J. Goubault–Larrecq

Un algorithme pour l'analyse de logs

Research Report LSV–02–18, Nov. 2002

Laboratoire
Spécification
et
Vérification

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

http://www.lsv.ens-cachan.fr/Publis/
Research Report LSV–02–18, Lab. Spécification et Vérification, CNRS & ENS de Cachan, France, Nov. 2002

Un algorithme pour l'analyse de logs

Jean Goubault-Larrecq

29 novembre 2002

Table des matières

1 Introduction

Le but de ce document est de préciser l'algorithme d'analyse de logs présenté en deuxième partie de l'article [Roger and Goubault-Larrecq, 2001], sous forme rigoureuse et complète. Il est à noter que l'algorithme décrit en figure 6 de cet article contient une erreur, qui se retrouve dans la preuve du théorème 4.11. Cette erreur a été corrigée dans l'outil log$\mathcal{W}$eaver le 9 juin 2001, par Jean Goubault-Larrecq au LSV. L'algorithme présenté ici, de même, corrige cette erreur.

Le but de ce document n'est en revanche pas de *prouver* cet algorithme, ni non plus de décrire précisément les structures de données qui peuvent être employées pour réaliser cet algorithme de manière efficace.

L'algorithme peut être lu à deux niveaux : un algorithme de base, et l'algorithme avec sa collection d'optimisations. Les optimisations sont effectués grâce à des considérations d'analyse statique, qui sont probablement relativement difficile à appréhender en première lecture. Ces considérations peuvent être ignorées en première lecture, et sont repérées par une barre en marge, comme pour le paragraphe que vous êtes en train de lire.

2 Expressions

On se fixe en premier un langage d'*expressions*, qui va servir à effectuer des calculs.

2.1 Ce que doit être un langage d'expressions

Le choix du langage d'expressions est relativement arbitraire, du moment qu'il s'agit d'un langage *fonctionnel*, c'est-à-dire que les calculs se font sans modifier d'état global, et *déterministe* ; en particulier, évaluer deux fois la même expression e doit produire deux fois le même résultat. Il est aussi souhaitable que l'on puisse garantir que l'exécution de toute expression e termine, si possible en un temps court. L'efficacité de l'algorithme dont il est question dans ce document dépend directement de la vitesse de calcul d'expressions stockées dans les transitions des formules (voir section 4).

On suppose que l'on s'est donné trois ensembles disjoints de variables :

- $\mathcal{V}_{\text{rigid}}$, l'ensemble des variables dites *rigides* ;
- $\mathcal{V}_{\text{flex}}$, l'ensemble des variables dites *flexibles* ;
- $\mathcal{V}_{\text{count}}$, l'ensemble des *compteurs*.

On notera $\mathcal{V} = \mathcal{V}_{\text{rigid}} \cup \mathcal{V}_{\text{flex}} \cup \mathcal{V}_{\text{count}}$ l'ensemble des *variables*, $\mathcal{V}_{\text{pat}} = \mathcal{V}_{\text{rigid}} \cup \mathcal{V}_{\text{flex}}$ l'ensemble des *variables de motifs*.

Par convention, on notera x, y, z, ..., les variables rigides ; $\$x$, $\$y$, $\$z$, ..., les variables flexibles ; $\_\$x$, $\_\$y$, $\_\$z$, ..., les compteurs. On notera d'autre part X, Y, Z, ..., des variables quelconques dans $\mathcal{V}$. Soit finalement $\_$ un symbole qui n'est pas dans $\mathcal{V}$; $\_$ représente l'absence de variable.

Un *environnement* ρ est une fonction de domaine fini, de $\mathcal{V}$ vers $\mathcal{D}$. Ici $\mathcal{D}$ est un ensemble dit de *valeurs*. Le choix de cet ensemble est aussi relativement arbitraire. Il devra typiquement au moins contenir l'ensemble String de toutes les chaînes de caractères. On supposera qu'il existe aussi un symbole $\perp$ qui n'est pas dans $\mathcal{D}$.

On notera $\mathcal{E}$ l'ensemble de tous les environnements.

Nous ne demandons qu'une chose d'un langage d'expressions : que l'on puisse définir une fonction

$$\mathcal{E}\,[\![e]\!]\,\rho$$

qui associe une valeur dans $\mathcal{D} \cup \{\perp\}$ à tout environnement ρ et toute expression e. Nous verrons un exemple d'un tel langage en section 2.2.

2.2 Un exemple de langage d'expressions

2.2.1 Valeurs

Soit Int l'ensemble des entiers machine, Time l'ensemble des dates. Typiquement, Int correspondra au type `int` de C, et Time correspondra au type `time_t`, le nombre de secondes depuis le 01 janvier 1970 à 0h00 UTC.

Les valeurs seront des chaînes, des entiers, des dates, ou bien des listes de valeurs, représentant l'accumulation des valeurs prises par les variables flexibles au cours du temps :

$$\begin{array}{rcl} \mathcal{D} & ::= & \text{String}|\text{Int}|\text{Time}|\text{Acc} \\ \text{Acc} & ::= & [\mathcal{D}]^+.\mathcal{D} \end{array}$$

où $[A]^+$ dénote une suite quelconque non vide d'objets de type A ; de même, $[A]^+$ dénote une suite quelconque, possiblement vide, d'objets de type A.

Une valeur de type Acc est appelée un *accumulateur*. On notera parfois $[a_n, \ldots, a_1.a_0]$ un accumulateur a, $n \geq 1$. Sa *longueur* est alors $|a| = n + 1$. Les fonctions $\texttt{hd} : \text{Acc} \to \mathcal{D}$ et $\texttt{tl} : \text{Acc} \to \mathcal{D}$ sont définies par $\texttt{hd}[a_n, \ldots, a_1.a_0] = a_n$, $\texttt{tl}[a_n, \ldots, a_1.a_0] = [a_{n-1}, \ldots, a_1.a_0]$ si $n \geq 2$, $\texttt{tl}[a_1.a_0] = a_0$.

Dans la suite, on notera $\mathbb{B} = \{\mathbf{0}, \mathbf{1}\}$ l'ensemble des *booléens* ; $\mathbf{0}$ est le *faux*, $\mathbf{1}$ est le *vrai*.

2.2.2 Fonctions auxiliaires

On pose :

- $[]$ dénote l'environnement vide (de domaine vide).
- $[X \mapsto v]$ dénote l'environnement de domaine $\{X\}$, qui à X associe v.
- Pour tout environnement ρ, $\text{dom}\rho$ est le domaine de ρ.
- Pour tout environnement ρ, toute variable X, toute valeur $v \in \mathcal{D}$, $\rho[X \mapsto v]$ est l'environnement de domaine $\text{dom}\rho \cup \{X\}$, et qui associe v à X, et $\rho(Y)$ à tout $Y \in \text{dom}\rho \setminus \{X\}$.

- Pour tout environnement ρ et tout sous-ensemble V de variables, $\rho_{|V}$ est la *restriction* de ρ à V, c'est-à-dire l'environnement de domaine $\mathrm{dom}\rho \cap V$ et qui à tout X dans ce domaine associe $\rho(X)$.
- Pour tout ensemble A, $[A]_\perp$ désigne l'union disjointe de A et de $\{\perp\}$.

Le langage d'expressions que nous allons définir est aussi laxiste que Perl [Christiansen et al., 2000]. On peut en particulier convertir tout type de donnée vers n'importe quel autre. Pour ceci, on définit les fonctions :

- $\mathtt{int\_of} : \mathcal{D} \to \mathrm{Int}$:
  - pour tout $n \in \mathrm{Int}$, $\mathtt{int\_of}(n) = n$;
  - pour tout $t \in \mathrm{Time}$, $\mathtt{int\_of}(t)$ est le nombre de secondes depuis le 01 janvier 1970, 0h00 UTC, spécifié par t ;
  - pour tout $s \in \mathrm{String}$, $\mathtt{int\_of}(s)$ est la valeur entière de s, telle que calculée par `atoi` (3) ;
  - pour tout $a \in \mathrm{Acc}$, $\mathtt{int\_of}(a) = \mathtt{int\_of}(\mathtt{hd}(a))$.
- $\mathtt{time\_of} : \mathcal{D} \to \mathrm{Time}$:
  - pour tout $n \in \mathrm{Int}$, $\mathtt{time\_of}(n)$ est la date située exactement n secondes après le 01 janvier 1970, 0h00 UTC ;
  - pour tout $t \in \mathrm{Time}$, $\mathtt{time\_of}(t) = t$;
  - pour tout $s \in \mathrm{String}$, $\mathtt{time\_of}(s)$ est la date telle que décrite par s au format de `ctime` (3). A titre d'exemple, la chaîne "`Wed Jun 30 21:49:08 1993\n`" spécifie le mercredi 30 juin 1993 à 21 heures 49 minutes et 8 secondes. Les champs jour de la semaine, mois, jour du mois, heure et année peuvent être mis dans n'importe quel ordre, et le nombre d'espaces et tabulations n'est pas important. Un champ absent est considéré comme valant 0. Les jours de la semaine sont `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, et `Sat`. Les mois sont `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, et `Dec`.
  - pour tout $a \in \mathrm{Acc}$, $\mathtt{time\_of}(a) = \mathtt{time\_of}(\mathtt{hd}(a))$.
- $\mathtt{string\_of} : \mathcal{D} \to \mathrm{String}$:
  - pour tout $n \in \mathrm{Int}$, $\mathtt{string\_of}(n)$ est la chaîne représentant n, au sens de `itoa` (3) ;
  - pour tout $t \in \mathrm{Time}$, $\mathtt{string\_of}(t)$ est la chaîne représentant t, au sens de `ctime` (3) ;
  - pour tout $s \in \mathrm{String}$, $\mathtt{string\_of}(s) = s$;
  - pour tout $a \in \mathrm{Acc}$, $\mathtt{string\_of}(a) = \mathtt{string\_of}(\mathtt{hd}(a))$.
- $\mathtt{current} : \mathcal{D} \to \mathcal{D}$:
  - pour tout $a \in \mathrm{Acc}$, $\mathtt{current}(a) = \mathtt{current}(\mathtt{hd}(a))$;
  - pour tout $v \notin \mathrm{Acc}$, $\mathtt{current}(v) = v$.
- $\mathtt{bool} : \mathcal{D} \to \mathbb{B}$ convertit une valeur en un booléen : $\mathtt{bool}(v)$ est $\mathbf{0}$ si $\mathtt{int\_of}(v) = 0$, $\mathbf{1}$ sinon.
- $\leftrightharpoons : \mathcal{D} \times \mathcal{D} \to \mathbb{B}$ teste l'égalité de deux valeurs, modulo conversion. On note $v_1 \leftrightharpoons v_2$ plutôt que $\leftrightharpoons (v_1, v_2)$.
  - si $v_1 \in \mathrm{Acc}$, alors $v_1 \leftrightharpoons v_2$ si et seulement si $\mathtt{hd}(v_1) \leftrightharpoons v_2$;
  - si $v_2 \in \mathrm{Acc}$, alors $v_1 \leftrightharpoons v_2$ si et seulement si $v_1 \leftrightharpoons \mathtt{hd}(v_2)$;
  - sinon, si $v_1 \in \mathrm{String}$ ou $v_2 \in \mathrm{String}$, alors $v_1 \leftrightharpoons v_2$ si et seulement si $\mathtt{string\_of}(v_1) = \mathtt{string\_of}(v_2)$;
  - sinon, si $v_1 \in \mathrm{Time}$ ou $v_2 \in \mathrm{Time}$, alors $v_1 \leftrightharpoons v_2$ si et seulement si $\mathtt{time\_of}(v_1) =$

time_of(v_2) ;
  - sinon, $v_1, v_2 \in$ Int, et $v_1 \leftrightharpoons v_2$ si et seulement si $v_1 = v_2$.
- set_of : $\mathcal{D} \to \mathbb{P}(\mathcal{D})$:
  - si $a = [a_n, \ldots, a_1.a_0] \in$ Acc, alors set_of(a) est l'ensemble $\{a_0, a_1, \ldots, a_n\}$; notamment, comme il s'agit d'un ensemble, les doublons sont éliminés (au sens de l'égalité usuelle, pas de $\leftrightharpoons$) ;
  - si $v \notin$ Acc, alors set_of(v) $= \{v\}$.
- sum_of : $\mathcal{D} \to$ Int effectue une somme sans débordement d'entiers positifs ou nuls :
  - si $a = [a_n, \ldots, a_1.a_0] \in$ Acc, alors sum_of(a) $= \min(\sum_{i=0}^{n} \max(a_i, 0), max\_int)$, où max_int est le plus grand entier machine positif ;
  - si $v \notin$ Acc, alors sum_of(v) $= \min(v, max\_int)$.

