CloudLens, un langage de script pour l'analyse de données semi-structurées

Guillaume Baudart[1], Louis Mandel[2], Olivier Tardieu[2], Mandana Vaziri[2]

1: École normale supérieure
2: IBM Research

Résumé

Lors de la mise au point d'applications qui s'exécutent dans le nuage, les programmeurs ne peuvent souvent que tracer l'exécution de leur code en multipliant les impressions. Cette pratique génère des quantités astronomiques de fichiers de traces qu'il faut ensuite analyser pour trouver les causes d'un bogue. De plus, les applications combinent souvent de nombreux micro-services, et les programmeurs n'ont qu'un contrôle partiel du format des données qu'ils manipulent. Cet article présente CloudLens, un langage dédié à l'analyse de ce type de données dites semi-structurées. Il repose sur un modèle de programmation flot de données qui permet à la fois d'analyser les sources d'une erreur et de surveiller une application en cours d'exécution.

1. Introduction

L'informatique en nuage se développe de plus en plus [6] et la mise au point d'applications qui s'exécutent dans le nuage amène de nouveaux défis. Ces applications combinent souvent de nombreux micro-services uniquement accessibles aux développeurs par des APIs (*Application Programming Interface*). Le code source des ces micro-services n'est, la plupart du temps, pas accessible. Bien souvent, les développeurs n'ont d'autre alternative que de multiplier les impressions pour tracer l'exécution de leurs applications. Comme l'écrivait déjà Kernighan en 1979 [11] : *"L'outil de débogage le plus efficace reste une réflexion approfondie associée à des impressions judicieusement placées."*[1]

En outre, certains micro-services génèrent eux aussi de nombreuses traces. Les développeurs qui utilisent ces services n'ont parfois aucun contrôle sur l'affichage ou le format de ces messages. Le formats des traces générées par une application n'est donc que partiellement connu. On parle alors de données *semi-structurées*. La multiplication des impressions génère des quantités astronomiques de fichiers de traces, ou *logs*. En cas d'erreur ou d'anomalie, il faut ensuite analyser efficacement toutes ces données pour en trouver les causes.

Nous présentons ici CloudLens, un langage réactif pour l'analyse de ces données semi-structurées qui repose sur un modèle de programmation flot de données. Conçu comme une extension de JavaScript, le langage possède des fonctionnalités dédiées à l'analyse de flots de données brutes. Les scripts CloudLens peuvent être utilisés pour 1) détecter les causes d'une panne, a posteriori, en analysant un fichier log, 2) surveiller un système en cours d'exécution pour signaler les états alarmants en consommant ses traces à la volée.

Nous avons choisi JavaScript pour sa popularité et sa facilité d'accès. En outre, il est possible d'exécuter du code JavaScript dans la machine virtuelle Java ce qui a permis un prototypage rapide de l'interprète du langage écrit en Java. Les idées présentées ici ne sont cependant pas spécifiques à Java ou JavaScript. On pourrait suivre une approche similaire avec Python, Ruby, ou OCaml.

1. "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."

Contributions Nous introduisons CloudLens un nouveau langage réactif pour l'analyse de données semi-structurées (section 2). Nous détaillons ensuite sa sémantique formelle (section 3). Enfin, nous présentons les idées clés de l'implémentation de l'interprète CloudLens (section 4). Cloudlens et les exemples de la section 2 sont disponibles sur GitHub : `https://cloudlens.github.io/jfla17`.

2. Présentation de CloudLens

CloudLens est un langage de programmation *flot de données*. Les programmes sont exécutés sur un flot d'objets JSON construit à partir d'un fichier d'entrée. Lors de la lecture d'un fichier, chaque ligne devient un objet JSON à un seul champs `message` qui contient le texte de la ligne. On appellera ces objets les *entrées* du log.

Illustrons les différentes fonctionnalité de CloudLens avec un fichier log produit par Travis [2]. Travis est un service d'intégration continue disponible, notamment, pour les utilisateurs de GitHub. Lorsqu'un utilisateur modifie le code source d'un projet, Travis lance automatiquement une suite de tests qui permet de s'assurer que cette modification ne produit pas de régression dans l'application développée.

Dans cette section nous utilisons le fichier `log.txt`, un rapport généré par Travis pour le projet OpenWhisk [3]. L'ensemble du fichier log se trouve à l'adresse suivante :
`https://s3.amazonaws.com/archive.travis-ci.org/jobs/144778470/log.txt`

2.1. Un premier script CloudLens

Notre premier script CloudLens extrait de `log.txt` le nom des tests qui ont échoué. Ces tests apparaissent avec le format suivant :

name `>` *description* `FAILED`

```
system.basic.WskBasicTests > Wsk Action CLI should reject delete of action that does not exist FAILED
```

Ci-dessous, le script CloudLens qui permet d'extraire ces informations de `log.txt` suivi de son résultat.

```
match {
    "(?<failure>.*) > .* FAILED"
}
stream (entry) when (entry.failure) {
    print("FAILED:", entry.failure);
}
```

```
FAILED: system.basic.WskBasicTests
FAILED: packages.UtilsTests
```

Ce script se décompose en deux parties. Pour chaque entrée : 1) une section `match` extrait des informations, 2) une section `stream` exécute un code JavaScript arbitraire.

La section `match` ajoute des champs aux entrées en utilisant les groupes de capture des expressions régulières Java. Un groupe de capture `(?<`*ident*`>`*regex*`)` définit un nouveau champs *ident* qui contient le texte correspondant à l'expression régulière *regex*. Dans le script précédent, l'expression régulière ajoute un champs `failure` qui contient le nom du test qui a échoué.

La section `stream` contient un code JavaScript arbitraire dans lequel la variable `entry` contient l'entrée courante du log. La condition `when (entry.failure)` permet de n'exécuter ce code que pour les entrées dont le champs `failure` est défini. Si aucune condition n'est précisée, une analyse de dépendance rudimentaire permet de n'exécuter une section `stream` que lorsque que les champs dont elle dépend sont définis pour l'entrée courante.

2. `https://travis-ci.org`
3. `http://openWhisk.org`

2.2. Variables d'état

Il est possible de partager des variables d'état entre différentes sections, comme le champs `failure` dans l'exemple précédent. Il est également possible de partager un état entre différentes itérations d'une même section.

