| *expr* | type | identifiant abstrait | contraintes générées |
|---|---|---|---|
| *const* | T scalaire | $\mathsf{T}(X^{0^+\sharp} \in D_T^{0\sharp})$ | $X^\sharp :=$*const* |
| { } | { } | { }() | |
| v | T | $\sigma(\mathsf{v})$ | |
| (e : T) | T (union) | $\mathsf{T}(U^{0\sharp})$ | $B_i^\sharp := \{T\}$, et pour $k \neq i, B_k^\sharp := \{F\}$ |
| | où $U^{0\sharp} = \{B_1^{0^+\sharp}(X_1^{0^+\sharp}); \ldots; B_i^{0^+\sharp}(X_\mathsf{e}^{0\sharp}); \ldots; B_n^{0^+\sharp}(X_n^{0^+\sharp})\}$, $i$ est l'index du cas du type de e dans le type union T, et $X_\mathsf{e}^{0\sharp} = \mathsf{absint\_expr}(\mathsf{e},\sigma)$ | | |
| **fun** v →e | $\mathsf{T}_1 \to \mathsf{T}_2$ | $(\mathsf{T}_1 \to \mathsf{T}_2)(X_1^{0^+\sharp} \to X_2^{0\sharp})$ | |
| | où $X_2^{0\sharp} = \mathsf{absint\_expr}(\mathsf{e}, \sigma[\mathsf{v} \leftarrow X_1^{0\sharp}])$ | | |
| $\mathsf{e}_1\ \mathsf{e}_2$ | $\mathsf{T}_2$ | $\mathsf{T}_2(X_{12}^{0\sharp})$ | $X_{11}^{0\sharp} := X_{11}^{0\sharp} \cup^{0\sharp} X_2^{0\sharp}$ |
| | où $(X_{11}^{0\sharp} \to X_{12}^{0\sharp})\langle\mu\rangle = \mathsf{absint\_expr}(\mathsf{e}_1,\sigma)$ et $X_2^{0\sharp} = \mu(\mathsf{absint\_expr},\ \mathsf{e}_2,\sigma)$ | | |
| **let** v = $\mathsf{e}_1$ **in** $\mathsf{e}_2$ | $\mathsf{T}_2$ | $\mathsf{T}_2(X_2^{0\sharp})$ | |
| | où $X_2^{0\sharp} = \mathsf{absint\_expr}(\mathsf{e}_2,\sigma[v \leftarrow X_1^{0\sharp}])$, $X_1^{0\sharp} = \mathsf{absint\_expr}(\mathsf{e}_1,\sigma)$ | | |
| **let rec** $\mathsf{v}_1 = \mathsf{e}_1$ [ **and** $\mathsf{v}_2 = \mathsf{e}_2$ [ ... ] ] **in** $\mathsf{e}_{n+1}$ | $\mathsf{T}_{n+1}$ | $\mathsf{T}_{n+1}(X_{n+1}^{0\sharp})$ | |
| | où $X_{n+1}^{0\sharp} = \mathsf{absint\_expr}(\mathsf{e}_{n1},\sigma')$ et $\sigma' = \sigma[v_1 \leftarrow X_1^{0\sharp}, v_2 \leftarrow X_2^{0\sharp}, \ldots, v_n \leftarrow X_n^{0\sharp}]$ et pour $1 \leq i \leq n, X_i^{0\sharp} = \mathsf{absint\_expr}(\mathsf{e}_i,\sigma')$ | | |
| { $\mathsf{f}_0 = \mathsf{e}_0$ } / $\mathsf{e}_1$ | T | $\mathsf{T}(\{f_0 = X_0^{0\sharp}; f_1 = X_1^{0\sharp}; \ldots; f_n = X_n^{0\sharp}\})$ | |
| | où $X_0^{0\sharp} = \mathsf{absint\_expr}(\mathsf{e}_0,\sigma)$ et $\{f_1 = X_1^{0\sharp}; \ldots; f_n = X_n^{0\sharp}\} = \mathsf{absint\_expr}(\mathsf{e}_1,\sigma)$ | | |
| e.f | $\mathsf{T}_f$ | $\mathsf{T}_f(X_f^{0\sharp})$ | |
| | si $\{\ldots; f = X_f^{0\sharp}; \ldots\} = \mathsf{absint\_expr}(\mathsf{e},\sigma)$ | | |
| **match** $\mathsf{e}_0$ **with** | $\mathsf{T}_1$ | $\mathsf{T}_1(X^{0\sharp})$ | |
| \| $\mathsf{pat}_1$ →$\mathsf{e}_1$ | | | $X^{0\sharp} := \bigcup_{1 \leq i \leq n}(B_1^{0^+\sharp} \gg^{0\sharp} X_1^{0\sharp})$ |
| [ \| $\mathsf{pat}_2$ →$\mathsf{e}_2$ [ ... ] ] | | | $B_i^{0\sharp} := B_{i-1}^{0\sharp} \wedge^{0\sharp} \mathsf{canmatch}(\mathsf{e}_0,\ \mathsf{pat}_i)$ |
| | où $X_i^{0\sharp} = \mu_{i\text{-}1}(\mathsf{absint\_expr},\ \mathsf{e}_i,\sigma)$ | | |
| %%*fonction*%% | $\mathsf{T}_1 \to \mathsf{T}_2$ | $(\mathsf{T}_1 \to \mathsf{T}_2)(Lib(fonction)\langle\mu_{fonction}\rangle)$ | |
| **assert** $\mathsf{e}_1$ **in** $\mathsf{e}_2$ | $\mathsf{T}_2$ | $\mathsf{T}_2(X_2^{0\sharp})$ | |
| | où $X_i^{0\sharp} = \mu_{i\text{-}1}(\mathsf{absint\_expr},\ \mathsf{e}_i,\sigma)$ | | |