  Si cette définition peut sembler bizarre, c'est qu'elle a été conçue pour garantir une propriété de monotonie : sum_of($v' \bullet v$) $\geq$ sum_of(v) pour toutes valeurs v et v' ($\bullet$ est définie ci-dessous). Ceci servira dans les optimisations de l'algorithme d'analyse de logs fondée sur la monotonie (voir en particulier la définition de monotony($\mathcal{M}$, sum(e), section 2.2.5).
- $\bullet$: $\mathcal{D} \times \mathcal{D} \to \mathcal{D}$ concatène deux accumulateurs :
  - si $a = [a_n, \ldots, a_1.a_0] \in$ Acc, alors $v \bullet a =$ [current(v), $a_n, \ldots, a_1.a_0$] ;
  - si $v' \notin$ Acc, alors $v \bullet v' =$ [current(v).v'].
- $\oplus$: $\mathcal{E} \times \mathcal{E} \to [\mathcal{E}]_\perp$M effectue l'union de deux environnements :
  - s'il existe $x \in \mathcal{V}_{\text{rigid}} \cap \text{dom}\rho \cap \text{dom}\rho'$ tel que $\rho(x) \not\leftrightharpoons \rho'(x)$, alors $\rho \oplus \rho' = \perp$ (conflit de valeurs de la variable rigide x) ;
  - sinon, $\rho \oplus \rho'$ est l'environnement de domaine $\text{dom}\rho \cup \text{dom}\rho'$, qui à toute variable $X \in \text{dom}\rho \setminus \text{dom}\rho'$ associe $\rho(X)$, à toute variable $X \in \text{dom}\rho' \setminus \text{dom}\rho$ associe $\rho'(X)$, à toute variable rigide $x \in \text{dom}\rho \cap \text{dom}\rho'$ associe $\rho'(x)$ (noter que $\rho(x) \leftrightharpoons \rho'(x)$), à toute variable flexible $\$x \in \text{dom}\rho \cap \text{dom}\rho'$ associe $\rho(\$x) \bullet \rho'(\$x)$, et à tout compteur $\_\$x \in \text{dom}\rho \cap \text{dom}\rho'$ associe $\rho'(\_\$x)$.

    En réalité, la dernière clause portant sur les compteurs est inutile : on n'utilisera jamais $\oplus$ que sur des environnements dont le domaine est inclus dans $\mathcal{V}_{\text{pat}} = \mathcal{V}_{\text{rigid}} \cup \mathcal{V}_{\text{flex}}$.
- $\dagger$: $\mathcal{E} \times \mathcal{E} \to \mathcal{E}$ est une fonction similaire : $\rho \dagger \rho'$ est l'environnement de domaine $\text{dom}\rho \cup \text{dom}\rho'$ qui à tout $X \in \text{dom}\rho'$ associe $\rho'(X)$ et à tout $X \in \text{dom}\rho \setminus \text{dom}\rho'$ associe $\rho(X)$.

Enfin, on suppose que l'on a un ensemble Regexp, non spécifié, dit d'*expressions régulières*, et une fonction regmatch : Regexp $\to$ String $\to$ [String $\to$ String]$_\perp$. On notera typiquement re un élément de Regexp.

Typiquement, Regexp est l'ensemble des expressions régulières reconnues par l'outil [Spencer, 1986], et regmatch(re)(s) retourne $\perp$ si la chaîne s n'appartient pas au langage $L(re)$ défini par re. Si s n'appartient pas à $L(re)$, on considère que regmatch(re)(s) ne retourne pas simplement un booléen "vrai", mais une fonction qui permet de substituer des sous-parties de s repérées par des constructions `\\1`, `\\2`, ..., dans l'expression régulière re. La substitution de ces constructions dans la chaîne s' est construite par l'appel regmatch(re)(s)(s'). Par exemple, si re est `a([bc]*)a`, et s est `bcabbcacc`, alors regmatch(re)(s)(`d\\1e`) = `dbbce`.

De façon concrète, regmatch(re) est réalisée à l'aide de la fonction `regcomp` de [Spencer, 1986] ; si $comp$ = regmatch(re), l'appel $comp(s)$ est effectué à l'aide de la fonction `regexec` (op.cit.) ; si $subst = comp(s)$ n'est pas $\perp$, $subst(s')$ est le résultat de l'appel de `regsub` (op.cit.)

2.2.3 Syntaxe

Les expressions obéissent à la syntaxe décrite en figure 1.

2.2.4 Sémantique

La sémantique $\mathcal{E}\llbracket e \rrbracket \rho$ des expressions e dans les environnements ρ est définie par récurrence structurelle sur e par :

- $\mathcal{E}\llbracket X \rrbracket \rho = \rho(X)$ si X est dans le domaine de ρ, 0 sinon.
- $\mathcal{E}\llbracket s \rrbracket \rho = s$ pour tout $s \in$ String.
- $\mathcal{E}\llbracket i \rrbracket \rho = i$ pour tout $i \in$ Int.
- $\mathcal{E}\llbracket \mathtt{let}([x_1, e_1, \ldots, x_n, e_n], e) \rrbracket \rho = \mathcal{E}\llbracket e \rrbracket \rho_n$, où $\rho_0 = \rho$ et pour tout i, $1 \leq i \leq n$, $\rho_i = \rho_{i-1}[x_i \mapsto \mathcal{E}\llbracket e_i \rrbracket \rho_{i-1}]$.
- $\mathcal{E}\llbracket \mathtt{switch}(e, [re_1, subst_1, e_1, \ldots, re_n, subst_n, e_n], e') \rrbracket \rho = v$, où :
  - $s = \mathtt{string\_of}(\mathcal{E}\llbracket e \rrbracket \rho)$;
  - soit $\mathtt{regmatch}(re_i)(s) = \perp$ pour tout i, $1 \leq i \leq n$, et $v = \mathcal{E}\llbracket e' \rrbracket \rho$;
    soit il existe un i, $1 \leq i \leq n$, minimal tel que $sub = \mathtt{regmatch}(re_i)(s) \neq \perp$, et $v = \mathcal{E}\llbracket e_i \rrbracket \rho_k$, où :
    - $subst_i$ est de la forme $[X_1, s_1, \ldots, X_k, s_k]$;
    - $\rho_0 = \rho$;
    - $\rho_j = \rho_{j-1}[X_j \mapsto sub(s_j)]$ pour tout j, $1 \leq j \leq k$.
- $\mathcal{E}\llbracket e_1 \&\& e_2 \rrbracket \rho$ vaut 0 si $\mathtt{bool}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$ ou $\mathtt{bool}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ est faux, $\mathcal{E}\llbracket e_2 \rrbracket \rho$ sinon.
- $\mathcal{E}\llbracket e_1 || e_2 \rrbracket \rho$ vaut $\mathcal{E}\llbracket e_1 \rrbracket \rho$ si $\mathtt{bool}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$ est vrai, $\mathcal{E}\llbracket e_2 \rrbracket \rho$ sino.
- $\mathcal{E}\llbracket !e \rrbracket \rho$ vaut 0 si $\mathtt{bool}(\mathcal{E}\llbracket e \rrbracket \rho)$ est vrai, 1 sinon.
- $\mathcal{E}\llbracket \mathtt{if}(e_1, e_2, e_3) \rrbracket \rho$ vaut $\mathcal{E}\llbracket e_2 \rrbracket \rho$ si $\mathtt{bool}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$ est vrai, $\mathcal{E}\llbracket e_3 \rrbracket \rho$ sinon.
- $\mathcal{E}\llbracket e_1 {==} e_2 \rrbracket \rho$ vaut 1 si $\mathcal{E}\llbracket e_1 \rrbracket \rho \leftrightharpoons \mathcal{E}\llbracket e_2 \rrbracket \rho$, 0 sinon.
- $\mathcal{E}\llbracket e_1 {!=} e_2 \rrbracket \rho$ vaut 0 si $\mathcal{E}\llbracket e_1 \rrbracket \rho \leftrightharpoons \mathcal{E}\llbracket e_2 \rrbracket \rho$, 1 sinon.
- $\mathcal{E}\llbracket e_1 {<} e_2 \rrbracket \rho$ vaut 1 si $\mathtt{int\_of}(\mathcal{E}\llbracket e_1 \rrbracket \rho) < \mathtt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$, 0 sinon.
- $\mathcal{E}\llbracket e_1 {<=} e_2 \rrbracket \rho$ vaut 1 si $\mathtt{int\_of}(\mathcal{E}\llbracket e_1 \rrbracket \rho) \leq \mathtt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$, 0 sinon.
- $\mathcal{E}\llbracket e_1 {>} e_2 \rrbracket \rho$ vaut 1 si $\mathtt{int\_of}(\mathcal{E}\llbracket e_1 \rrbracket \rho) > \mathtt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$, 0 sinon.
- $\mathcal{E}\llbracket e_1 {>=} e_2 \rrbracket \rho$ vaut 1 si $\mathtt{int\_of}(\mathcal{E}\llbracket e_1 \rrbracket \rho) \geq \mathtt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$, 0 sinon.
- $\mathcal{E}\llbracket e_1 + e_2 \rrbracket \rho$ vaut :
  - $v_1 + \mathtt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Int ;
  - $v_1 + \mathtt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Time ;
  - $v_1 \hat{\ } \mathtt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ String

  où $v_1 = \mathtt{current}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$. On note $+$ l'addition des entiers machine (y compris les `time_t`), et ˆ la concaténation de chaînes de caractères.
- $\mathcal{E}\llbracket e_1 - e_2 \rrbracket \rho$ vaut :
  - $v_1 - \mathtt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Int ;
  - $v_1 - \mathtt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Time ;
  - $\mathtt{int\_of}(v_1) - \mathtt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ String

e	$::=$	X	$(X \in \mathcal{V})$ variable
	$\mid$	s	$(s \in \text{String})$ chaîne constante
	$\mid$	i	$(i \in \text{Int})$ entier constant
	$\mid$	$\texttt{let}([x, e]^*, e)$	abréviation
	$\mid$	$\texttt{switch}(e, [re, subst, e]^*, e)$	$(re \in \text{Regexp})$ matching d'expression régulière
	$\mid$	$e\&\&e$	et logique
	$\mid$	$e\vert\vert e$	ou logique
	$\mid$	$!e$	négation
	$\mid$	$\texttt{if}(e, e, e)$	conditionnelle
	$\mid$	$e{==}e$	test d'égalité
	$\mid$	$e! = e$	test de différence
	$\mid$	$e < e$	inférieur strictement
	$\mid$	$e <= e$	inférieur ou égal
	$\mid$	$e > e$	supérieur strictement
	$\mid$	$e >= e$	supérieur ou égal
	$\mid$	$e + e$	addition, concaténation
	$\mid$	$e - e$	soustraction
	$\mid$	$e * e$	multiplication
	$\mid$	e/e	division
	$\mid$	$e\%e$	reste
	$\mid$	$-e$	opposé
	$\mid$	$\texttt{len}(e)$	longueur de chaîne
	$\mid$	$\texttt{substr}(e, e, e)$	extraction de sous-chaîne
	$\mid$	$\texttt{int}(e)$	conversion vers entier
	$\mid$	$\texttt{time}(e)$	conversion vers date
	$\mid$	$\texttt{string}(e)$	conversion vers chaîne
	$\mid$	$\#e$	nombre d'éléments dans accumulateur
	$\mid$	$\texttt{in}(e, e)$	test d'appartenance à un accumulateur
	$\mid$	$`e$	valeurs précédentes dans accumulateur
	$\mid$	$\texttt{new}(e)$	test de nouveauté par rapport à un accumulateur
	$\mid$	$\texttt{sum}(e)$	somme d'accumulateur
	$\mid$	$\vert e \vert$	longueur d'accumulateur
$subst$	$::=$	$[var, \text{String}]^*$	$(var \in \mathcal{V} \cup \{\_\})$ substitution

FIG. 1 – Syntaxe des expressions

où $v_1 = \texttt{current}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$. On note $-$ la soustraction des entiers machine (y compris les `time_t`).

- $\mathcal{E}\llbracket e_1 * e_2 \rrbracket \rho$ vaut :
  - $v_1 \times \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Int ;
  - $v_1 \times \texttt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Time ;
  - $\texttt{int\_of}(v_1) \times \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ String

  où $v_1 = \texttt{current}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$. On note $\times$ la multiplication des entiers machine (y compris les `time_t`).
- $\mathcal{E}\llbracket e_1 / e_2 \rrbracket \rho$ vaut :
  - $v_1 / \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Int ;
  - $v_1 / \texttt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Time ;
  - $\texttt{int\_of}(v_1) / \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ String

  où $v_1 = \texttt{current}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$. On note $/$ la division des entiers machine (y compris les `time_t`). Par ceci on entend le quotient de la division entière. Les divisions par zéro retournent un résultat indéfini.
- $\mathcal{E}\llbracket e_1 \% e_2 \rrbracket \rho$ vaut :
  - $v_1 \bmod \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Int ;
  - $v_1 \bmod \texttt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Time ;
  - $\texttt{int\_of}(v_1) \bmod \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ String

  où $v_1 = \texttt{current}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$. On note mod la reste de la division entière des entiers machine (y compris les `time_t`). Les restes de divisions par zéro retournent un résultat indéfini.
- $\mathcal{E}\llbracket -e \rrbracket \rho$ vaut $-v$ si $v \in$ Time, $-\texttt{int\_of}(v)$ si $v \in$ Int $\cup$ String, où $v = \texttt{current}(\mathcal{E}\llbracket e \rrbracket \rho)$.
- $\mathcal{E}\llbracket \texttt{len}(e) \rrbracket \rho$ est la longueur de la chaîne $\texttt{string\_of}(\mathcal{E}\llbracket e \rrbracket \rho)$.
- $\mathcal{E}\llbracket \texttt{substr}(e_1, e_2, e_3) \rrbracket \rho$ est la chaîne vide si $i > j$, et composée des caractères $s_i, s_{i+1}, \ldots, s_{j-1}$ sinon, où :
  - $s = \texttt{string\_of}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$;
  - n est la longueur de s, et s est composée des caractères $s_0, s_1, \ldots, s_{n-1}$;
  - $i = \max(\texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho), 0)$;
  - $j = \min(\texttt{int\_of}(\mathcal{E}\llbracket e_3 \rrbracket \rho), n)$.
- $\mathcal{E}\llbracket \texttt{int}(e) \rrbracket \rho = \texttt{int\_of}(\mathcal{E}\llbracket e \rrbracket \rho)$.
- $\mathcal{E}\llbracket \texttt{time}(e) \rrbracket \rho = \texttt{time\_of}(\mathcal{E}\llbracket e \rrbracket \rho)$.
- $\mathcal{E}\llbracket \texttt{string}(e) \rrbracket \rho = \texttt{string\_of}(\mathcal{E}\llbracket e \rrbracket \rho)$.
- $\mathcal{E}\llbracket \# e \rrbracket \rho$ est le cardinal de l'ensemble $\texttt{set\_of}(\mathcal{E}\llbracket e \rrbracket \rho)$.
- $\mathcal{E}\llbracket \texttt{in}(e_1, e_2) \rrbracket \rho$ vaut 1 si $\texttt{current}(\mathcal{E}\llbracket e_1 \rrbracket \rho) \in \texttt{set\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$, 0 sinon. Noter que la relation d'appartenance est fondée sur l'égalité $=$, et non sur la relation $\leftrightharpoons$.
- $\mathcal{E}\llbracket \text{`} e \rrbracket \rho$ vaut $\texttt{tl}(v)$ si $v \in$ Acc, 0 sinon, où $v = \mathcal{E}\llbracket e \rrbracket \rho$.
- $\mathcal{E}\llbracket \texttt{new}(e) \rrbracket \rho$ vaut 0 si $\texttt{current}(\mathcal{E}\llbracket e \rrbracket \rho) \in \texttt{set\_of}(\texttt{tl}(\mathcal{E}\llbracket e \rrbracket \rho))$, 1 sinon ; autrement dit, $\texttt{new}(e)$ retourne vrai si et seulement si la valeur courante de e est *fraîche*, c'est-à-dire pas dans la liste des anciennes valeurs de e.
- $\mathcal{E}\llbracket \texttt{sum}(e) \rrbracket \rho = \texttt{sum\_of}(\mathcal{E}\llbracket e \rrbracket \rho)$.
- $\mathcal{E}\llbracket |e| \rrbracket \rho$ vaut $n + 1$, si $\mathcal{E}\llbracket e \rrbracket \rho = [a_n, \ldots, a_1.a_0] \in$ Acc, 1 sinon.