Dans le fichier `log.txt`, les débuts et fin de tests sont annoncés avec le format suivant :

`Starting test` *description* `at` *date*
`Finished test` *description* `at` *date*

Le script suivant permet d'afficher la description des tests qui ont duré plus de 12 secondes.

```
var dateFormat = "# date:Date[yyyy-MM-dd' 'HH:mm:ss.SSS]";
match {
    "(?<start>Starting) test (?<desc>.*) at (?<date>.*)" + dateFormat;
    "(?<end>Finished) test (?<desc>.*) at (?<date>.*)" + dateFormat
}
var start;
stream (entry) when (entry.start) {
    start = entry.date;
}
stream (entry) when (entry.end) {
    entry.duration = entry.date - start;
    if (entry.duration > 12000) {
        print(entry.duration, "\t", entry.desc);
    }
}
```

```
13107    Actions CLI should error with a proper warning if the action exceeds its ...
14282    Wsk Activation Console should show an activation log message for hello world
15563    Wsk Activation Console should show repeated activations
31496    Util Actions should extract first n elements of an array of strings using...
```

La variable `dateFormat` permet de préciser le type du champs `date` défini par les deux expressions régulières de la section `match`. Ce format permet d'exprimer une date complète en millisecondes.

La variable `start`, partagée entre les deux sections `stream`, est mise à jour dès qu'un début de test est détecté, c'est à dire quand le champs `start` est défini. À la fin du test, quand le champs `end` est défini, on ajoute un champs `duration` contenant le temps écoulé pendant le test. Si ce temps est supérieur à 12 secondes (12000ms), on affiche la description du test `entry.desc`.

2.3. Blocs et parcours multiples

Les blocs permettent d'exécuter un code JavaScript arbitraire une seule fois au début ou à la fin du flot. Cette construction est, par exemple, utile pour initialiser un ensemble de variables, ou afficher le résultat d'une exécution. Le script suivant permet ainsi de compter le nombre de tests qui ont échoué.

```
var failed = 0;
stream (entry) when (entry.failure) {
    failed++;
}
{ print(failed, "failed tests"); }
```

```
2 failed tests
```

Par convention, on rembobine le flot d'entrées après un bloc. Il est donc possible d'itérer deux fois de suite sur un même flot. Le script suivant permet ainsi d'afficher la description des tests qui ont pris plus de 10% de la durée totale de test.

```
var totalTime = 0;
stream (entry) when (entry.duration) {
    totalTime += entry.duration;
}
{
  print("Total Time:", totalTime/1000, "seconds");
}
stream (entry) when (entry.duration) {
    entry.prop = entry.duration*100 / totalTime;
    if (entry.prop > 10) {
        print(entry.prop.toFixed(2)+"%", entry.desc)
    }
}
```

```
Warning: time.lens line 29, implicit restart of stream log!
Total Time: 226.309 seconds
13.92% Util Actions should extract first n elements of an array of strings using ...
```

Dès qu'un test est terminé, on ajoute la durée de ce test `entry.duration` au temps de test total `totalTime`. À la fin du flot, le bloc affiche la durée totale de test. On peut alors parcourir à nouveau le flot pour calculer la proportion de temps passée dans chacun des tests : `entry.duration / totalTime`.

CloudLens alerte l'utilisateur que le flot a été rembobiné. On signifie ainsi à l'utilisateur que lors de l'exécution du programme, il a fallu garder en mémoire l'ensemble des entrées qui ont été traitées pour pouvoir les reparcourir. De plus, cela prévient l'utilisateur que ce script CloudLens ne peut pas être utilisé pour faire de la surveillance (traitement de logs infini).

Pour désactiver cette alerte, il est possible de rembobiner explicitement le flot en utilisant l'instruction `restart log` juste après le bloc. Il est également possible d'utiliser cette instruction `restart log` seule, sans la précéder par un bloc.

2.4. Structure hiérarchique et lenses

Il est parfois utile de structurer hiérarchiquement les entrées. Par exemple, lorsqu'un test échoue, Travis imprime la pile d'appels. Bien que cette pile apparaisse sur plusieurs lignes consécutives, chaque ligne n'est qu'un fragment d'information. Toute cette trace peut donc être regroupée en une seule entrée. Plus généralement, il n'est pas rare qu'une entrée s'étale sur plus d'une ligne. Dans notre exemple, on supposera par convention qu'une ligne qui commence par un espace est un fragment de l'entrée précédente.

```
group in messages {
    "^[^ ]"
}
```

Une section `group` regroupe plusieurs entrées successives en une seule entrée contenant un tableau, ici `messages`. On transforme ainsi le flot d'objets JSON contenant chacun une ligne du log en un flot d'objets contenant un tableau d'objets. Une nouvelle entrée est initiée chaque fois que l'expression régulière associée à la section `group` est détectée. Ici, on démarre une nouvelle entrée lorsqu'on rencontre une ligne qui ne commence pas par un espace. Chaque entrée contient donc tous les fragments qui lui sont associés. C'est à dire toutes les lignes qui commencent par un espace qui lui succèdent.

Pour chaque test qui a échoué, on a ainsi un tableau contenant toute la pile d'appels. Cette trace contient beaucoup d'informations inutiles (bibliothèque de test, API externe, etc.). On voudrait élaguer ces traces pour ne retenir que les appels associés au code source de OpenWhisk.

Le script suivant définit une *lens*, c'est à dire une fonction qui permet d'exécuter un script CloudLens.

```
lens stackCleanup() {
    match {
        "at .*\((?<whisk>Wsk.*)\)";
    }
    stream (line) when (line.whisk) {
        print('    at', line.whisk)
    }
}
```

Exécutée sur une pile d'appels complète, la lens `stackCleanup` n'imprime que les lignes qui contiennent le motif `Wsk`. Par convention, le nom d'une classe OpenWhisk contient ce motif.

Pour chaque test qui a échoué, le script CloudLens suivant imprime une trace d'appels élaguée.

```
stream (entry) {
    if (entry.messages[0].failure !== undefined) {
        print("FAILED", entry.messages[0].failure);
        stackCleanup(entry.messages)
    }
}
```

```
FAILED system.basic.WskBasicTests
    at WskBasicTests.scala:295
    at WskBasicTests.scala:295
    at WskBasicTests.scala:295
    at WskBasicTests.scala:50
    at WskBasicTests.scala:50
FAILED packages.UtilsTests
    at WskTestHelpers.scala:111
    at WskTestHelpers.scala:65
```

Si un test a échoué, le champs `failure` de la première entrée du tableau `entry.messages` est défini. Lors de l'appel d'une lens, un argument optionnel permet de préciser le flot d'entrées. Par défaut les lenses sont exécutées sur le flot d'entrées courant, mais ce flot peut également être un tableau JavaScript arbitraire. Ici, on commence par imprimer le nom du test, puis on exécute la lens `stackCleanup` sur le tableau `entry.messages`. On n'imprime donc que les lignes de la trace d'appels qui correspondent à des classes OpenWhisk.