Remarquez

la notation $(X_1^{0\sharp} \to X_2^{0\sharp})\langle\mu\rangle =$ qui indique qu'une fonction $\mu$ est associée à l'identifiant abstrait $(X_1^{0\sharp} \to X_2^{0\sharp})$ — en cas d'absence, on pourra considérer qu'il s'agit de l'identité. Cette fonction $\mu$ associe à une fonction de même signature que absint_expr une autre fonction de même signature, mais dont le comportement a pu être modifié. Les fonctions de la bibliothèque possèdent toutes une telle fonction $\mu$ qui permet de modifier le contexte ou de lier à une sortie de fonction la valeur issue du calcul de cette fonction.

De telles fonctions associées à un identifiant abstrait de booléen permet d'obtenir des informations sur les disjonctions de cas. Exemple :

```
let b = (x > 1) in
```

À l'identifiant abstrait $B^{0\sharp}$ du booléen b sont associées deux fonctions qui correspondent respectivement au cas true et au cas false :

$\mu_T(\mathsf{absint\_expr}, \mathsf{e}, \sigma) = \mathsf{absint\_expr}(\mathsf{e},\ \sigma[X^{0\sharp} \leftarrow X_T^{0^+\sharp}])$ avec $X_T^{0\sharp} = X^{0\sharp} \cap^{0\sharp} [2, +\infty]^{0\sharp}$
$\mu_F(\mathsf{absint\_expr}, \mathsf{e}, \sigma) = \mathsf{absint\_expr}(\mathsf{e},\ \sigma[X^{0\sharp} \leftarrow X_F^{0^+\sharp}])$ avec $X_T^{0\sharp} = X^{0\sharp} \cap^{0\sharp} [-\infty, 1]^{0\sharp}$
où $X^{0\sharp}$ est l'identifiant abstrait de x

Enfin précisons que l'intégration de la base de données à l'analyseur est aisée puisque cette partie impérative du langage est unique et globale. Il suffit en effet de faire l'union de toutes les modifications de valeurs de la base de données.

## 5.2 Calcul des valeurs abstraites

Le système de contraintes précédemment généré est représenté sous la forme d'un graphe $G$ dans lequel les nœuds sont les identifiants abstraits. Un arc de $X^{0\sharp}$ vers $Y^{0\sharp}$ est présent si et seulement si il existe une contrainte de la forme $X^{0\sharp} := f(Y^{0\sharp})$.

Le graphe est réduit au sous-graphe accessible depuis au moins un identifiant abstrait présent dans une vérification.

Le système de contraintes est simplifié (mise en commun des unions) et factorisé (fusion des identifiants dont les contraintes sont identiques). D'où un nouveau graphe de contraintes $G'$.

On en calcule les composantes fortement connexes dont on fait le tri topologique. La méthode d'approximation de point fixe peut ensuite être appliquée sur chacune des composantes fortement connexes indépendamment, prises dans l'ordre du tri topologique (propagation des valeurs calculées).

Effectuer un calcul de point fixe sur chaque composante offre donc un gain de performance par rapport à un calcul effectué sur le système de contraintes en entier.