Il est à noter que $\mathcal{E}\llbracket e \rrbracket \rho$ ne retourne jamais $\perp$.

2.2.5 Fonctions d'analyse statique

Variables libres. On définit d'abord l'ensemble $\mathrm{fv}(e)$ des *variables libres* dans e :

- $\mathrm{fv}(X) = \{X\}$;
- $\mathrm{fv}(s) = \mathrm{fv}(i) = \emptyset$;
- $\mathrm{fv}(\mathtt{let}([x_1, e_1, \ldots, x_n, e_n], e) = (((\mathrm{fv}(e) \setminus \{x_n\}) \cup \mathrm{fv}(e_n)) \setminus \ldots \setminus \{x_1\}) \cup \mathrm{fv}(e_1)$;
- $\mathrm{fv}(\mathtt{switch}(e, [re_1, subst_1, e_1, \ldots, re_n, subst_n, e_n], e')) = \mathrm{fv}(e) \cup \bigcup_{i=1}^{n}(\mathrm{fv}(e_i) - sbv(subst_i)) \cup \mathrm{fv}(e')$, où $sbv(subst) = \{x_1, \ldots, x_k\}$ pour tout $subst = [x_1, s_1, \ldots, x_k, s_k]$;
- $\mathrm{fv}(e_1 \&\& e_2) = \mathrm{fv}(e_1 || e_2) = \mathrm{fv}(e_1 {==} e_2) = \mathrm{fv}(e_1 ! {=} e_2) = \mathrm{fv}(e_1 {<} e_2) = \mathrm{fv}(e_1 {<=} e_2) = \mathrm{fv}(e_1 {>} e_2) = \mathrm{fv}(e_1 {>=} e_2) = \mathrm{fv}(e_1 + e_2) = \mathrm{fv}(e_1 - e_2) = \mathrm{fv}(e_1 * e_2) = \mathrm{fv}(e_1 / e_2) = \mathrm{fv}(e_1 \% e_2) = \mathrm{fv}(\mathtt{in}(e_1, e_2)) = \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$;
- $\mathrm{fv}(!e) = \mathrm{fv}(-e) = \mathrm{fv}(\mathtt{len}(e)) = \mathrm{fv}(\mathtt{int}(e)) = \mathrm{fv}(\mathtt{time}(e)) = \mathrm{fv}(\mathtt{string}(e)) = \mathrm{fv}(\#e) = \mathrm{fv}(`e) = \mathrm{fv}(\mathtt{new}(e)) = \mathrm{fv}(\mathtt{sum}(e)) = \mathrm{fv}(|e|) = \mathrm{fv}(e)$;
- $\mathrm{fv}(\mathtt{if}(e_1, e_2, e_3)) = \mathrm{fv}(\mathtt{substr}(e_1, e_2, e_3)) = \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup \mathrm{fv}(e_3)$.

Il est clair que $\mathcal{E}\llbracket e \rrbracket \rho$ ne dépend que des variables libres de e, plus précisément : si ρ et ρ' sont deux environnements tels que $\rho(x) = \rho'(x)$ pour tout $x \in \mathrm{fv}(e)$, alors $\mathcal{E}\llbracket e \rrbracket \rho = \mathcal{E}\llbracket e \rrbracket \rho'$.

Monotonie des expressions. Soient $\mathtt{MONO}$ et $\mathtt{ANTI}$ deux constantes. On définit une fonction $\mathtt{monotony}(\mathcal{M}, e)$ qui à toute expression e associe un sous-ensemble de $\{\mathtt{MONO}, \mathtt{ANTI}\}$. Si ce sous-ensemble contient $\mathtt{MONO}$, alors e est *monotone*, c'est-à-dire que la valeur entière de e ne peut que croître au fur et à mesure où on avance dans le log. Si ce sous-ensemble contient $\mathtt{ANTI}$, alors e est *antitone*, et sa valeur ne peut que décroître. En particulier, si l'ensemble retourné est $\{\mathtt{MONO}, \mathtt{ANTI}\}$, alors e est *constante*. S'il est vide, alors on ne sait rien sur l'évolution de e.

Ici $\mathcal{M}$ est un *environnement de monotonie* qui à un certain nombre de variables associe un sous-ensemble de $\{\mathtt{MONO}, \mathtt{ANTI}\}$, et $\mathtt{monotony}(\mathcal{M}, e)$ calcule plus précisément si e est croissante, décroissante ou constante sous les hypothèses que toutes les variables X de $\mathrm{dom}\mathcal{M}$ croissent ou décroissent comme spécifié par $\mathcal{M}(X)$.

- $\mathtt{monotony}(\mathcal{M}, \_\$x) = \{\mathtt{MONO}\}$ (les compteurs ne font que croître) ;
- $\mathtt{monotony}(\mathcal{M}, X) = \mathcal{M}(X)$ si $X \in \mathcal{V}_{\mathrm{pat}}$ et $X \in \mathrm{dom}\mathcal{M}$;
- $\mathtt{monotony}(\mathcal{M}, X) = \emptyset$ sinon ;
- $\mathtt{monotony}(\mathcal{M}, s) = \{\mathtt{MONO}, \mathtt{ANTI}\}$;
- $\mathtt{monotony}(\mathcal{M}, i) = \{\mathtt{MONO}, \mathtt{ANTI}\}$;
- $\mathtt{monotony}(\mathcal{M}, \mathtt{let}([x_1, e_1, \ldots, x_n, e_n], e) = \mathtt{monotony}(\mathcal{M}_n, e)$, où $\mathcal{M}_0 = \mathcal{M}$ et pout tout i, $1 \leq i \leq n$, $\mathcal{M}_i = \mathcal{M}_{i-1}[x_i \mapsto \mathtt{monotony}(\mathcal{M}_{i-1}, e_i)]$.
- $\mathtt{monotony}(\mathcal{M}, \mathtt{switch}(e, [re_1, subst_1, e_1, \ldots, re_n, subst_n, e_n], e'))$ vaut $\emptyset$ si $\mathtt{monotony}(\mathcal{M}, e) \neq \{\mathtt{MONO}, \mathtt{ANTI}\}$; sinon, il vaut

$$\bigcap_{i=1}^{n} \mathtt{monotony}(\mathcal{M} \dagger [X_{i1} \mapsto \emptyset, \ldots, X_{ik_i} \mapsto \emptyset], e_i) \cap \mathtt{monotony}(\mathcal{M}, e')$$

  où $subst_i = [X_{i1}, s_{i1}, \ldots, X_{ik_i}, s_{ik_i}]$, $1 \leq i \leq n$. (Une intersection vide est supposée égale à $\{\mathtt{MONO}, \mathtt{ANTI}\}$.)
- $\mathtt{monotony}(\mathcal{M}, e_1 \&\& e_2) = \mathtt{monotony}(\mathcal{M}, e_1 || e_2) = \mathtt{monotony}(\mathcal{M}, e_1 + e_2) = \mathtt{monotony}(\mathcal{M}, e_1) \cap \mathtt{monotony}(\mathcal{M}, e_2)$.

- $\texttt{monotony}(\mathcal{M}, \texttt{if}(e_1, e_2, e_3)) = \texttt{monotony}(\mathcal{M}, e_2) \cap \texttt{monotony}(\mathcal{M}, e_3)$.
- $\texttt{monotony}(\mathcal{M}, e_1{=}{=}e_2) = \texttt{monotony}(\mathcal{M}, e_1!{=}e_2) = \texttt{monotony}(\mathcal{M}, e_1\%e_2) = \{\texttt{MONO}, \texttt{ANTI}\}$ si $\texttt{monotony}(\mathcal{M}, e_1) = \texttt{monotony}(\mathcal{M}, e_2) = \{\texttt{MONO}, \texttt{ANTI}\}$, $\emptyset$ sinon.
- $\texttt{monotony}(\mathcal{M}, e_1{<}e_2) = \texttt{monotony}(\mathcal{M}, e_1{<=}e_2) = rev(\texttt{monotony}(\mathcal{M}, e_1)) \cap \texttt{monotony}(\mathcal{M}, e_2)$, où rev est la fonction qui échange MONO et ANTI ($rev(\emptyset) = \emptyset$, $rev(\{\texttt{MONO}\}) = \{\texttt{ANTI}\}$, $rev(\{\texttt{ANTI}\}) = \{\texttt{MONO}\}$, $rev(\{\texttt{MONO}, \texttt{ANTI}\}) = \{\texttt{ANTI}, \texttt{MONO}\}$).
- $\texttt{monotony}(\mathcal{M}, e_1{>}e_2) = \texttt{monotony}(\mathcal{M}, e_1{>=}e_2) = \texttt{monotony}(\mathcal{M}, e_1 - e_2) = \texttt{monotony}(\mathcal{M}, e_1) \cap rev(\texttt{monotony}(\mathcal{M}, e_2))$.
- $\texttt{monotony}(\mathcal{M}, e_1 * e_2)$ vaut $\{\texttt{MONO}, \texttt{ANTI}\}$ si $sign(e_i) = \{0\}$ pour un certain $i \in \{0, 1\}$, vaut $\texttt{monotony}(\mathcal{M}, e_2)$ si $sign(e_1) = \{+\}$, $rev(\texttt{monotony}(\mathcal{M}, e_2))$ si $sign(e_1) = \{-\}$, et $\emptyset$ sinon ; la fonction $sign$ est définie comme suit :
  - $sign(c) = \{+\}$ si c est une constante et $\texttt{int\_of}(c) > 0$;
  - $sign(c) = \{-\}$ si c est une constante et $\texttt{int\_of}(c) < 0$;
  - $sign(c) = \{0\}$ si c est une constante et $\texttt{int\_of}(c) = 0$;
  - $sign(e_1 * e_2) = \{s_1 \bar{*} s_2 | s_1 \in sign(e_1), s_2 \in sign(e_2)\}$, où $+\bar{*}+ = +$, $+\bar{*}- = -$, $+\bar{*}0 = 0$, $-\bar{*}- = +$, et $\bar{*}$ est supposée commutative ;
  - $sign(e) = \{+, -, 0\}$ dans tous les autres cas.

  Une constante est une chaîne s ou un entier i. On pourrait bien sûr raffiner la définition de la fonction $sign$, du moment qu'elle retourne un sur-ensemble de tous les signes que peut prendre son argument.
- $\texttt{monotony}(\mathcal{M}, e_1/e_2)$ vaut $\{\texttt{MONO}, \texttt{ANTI}\}$ si $sign(e_i) = \{0\}$ pour un certain $i \in \{0, 1\}$, vaut $\texttt{monotony}(\mathcal{M}, e_1)$ si $sign(e_2) = \{+\}$, $rev(\texttt{monotony}(\mathcal{M}, e_1))$ si $sign(e_2) = \{-\}$, et $\emptyset$ sinon.
- $\texttt{monotony}(\mathcal{M}, -e) = \texttt{monotony}(\mathcal{M}, !e) = rev(\texttt{monotony}(\mathcal{M}, e))$.
- $\texttt{monotony}(\mathcal{M}, \texttt{len}(e)) = \texttt{monotony}(\mathcal{M}, \texttt{new}(e)) = \{\texttt{MONO}, \texttt{ANTI}\}$ si $\texttt{monotony}(\mathcal{M}, e) = \{\texttt{MONO}, \texttt{ANTI}\}$, $\emptyset$ sinon.
- $\texttt{monotony}(\mathcal{M}, \texttt{substr}(e_1, e_2, e_3)) = \{\texttt{MONO}, \texttt{ANTI}\}$ si $\texttt{monotony}(\mathcal{M}, e_1) = \texttt{monotony}(\mathcal{M}, e_2) = \texttt{monotony}(\mathcal{M}, e_3) = \{\texttt{MONO}, \texttt{ANTI}\}$, $\emptyset$ sinon.
- $\texttt{monotony}(\mathcal{M}, \texttt{int}(e)) = \texttt{monotony}(\mathcal{M}, \texttt{time}(e)) = \texttt{monotony}(\mathcal{M}, \texttt{string}(e)) = \texttt{monotony}(\mathcal{M}, \text{`}e) = \texttt{monotony}(\mathcal{M}, e)$.
- $\texttt{monotony}(\mathcal{M}, \#e) = \texttt{monotony}(\mathcal{M}, |e|) = \texttt{monotony}(\mathcal{M}, \texttt{sum}(e)) = \{\texttt{MONO}\}$. En effet, le nombre d'éléments, la longueur, et la somme d'un accumulateur ne peuvent que croître. Dans le cas de la somme, c'est à cause de la forme spéciale de la définition de sum_of (section 2.2.2).
- $\texttt{monotony}(\mathcal{M}, \texttt{in}(e_1, e_2)) = \{\texttt{MONO}\}$ si $\texttt{monotony}(\mathcal{M}, e_1) = \{\texttt{MONO}, \texttt{ANTI}\}$, $\emptyset$ sinon. Le premier cas vient du fait qu'un accumulateur e_2 ne peut que croître dans l'ordre d'inclusion.