Exécution des lenses Il existe deux manières d'exécuter une lens. Pour illustrer ces deux options, commençons par définir deux lenses `testFailed` et `testTime` à partir des exemples des sections 2.1 et 2.2. La première option, utilisée dans l'exemple précédent, est d'appeler les fonctions correspondantes dans un bloc JavaScript.

```
{ testFailed(); testTime(); }
```

```
FAILED: system.basic.WskBasicTests
FAILED: packages.UtilsTests
13107    Actions CLI should error with a proper warning if the action exceeds its ...
14282    Wsk Activation Console should show an activation log message for hello world
15563    Wsk Activation Console should show repeated activations
31496    Util Actions should extract first n elements of an array of strings using...
```

Lors de l'exécution on exécute les deux lenses en séquence. On commence par parcourir l'ensemble du log pour extraire le nom des tests qui ont échoué. Puis on reparcourt l'ensemble du log pour afficher la description des tests qui ont duré plus de 12 secondes.

La seconde option permet d'exécuter ces deux analyses de manière concurrente en ne parcourant qu'une seule fois le log. Tout se passe alors comme si on fusionnait le corps des deux lenses en une seule analyse.

```
run testFailed()
run testTime()
```

```
13107    Actions CLI should error with a proper warning if the action exceeds its ...
FAILED: system.basic.WskBasicTests
14282    Wsk Activation Console should show an activation log message for hello world
15563    Wsk Activation Console should show repeated activations
FAILED: packages.UtilsTests
31496    Util Actions should extract first n elements of an array of strings using...
```

Pour chaque entrée, on exécute d'abord `testFailed` puis `testTime`. Comparé à l'exemple précédent les résultats des deux analyses sont donc entrelacés.

3. Sémantique

Nous présentons ici la sémantique d'un noyau de CloudLens appelé CoreCL. Les instructions CloudLens manquantes (`match`, `restart` et déclaration de fonctions) peuvent facilement être obtenues à partir de CoreCL.

3.1. CoreCL

La grammaire de CoreCL est la suivante :

$$
\begin{array}{rcl}
\mathit{script} & ::= & \emptyset \mid \mathit{section}\ \ \mathit{script} \\
\mathit{section} & ::= & \texttt{source}\ (\mathit{log}) \\
& \mid & \texttt{\{}\ \mathit{js}\ \texttt{\}} \\
& \mid & \texttt{stream}\ (x)\ \texttt{when}\ (c)\ \texttt{\{}\ \mathit{js}\ \texttt{\}} \\
& \mid & \texttt{group \{}\ \mathit{regex}\ \texttt{\}} \\
& \mid & \texttt{var}\ x\ \texttt{=}\ \mathit{js}\texttt{;} \\
& \mid & \texttt{lens}\ f(x)\ \texttt{\{}\ \mathit{script}\ \texttt{\}} \\
& \mid & \texttt{run}\ f(\mathit{js}) \\
c & ::= & x.\mathit{path} \mid x.\mathit{path}\ \ c \mid x.\mathit{path},\ c \\
\mathit{js} & ::= & \ldots \mid \texttt{CL.run}(f, \mathit{js}, \ell)
\end{array}
$$

Un *script* est une succession de *sections*. Il y a quatre catégories de sections :

- les blocs (`source` et `{ . }`),
- les sections de *pipeline* (`stream` et `group`),
- les déclarations (`var` et `lens`),
- l'invocation de lenses (`run`).

La structure de la sémantique est la suivante. Elle commence par décrire l'évaluation d'un script ($\Longrightarrow_s$, section 3.2) alternant l'exécution de *pipelines* et de blocs. Un pipeline est constitué des sections de pipeline et déclarations définies entre deux blocs. La sémantique précise ensuite l'évaluation d'un pipeline ($\Longrightarrow_p$, section 3.3). Cette évaluation traite successivement toutes les entrées d'un log. Le traitement de chacune des entrées est définie par l'évaluation d'un pas de pipeline ($\longrightarrow_p$, section 3.4), c'est-à-dire la transformation d'une entrée par chacune des sections du pipeline. Enfin, section 3.5, la sémantique définit le comportement des lenses qui peuvent être utilisées au sein d'un pipeline (`run`), ou dans un bloc (`CL.run`).

$$\frac{M, \ell \vdash p \Longrightarrow_p M', \ell'}{M, \ell, p \vdash \emptyset \Longrightarrow_s M', \ell'}$$

$$\frac{M, \ell \vdash p \Longrightarrow_p M_p, \ell_p \quad (\texttt{function () \{ } js \texttt{ \})()}_{/M_p} \Downarrow \cdot_{/M_j} \quad M_j, \ell_p, [] \vdash script \Longrightarrow_s M', \ell'}{M, \ell, p \vdash \texttt{\{ } js \texttt{ \}} \; script \Longrightarrow_s M', \ell'}$$

$$\frac{M, \ell \vdash p \Longrightarrow_p M_p, \ell_p \quad source(log)_{/M_p} \Downarrow \ell_{s/M_s} \quad M_s, \ell_s, [] \vdash script \Longrightarrow_s M', \ell'}{M, \ell, p \vdash \texttt{source (}log\texttt{)} \; script \Longrightarrow_s M', \ell'}$$

$$\frac{M, \ell, p :: \texttt{stream (}x\texttt{) when (}c\texttt{) \{ } js \texttt{ \}} \vdash script \Longrightarrow_s M', \ell'}{M, \ell, p \vdash \texttt{stream (}x\texttt{) when (}c\texttt{) \{ } js \texttt{ \}} \; script \Longrightarrow_s M', \ell'}$$

$$\frac{\gamma = fresh(M) \quad M[\gamma \leftarrow \texttt{\{\}}], \ell, p :: \texttt{group}_\gamma \texttt{ \{ } regex \texttt{ \}} \vdash script \Longrightarrow_s M', \ell'}{M, \ell, p \vdash \texttt{group \{ } regex \texttt{ \}} \; script \Longrightarrow_s M', \ell'}$$

$$\frac{\alpha = fresh(M) \quad js_{/M} \Downarrow v_{/M_v} \quad M_v[\alpha \leftarrow v], \ell, p \vdash script[x \leftarrow \alpha] \Longrightarrow_s M', \ell'}{M, \ell, p \vdash \texttt{var } x \texttt{ = } js\texttt{;} \; script \Longrightarrow_s M', \ell'}$$

$$\frac{\alpha = fresh(M) \quad M_f = M[\alpha \leftarrow \texttt{lens } f\texttt{(}x\texttt{) \{ }script_f \texttt{ \}}] \quad M_f, \ell, p \vdash script[f \leftarrow \alpha] \Longrightarrow_s M', \ell'}{M, \ell, p \vdash \texttt{lens } f\texttt{(}x\texttt{) \{ }script_f \texttt{ \}} \; script \Longrightarrow_s M', \ell'}$$

FIGURE 1 – Évaluation d'un script CoreCL.

3.2. Évaluer un script

La figure 1 présente les règles qui permettent d'évaluer un script CoreCL. Le jugement

$$M, \ell, p \vdash script \Longrightarrow_s M', \ell'$$

signifie que dans l'environnement M, avec un flot d'entrées ℓ et un pipeline en construction p, le script CoreCL *script* transforme M en M' pour capturer les éventuels effets de bord, et ℓ en ℓ'.