Pour calculer les valeurs abstraites $X_1^\sharp, \ldots, X_n^\sharp$ d'un sous-ensemble de contraintes

$$\begin{array}{ccc} X_1^{0\sharp} & := & f_1^{0\sharp}(Y_{11}^{0\sharp}, \ldots, Y_{1k_1}^{0\sharp}) \\ \vdots & \vdots & \vdots \\ X_n^{0\sharp} & := & f_n^{0\sharp}(Y_{n1}^{0\sharp}, \ldots, Y_{nk_n}^{0\sharp}) \end{array}$$

on pose tout d'abord $X_i^\sharp = \bot^\sharp$ pour $0 \leq i \leq n$. Puis chacune des contraintes est appliquée, c'est-à-dire qu'on calcule pour $1 \leq i \leq n$, la valeur $X_i'^\sharp = f_i^\sharp(Y_{i1}^\sharp, \ldots, Y_{ik_i}^\sharp)$. Si le type de $X$ possède un opérateur d'élargissement, on remplace $X_i^\sharp$ par $X_i^\sharp \bigtriangledown^\sharp X_i'^\sharp$ ; sinon on le remplace par $X_i'^\sharp$. L'itération est répétée jusqu'à obtenir un point fixe, i.e. pour $1 \leq i \leq n$, $X_i^\sharp = X_i'^\sharp$.

Si un des types des variables des contraintes possède un opérateur de rétrécissement, alors les mêmes opérations sont ensuite effectuées en remplaçant les opérateurs d'élargissement par les opérateurs de rétrécissement.

## 5.3 Vérifications

Une vérification est un couple $(X^{0\sharp}, \nu)$ d'un identifiant abstrait et d'une fonction qui prend en paramètre une valeur abstraite, et dont le but est d'effectuer une certaine action dépendant de la valeur abstraite calculée correspondant à cet identifiant abstrait.

Des vérifications sont créées à chaque **assert** (si la valeur n'est pas toujours vraie, afficher un message d'avertissement), mais aussi par des fonctions de la bibliothèque (par exemple pour tester les divisions par 0 ou vérifier que les paramètres entiers de sub sont positifs ou nuls).

Une fois les valeurs abstraites calculées, les vérifications sont exécutées. Suivant le résultat, le programme pourra être rejeté avant la compilation.

## 5.4 Implémentation

L'analyseur a été écrit en OCaml et fait environ 5000 lignes. Il utilise les fonctions déjà implémentées des analyseurs lexical et syntaxique et du typeur de QML, ce qui permet de disposer directement de l'arbre abstrait du programme annoté par les types des expressions.

Un *pretty printer* de chaque structure de données a été implémenté afin de suivre facilement les étapes de calcul et d'afficher les valeurs abstraites calculées en fin d'analyse.

Les opérations sur le graphe des contraintes ont été réalisées grâce à la bibliothèque OCamlGraph [Cochon et al., 2008]. Une sortie GraphViz DOT [Gansner et al., 2008] peut être produite.

À titre d'exemple, deux types de propriétés CSS ont été implémentées : les longueurs et les couleurs. Elles s'expriment très simplement grâce aux chaînes rationnelles avec ensembles d'entiers.

## 5.5 Exemples

1. Ce premier exemple est tiré de [Cousot et Cousot, 1977].

```
val foo = let rec f x = if x <= 100 then f (x+1) else x
          in f 1
```

   La réponse calculée est exacte : foo $\in [101, 101]$.

2. l'exemple suivant montre le fonctionnement sur les types unions. Un x de t2 est toujours associé à un a de t1, et un y est toujours associé à un b.

```
type t2 = { x : int } / { y : int }
type t1 = { a : unit ; c : t2 } / { b : unit ; c : t2 }
val bar =
    let rec f z = match (z : t1) with
    | { a = () ; c = { x = i } } →
        if i < 1000 then
            f ({ b = () ; c = ({ y = i + 1 } : t2) } : t1)
        else
            z
    | { b = () ; c = { y = i } } →
        if i < 2000 then
            f ({ a = () ; c = ({ x = i + 1 } : t2) } : t1)
        else
            z
    | _ → z
    in f ({ a = () ; c = ({ x = 0 } : t2)} : t1)
```

   Notre analyseur fournit la réponse suivante :

$$\begin{array}{rl} & \{T\}\{a = \textbf{unit}; c = \{T\}\{x \in [0,0] \cup [2,2] \cup [4,2000]\}/\{F\}\{y \in \varnothing\}\} \\ / & \{F\}\{b = \textbf{unit}; c = \{F\}\{x \in \varnothing\}/\{T\}\{y \in [1,1] \cup [3,1000]\}\} \end{array}$$

   qu'il faut lire ainsi : la valeur est toujours un a dans le type union t1, dont le champ c est toujours un x dans l'ensemble $[0,0] \cup [2,2] \cup [4,2000]$ (on devine les entiers pairs). Remarquez qu'on a également une information dans le cas b, même s'il est indiqué que celui-ci ne se produit jamais.