3 Motifs

Les motifs sont des prédicats portant sur des événements dans le log. Comme pour les expressions, on a une certaine latitude pour définir un langage de motifs. Essentiellement, on a besoin de définir deux jugements $R, \varrho \models pat \Rightarrow \rho$ et $\square(R, \varrho, pat)$, où pat est un motif quelconque (cf. section 3.3). Le premier jugement doit être *déterministe*, à savoir que si $R, \varrho \models pat \Rightarrow \rho$ et $R, \varrho \models pat \Rightarrow \rho'$, alors $\rho = \rho'$. Les conditions portant sur le second jugement sont plus difficiles à préciser. Nous nous fixerons donc sur un langage de motifs précis, sans montrer qu'on pourrait l'échanger contre un autre.

3.1 Événements, enregistrements

Un *événement* est un enregistrement (record) ou le symbole spécial `EOF`, dénotant un événement de *fin de fichier*.

On va considérer, pour simplifier ainsi que par souci d'efficacité, qu'un enregistrement R est juste un N-uplet, pour un entier $N \geq 0$ fixé, dépendant uniquement du format des logs que l'on cherche à analyser. On pose $R = (R_1, \ldots, R_N)$. Les champs R_1, …, R_N sont des valeurs dans $\mathcal{D}$.

Le format des enregistrements est *typé*. Autrement dit, on dispose d'un N-uplet Type de types. Les types sont `A` (chaîne de caractères), `I` (entier machine), `M` (entier machine monotone), `T` (date), `D` (date monotone). Pour tout i, $1 \leq i \leq N$, si $\mathrm{Type}_i = \mathtt{A}$, alors $R_i \in \mathrm{String}$; si $\mathrm{Type}_i \in \{\mathtt{I}, \mathtt{M}\}$, alors $R_i \in \mathrm{Int}$; si $\mathrm{Type}_i \in \{\mathtt{T}, \mathtt{D}\}$, alors $R_i \in \mathrm{Time}$. Les champs dits *monotones*, c'est-à-dire de type `M` ou `D`, sont des champs qui ne peuvent que croître au fur et à mesure où l'on avance dans le log.

3.2 Syntaxe

Les motifs obéissent à la syntaxe :

$$
\begin{array}{rcl}
pat & ::= & field\text{-}pat \\
& | & cmp\text{-}pat \\
& | & \langle\langle \mathtt{EOF} \rangle\rangle \\
& | & pat \wedge pat
\end{array}
$$

$$
\begin{array}{rcl}
field\text{-}pat & ::= & i, re, var, [var, \mathrm{String}]^* \\
& & (1 \leq i \leq N, re \in \mathrm{Regexp}, var \in \mathcal{V}_{\mathrm{pat}} \cup \{\_\} \text{ disjointes deux à deux}) \\
cmp\text{-}pat & ::= & i, var, [cmp\text{-}op, e]^* \\
& & (1 \leq i \leq N, var \in \mathcal{V}_{\mathrm{pat}} \cup \{\_\}) \\
cmp\text{-}op & ::= & ==, \,!=, <, >, <=, >=
\end{array}
$$

où Regexp est l'ensemble des expressions régulières.

De plus, les motifs obéissent aux restrictions suivantes :

- pour tout motif $field\text{-}pat = i, re, var, [var_1, s_1, \ldots, var_n, s_n]$, alors $\mathrm{Type}_i = verb/A/$ (le test d'expression régulière n'a de sens que sur des champs texte) ;
- pour tout motif $cmp\text{-}pat = i, var, [cmp\text{-}op_1, e_1, \ldots, cmp\text{-}op_n, e_n]$, on a $\mathrm{fv}(e_j) \subseteq \mathcal{V}_{\mathrm{rigid}}$, $1 \leq j \leq n$;

– finalement, les restrictions de la section 4.5. (Nous préférons les reléguer plus loin, car elles font appel à des fonctions d'analyse statique que nous n'avons pas encore définies.)

Soit $cmp\text{-}pat = i, var, [cmp\text{-}op_1, e_1, \ldots, cmp\text{-}op_n, e_n]$. La deuxième condition, alliée à la restriction syntaxique de la section 4.5.1, assure que e_j, une fois évaluée, ne changera plus jamais de valeur. Ceci est utilisé pour justifier la correction de la définition du prédicat $\Box(R, \varrho, pat)$ en section 3.3.

3.3 Sémantique

On définit maintenant une relation

$$R, \varrho \models pat \Rightarrow \rho$$

qui décrit quand le motif pat est vérifié par l'événement R dans un environnement ϱ, donnant un nouvel environnement ρ. On remarquera qu'on aura dans tous les cas $\mathrm{dom}\rho \subseteq \mathcal{V}_{\mathrm{pat}}$.

- si $field\text{-}pat = i, re, var, [var_1, s_1, \ldots, var_n, s_n]$, alors $R, \varrho \models field\text{-}pat \Rightarrow \rho_n$, où
  - $R \neq \texttt{EOF}$;
  - $s = \texttt{string\_of}(R_i)$;
  - $sub = \texttt{regmatch}(re)(s) \neq \bot$;
  - $\rho_0 = []$ si $var = \_$, $\rho_0 = [var \mapsto s]$ sinon ;
  - pour tout j, $1 \leq j \leq n$, $\rho_j = \rho_{j-1}$ si $var_j = \_$, $\rho_j = \rho_{j-1}[var_j \mapsto sub(s_j)]$ sinon.
- si $cmp\text{-}pat = i, var, [cmp\text{-}op_1, e_1, \ldots, cmp\text{-}op_n, e_n]$, alors $R, \varrho \models cmp\text{-}pat \Rightarrow \rho$ si et seulement si :
  - $R \neq \texttt{EOF}$;
  - $\rho = []$ si $var = \_$, $\rho = [var \mapsto R_i]$ sinon ;
  - $k = \texttt{int\_of}(R_i)$;
  - pour tout j, $1 \leq j \leq n$, $\texttt{bool}(\mathcal{E}\,[\![z\ cmp\text{-}op_j\ e_j]\!]\,\varrho[z \mapsto k])$ est vrai, où z est une variable fraîche.
- $R, \varrho \models \langle\langle \texttt{EOF} \rangle\rangle \Rightarrow \rho$ si et seulement si :
  - $R = \texttt{EOF}$;
  - et $\rho = []$.
- si $pat = pat_1 \wedge pat_2$, alors $R, \varrho \models pat \Rightarrow \rho$ si et seulement si :
  - $R, \varrho \models pat_1 \Rightarrow \rho_1$;
  - $R, \varrho \models pat_2 \Rightarrow \rho_2$;
  - $\rho = \rho_1 \oplus \rho_2 \neq \bot$.

3.4 Fonctions d'analyse statique

Test d'échec éternel des motifs. Notons $R, \varrho \models pat \not\Rightarrow$ le prédicat exprimant que $R, \varrho \models pat \Rightarrow \rho$ pour aucun environnement ρ. On définit aussi un prédicat $\Box(R, \varrho, pat)$, qui est intuitivement vrai à condition que, si $R, \varrho \models pat \not\Rightarrow$, alors pour tout événement R' venant après R dans le log, pour tout environnement ϱ' qui coincide avec ϱ sur les variables rigides (autrement dit, $\varrho'(x) = \varrho(x)$ pour tout $x \in \mathcal{V}_{\mathrm{rigid}}$), alors $R', \varrho' \models pat \not\Rightarrow$. Ceci exploite le fait que certains champs des enregistrements sont *monotones* (ceux de type `M` ou `D`), et ne peuvent que croître d'un enregistrement R à un enregistrement ultérieur R' :

- $\Box(R, \varrho, field\text{-}pat)$ est toujours faux ;
- $\Box(R, \varrho, cmp\text{-}pat)$ est vrai, où $cmp\text{-}pat = i, var, [cmp\text{-}op_1, e_1, \ldots, cmp\text{-}op_n, e_n]$, si et seulement s'il existe j_0, $1 \leq j_0 \leq n$ tel que :
  - $R \neq \texttt{EOF}$;
  - $\text{Type}_{j_0} \in \{\texttt{M}, \texttt{D}\}$;
  - $k = \texttt{int\_of}(R_i)$;
  - pour tout j, $1 \leq j < j_0$, , $\texttt{bool}(\mathcal{E} [\![z\ cmp\text{-}op_j\ e_j]\!]\ \varrho[z \mapsto k])$ est vrai, où z est une variable fraîche ;
  - $\texttt{bool}(\mathcal{E} [\![z\ cmp\text{-}op_{j_0}\ e_{j_0}]\!]\ \varrho[z \mapsto k])$ est faux, où z est une variable fraîche ;
  - $cmp\text{-}op_{j_0} \in \{\texttt{<}, \texttt{<=}\}$.

  Noter qu'on pourrait être moins exigeant et seulement imposer qu'il existe un j_0 tel que $\texttt{bool}(\mathcal{E} [\![z\ cmp\text{-}op_{j_0}\ e_{j_0}]\!]\ \varrho[z \mapsto k])$ est faux et $cmp\text{-}op_{j_0} \in \{\texttt{<}, \texttt{<=}\}$, mais ceci demanderait davantage de calculs.

  Ceci est correct : si ϱ' coincide avec ϱ sur les variables rigides, alors $\mathcal{E} [\![e_{j_0}]\!]\ \varrho = \mathcal{E} [\![e_{j_0}]\!]\ \varrho'$ (ceci demande que $\text{fv}(e_{j_0}) \subseteq \text{dom}\varrho$ et $\text{fv}(e_{j_0}) \subseteq \text{dom}\varrho'$, ce qui sera assuré par la condition de la section 4.5.1 et le fait que toutes les variables libres de e_{j_0} sont rigides) ; de plus, le champ i étant monotone, $\texttt{int\_of}(R_i) \leq \texttt{int\_of}(R'_i)$; donc si $\texttt{bool}(\mathcal{E} [\![z\ cmp\text{-}op_{j_0}\ e_{j_0}]\!]\ \varrho[z \mapsto \texttt{int\_of}(R_i)])$ est faux , c'est-à-dire si $\texttt{int\_of}(R_i) < \mathcal{E} [\![e_{j_0}]\!]\ \varrho$ est faux (resp. $\leq$), alors a fortiori $\texttt{int\_of}(R'_i) < \mathcal{E} [\![e_{j_0}]\!]\ \varrho$ (resp. $\leq$) $= \mathcal{E} [\![e_{j_0}]\!]\ \varrho'$ est faux aussi, c'est-à-dire que $\texttt{bool}(\mathcal{E} [\![z\ cmp\text{-}op_{j_0}\ e_{j_0}]\!]\ \varrho[z \mapsto \texttt{int\_of}(R'_i)])$ est faux.
- $\Box(R, \varrho, \langle\langle \texttt{EOF} \rangle\rangle)$ est toujours faux.

  Ceci peut paraître surprenant. Mais rien n'interdit de rajouter de nouveaux enregistrements à la fin d'un log après que l'on a vu sa fin de fichier. Les événements `EOF` sont de ce point de vue des événements synthétiques, servant à représenter une fin de fichier ; on peut en insérer un lorsqu'on rencontre une fin de fichier, ou zéro si l'on ne souhaite pas déclencher d'action spéciale sur fin de fichier.
- $\Box(R, \varrho, pat_1 \wedge pat_2)$ est vrai si et seulement si $R, \varrho \models pat_1 \not\Rightarrow$ et $\Box(R, \varrho, pat_1)$, ou bien $R, \varrho \models pat_1 \Rightarrow \rho_1$, $R, \varrho \models pat_2 \not\Rightarrow$, et $\Box(R, \varrho, pat_2)$.

  Encore une fois, on pourrait être moins exigeant et seulement imposer que $R, \varrho \models pat_1 \not\Rightarrow$ et $\Box(R, \varrho, pat_1)$, ou bien $R, \varrho \models pat_2 \not\Rightarrow$, et $\Box(R, \varrho, pat_2)$.

Variables liées à des champs monotones. On définit de plus l'ensemble $\texttt{monovars}(pat)$ des variables liées par un motif pat à un champ de type monotone.

- $\texttt{monovars}(field\text{-}pat) = \begin{cases} \{var\} & \text{si } var \in \mathcal{V}, \text{Type}_i \in \{\texttt{M}, \texttt{D}\} \\ \emptyset & \text{sinon} \end{cases}$

  où $field\text{-}pat = i, re, var, [var_1, s_1, \ldots, var_n, s_n]$.
- $\texttt{monovars}(cmp\text{-}pat) = \begin{cases} \{var\} & \text{si } var \in \mathcal{V}, \text{Type}_i \in \{\texttt{M}, \texttt{D}\} \\ \emptyset & \text{sinon} \end{cases}$

  où $cmp\text{-}pat = i, var, [cmp\text{-}op_1, e_1, \ldots, cmp\text{-}op_n, e_n]$.
- $\texttt{monovars}(\langle\langle \texttt{EOF} \rangle\rangle) = \emptyset$.