Formellement un pipeline p est une liste de sections `stream` et `group` :

$$\begin{array}{rcl} p & ::= & [] \mid pe :: p \\ pe & ::= & \texttt{stream (}x\texttt{) when (}c\texttt{) \{ } js \texttt{ \}} \\ & \mid & \texttt{group}_\gamma \texttt{ \{ } regex \texttt{ \}} \end{array}$$

A l'exécution les groupes sont décorés par une variable γ contenant un tableau γ`.group` qui permet d'accumuler plusieurs entrées successives.

La première règle indique qu'à la fin du script, la construction du pipeline est terminée. On exécute alors le pipeline en appliquant le jugement $M, \ell \vdash p \Longrightarrow_p M', \ell'$ défini section 3.3.

Les deux règles suivantes définissent l'évaluation des blocs. Lorsqu'on rencontre un bloc, on commence par évaluer le pipeline : $M, \ell \vdash p \Longrightarrow_p M_p, \ell_p$. Puis on évalue le bloc avec la sémantique de JavaScript : $e_{/M_p} \Downarrow v_{/M_j}$. Ce jugement signifie que dans l'environnement M_p, l'expression e s'évalue en v et l'environnement devient M_j. Enfin on évalue le reste du script en partant d'un pipeline vide [] : $M_j, \ell_p, [] \vdash script \Longrightarrow_s M', \ell'$.

Les blocs qui contiennent du code JavaScript arbitraire sont évalués à l'aide de clôtures JavaScript (`function () { ` *js* ` })()`. Les variables définies à l'intérieur d'un bloc ont donc une portée locale. Un bloc `source` permet de réinitialiser le flot d'entrées. La fonction *source* transforme un fichier en flot d'objets JSON. Chaque ligne du fichier devient un objet à un seul champs `message` qui contient le texte de la ligne (cf. figure 6 en annexe).

Les deux règles suivantes montrent le traitement des sections de pipeline. Lorsqu'on rencontre une section `stream` ou `group`, on se contente de l'ajouter au pipeline en construction p. Au passage, on

$$\frac{M, \alpha \vdash p \longrightarrow_p M_\alpha, \alpha' \quad M_\alpha, \ell \vdash p \Longrightarrow_p M', \ell'}{M, \alpha :: \ell \vdash p \Longrightarrow_p M', \alpha' :: \ell'} \qquad \frac{M, \alpha \vdash p \longrightarrow_p M_\alpha, \bot \quad M_\alpha, \ell \vdash p \Longrightarrow_p M', \ell'}{M, \alpha :: \ell \vdash p \Longrightarrow_p M', \ell'}$$

$$\frac{}{M, [] \vdash [] \Longrightarrow_p M, []}$$

$$\frac{M, [] \vdash p \Longrightarrow_p M', \ell'}{M, [] \vdash \texttt{stream (}x\texttt{) when (}c\texttt{) \{ }js\texttt{ \}} :: p \Longrightarrow_p M', \ell'} \qquad \frac{M, M(\gamma) :: [] \vdash p \Longrightarrow_p M', \ell'}{M, [] \vdash \texttt{group}_\gamma \texttt{ \{ }regex\texttt{ \}} :: p \Longrightarrow_p M', \ell'}$$

FIGURE 2 – Évaluation d'un pipeline.

associe aux sections `group` une variable fraîche γ qui servira d'accumulateur au moment de l'exécution du pipeline.

Les blocs de déclarations (variables et lens) sont évalués une seule fois (même entre deux sections `stream`) lors de la construction du pipeline. La valeur d'une variables est stockée en mémoire et le nom de la variable est substitué dans le script par l'adresse où sa valeur est stockée. Après l'évaluation d'une lens f, la variable f contient l'AST associé à la lens.

3.3. Évaluer un pipeline

La figure 2 présente les règles d'évaluation d'un pipeline. Le jugement

$$M, \ell \vdash p \Longrightarrow_p M', \ell'$$

signifie que dans l'environnement M, le pipeline p transforme M en M' et le flot d'entrées ℓ en ℓ'.

Les deux premières règles définissent l'exécution d'un pipeline p sur un flot $\alpha :: \ell$. On commence par exécuter un pas de pipeline, c'est à dire exécuter une fois toutes les sections de pipeline, sur l'entrée α puis sur le reste du flot d'entrées. L'exécution d'un pas du pipeline peut retourner une valeur ($M, \alpha \vdash p \longrightarrow_p M_e, \alpha'$) ou pas de valeur ($M, \alpha \vdash p \longrightarrow_p M_\alpha, \bot$). L'absence de valeur est introduite par la présence d'un groupe qui peut accumuler une entrée au lieu de faire un calcul.

Les trois dernières règles permettent de traiter le cas particulier d'un flot d'entrées vide. En particulier, la dernière règle montre qu'arrivé à la fin du flot, une section `group` exécute la suite du pipeline sur le contenu de l'accumulateur γ.

3.4. Évaluer un pas de pipeline

Lors de l'exécution d'un pas de pipeline, pour chaque entrée, les sections de pipeline sont exécutées en séquence. La figure 3 présente les règles d'évaluation d'un pas de pipeline. Le jugement

$$M, \alpha \vdash p \longrightarrow_p M', \alpha_\bot$$

signifie que dans l'environnement M, évaluer un pas du pipeline p sur l'entrée α retourne une nouvelle entrée $\alpha_\bot$ (potentiellement $\bot$) et transforme M en M'.

Les trois premières règles présentent le principe d'induction sur le pipeline. Les suivantes présentent l'exécution d'une section de pipeline.

Pour un pipeline $pe :: p$, si la première section de pipeline pe retourne une nouvelle entrée α', on exécute le reste du pipeline p avec cette nouvelle entrée pour obtenir $\alpha_\bot$. Si la première section retourne $\bot$, il n'y a plus d'entrée à consommer. On interrompt donc le pas d'exécution. Le résultat du pas est alors $\bot$.