   Je m'attendais à n'obtenir une information que sur l'association a/x, b/y ; donc j'ai été surpris de découvrir que la valeur finale était bien un a.

3. Enfin, précisons que notre analyseur a été testé avec des applications *web* de taille plus importante. À titre d'exemple, un fichier QML de 1500 lignes (90 Ko) a été analysé en 3 secondes sur une machine moderne. Les résultats obtenus – après calcul sur plus de 45000 valeurs abstraites – sur les chaînes de caractères et la base de données fournissaient des informations pour la plupart suffisamment précises pour être utiles.

# 6 Conclusion

## 6.1 Résumé

Nous avons implémenté un analyseur de programmes QML par interprétation abstraite dans un domaine non relationnel mais offrant des résultats relativement précis sur les valeurs de tout type de données, notamment entiers (domaine des multi-intervalles), chaînes de caractères (domaine des chaînes rationnelles avec ensembles d'entiers) et bases de données.

## 6.2 Autres contributions

Mon travail au sein de la société MLstate ne s'est pas limité à l'analyseur, j'ai aussi pu contribuer en optimisant localement la grammaire PEG de QML ; en assemblant table et chaises nécessaires face à l'agrandissement de l'équipe ; et surtout en participant, en tant que conseiller, à la rédaction d'un brevet sur la technologie MLstate, où j'ai pu découvrir le vocabulaire et les difficultés de rédaction et du cadre juridique pour ce type de brevet.

## 6.3 Idées pour aller plus loin

Deux mois ne sont bien sûr pas suffisant pour la réalisation d'outils d'analyse, de vérification et de preuve pour la technologie MLstate en entier. Je laisse ici, comme une *todo list*, une liste de sujets sur lesquels j'ai commencé des réflexions qui me semblent intéressantes sans pour autant avoir le temps de les poursuivre :

- vérification de l'AML (langage de développement *web*, transformé en QML avant compilation) par typage systématique, précis mais aussi non contraignant (par des unions) des balises et attributs HTML, des propriétés CSS, ...afin d'assurer une confirmité parfaite aux standarts du *web*.
- implémentation de domaines relationnels, ou au moins avec bornes symboliques, pouvant s'inspirer de [Miné, 2004].
- preuve d'AJAX, c'est-à-dire prouver qu'à chaque instant le JavaScript généré par AML ne peut accéder qu'à des éléments du DOM présents actuellement sur la page. Cela peut se faire facilement dans certains cas mais nécessite des domaines plus précis, comme ceux du point précédent.
- utilisation des résultats de l'analyseur (conseils au développeur, optimisations, ...).
- preuves formelles de la correction et de la terminaison de l'analyseur.

## 6.4 Remerciements

Je souhaiterais tout d'abord exprimer ma gratitude à mon camarade de classe et ami Mathieu Barbin, qui m'a fait découvrir la société MLstate.

J'ai également à cœur d'exprimer ma profonde reconnaissance à Henri Binsztok qui m'a offert ce stage des plus intéressants, à la fois théorique et pratique, à la frontière entre la recherche et l'application industrielle. L'intérêt et le plaisir que j'y ai pris lui doivent beaucoup : ses conseils,

son attention à mon travail n'ont jamais entamé la liberté d'action qui a été la mienne. Il a su être disponible quand j'en avais besoin, malgré son agenda très chargé. Il restera pour moi un modèle de dirigeant d'entreprise et de chef d'équipe.

Mes remerciements vont aussi à chacun des membres de l'équipe MLstate, accueillante et de bonne humeur, avec qui il fait bon travailler et engager des discussions. J'en profite pour remercier également la boulangerie Colin, fournisseur officiel de sandwichs de l'équipe, certainement parmi les meilleurs de la ville.

Merci encore, à camarades du cours d'arabe, avec qui j'effectue actuellement un stage linguistique en Égypte, qui m'ont soutenu durant la rédaction de ce rapport ; rapport qui m'a permis de tester mes facultés de travail en conditions difficiles[1].

Enfin, je tiens à exprimer mon regret de ne pouvoir effectuer de soutenance. J'aurais aimé présenter mon travail sous une forme plus vivante. Je m'en excuse donc auprès de Michel Pocchiola et de la direction des études du Département d'Informatique de l'ENS, Patrick Cousot et Laurent Mauborgne. Il m'est par ailleurs impossible de ne pas les remercier pour leur travail et leur contribution au monde de l'informatique. C'est pour moi une chance inestimable de les avoir comme professeurs.