– $\texttt{monovars}(pat_1 \wedge pat_2) = \texttt{monovars}(pat_1) \cup \texttt{monovars}(pat_2)$.

Variables libres et liées par les motifs. On définit maintenant le couple $(free, bound) = \texttt{pat\_vars}(pat)$ de l'ensemble des variables *libres* dans le motif pat et de celui des variables *liées* dans le motif pat :

– $\texttt{pat\_vars}(field\text{-}pat) = (\emptyset, \{var, var_1, \ldots, var_n\} \setminus \{\_\})$ si $field\text{-}pat = i, re, var, [var_1, s_1, \ldots, var_n, s_n]$.
– $\texttt{pat\_vars}(cmp\text{-}pat) = (\bigcup_{i=1}^{n} \mathrm{fv}(e_i), \{var\} \setminus \{\_\})$ si $cmp\text{-}pat = i, var, [cmp\text{-}op_1, e_1, \ldots, cmp\text{-}op_n, e_n]$.
– $\texttt{pat\_vars}(\langle\langle \texttt{EOF} \rangle\rangle) = (\emptyset, \emptyset)$.
– $\texttt{pat\_vars}(pat_1 \wedge pat_2) = (free_1 \cup free_2, bound_1 \cup bound_2)$, où $(free_1, bound_1) = \texttt{pat\_vars}(pat_1)$ et $(free_2, bound_2) = \texttt{pat\_vars}(pat_2)$.

4 Machines, formules

Les formules, enfin, servent à détecter des attaques, exprimées sous forme de corrélations temporelles entre événements repérés par des motifs.

4.1 Syntaxe

Machines. Une *machine* M est un automate $(Q, q_{\mathrm{acc}}, q_{\mathrm{rej}}, Q^*, \delta)$, où :

– Q est un ensemble fini de *sommets*, ou *états*. On considérera que $Q \subseteq \mathcal{V}_{\mathrm{count}}$. Cette astuce nous permet d'associer naturellement à chaque état q un compteur $\_\$x_q$, à savoir l'état q lui-même. Ce compteur comptabilise le nombre de fois où un itinéraire (section 4.2) passe par l'état q.
– q_{acc} et q_{rej} sont deux états (dans Q) distincts, appelés respectivement état d'*acceptation* et état de *rejet*.
  Le nom d'état de rejet est peut-être un peu trompeur : les deux états q_{acc} et q_{rej} sont en fait des états acceptants, mais seul le premier donne lieu à une alerte.
– Q^* est un sous-ensemble de Q. Les états de Q^* sont appelés états de *choix commis*.
– δ est une *fonction de transition*, qui à chaque état q associe une liste finie de triplets (pat_i, e_i, q_i), $(1 \leq i \leq n)$, où pat_i est un motif optionnel (c'est-à-dire un motif, ou le symbole spécial ϵ), e_i est une expression (la *garde*), et q_i est un état.
  Si $q \in Q^*$, on impose de surcroît que $pat_i \neq \epsilon$ pour tout i, $1 \leq i \leq n$.
  On notera parfois $q \xrightarrow[e]{pat} q'$ pour exprimer que le triplet (pat, e, q') est dans la liste $\delta(q)$.
  S'il s'agit du triplet (pat_i, e_i, q_i) on dira que $q \xrightarrow[e]{pat} q'$ est la *iième transition sortant de* q.
  Le nombre n de transitions sortant de q est appelé le *degré* de q.
  Les états q_i, $1 \leq i \leq n$, sont appelés les *successeurs* de q. Si $q \in Q^*$, $\delta(q)$ est vue comme une liste ordonnée, sinon $\delta(q)$ est juste une collection non ordonnée de transitions.

Formules. Une *formule* F est un quadruplet $(M, q, sync\text{-}vars, \lightning)$ formé d'une machine $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$, d'un état dit *initial* $q \in Q$, d'un ensemble fini $sync\text{-}vars$ de variables rigides, dites de *synchronisation*, et d'un booléen $\lightning$ (vrai si la formule est `anchored` au sens de [Goubault-Larrecq, 2001]).

Les machines $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$ sont restreintes de sorte que [Roger and Goubault-Larrecq, 2001] :

– Acyclicité : M ne contient pas de cycles d'ϵ-transitions, autrement dit il n'y a pas de suite non vide de transitions de la forme $q_1 \xrightarrow[e_1]{\epsilon} q_2 \xrightarrow[e_2]{\epsilon} \cdots \quad q_n \xrightarrow[e_n]{\epsilon} q_1$.
  Ceci est pour une question de terminaison : sinon l'algorithme de la section 4.3 bouclerait en arrivant sur l'état q_1.
– ϵ-déterminisme : pour tout $q \in Q \setminus Q^*$ tel qu'il y a deux transitions distinctes $q \xrightarrow[e_1]{\epsilon} q_1'$ et $q \xrightarrow[e_2]{\epsilon} q_2'$, alors pour aucun environnement ρ ne sont $\mathtt{bool}(\mathcal{E}\,[\![e_1]\!]\,\rho)$ et $\mathtt{bool}(\mathcal{E}\,[\![e_2]\!]\,\rho)$ tous les deux vrais.
  Ceci est pour des raisons de correction de l'algorithme de la section 4.3.

Les formules $F = (M, q, sync\text{-}vars, \lightning)$ sont de plus restreintes de sorte que :

– il n'y a pas de chemin d'ϵ-transitions de q ni à q_{acc} ni à q_{rej}, autrement dit il n'y a pas de suite (possiblement vide) de transitions de la forme $q = q_1 \xrightarrow[e_1]{\epsilon} q_2 \xrightarrow[e_2]{\epsilon} \cdots \quad q_{n-1} \xrightarrow[e_{n-1}]{\epsilon} q_n$, où $q_n \in \{q_{\text{acc}}, q_{\text{rej}}\}$.
  Sinon F sera trivialement satisfaite à tout état ; de telles formules ne sont donc pas spécialement utiles, et compliqueraient l'algorithme de la section 4.3.

Les formules obéissent à des restrictions additionnelles, dites de bonne formation, qui seront décrites en section 4.5.

4.2 Sémantique

La sémantique est celle décrite dans [Roger and Goubault-Larrecq, 2001].

Un *log* S est une suite, finie ou infinie, d'événements tels que définis en section 3.1. Noter que le nombre N de champs dans chaque enregistrement R de S est constant, de même que l'information de typage Type.

On note S_k, $k \geq 1$, le $k$ième événement de S, quand ceci est défini. Le *domaine* $\text{dom}S$ est l'ensemble des indices k pour lesquels S_k est défini, et est un intervalle $[1, n]$ ou $[1, +\infty[$. La longueur $|S|$ du log S est le sup des indices $k \in \text{dom}S$, n dans le premier cas, $+\infty$ dans le second.

Pour tout environnement ρ, posons $\rho_{\text{rigid}} = \rho_{|\mathcal{V}_{\text{rigid}}}$, $\rho_{\text{flex}} = \rho_{|\mathcal{V}_{\text{flex}}}$, et $\rho_{\text{count}} = \rho_{|\mathcal{V}_{\text{count}}}$.

Rappelons que tout état est aussi un compteur. Notons $\rho[q{+}{+}]$, pour tout état q, l'environnement de domaine $\text{dom}\rho \cup \{q\}$, qui à q associe $\rho(q) + 1$ si $q \in \text{dom}\rho$, 1 sinon ; et qui à tout $X \in \text{dom}\rho \setminus \{q\}$ associe $\rho(X)$. Ceci a pour effet d'incrémenter le compteur associé à l'état q ; un compteur non initialisé (hors de $\text{dom}\rho$) étant lui mis à 1. Ceci est cohérent avec le fait que $\mathcal{E}\,[\![q]\!]\,\rho = 0$ si $q \notin \text{dom}\rho$.

Un *itinéraire partiel* d'une formule $F = (M, q, sync\text{-}vars, \lightning)$, $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$ dans un log S est une suite finie $\sigma = (\sigma_1, \ldots, \sigma_\ell)$, $\ell \in \mathbb{N}$, où pour chaque j, $1 \leq j \leq \ell$, σ_j est un triplet (q_j, i_j, ρ_j), $q_j \in Q$, $i_j \in \mathrm{dom} S$, et ρ_j est un environnement tels que, si $\ell \geq 1$ alors :

- $q_1 = q$, $\rho_1 = []$;
- pour tout j, $1 < j \leq \ell$, soient $q_{j-1} \xrightarrow[e_{j1}]{pat_{j1}} q_{j1}$, …, $q_{j-1} \xrightarrow[e_{jn_j}]{pat_{jn_j}} q_{jn_j}$ les transitions sortant de q_{j-1} dans cet ordre (formellement, $\delta(q_{j-1})$ est la liste $(pat_{j1}, e_{j1}, q_{j1}), \ldots, (pat_{jn_j}, e_{jn_j}, q_{jn_j})$), alors il existe un indice k, $1 \leq k \leq n_j$, tel que :
  - $q_j = q_{jk}$;
  - pat_{jk} est un enregistrement, $S_{i_{j-1}}, \rho_{j-1} \models pat_{jk} \Rightarrow \rho$, $\rho_j = (\rho_{j-1} \oplus \rho)[q_j{++}]$, et $i_j > i_{j-1}$; ou $pat_{jk} = \epsilon$, $\rho_j = \rho_{j-1}[q_j{++}]$, et $i_j = i_{j-1}$;
  - $\mathtt{bool}(\mathcal{E}\,[\![e_{jk}]\!]\,\rho_j)$ est vrai ;
  - si $q_{j-1} \in Q^*$ (en particulier, tous les $pat_{jk'}$, $1 \leq k' \leq n_j$, sont des enregistrements), alors pour tout k', $1 \leq k' < k$, pour tout ρ tel que $S_{i_{j-1}}, \rho_{j-1} \models pat_{jk'} \Rightarrow \rho$, et tel que $\rho' = \rho_{j-1} \oplus \rho$ est bien défini, $\mathtt{bool}(\mathcal{E}\,[\![e_{jk'}]\!]\,\rho'[q_{j-1}{++}])$ est faux.
    De plus, si $j > 2$, alors pour tout i tel que $i_{j-2} < i < i_{j-1}$, pour tout k', $1 \leq k' \leq n_j$, pour tout ρ tel que $S_i, \rho_{j-1} \models pat_{jk'} \Rightarrow \rho$ et $\rho' = \rho_{j-1} \oplus \rho$ est bien défini, $\mathtt{bool}(\mathcal{E}\,[\![e_{jk'}]\!]\,\rho'[q_{j-1}{++}])$ est faux.

Un *itinéraire* de F dans S est un itinéraire partiel comme ci-dessus tel que $\ell \geq 1$ et $q_\ell = q_{\mathrm{acc}}$ (auquel cas l'itinéraire est *acceptant*) ou $q_\ell = q_{\mathrm{rej}}$ (auquel cas l'itinéraire est *rejetant*).

Cette définition est relativement illisible, expliquons-là en termes plus simples, qui regardent les machines M comme de véritables machines, quoique des machines non-déterministes. La donnée d'un état q_j, d'un entier i_j numérotant un événement dans le log S, et d'un environnement ρ_j, décrit la *configuration instantanée* de la machine M.

Lorsqu'elle est dans la configuration $(q_{j-1}, i_{j-1}, \rho_{j-1})$, la machine peut alors :

- accepter si $q_{j-1} = q_{\mathrm{acc}}$;
- ou rejeter si $q_{j-1} = q_{\mathrm{rej}}$;
- ou franchir une ϵ-transition $q_{j-1} \xrightarrow[e]{\epsilon} q_j$, et se retrouver dans la configuration $(q_j, i_j = i_{j-1}, \rho_j = \rho_{j-1}[q_{j-1}{++}])$, à condition que la garde e soit vraie ; l'opération $[q_{j-1}{++}]$ effectue un comptage en ajoutant un au nombre de fois où l'on est passé par l'état q_{j-1}, mais on n'avance pas dans le log ($i_j = i_{j-1}$) ;
- ou avancer d'un nombre indéterminé d'événements dans le log, pour se retrouver en $i_j > i_{j-1}$, sur un événement S_{i_j} reconnu par un motif pat_{jk} et rendant la garde e_{jk} vraie, pour une certaine transition $q_{j-1} \xrightarrow[e_{jk}]{pat_{jk}} q_j$; de même que ci-dessus, le compteur q_{j-1} est incrémenté.

De plus, si $q_{j-1} \in Q^*$, alors aucune transition sortant de q_{j-1} n'est une ϵ-transition (on est donc dans le deuxième cas), et le non-déterminisme est réduit par le fait qu'au lieu d'avancer sur une transition numéro k quelconque, on choisit k minimal, autrement dit, on avance sur la première transition franchissable. De plus, si une transition est franchissable, alors elle doit être franchie, et on n'a pas le choix de ne pas la franchir et d'attendre un nombre indéterminé d'événements pour réessayer de la franchir, comme ci-dessus. (Il s'agit d'un comportement similaire à la coupure ! de Prolog [Goubault-Larrecq et al., 2002].)

Le *domaine* $\mathrm{dom}\sigma$ de σ est $\{i_1 < i_2 < \ldots < i_\ell\}$. L'*étendue* de $\{i_1, i_2, \ldots, i_\ell\}$, et par extension de l'itinéraire partiel σ, est l'intervalle $\mathrm{span}\,\sigma = [i_1, i_\ell]$ (l'ensemble vide si $\ell = 0$). La *longueur*$\mathrm{len}\,\sigma$ est ℓ. Si $\ell \geq 1$, le *début* de σ est i_1, sa *fin* est i_ℓ, son *état final* est q_ℓ, et son *environnement final* $end\rho(\sigma)$ est ρ_ℓ.