Une section `stream` n'est exécutée que si la condition d'activation est vérifiée. Cette condition est spécifiée sous forme de clauses logiques. Par exemple la section

$$\frac{}{M, \alpha \vdash [] \longrightarrow_p M, \alpha}$$

$$\frac{M, \alpha \vdash pe \longrightarrow_{pe} M_\alpha, \alpha' \quad M_\alpha, \alpha' \vdash p \longrightarrow_p M', \alpha_\perp}{M, \alpha \vdash pe :: p \longrightarrow_p M', \alpha_\perp} \qquad \frac{M, \alpha \vdash pe \longrightarrow_{pe} M_\alpha, \perp}{M, \alpha \vdash pe :: p \longrightarrow_p M', \perp}$$

$$\frac{c[x \leftarrow M(\alpha)] \Downarrow_c \mathit{false}}{M, \alpha \vdash \texttt{stream (}x\texttt{) when (}c\texttt{) \{ }js\texttt{ \}} \longrightarrow_{pe} M, \alpha} \qquad \frac{c[x \leftarrow M(\alpha)] \Downarrow_c \mathit{true} \quad \texttt{(function (}x\texttt{) \{ }js\texttt{ \})(}\alpha\texttt{)}_{/M} \Downarrow \cdot_{/M'}}{M, \alpha \vdash \texttt{stream (}x\texttt{) when (}c\texttt{) \{ }js\texttt{ \}} \longrightarrow_{pe} M', \alpha}$$

$$\frac{\mathit{regex}(M(\alpha)) = \mathit{false}}{M, \alpha \vdash \texttt{group}_\gamma\ \texttt{\{ }\mathit{regex}\texttt{ \}} \longrightarrow_{pe} M[\gamma\texttt{.group} \leftarrow \gamma\texttt{.group.append}(M(\alpha))], \perp}$$

$$\frac{\mathit{regex}(M(\alpha)) = \mathit{true} \quad \alpha' = \mathit{fresh}(M) \quad M_\gamma = M[\alpha' \leftarrow M(\gamma)]}{M, \alpha \vdash \texttt{group}_\gamma\ \texttt{\{ }\mathit{regex}\texttt{ \}} \longrightarrow_{pe} M_\gamma[\gamma\texttt{.group} \leftarrow [M(\alpha)]], \alpha'}$$

FIGURE 3 – Évaluation d'un pas de pipeline.

```
stream (entry) when (entry.foo entry.bar, entry.foobar) { ... }
```

n'est exécutée que si les champs `entry.foo` et `entry.bar` sont définis, ou si le champs `entry.foobar` est défini. Les espaces indiquent un *et* logique, les virgules un *ou* logique (cf. figure 7 en annexe).

A l'instar des blocs, les sections `stream` sont évaluées sous forme de clôtures JavaScript qui prennent en argument l'entrée courante : `(function (`x`) { `js` })(`α`)`. Les variables définies à l'intérieur de la section ont donc une portée locale.

Pour une section `group`, si l'expression régulière *regex* n'est pas détectée, on renvoie $\perp$. La valeur de l'entrée courante $M(\alpha)$ est ajoutée à la fin de l'accumulateur γ`.group` (où γ est le paramètre du groupe). Si l'expression régulière est détectée, on renvoie la valeur précédente de l'accumulateur qu'on stocke dans une variable fraîche α'. L'accumulateur γ est alors réinitialisé avec un tableau qui ne contient que la valeur de l'entrée courante $[M(\alpha)]$. On n'accumule pas les entrées avant que le champs `group` soit initialisé, c'est-à-dire lorsqu'on détecte l'expression regulière pour la première fois.

3.5. Évaluer une lens

Il existe deux modes d'exécution pour les lenses.

— `run f(x)` exécute une lens `f` de manière concurrente avec le reste du script sur le flot d'entrées courant. Lors de l'évaluation, les sections de pipeline contenues dans le corps de la lens sont ajoutées au pipeline global.

— `CL.run(f, x, log)` permet d'exécuter une lens au sein d'un bloc de code JavaScript. On parcourt alors l'ensemble du flot d'entrées `log` avant d'exécuter l'instruction suivante.

Pour une lens `f`, en CloudLens, `f(x)` est un raccourci syntaxique pour invoquer `CL.run(f, x, current)` sur le flot d'entrées courant `current`. C'est cette notation qui est utilisée dans l'exemple de la section 2.4.

Considérons par exemple le script suivant :

```
lens f(x) {
    stream { print(x); }
}
run f(1)
stream { print(2); }
```

L'exécution de ce script produit une alternance `1 2 1 2...` avec un couple `1 2` par ligne du fichier log. Ce résultat est à comparer avec l'exécution du script suivant

$$\frac{M(f) = \texttt{lens } f(x) \texttt{ \{ } script \texttt{ \}} \quad js_{/M} \Downarrow v_{/M_v} \quad \alpha = fresh(M_v) \quad M' = M_v[\alpha \leftarrow v] \quad script' = script[x \leftarrow \alpha]}{instantiate(f, js)_{/M} \Downarrow script'_{/M'}}$$

$$\frac{instantiate(f, js)_{/M} \Downarrow script_{f\,/M_i} \quad M_i, \ell, p \vdash script_f\ script \Longrightarrow_s M', \ell'}{M, \ell, p \vdash \texttt{run } f(js)\ script \Longrightarrow_s M', \ell'}$$

$$\frac{instantiate(f, js)_{/M} \Downarrow script_{/M_i} \quad source(log)_{/M_i} \Downarrow \ell_{/M_r} \quad M_r, \ell, [] \vdash script \Longrightarrow_s M', \ell'}{\texttt{CL.run}(f, js, log)_{/M} \Downarrow \ell'_{/M'}}$$

FIGURE 4 – Évaluation d'une lens

```
{ CL.run(f, 1, current); }
stream { print(2); }
```

qui produit la sortie `1 1 1... 2 2 2...`, autant de `1` que de lignes dans le fichier log, puis autant de `2`.

La figure 4 présente les règles d'évaluation d'une lens. La fonction *instantiate* permet d'instancier le corps d'une lens avec la valeur de ses arguments. Cette fonction retourne une liste de sections *script* que l'on peut ensuite évaluer en utilisant les règles de la figure 1.

Pour évaluer une section `run`, on concatène le script obtenu après l'instanciation avec les sections qu'il reste à évaluer. Il suffit ensuite d'évaluer le script obtenu. Contrairement aux blocs, une section `run` ne déclenche pas l'évaluation d'un pipeline. On ne réinitialise donc pas le pipeline en construction p.

La fonction `CL.run` ne peut être utilisée que dans un bloc JavaScript. L'argument optionnel *log* permet de préciser le flot d'entrée. Par défaut le flot d'entrée est l'ensemble du fichier log. On commence par instancier le corps de la lens. Puis on évalue le script obtenu avec le flot d'entrée produit à partir de l'argument *log*. Le pipeline est initialement vide comme pour l'évaluation d'un script complet. L'appel `CL.run(f, x)` renvoie le flot d'entrées obtenu après le parcours de la lens `f`.

4. Implémentation

Nous décrivons ici l'implémentation de l'interprète CloudLens. L'interprète, écrit en Java, permet d'orchestrer l'exécution des différentes sections CloudLens. Nous utilisons la machine d'exécution Nashorn [4] qui permet d'exécuter du code JavaScript dans la machine virtuelle Java.

Nous avons choisi JavaScript comme base pour CloudLens pour deux raisons principales : 1) JavaScript est un langage très populaire et facile d'accès. On ne demande donc pas aux utilisateurs d'apprendre un nouveau langage. 2) L'existence de la machine d'exécution Nashorn a permis un prototypage rapide. Nous avons ainsi pu mettre en place et tester rapidement plusieurs fonctionnalités et choix de conception. Cela étant, les idées évoquées dans cet article ne sont pas spécifiques à Java ou JavaScript. Le langage de base pourrait aussi bien être Python, Ruby, ou OCaml.