[1] Louxor, Le Caire, fin août, début septembre, souvent sans climatisation

# Bibliographie

[Anley, 2002] Anley C. (2002). Advanced SQL Injection in SQL Server Applications. Technical report, Next Generation Security Software, Ltd.

[Choi et al., 2006] Choi T.-H., Lee O., Kim H., et Doh K.-G. (2006). A practical string analyzer by the widening approach. In *4th Asian Symposium on Programming Languages and Systems (APLAS 2006)*, pages 374–388, Sydney, Australia. Springer-Verlag, New York.

[Christensen et al., 2003] Christensen A. S., Møller A., et Schwartzbach M. I. (2003). Precise analysis of string expressions. In *10th International Static Analysis Symposium, SAS '03, volume 2694 of LNCS*, volume 2694, pages 1–18. Springer-Verlag.

[Cochon et al., 2008] Cochon S., Filliâtre J.-C., et Signoles J. (2004–2008). La bibliothèque OCamlGraph. http ://ocamlgraph.lri.fr/.

[Cousot, 1997] Cousot P. (1997). Types as abstract interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France. ACM Press, New York, NY.

[Cousot, 2000] Cousot P. (2000). Interprétation abstraite. *Technique et science informatique*, 19(1-2-3) :155–164.

[Cousot, 2001] Cousot P. (2001). Abstract interpretation based formal methods and future challenges, invited paper. In Wilhelm R. editor, *« Informatics — 10 Years Back, 10 Years Ahead »*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag.

[Cousot et Cousot, 1977] Cousot P. et Cousot R. (1977). Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California. ACM Press, New York, NY.

[Cousot et al., 2008] Cousot P., Cousot R., Feret J., Mauborgne L., Miné A., et Rival X. (2008). *Interprétation abstraite : application à la vérification et à l'analyse statique.* Master Parisien de Recherche en Informatique, Paris.

[Dor et al., 2001] Dor N., Rodeh M., et Sagiv S. (2001). Cleanness checking of string manipulations in C programs via integer analysis. In *SAS '01 : Proceedings of the 8th International Symposium on Static Analysis*, pages 194–212, London, UK. Springer-Verlag.

[Ford, 2004] Ford B. (2004). Parsing expression grammars : A recognition-based syntactic foundation. In *Symposium on Principles of Programming Languages*, Venice, Italy. Massachusetts Institute of Technology.

[Gansner et al., 2008] Gansner E., North S., Koren Y., Hu Y., Bilgin A., et Ellson J. (2001–2008). GraphViz. http ://www.graphviz.org/.

[Hors et al., 2000] Hors A. L., Hégaret P. L., Wood L., Nicol G., Robie J., Champion M., et Byrne S. (2000). Document object model (dom) level 2 core specification. Technical report, World Wide Web Consortium (W3C).

[INRIA, 2008] INRIA (1995-2008). Objective Caml. http ://caml.inria.fr/ocaml/.

[Klein, 2002] Klein A. (2002). Cross Site Scripting Explained. Technical report, Sanctum Inc.

[Mauborgne, 2008] Mauborgne L. (2008). *Cours INF 588 : Analyse statique de programmes.* École Polytechnique, Palaiseau.

[Minamide, 2005] Minamide Y. (2005). Static approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference Committee*, pages 432–441.

[Minamide, 2007] Minamide Y. (2007). Verified Decision Procedures on Context-Free Grammars.

[Minamide et Tozawa, 2006] Minamide Y. et Tozawa A. (2006). XML validation for context-free grammars. In *The 4th ASIAN Symposium on Programming Languages and Systems*, volume LNCS 4279, pages 357–373.

[Miné, 2004] Miné A. (2004). *Domaines numériques abstraits faiblement relationnels.* These de doctorat, École polytechnique.

[Mohri et Nederhof, 2001] Mohri M. et Nederhof M.-J. (2001). Regular approximation of context-free grammars through transformation. *Robustness in Language and Speech Technology*, pages 153–163.

[Simon et King, 2002] Simon A. et King A. (2002). Analyzing String Buffers in C. In Kirchner H. et Ringeissen C. editors, *International Conference on Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 365–379. Springer. Also see http ://www.springer.de/comp/lncs/index.html.

[Spett, 2003] Spett K. (2003). Blind SQL Injection. Technical report, SPI Dynamics.