Les ordres stricts $<_{lex}$ et $\prec$ sur les domaines d'itinéraires partiels, et par extension, sur les itinéraires partiels eux-mêmes, sont définis par : pour tous domaines $D = \{i_1 < i_2 < \ldots < i_\ell\}$ et $D' = \{i'_1 < i'_2 < \ldots < i'_{\ell'}\}$, $D <_{lex} D'$ si et seulement si $i_1 = i'_1$, $i_2 = i'_2$, …, $i_{j-1} = i'_{j-1}$, $i_j < i'_j$ pour un certain j, $1 \leq j \leq \min(\ell, \ell')$. On pose $D \prec D'$ si et seulement si $[i_1, i_\ell] \subsetneq [i'_1, i'_{\ell'}]$, ou $[i_1, i_\ell] = [i'_1, i'_{\ell'}]$ et $D <_{lex} D'$.

Un *itinéraire minimal* en est un qui est minimal pour $\prec$. La justification de cette notion (*shortest run*) se trouve dans l'article [Roger and Goubault-Larrecq, 2001].

Le but de l'analyse de logs est, étant donné un log S, pour chaque $i \in \mathrm{dom} S$, pour chaque formule $F = (M, q, sync\text{-}vars, \lightning)$, de trouver les itinéraires minimaux σ^i de F de début i. Les *alertes* sont ces itinéraires qui :

- sont acceptants : les itinéraires minimaux rejetants ne donnent pas lieu à alerte ;
- sont non subsumés : il ne doit pas exister d'itinéraire (minimal) $\sigma^{i'}$ de début i', $1 \leq i' < i$, avec $\mathrm{span}(\sigma^{i'}) \cap \mathrm{span}(\sigma^i) \neq \emptyset$ et tel que les restrictions de $end\rho(\sigma^i)$ et de $end\rho(\sigma^{i'})$ à $sync\text{-}vars$ coincident. (La notion d'égalité est ici l'égalité standard, pas $\leftrightharpoons$.)

Lorsque $\lightning$ est faux, c'est exactement la sémantique des formules. Lorsque $\lightning$ est vrai, la propriété de non-subsomption est modifiée d'une façon qui n'est pas clairement définissable dans la sémantique ci-dessus, et qui sera définie au travers de l'algorithme (section 4.3). Comme expliqué dans [Goubault-Larrecq, 2001], le mot-clé `anchored` de la syntaxe de log$\mathcal{W}$eaver, qui correspond à $\lightning = \mathbf{1}$, est un hack... mais un hack utile, similaire à la coupure `!` de Prolog [Goubault-Larrecq et al., 2002].

4.3 Algorithme

On se fixe un nombre P de formules F_p, $1 \leq p \leq P$. Elles constituent la base de signatures d'attaques. On considère que P est fixe, de même que le nombre N de champs des enregistrements du log est fixe, surtout par souci de simplification.

Un *thread* τ est un septuplet $(pid, F, q, t, \rho, flags, lock)$ où $pid \in \mathbb{N}$ est appelé l'*identificateur*, F est une formule $(M, q_0, sync\text{-}vars, \lightning)$, où $M = (Q, q_{\mathrm{acc}}, q_{\mathrm{rej}}, Q^*, \delta)$, $q \in Q$, ρ est un environnement, $flags$ est un sous-ensemble de l'ensemble fini $\{\mathtt{RETRIGGER}, \mathtt{ONLY\_ONCE}\}$, et $lock$ est un environnement de domaine $sync\text{-}vars$ ou le symbole spécial $\perp$. De plus, t est un entier entre 1 et le degré de q et désigne donc une transition sortant de q si $q \in Q \setminus Q^*$, et vaut 0 si $q \in Q^*$.

Il est à noter qu'il peut très bien y avoir plusieurs threads ayant le même identificateur pid. Cette possibilité est importante dans la suite de l'algorithme. Dans la plupart des cas, un thread comme ci-dessus est en train d'attendre de trouver un événement dans le log qui permette de franchir la transition numéro t sortant de q, les valeurs de ses variables étant données par l'environnement ρ. Ceci est le cas si $q \notin Q^*$, sinon $t = 0$, et le thread attend de trouver un événement dans le log qui permette de franchir l'une des transitions sortant de q.

Ce comportement est modifié selon la valeur de $flags$. Notamment, si $\mathtt{ONLY\_ONCE} \in flags$, alors l'algorithme n'attend pas un événement, mais requiert que ce soit l'événement courant qui permette de franchir la transition sortant de q. Si RETRIGGER $\in flags$ (le cas général), et que l'événement courant permette de franchir la transition, alors l'algorithme va dupliquer le thread courant en deux threads, l'un franchissant la transition, l'autre ne la franchissant pas. Si RETRIGGER $\not\in flags$, c'est qu'une fonction d'analyse statique (section 4.4) a permis de déduire que le thread ne franchissant pas la transition ne mènera jamais à un itinéraire minimal, et n'a pas à être créé.

Un *contexte* Γ est un triplet formé de :

– deux files $\mathbf{q}_{\text{new}}$ et $\mathbf{q}_{\text{retrig}}$; $\mathbf{q}_{\text{new}}$ et $\mathbf{q}_{\text{retrig}}$ sont des files contenant des threads ou le symbole spécial $\curvearrowright$;

– un ensemble fini $alerts$ d'environnements, appelées *alertes*.

Les files sont juste des listes finies : $[\tau_1, \ldots, \tau_k]$ est la liste formée des k threads τ_1, …, τ_k, et on note @ l'opérateur de concaténation des listes. Le format de $\mathbf{q}_{\text{new}}$ et de $\mathbf{q}_{\text{retrig}}$ est un peu spécial, à cause de la présence possible du symbole $\curvearrowright$. Il y a plusieurs façons de comprendre cette représentation. Une façon simple, recommandée en première lecture, est d'ignorer les symboles $\curvearrowright$: la file $[\tau_1, \curvearrowright, \tau_2, \tau_3, \curvearrowright, \tau_4]$ est essentiellement la file formée des quatre threads $\tau_1, \tau_2, \tau_3, \tau_4$. Une façon plus précise de voir ce genre de files est de comprendre que τ_1 doit être traitée avant τ_2 et τ_3, qui doivent être traitées avant τ_4, mais τ_2 et τ_3 peuvent être traitées dans n'importe quel ordre. (Le bug de [Roger and Goubault-Larrecq, 2001] consistait essentiellement en ce que seul un ordre total $\tau_1 < \tau_2 < \tau_3 < \tau_4$ était considéré, ce qui violait parfois la politique de l'itinéraire minimal.)

On notera de plus $kill$ un ensemble fini d'entiers, représentant les identificateurs de threads à supprimer, c'est-à-dire tels qu'il existe un thread de même identificateur pid qui a déjà atteint son état d'acceptation ou de rejet.

Dans la suite, nous définissons une série de jugements, qui sont tous de la forme $X \vdash_{op} Y \Rightarrow X'$. Intuitivement, ces jugements se lisent : "partant d'un état X, l'opération op appliquée aux arguments Y résulte en le nouvel état X'". Pour chaque jugement, nous donnons sont format dans un encadré à droite, comme suit :

$$\boxed{X \vdash_{op} Y \Rightarrow X'}$$

puis nous décrivons l'idée intuitive de ce que doit faire l'opération op, les abréviations utilisées dans la définition de ce jugement, et les règles permettant de déduire des jugements de cette forme.

4.3.1 Ajout d'un état dans la file d'attente

$$\boxed{\Gamma, kill \vdash_{\texttt{q-}add} F, q, \rho, pid, flags, lock \Rightarrow \Gamma', kill'}$$

Ce jugement est utilisé pour ajouter à la file des threads un thread $(pid, F, q, t, \rho, flags, lock)$, modulo quelques détails : on n'empile rien si q est un état d'acceptation ou de rejet, et on franchit les ϵ-transitions immédiatement.

On supposera ici que $F = (M, q, sync\text{-}vars, \lightning)$, et $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$.

Si $\Gamma = \mathbf{q}_{\text{new}}, \mathbf{q}_{\text{retrig}}, alerts$, et τ est un thread, on notera $\Gamma +_{\text{new}} \tau$ le contexte $\mathbf{q}_{\text{new}}@[\tau], \mathbf{q}_{\text{retrig}}, alerts$. $\Gamma +_{\text{new}} \tau$ est le résultat de l'empilement de τ sur la file $\mathbf{q}_{\text{new}}$. On notera aussi $\Gamma +_{\text{retrig}} \tau$ le contexte $\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{retrig}}@[\tau], alerts$, résultat de l'empilement de τ sur la file $\mathbf{q}_{\text{retrig}}$.

Acceptation. Si $q = q_{\text{acc}}$, on vient de trouver un itinéraire acceptant, et on l'ajoute à la liste des alertes. D'autre part, on tue tous les autres threads de même identificateur pid :

$$\frac{}{\begin{array}{rcl}\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{retrig}}, alerts, kill & \vdash_{\mathbf{q}\text{-}add} & F, q_{\text{acc}}, \rho, pid, flags, lock \\ & \Rightarrow & \mathbf{q}_{\text{new}}, \mathbf{q}_{\text{retrig}}, alerts \cup \{\rho\}, kill \cup \{pid\}\end{array}} \tag{1}$$

Le thread à partir duquel on fabrique l'alerte est nécessairement minimal et non subsumé ; c'est une conséquence des invariants sur l'ordre des threads dans les files, et ce n'est pas une propriété triviale. Les autres threads de même identificateur pid représentent toutes les autres tentatives de trouver un itinéraire pour la même formule avec le même début dans le log, et ne sont pas minimales : ils doivent donc être éliminés.

Au lieu de parcourir les files $\mathbf{q}_{\text{new}}$ et $\mathbf{q}_{\text{retrig}}$ pour enlever ces threads, on ajoute leurs identificateurs à $kill$, et un processus ultérieur enlèvera les threads d'identificateurs tués (dans $kill$) petit à petit. Ceci est pour des raisons d'efficacité.

En fait, les autres threads de même identificateur pid contiennent aussi les autres tentatives de trouver un itinéraire pour la même formule, à condition que l'un de ces itinéraires soit nécessairement subsumé par un autre. Ceci est détecté par une autre règle (la règle (7)), qui se contente de remplacer son identificateur courant pid par l'identificateur pid' d'un thread qu'elle doit subsumer ou par lequel elle doit être subsumé.

Rejet. Si $q = q_{\text{rej}}$, on vient de trouver un itinéraire rejetant.

$$\frac{}{\Gamma, kill \vdash_{\mathbf{q}\text{-}add} F, q_{\text{rej}}, \rho, pid, flags, lock \Rightarrow \Gamma, kill} \tag{2}$$

On n'empile donc rien, et le thread expire tout seul.

Etats de choix commis. Si $q \in Q^*$, on souhaite ne franchir qu'une seule des transitions sortant de q, à savoir la première qui convienne. On empile le thread adéquat, en attente sur l'état q, sur la file $\mathbf{q}_{\text{new}}$:

$$\frac{q \in Q^* \quad \tau = (pid, F, q, 0, \rho[q{+}{+}], flags, lock)}{\Gamma, kill \vdash_{\mathbf{q}\text{-}add} F, q, \rho, pid, flags, lock \Rightarrow \Gamma +_{\text{new}} \tau, kill} \tag{3}$$

A noter que l'on incrémente le compteur correspondant à l'état q.

États ordinaires. Si $q \notin Q^*$, on empile autant de threads qu'il y a de transitions sortant de q. Soient (pat_i, e_i, q_i) les éléments de la liste $\delta(q)$, $1 \leq i \leq n$.

$$\frac{\begin{array}{c} q \notin Q^* \quad \rho' = \rho[q{+}{+}] \quad \Gamma_0 = \Gamma \quad kill_0 = kill \\ (1 \leq i \leq n) \left\{ \begin{array}{ll} \begin{array}{lll} \Gamma_{i-1}, kill_{i-1} & \vdash_{\mathtt{q}\text{-}add} & F, q_i, \rho', \\ & & pid, flags, lock \\ & \Rightarrow & \Gamma_i, kill_i \end{array} & \text{si } pat_i = \epsilon, \mathtt{bool}(\mathcal{E}\,[\![e_i]\!]\,\rho') = \mathbf{1} \\ \Gamma_i = \Gamma_{i-1}, kill_i = kill_{i-1} & \text{si } pat_i = \epsilon, \mathtt{bool}(\mathcal{E}\,[\![e_i]\!]\,\rho') = \mathbf{0} \\ \Gamma_i = \Gamma_{i-1} +_{\text{new}} (pid, F, q, i, \rho', flags', lock), & \\ \quad kill_i = kill_{i-1} & \text{si } pat_i \neq \epsilon \\ \text{où } flags' = \left\{ \begin{array}{ll} flags & \text{si } \mathtt{no\_backtrack\_needed}(F, q, i, \rho') \\ flags \cup \{\mathtt{RETRIGGER}\} & \text{sinon} \end{array} \right. & \end{array} \right. \end{array}}{\Gamma, kill \vdash_{\mathtt{q}\text{-}add} F, q, \rho, pid, flags, lock \Rightarrow \Gamma_n, kill_n} \tag{4}$$

où $\mathtt{no\_backtrack\_needed}(F, q, i, \rho')$ est une fonction qui, si elle retourne $\mathbf{1}$, alors tout itinéraire minimal pour F partant de q, avec les valeurs de variables telles que décrites par ρ', doit d'abord franchir la $i$ième transition sortant de q. Une solution qui est toujours correcte est de définir `no_backtrack_needed` comme retournant toujours $\mathbf{0}$. Une solution plus pertinente est fournie en section 4.4.