L'interprète CloudLens suit la sémantique présentée dans la section 3. Le rôle de l'interprète CloudLens est d'orchestrer la construction et l'exécution de l'alternance de blocs et de pipelines qui rythme l'exécution d'un script.

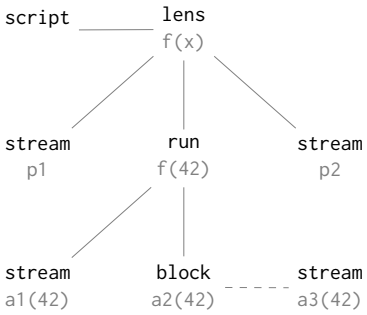
4.1. Gestion de l'environnement

Gérer correctement l'environnement, c'est à dire la portée des variables, est un problème difficile que nous avons choisi de déléguer le plus possible à Nashorn, la machine d'exécution JavaScript, en utilisant des continuations.

4. http://openjdk.java.net/projects/nashorn/

```
lens f(x) {
    stream {a1(x)}
    {a2(x)}
    stream {a3(x)}
}
stream {p1}
run f(42)
stream {p2}
```

(a) Exemple de script



(b) Arbre de continuations

FIGURE 5 – Exemple de script et l'arbre de continuations associé. L'exécution de la racine du script peuple toutes les sections jusqu'au bloc `a2(42)` (lignes continues). Puis l'exécution du bloc permet de peupler la dernière section `stream` (ligne pointillée).

Lors de l'exécution d'un script CloudLens, on crée dynamiquement un arbre de continuations, une pour chaque section du script. Les noeuds de l'arbre sont les blocs, les déclarations (variables et lenses), et les sections `run` qui permettent d'exécuter les lenses. Les feuilles sont les actions JavaScript associées aux sections de pipeline : des clôtures qui prennent en argument la valeur de l'entrée courante. La racine de l'arbre correspond au script complet.

Les sections qui suivent une déclaration de variable ou de lens doivent avoir accès à sa valeur. Le sous-arbre issu d'un noeud de déclaration contient donc toutes les sections suivantes. Les blocs interrompent l'exécution du script. C'est par exemple le moment de redéfinir la valeur de certaines variables. Le sous-arbre issu d'un bloc contient donc lui aussi toutes les sections suivantes. Enfin, le sous-arbre issu d'un noeud `run` correspond au corps de sa lens : un arbre de continuation paramétré par la valeur de ses arguments. La figure 5 donne un exemple de script et l'arbre de continuations associé.

Initialement, on ne connaît que la structure de l'arbre (donnée par l'AST du script) et la continuation associée à la racine de l'arbre. Le peuplement de l'arbre de continuations est rythmé par l'évaluation des blocs (y compris le début du programme qui peut être considéré comme un bloc vide). Lors de l'exécution d'un bloc, on commence par évaluer sa continuation qui exécute le corps du bloc et renvoie une liste de continuations associées aux sections suivantes. L'interprète CloudLens peuple alors l'arbre autant que possible et évalue récursivement les continuations des noeuds de l'arbre jusqu'au prochain bloc. Lors de l'évaluation d'un noeud `run`, on commence par évaluer la valeur des arguments avant de peupler le sous-arbre associé au corps de la lens.

Dans l'exemple de la figure 5, l'évaluation de la racine permet de peupler la déclaration de lens. La continuation de cette déclaration ajoute la valeur de `f(x)` dans l'environnement et retourne les continuations associées aux trois sections de la seconde ligne. On évalue ensuite l'argument de la section `run`, ici `42`, qui permet de commencer à peupler le sous-arbre associé à la lens `f` (troisième ligne). On arrête le peuplement au premier bloc `a2(42)`.

Par construction toutes les sections de pipeline jusqu'au prochain bloc sont maintenant associées à leurs continuations. L'interprète CloudLens parcourt alors l'arbre en profondeur et construit un pipeline qui contient toutes ces sections de pipeline. Dans l'exemple de la figure 5 ce pipeline contient ainsi `p1` et `a1(42)`.

On exécute ensuite le pipeline sur l'ensemble du fichier log, avant de recommencer les opérations précédentes à partir du bloc suivant. Dans l'exemple de la figure 5 l'exécution du bloc `a2(42)` permet de peupler la dernière section `stream`. On poursuit ensuite le parcourt en profondeur de l'arbre pour construire le second pipeline qui contient alors `a3(42)` et `p2`.

L'interprète CloudLens suit donc le schéma d'exécution présenté au début de la section 3, mais l'étape de construction du pipeline se décompose en deux parties :

1. Peuplement de l'arbre et évaluation des déclarations
2. Parcours de l'arbre et construction du pipeline

4.2. Pipelines et itérateurs

L'exécution d'un pas pipeline suit les règles de la figure 3. Chaque section de pipeline implémente une méthode `apply` qui transforme un itérateur d'objets JSON (le flot d'entrées) en un nouvel itérateur d'objets JSON.

```
Iterator<JsonObject> apply (Iterator<JsonObject> it);
```

Un itérateur Java, est une collection qu'on peut parcourir en utilisant les fonctions `hasNext`, qui renvoie vrai s'il existe un élément suivant, et `next`, qui renvoie l'élément suivant. Ces itérateurs nous permettent de manipuler les flots d'entrées de manière fonctionnelle. L'effet d'une section de pipeline est capturé dans la définition des fonctions `next` et `hasNext` de l'itérateur de sortie en fonction de celles de l'itérateur d'entrée.

Ainsi une section `stream` définit une méthode `apply` selon le schéma suivant :

```
Iterator<JsonObject> apply (Iterator<JsonObject> it) {
    return new Iterator<JsonObject> () {
        @override
        public boolean hasNext () {
            return it.hasNext()
        }

        @override
        public JsonObject next () {
            JsonObject entry = it.next()
            if evalCond (entry) {
                body.call (entry)
            }
            return entry
        }
    }
```

Une section `stream` ne modifie pas la longueur du flot d'entrée. La fonction `hasNext` n'est donc pas modifiée. L'entrée courante `entry` correspond au résultat de la fonction `next` de l'itérateur d'entrée. Si la condition d'activation de la section `stream` est vérifiée pour `entry`, on lui applique la clôture associée à la section `stream`. Cette application peut faire des effets de bords sur `entry` ou plus généralement modifier l'environnement JavaScript global. Dans tous les cas on renvoie ensuite l'entrée courante.

Les sections `group` implémentent également une méthode `apply`, mais celle-ci raccourcit la longueur du flot. Les groupes sont paramétrés par un accumulateur γ. Tant que la condition associée au groupe n'est pas vérifiée, la fonction `next` parcourt et accumule les éléments de l'itérateur d'entrée dans γ. Quand une entrée vérifie la condition, on renvoie la valeur courante de l'accumulateur et on le réinitialise avant de poursuivre le parcourt du flot d'entrées.

Finalement, un pipeline complet correspond à une chaîne d'itérateurs. Pour exécuter ce pipeline, il suffit de parcourir l'itérateur de sortie qu'on récupère en bout de chaîne. La fonction `next` de cet itérateur appelle la fonction `next` de son itérateur d'entrée, qui appelle la fonction `next` de son itérateur d'entrée, et ainsi de suite jusqu'à retrouver l'itérateur initial, c'est à dire le flot d'entrées. Ainsi, les sections de pipeline sont exécutées en séquence pour chacune des entrées du flot. C'est le comportement décrit dans la figure 2.

4.3. Réification

La fonction JavaScript `CL.run` permet d'exécuter un script CloudLens au sein d'un bloc de code JavaScript. Cette fonction *réifie* [7], l'interprète CloudLens, c'est à dire qu'elle concrétise l'interprète en le rendant accessible au développeur par le biais d'une fonction JavaScript. Tout se passe alors comme si le développeur pouvait appeler l'exécutable CloudLens au sein d'un bloc de code JavaScript pour exécuter une lens qu'il a préalablement définit. Les instructions JavaScript qui suivent sont exécutées quand l'interprète rend la main, c'est à dire à la fin de l'exécution de la lens.

L'implémentation de la réification ne pose pas de problème particulier. Il suffit d'exporter la fonction d'évaluation principale `CL.run` dans l'environnement JavaScript.

5. Travaux apparentés

L'approche traditionnelle de l'analyse de logs consiste à utiliser la commande `grep`, ou à écrire des script ad hoc (par exemple en Perl ou Python). De nombreuses techniques ont été développées pour faciliter ces analyses : apprentissage automatique pour détecter les motifs récurrents et les anomalies, techniques de visualisation efficaces pour aider l'utilisateur, langages de programmation et de requêtes spécialisés.

Il existe de nombreux travaux d'apprentissage automatique appliqué à l'analyse de logs [14, 17, 18, 19, 20]. Xu et al. appliquent leur technique d'extraction d'information [18, 19] à un large volume de logs provenant des systèmes de production de Google [20]. Leur technique consiste à extraire du code source, par analyse statique, tout ce qui peut produire des messages de log (`printf` par exemple). Ces données sont utilisées pour générer des expressions régulières qui permettent d'analyser les logs produit par le système. Ils identifient ensuite les motifs récurrents par apprentissage automatique et peuvent ainsi détecter des anomalies. Cependant, la détection d'anomalies par analyse statistique a ses limites : un message de log peut très bien être unique en son genre sans pour autant signaler un problème ou une erreur. Il est donc nécessaire de poursuivre l'analyse sur ces irrégularités. CloudLens est complémentaire de ces techniques. Nous proposons un langage de programmation spécialisé qui permet d'explorer les logs de manière plus détaillée.

La visualisation efficace de logs [1, 8, 15, 16] permet à l'utilisateur d'extraire des informations utiles de grandes quantités de données. Notre approche est, là encore, complémentaire. Nous proposons un langage expressif qui permet de traiter les logs avant ou après cette visualisation.

Il existe de nombreux langages dédié à l'analyse de logs. On peut remonter jusqu'au langage Awk [2] conçu pour extraire des informations de fichiers textuels. Awk est inclus dans la distribution standard de la plupart des systèmes d'exploitation Unix. Un script Awk est composé d'une série de couples motif/action. Une action est exécutée dès que le motif associé correspond à une ligne du fichier d'entrée. CloudLens repose sur des choix de conceptions similaires. Par ailleurs, un script CloudLens permet de structurer le log en le parcourant. Il est donc possible de traiter un flot d'entrées structuré par les passes précédentes. Ce genre de traitements itératifs est typiquement difficile à traduire en Awk/Bash.

Sawzall [9] est un langage de programmation spécialisé qui permet d'écrire des requêtes sur de larges volumes de données. L'expressivité de ce langage, construit sur une infrastructure *map/reduce*, est volontairement restreinte afin de maximiser le parallélisme. Un programme Sawzall précise les opérations à effectuer sur chacune des entrées du log (*map*) et agrège les résultats obtenus (*reduce*). Ce langage à été conçu pour calculer efficacement un ensemble de requêtes simples. En comparaison, nous avons choisi de mettre en avant l'expressivité de CloudLens.

Swatch [10] est un système automatisé de surveillance. Les entrées du log sont analysées à l'aide d'expressions régulières. Lors de l'exécution, la détection d'un motif peut déclencher des actions, par exemple exécuter un script. Logsurfer [12] est assez similaire. Un script est constitué d'un ensemble de règles : expression régulière/action. A la différence de Swatch les actions permettent en outre de créer

ou supprimer des règles. L'ensemble de règles évolue donc dynamiquement. Par ailleurs, Logsurfer permet de grouper les messages par *contextes* qui peuvent ensuite être analysés avec le même ensemble de règles. CloudLens est plus expressif. Il permet de partager des informations entre différentes règles grâce aux variables d'état et aux effets de bords effectués sur les entrées du log. Il est par exemple possible de réagir uniquement si un champs créé en amont dans un script est détecté pour l'entrée courante.

Splunk [5] est un outil commercial qui permet d'interroger et de visualiser des données textuelles, par exemple des logs. Il est possible d'extraire et traiter ces données en chaînant des requêtes écrites en SPL (*Search Processing Language*). L'utilisateur peut définir de nouveaux champs à l'aide d'expressions régulières incluses dans les requêtes. Ces requêtes peuvent également embarquer et évaluer des scripts arbitraires. CloudLens permet plus facilement de partager des informations au sein d'un script, notamment à l'aide des variables d'états.

Enfin, il existe des systèmes d'analyse et de modélisation qui s'appuient sur l'analyse de logs. Sherlog [21] est un outil qui utilise l'analyse de logs pour trouver les causes d'une erreur, par analyse statique, dans le code source de l'application. CloudLens est conçu pour l'analyse des logs provenant d'applications qui s'exécutent dans le nuage. Le code source n'est donc pas forcément disponible.

Synoptic [4, 13] permet d'inférer un graphe qui modélise un système en analysant ses logs. Dans la même veine, CSight [3] infère, lui, des automates finis communiquants. Ces approches peuvent être utilisées en complément de CloudLens pour affiner l'analyse d'une application.

6. Conclusion

Nous avons présenté CloudLens, un nouveau langage pour l'analyse de données semi-structurées qui repose sur un modèle de programmation flot de données. Nous avons implémenté deux modes d'exécution qui correspondent a deux objectifs distincts : *diagnostic* et *surveillance*.