La règle ci-dessus se lit comme suit. On parcourt toutes les transitions $q \xrightarrow[e_i]{pat_i} q_i$ qui sortent de q. On franchit directement les ϵ-transitions (cas $pat_i = \epsilon$, $\mathtt{bool}(\mathcal{E}\,[\![e_i]\!]\,\rho') = \mathbf{1}$), en rappelant récursivement le prédicat $\vdash_{\mathtt{q}\text{-}add}$ — ceci termine parce que la formule est ϵ-acyclique. Les ϵ-transitions infranchissables (cas $pat_i = \epsilon$, $\mathtt{bool}(\mathcal{E}\,[\![e_i]\!]\,\rho') = \mathbf{0}$) sont ignorées, exactement comme s'il s'agissait d'un état de rejet : en effet, ces transitions resteront à jamais infranchissables et ne mèneront jamais à un état d'acceptation, car le langage des expressions est fonctionnel et déterministe. Finalement, on empile les transitions qui restent (cas $pat_i \neq \epsilon$).

A noter que l'on incrémente le compteur correspondant à l'état q encore une fois ($\rho' = \rho[q{+}{+}]$).

4.3.2 Synchronisation, non-subsomption

$$\boxed{pid, lock \vdash_{sync} F, \rho \Rightarrow pid', lock'}$$

Ce jugement gère la synchronisation, c'est-à-dire les tests de non-subsomption.

Posons $F = (M, q, sync\text{-}vars, \lightning)$.

D'abord, si l'on a déjà une information de synchronisation, on la retourne. Autrement dit, si $lock \neq \bot$, on le retourne.

$$\frac{lock \neq \bot}{pid, lock \vdash_{sync} F, \rho \Rightarrow pid, lock} \tag{5}$$

Sinon, si $\mathrm{dom}\rho \supseteq sync\text{-}vars$, c'est que toutes les variables rigides de $sync\text{-}vars$ ont reçu une valeur, et on doit tester s'il existe ou non un thread dans l'une des files $\mathbf{q}_{\text{new}}$, $\mathbf{q}_{\text{retrig}}$ de même formule F et dont le champ $lock$ vaut exactement $\rho_{|sync\text{-}vars}$. Si non, alors on retourne $\rho_{|sync\text{-}vars}$ comme valeur du champ $lock$ du thread à créer, d'identificateur pid :

$$\frac{\mathrm{dom}\rho \supseteq \mathit{sync\text{-}vars} \quad \neg\exists(\star, F, \star, \star, \star, \star, \rho_{|sync\text{-}vars}) \in \mathbf{q}_{\mathrm{new}}@\mathbf{q}_{\mathrm{retrig}}}{pid, \bot \vdash_{sync} F, \rho \Rightarrow pid, \rho_{|sync\text{-}vars}} \tag{6}$$

La notation $\star$ représente ici un champ de valeur quelconque. La ligne $\neg\exists(\star, F, \star, \star, \star, \star, \rho_{|sync\text{-}vars}) \in \mathbf{q}_{\mathrm{new}}@\mathbf{q}_{\mathrm{retrig}}$ se lit : il n'y a pas de thread portant sur la formule F et ayant pour champ $lock$ exactement $\rho_{|sync\text{-}vars}$ dans la liste $\mathbf{q}_{\mathrm{new}}@\mathbf{q}_{\mathrm{retrig}}$.

S'il existe un thread, disons d'identificateur pid', qui subsume le thread courant d'identificateur pid, alors on a deux cas. Si $\lightning$ est vrai, alors on tue directement le thread pid. Sinon, on ne sait pas quel thread tuer (on le saura lorsqu'un des deux accepte). On retourne donc $\rho_{|sync\text{-}vars}$ comme valeur du champ $lock$ du thread à créer, comme plus haut, mais avec identificateur pid' et non pid. Ainsi le premier des deux threads qui acceptera tuera l'autre :

$$\frac{\lightning = \mathbf{0} \quad \mathrm{dom}\rho \supseteq \mathit{sync\text{-}vars} \quad (pid', F, \star, \star, \star, \star, \rho_{|sync\text{-}vars}) \in \mathbf{q}_{\mathrm{new}}@\mathbf{q}_{\mathrm{retrig}}}{pid, \bot \vdash_{sync} F, \rho \Rightarrow pid', \rho_{|sync\text{-}vars}} \tag{7}$$

La notation $\star$ représente encore une fois un champ de valeur quelconque. La condition $(pid', F, \star, \star, \star, \star, \rho_{|sync\text{-}vars}) \in \mathbf{q}_{\mathrm{new}}@\mathbf{q}_{\mathrm{retrig}}$ exprime qu'il existe un thread dans $\mathbf{q}_{\mathrm{new}}@\mathbf{q}_{\mathrm{retrig}}$ qui porte sur la formule F et qui a exactement $\rho_{|sync\text{-}vars}$ comme valeur du champ $lock$, et que pid' est son identificateur.

Noter que cette règle est en fait déterministe en pratique, même si elle est écrite dans un format qui la fait apparaître comme non-déterministe, car nécessairement l'usage de cette règle aura déjà fourni le même identificateur pid' à tous les threads sur la formule F et de même champ $lock$ égal à $\rho_{|sync\text{-}vars}$. (Pour rendre explicite le fait que cette règle est déterministe, il faudrait formellement restreindre Γ par un invariant exprimant que si deux threads dans $\mathbf{q}_{\mathrm{new}}@\mathbf{q}_{\mathrm{retrig}}$ portent sur la même formule et ont même champ $lock \neq \bot$, alors ils ont le même identificateur.)

4.3.3 Franchissement d'une transition

$$\boxed{R, kill, pid, \rho, lock \vdash_{trans} F, pat, e \Rightarrow pid', \rho', lock'}$$

Ce jugement exprime si l'on peut franchir une transition, et si oui quel environnement on obtient après l'avoir franchie. Pour ceci, on vérifie que le motif pat reconnaît l'événement R, que la garde e est vérifiée, que les contraintes de synchronisation (non-subsomption) sont vérifiées et que l'identificateur pid' du nouveau thread n'a pas été tué :

$$\frac{\begin{array}{c} R, \rho \models pat \Rightarrow \rho_1 \quad \rho' = \rho \oplus \rho_1 \neq \bot \quad \mathtt{bool}(\mathcal{E}\,[\![e]\!]\,\rho') = \mathbf{1} \\ pid, lock \vdash_{sync} F, \rho' \Rightarrow pid', lock' \\ pid' \notin kill \end{array}}{R, kill, pid, \rho, lock \vdash_{trans} F, pat, e \Rightarrow pid', \rho', lock'} \tag{8}$$

Cette règle ne s'applique que si pat est un motif, différent de ϵ. Noter aussi que le test $pid' \notin kill$ n'a pas à être effectué si $pid' = pid$ (un cas fréquent), car cette règle sera toujours appelée avec $pid \notin kill$.

4.3.4 Impossibilité de franchir une transition à jamais

$$\boxed{R, kill, pid, \rho, lock \vdash_{never} F, q, pat, e}$$

Ce jugement, a contrario, exprime que non seulement on ne peut pas franchir la transition, mais encore qu'on ne pourra jamais la franchir.

$$\frac{\Box(R, \rho, pat)}{R, kill, pid, \rho, lock \vdash_{never} F, q, pat, e} \tag{9}$$

Un deuxième cas se produit lorsque le motif pat reconnaît l'événement R, mais la garde e est fausse, et l'on peut de plus montrer que e ne fait que décroître en fonction des variables rigides monotones libres dans e.

$$\frac{\begin{array}{c} R, \rho \models pat \Rightarrow \rho_1 \quad \rho' = \rho \oplus \rho_1 \neq \bot \\ mono = \begin{cases} \emptyset & \text{si } pat = \epsilon \\ \texttt{monovars}(pat) & \text{sinon} \end{cases} \\ F = (M, q_0, sync\text{-}vars, \lightning) \quad def = \texttt{defined}(M, q_0)(q) \neq \top \\ \mathcal{M} = [x \mapsto \{\texttt{MONO}, \texttt{ANTI}\} | x \in def] \dagger [X \mapsto \{\texttt{MONO}\} | X \in mono] \\ \texttt{ANTI} \in \texttt{monotony}(\mathcal{M}, e) \end{array}}{R, kill, pid, \rho, lock \vdash_{never} F, q, pat, e} \tag{10}$$

Pour montrer que e ne peut que décroître, on vérifie que e évolue de façon antitone (décroît) au cours du log, en utilisant `monotony` (section 2.2.5), en supposant que :

– toutes les variables de $mono$, c'est-à-dire les variables liées à des champs monotones dans le motif pat, sont croissantes (`monovars` est définie en section 3.4) ;
– toutes les variables rigides dans def ne peuvent que rester constantes. La définition de `defined` et de $\top$ est en section 4.4, et on peut montrer qu'au cours de l'algorithme $\texttt{defined}(M, q_0)(q) \subseteq \mathrm{dom}\rho$. En particulier, on pourrait poser $def = \mathrm{dom}\rho \cap \mathcal{V}_{\text{rigid}}$ ci-dessus, mais ceci demanderait à effectuer un test à l'exécution ; avec la règle ci-dessus, on peut précompiler le résultat du test $\texttt{ANTI} \in \texttt{monotony}(\mathcal{M}, e)$ pour chaque transition.

On note de plus $[y \mapsto z | y \in A]$ la fonction partielle de domaine A qui à tout y de A associe z.

4.3.5 Traitement d'un thread

$$\boxed{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma', kill'}$$

Posons ici $\tau = (pid, F, q, t, \rho, flags, lock)$, $\delta(q) = (pat_1, e_1, q_1), \ldots, (pat_n, e_n, q_n)$, où δ est la fonction de transition de F.

Choix commis, cas où une transition est franchie. La règle suivante exprime que la $k$ième transition est franchie. Ceci demande que l'on puisse la franchir, bien sûr, mais aussi qu'on ne puisse franchir aucune des transitions précédentes dans la liste des transitions sortant de q.

$$\frac{\begin{array}{c} q \in Q^* \quad 1 \leq k \leq n \\ R, kill, pid, \rho, lock \vdash_{trans} F, pat_k, e_k \Rightarrow pid', \rho', lock' \\ \bigwedge_{i=1}^{k-1} \neg\exists \rho'', pid'', lock'' \cdot R, kill, pid, \rho, lock \vdash_{trans} F, pat_i, e_i \Rightarrow pid'', \rho'', lock'' \\ \Gamma, kill \vdash_{\mathtt{q}\text{-}add} F, q_k, \rho', pid', flags, lock' \Rightarrow \Gamma', kill' \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma', kill'} \tag{11}$$

Noter que comme $q \in Q^*$, toutes les transitions sortant de q sont étiquetées par un motif $pat_i \neq \epsilon$.

Choix commis, cas où aucune transition n'est franchie. A l'opposé, si aucune transition sortant d'un état de choix commis n'est franchissable, alors il y a deux possibilités. Soit $\texttt{ONLY\_ONCE} \in flags$ et l'on tue le thread, qui n'a pas réussi à franchir de transition sur l'événement courant :

$$\frac{\begin{array}{c} q \in Q^* \quad \texttt{ONLY\_ONCE} \in flags \\ \bigwedge_{i=1}^{n} \neg\exists \rho'', pid'', lock'' \cdot R, kill, pid, \rho, lock \vdash_{trans} F, pat_i, e_i \Rightarrow pid'', \rho'', lock'' \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma, kill} \tag{12}$$

Soit $\texttt{ONLY\_ONCE} \notin flags$, et on rempile q sur $\mathbf{q}_{\text{retrig}}$ pour attendre de trouver un événement ultérieur où une des transitions sortant de q pourra être franchie.

$$\frac{\begin{array}{c} q \in Q^* \quad \texttt{ONLY\_ONCE} \notin flags \\ \bigwedge_{i=1}^{n} \neg\exists \rho'', pid'', lock'' \cdot R, kill, pid, \rho, lock \vdash_{trans} F, pat_i, e_i \Rightarrow pid'', \rho'', lock'' \\ \Gamma' = \Gamma +_{\text{retrig}} (pid, F, q, 0, \rho, flags, lock) \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma', kill} \tag{13}$$

Autres états. Dans le cas des autres états, $\tau = (pid, F, q, t, \rho, flags, lock)$, alors $q \notin Q^*$, et $t \neq 0$ désigne un numéro de transition que l'on essaie de franchir. Nous convenons de noter $\delta(q)[t]$ la $t$ième transition (pat, e, q') de la liste $\delta(q)$.