— L'analyse *a posteriori* traite un fichier de log complet. On peut parcourir l'ensemble du flot autant de fois qu'on le souhaite pour enchaîner les analyses. Ce mode d'exécution ne permet pas de traiter des fichiers potentiellement infinis, mais peut être utile pour affiner un diagnostic.

— L'analyse *à la volée* permet de surveiller un système en cours d'exécution en consommant les sorties affichées sur la sortie standard. Le flot d'entrée est ici potentiellement infini et il n'est pas question de le reparcourir.

CloudLens a été utilisé dans le cadre du projet OpenWhisk. OpenWhisk permet à un utilisateur de déployer des service dans un nuage sans se soucier des problèmes souvent associés au déploiement d'applications (communication, sécurité, mémoire, etc.). Ce projet utilise massivement des outils d'intégration continue, comme Travis, pour contrôler chaque étape de développement. Ainsi, chaque fois qu'un développeur tente de reconstruire le projet, par exemple après avoir ajouté de nouvelles fonctionnalités, Travis génère de longs fichiers log qu'il convient d'analyser rapidement en cas d'échec. Des scripts CloudLens ont été développés pour faciliter cette analyse. Ces scripts sont exécutés par OpenWhisk, dans un nuage, dès qu'une erreur est détectée dans le processus de construction. Un robot envoie alors un message personnel au développeur pour lui préciser quels sont les tests qui ont échoué.

On peut bien sûr penser à de nombreuses autres applications : analyse de données météo, filtres d'email, systèmes de conversation automatique etc. Les flots de données semi-structurées ne manquent pas et CloudLens permet aux utilisateurs déjà familiers de JavaScript de développer rapidement leurs propres scripts d'analyse.

Un axe de recherche futur est d'intégrer des techniques d'apprentissage automatique a CloudLens. Un script CloudLens pourrait par exemple détecter des anomalies dans le flot d'entrées par rapport aux données acceptables sur lesquelles il a été entraîné. De tels scripts CloudLens seraient à la fois plus robustes et plus adaptables.

Références

[1] Jenny Abrahamson, Ivan Beschastnikh, Yuriy Brun et Michael D. Ernst : Shedding Light on Distributed System Executions. *In ICSE'14*, 2014.

[2] Alfred H. Aho, Brian W. Kernighan et Peter J. Weinberger : *The AWK Programming Language*. Addison-Wesley, 1988.

[3] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst et Arvind Krishnamurthy : Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. *In ICSE'14*, 2014.

[4] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan et Michael D. Ernst : Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. *In FSE'11*, 2011.

[5] David Carasso : *Exploring Splunk, Search Processing Language (SPL) Primer and Cookbook*. CITO Research, 2000.

[6] Louis Columbus : 2016 Big Data, Advanced Analytics & Cloud Developer Update : 5.4M Developers Now Building Cloud Apps. *Forbes*, octobre 2016. http://www.forbes.com/sites/louiscolumbus/2016/10/16/2016-big-data-advanced-analytics-cloud-developer-update-5-4m-developers-now-building-cloud-apps/.

[7] D.P. Friedman et M. Wand : Reification : Reflection without metaphysics. *In LFP'84*, 1984.

[8] Luc Girardin et Dominique Brodbeck : A Visual Approach for Monitoring Logs. *In LISA'98*, 1998.

[9] Robert Griesemer : Parallelism by Design : Data Analysis with Sawzall. *In CGO'08*, 2008.

[10] Stephen E. Hansen et E. Todd Atkins : Automated System Monitoring and Notification with Swatch. *In LISA'93*, 1993.

[11] Brian W Kernighan : *UNIX for Beginners*. Bell Laboratories, 1978.

[12] James E. Prewett : Analyzing cluster log files using Logsurfer. *In ACLC'03*, 2003.

[13] Sigurd Schneider, Ivan Beschastnikh, Slava Chernyak, Michael D. Ernst et Yuriy Brun : Synoptic : Summarizing System Logs with Refinement. *In SLAML'10*, 2010.

[14] Jon Stearley : Towards Informatic Analysis of Syslogs. *In Cluster'04*, 2004.

[15] Tetsuji Takada et Hideki Koike : Information Visualization System for Monitoring and Auditing Computer Logs. *In iV'02*, 2002.

[16] Tetsuji Takada et Hideki Koike : Mielog : A Highly Interactive Visual Log Browser Using Information Visualization and Statistical Analysis. *In LISA'02*, 2002.

[17] Ricardo Vaarandi : A Data Clustering Algorithm for Mining Patterns from Event Logs. *In IPOM'03*, 2003.

[18] Wei Xu, Ling Huang, Armando Fox, David Patterson et Michael I. Jordan : Detecting Large-Scale System Problems by Mining Console Logs. *In SOSP'09*, 2009.

[19] Wei Xu, Ling Huang, Armando Fox, David Patterson et Michael I. Jordan : Online System Problem Detection by Mining Patterns of Console Logs. *In ICDM'09*, 2009.

[20] Wei Xu, Ling Huang, Armando Fox, David Patterson et Michael I. Jordan : Experience Mining Google's Production Logs. *In SLAML'10*, 2010.

[21] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou et Shankar Pasupathy : Sherlog : error diagnosis by connecting clues from run-time logs. *In ASPLOS'10*, 2010.

A. Source et conditions d'activation

$$\frac{}{source([])_{/M} \Downarrow []_{/M}}$$

$$\frac{\alpha = fresh(M) \quad M_\alpha = M[\alpha.\texttt{message} \leftarrow v] \quad source(\ell)_{/M_\alpha} \Downarrow \ell'_{/M'}}{source(v :: \ell)_{/M} \Downarrow \alpha :: \ell'_{/M'}}$$

FIGURE 6 – Fonction source.

La figure 6 présente la sémantique de la fonction *source* qui transforme un fichier log en flot d'objets JSON. Chaque ligne du fichier devient un objet à un seul champs `message` qui contient le texte de la ligne. Appliquée sur un fichier vide, la fonction *source* renvoie un flot vide.

$$\frac{x.path \neq \texttt{undefined}}{x.path \Downarrow_c true} \qquad \frac{x.path = \texttt{undefined}}{x.path \Downarrow_c false}$$

$$\frac{x.path \Downarrow_c b_1 \quad c \Downarrow_c b_2}{x.path \;\; c \Downarrow_c b_1 \wedge b_2} \qquad \frac{x.path \Downarrow_c b_1 \quad c \Downarrow_c b_2}{x.path, \;\; c \Downarrow_c b_1 \vee b_2}$$

FIGURE 7 – Évaluation des conditions d'activation.

La figure 7 présente les règles d'évaluation des conditions d'activation des sections `stream`. Ces conditions sont écrites sous la forme de clauses logiques qui portent sur le fait qu'un champs *path* est défini, ou non, pour l'entrée courante x.

Les espaces indiquent un *et* logique, les virgules un *ou* logique.