Lorsque la transition numéro t peut être franchie, il y a deux cas, selon que $flags$ contient `RETRIGGER` ou non ; et si oui, selon que `ONLY_ONCE` est dans $flags$ ou non.

$$\frac{\begin{array}{c} q \notin Q^* \quad \delta(q)[t] = (pat, e, q') \\ R, kill, pid, \rho, lock \vdash_{trans} F, pat, e \Rightarrow pid', \rho', lock' \\ \Gamma, kill \vdash_{\mathtt{q}\text{-}add} F, q', \rho', pid', flags, lock' \Rightarrow \Gamma_1, kill' \\ \Gamma' = \begin{cases} \Gamma_1 +_{\text{retrig}} \tau & \text{si } \texttt{RETRIGGER} \in flags \wedge \texttt{ONLY\_ONCE} \notin flags \\ \Gamma_1 & \text{sinon} \end{cases} \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma', kill'} \tag{14}$$

Lorsque la transition numéro t ne peut pas être franchie, on considère de nouveau deux cas. En général, on doit réempiler le thread τ pour attendre un événement où la transition t sera franchissable ; sauf si l'on peut montrer qu'elle ne sera plus jamais franchissable, auquel cas on tue le thread. Ce dernier cas est traité dans la règle suivante :

$$\frac{\begin{array}{c} q \notin Q^* \quad \delta(q)[t] = (pat, e, q') \\ R, kill, pid, \rho, lock \vdash_{never} F, q, pat, e \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma, kill} \tag{15}$$

Sinon, le cas général consiste à réempiler le thread :

$$\frac{\begin{array}{c} q \notin Q^* \quad \delta(q)[t] = (pat, e, q') \\ \neg\exists pid', \rho', lock' \cdot R, kill, pid, \rho, lock \vdash_{trans} F, pat, e \Rightarrow pid', \rho', lock' \\ \neg(R, kill, pid, \rho, lock \vdash_{never} F, q, pat, e) \\ \Gamma' = \Gamma +_{\text{retrig}} \tau \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma', kill} \tag{16}$$

4.3.6 Initialisation avant traitement d'un événement

$$\boxed{\Gamma \vdash_{\mathbf{q}\text{-}init} \Rightarrow \mathbf{q}, alerts}$$

Rappelons que l'on a P formules F_1, …, F_P. On doit, à chaque nouvel événement dans le log, créer P threads, un pour chaque formule F_p, qui doivent vérifier si F_p est satisfiable par un itinéraire commençant à cet événement précis. Posons $F_p = (M_p, q_p, sync\text{-}vars_p, \lightning_p)$.

$$\frac{\begin{array}{c} \Gamma_0 = \Gamma \\ \Gamma_0, \emptyset \vdash_{\mathbf{q}\text{-}add} F_1, q_1, [], pid_1, \{\texttt{ONLY\_ONCE}\}, \bot \Rightarrow \Gamma'_1, \star \\ \Gamma_1 = \Gamma'_1 +_{\text{new}} \curvearrowright \\ \ldots \\ \Gamma_{P-1}, \emptyset \vdash_{\mathbf{q}\text{-}add} F_P, q_P, [], pid_P, \{\texttt{ONLY\_ONCE}\}, \bot \Rightarrow \Gamma'_P, \star \\ \Gamma_P = \Gamma'_P +_{\text{new}} \curvearrowright \\ pid_1, \ldots, pid_P \text{ frais et distincts deux à deux} \\ (\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{retrig}}, alerts) = \Gamma_P \end{array}}{\Gamma \vdash_{\mathbf{q}\text{-}init} \Rightarrow \mathbf{q}_{\text{new}} @ \mathbf{q}_{\text{retrig}}, alerts} \tag{17}$$

Noter que les jugements $\vdash_{\mathbf{q}\text{-}add}$ sont fournis avec $kill = \emptyset$, et on ignore le $kill$ retourné : comme les identificateurs pid_p, $1 \leq p \leq P$, sont frais, les threads correspondants ne peuvent pas être tués de toute façon.

4.3.7 Traitement des différents threads en attente

$$\boxed{R, \mathbf{q}, \Gamma, kill, newkill \vdash_{evt\text{-}loop} \Rightarrow \Gamma', newkill'}$$

Ce jugement traite en série de tous les threads de la liste $\mathbf{q}$, en appelant répétitivement des jugements $\vdash_{one\text{-}step}$. On doit tuer les threads d'identificateurs dans $kill$. Ce sont ceux qui ont été tués lors de l'examen de l'événement précédent dans le log. On collectionne dans $newkill$ les threads qui seront tués à l'examen de l'événement courant. De même, on empile dans Γ, Γ' les threads qu'il faudra relancer sur l'événement suivant, autrement dit ceux qui sont en attente.

D'abord, le cas où $\mathbf{q}$ est la liste vide $[]$:

$$\frac{}{R, [], \Gamma, kill, newkill \vdash_{\textit{evt-loop}} \Rightarrow \Gamma, newkill} \tag{18}$$

Ensuite le cas où le premier élément de la liste $\mathbf{q}$ est un thread, qui est traité par $\vdash_{\textit{one-step}}$. Il y a deux sous-cas, selon que le thread est marqué comme tué ou non :

$$\frac{\begin{array}{c}\tau = (pid, F, q, t, \rho, flags, lock) \quad pid \notin kill \\ R, \Gamma, newkill \vdash_{\textit{one-step}} \tau \Rightarrow \Gamma_1, newkill_1 \\ R, \mathbf{q}, \Gamma_1, kill, newkill_1 \vdash_{\textit{evt-loop}} \Rightarrow \Gamma', newkill'\end{array}}{R, [\tau]@\mathbf{q}, \Gamma, kill, newkill \vdash_{\textit{evt-loop}} \Rightarrow \Gamma', newkill'} \tag{19}$$

$$\frac{\begin{array}{c}\tau = (pid, F, q, t, \rho, flags, lock) \quad pid \in kill \\ R, \mathbf{q}, \Gamma, kill, newkill \vdash_{\textit{evt-loop}} \Rightarrow \Gamma', newkill'\end{array}}{R, [\tau]@\mathbf{q}, \Gamma, kill, newkill \vdash_{\textit{evt-loop}} \Rightarrow \Gamma', newkill'} \tag{20}$$

Finalement, on a le cas où le premier élément de $\mathbf{q}$ est l'élément spécial $\curvearrowright$, auquel cas on fusionne les files $\mathbf{q}_{\text{new}}$ et $\mathbf{q}_{\text{retrig}}$, en insérant des $\curvearrowright$ aux endroits appropriés.

$$\frac{\begin{array}{c}(\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{retrig}}, alerts) = \Gamma \\ \mathbf{q}'_{\text{new}} = \begin{cases} \mathbf{q}_{\text{new}} & \text{si } \mathbf{q}_{\text{new}} = [] \text{ ou } \mathbf{q}_{\text{new}} \text{ se termine par } \curvearrowright \\ \mathbf{q}_{\text{new}}@[\curvearrowright] & \text{sinon} \end{cases} \\ \mathbf{q}'_{\text{retrig}} = \begin{cases} \mathbf{q}_{\text{retrig}} & \text{si } \mathbf{q}_{\text{retrig}} = [] \text{ ou } \mathbf{q}_{\text{retrig}} \text{ se termine par } \curvearrowright \\ \mathbf{q}_{\text{retrig}}@[\curvearrowright] & \text{sinon} \end{cases} \\ \Gamma' = (\mathbf{q}'_{\text{new}}@\mathbf{q}'_{\text{retrig}}, [], alerts)\end{array}}{R, [\curvearrowright]@\mathbf{q}, \Gamma, kill, newkill \vdash_{\textit{evt-loop}} \Rightarrow \Gamma', newkill} \tag{21}$$

4.3.8 Traitement d'un événement

$$\boxed{R, \Gamma, kill \vdash_{\textit{one-evt}} \Rightarrow \Gamma', kill'}$$

On peut maintenant définir le jugement $R, \Gamma, kill \vdash_{\textit{one-evt}} \Rightarrow \Gamma', kill'$, qui examine l'événement R, sachant que les threads en attente et les alertes venant des événements précédents dans le log sont dans Γ, et les threads tués précédemment sont dans $kill$, et retourne les nouveaux threads en attente pour les événements suivants et enrichit l'ensemble des alertes dans Γ', et l'ensemble des nouveaux threads tués dans $kill'$:

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathbf{q}\textit{-init}} \Rightarrow \mathbf{q}, alerts \\ R, \mathbf{q}, ([], [], alerts), kill, \emptyset \vdash_{\textit{evt-loop}} \Rightarrow \Gamma', kill'\end{array}}{R, \Gamma, kill \vdash_{\textit{one-evt}} \Rightarrow \Gamma', kill'} \tag{22}$$

4.3.9 Analyse de logs

$\boxed{S, \Gamma, kill \vdash_{log} \Gamma', kill'}$

Il ne reste plus qu'à enchaîner les appels à $\vdash_{one\text{-}evt}$ pour analyser un log fini S. Noter qu'un log fini est soit le log vide [], soit de la forme $[R]@S'$, où R est le premier événement de S.

$$\frac{S = []}{S, \Gamma, kill \vdash_{log} \Gamma, kill} \tag{23}$$

$$\frac{S = [R]@S' \quad R, \Gamma, kill \vdash_{one\text{-}evt} \Gamma_1, kill_1 \quad S', \Gamma_1, kill_1 \vdash_{log} \Gamma', kill'}{S, \Gamma, kill \vdash_{log} \Gamma', kill'} \tag{24}$$

$\boxed{\mathcal{A}[\![S]\!]}$

$\mathcal{A}[\![S]\!]$—textbf

On définit maintenant l'ensemble des alertes engendrées par le log S dans le contexte des formules F_1, ..., F_P. Ceci sera défini dans le cas général où S peut être un log infini :

- si S est un log fini, alors $\mathcal{A}[\![S]\!] = \{\rho | \rho \in alerts, S, ([], [], \emptyset), \emptyset \vdash_{log} (\star, \star, alerts), \star\}$;
- si S est un log infini, alors $\mathcal{A}[\![S]\!] = \bigcup_{k \in \mathbb{N}} \mathcal{A}[\![S]\!]_{|k}$, où $S_{|k}$ est le préfixe fini de S consistant en les événements $S_1, \ldots, S_k$.

4.4 Fonctions d'analyse statique

Variables rigides définies en un état. On définit en premier une fonction $\texttt{defined}(M, q_0)(q)$ qui à toute machine M d'état initial q_0 et à tout état q de M associe l'ensemble des variables rigides qui seront définies (dans le domaine de ρ) de tout thread aboutissant à l'état q, reconnaissant une formule $F = (M, q_0, sync\text{-}vars, \lightning)$.

Soit $\top$ un nouveau symbole, et posons $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$. Précisément, $\texttt{defined}(M, q_0)$ est une fonction de Q vers $\mathbb{P}(\mathcal{V}) \cup \{\top\}$ définie comme suit.

Étendons la définition de $\subseteq$ à $\top$ par : $X \subseteq \top$ pour tout $X \in \mathbb{P}(\mathcal{V}) \cup \{\top\}$. Pour toute paire de fonctions $f, g : Q \to \mathbb{P}(\mathcal{V}) \cup \{\top\}$, posons $f \subseteq^* g$ si et seulement si $f(q) \subseteq g(q)$ pour tout $q \in Q$.

Alors $\texttt{defined}(M, q_0)$ est la plus grande fonction f, pour $\subseteq^*$, telle que :

- $f(q_0) = \emptyset$ (initialement, aucune variable n'est définie) ;
- pour toute transition $q \xrightarrow[e]{\epsilon} q'$, $f(q') \subseteq f(q)$ (toute variable définie en q' l'est déjà en q) ;
- pour toute transition $q \xrightarrow[e]{pat} q'$, avec $pat \neq \epsilon$, $f(q') \subseteq f(q) \cup bound \cap \mathcal{V}_{\text{rigid}}$, où $(free, bound) = \texttt{pat\_vars}(pat)$ (toute variable définie en q' est liée par pat ou bien définie en q).

Variables rigides vivantes. La fonction $\texttt{live}(F)(q)$ calcule l'ensemble des variables rigides *vivantes* à l'état q de la formule F, c'est-à-dire celles qui sont libres ou liées dans au moins une transition accessible depuis q dans F. Si f et g sont deux fonctions de Q vers $\mathbb{P}(\mathcal{V}_{\text{rigid}})$, où Q est l'ensemble des états de F, définissons $f \subseteq^* g$ par $f(q) \subseteq g(q)$ pour tout $q \in Q$. Posons $F = (M, q_0, \mathit{sync\text{-}vars}, \lightning)$, $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$. Alors $\texttt{live}(F)$ est la plus petite fonction $f : Q \to \mathbb{P}(\mathcal{V}_{\text{rigid}})$ telle que :

- $f(q_{\text{acc}}) = f(q_{\text{rej}}) = \emptyset$;
- pour toute transition $q \xrightarrow[e]{\epsilon} q'$ dans M, $f(q') \cup (\text{fv}(e) \cap \mathcal{V}_{\text{rigid}}) \subseteq f(q)$;
- pour toute transition $q \xrightarrow[e]{pat} q'$ dans M où $pat \neq \epsilon$, $f(q') \cup ((\mathit{free} \cup \mathit{bound} \cup \text{fv}(e)) \cap \mathcal{V}_{\text{rigid}}) \subseteq f(q)$, où $(\mathit{free}, \mathit{bound}) = \texttt{pat\_vars}(pat)$.

La fonction $\texttt{live\_trans}(F, pat, e, q')$, où $q \xrightarrow[e]{pat} q'$ est une transition de F, calcule l'ensemble des variables rigides qui sont *vivantes* à cette transition, c'est-à-dire $\texttt{live}(F)(q') \cup ((\mathit{free} \cup \mathit{bound} \cup \text{fv}(e)) \cap \mathcal{V}_{\text{rigid}})$, où $(\mathit{free}, \mathit{bound}) = \texttt{pat\_vars}(pat)$ si $pat \neq \epsilon$, et $\mathit{free} = \mathit{bound} = \emptyset$ sinon.

Variables flexibles utilisées de façon antitone. Soit F une formule, def un ensemble de variables rigides.

On définit l'ensemble $\texttt{used\_antitonically}(F, def)$ de tous les couples (q, mono) où q est un état de F et mono un ensemble fini de variables flexibles, tels que si l'on diminue toutes les valeurs des variables de mono, les transitions à franchir en partant de q restent franchissables. En l'absence d'états de choix commis, ceci revient à demander que, sur tout chemin dans la machine sous-jacente à F, partant de q, toute garde que l'on atteint est antitone en les variables de mono — en tous cas, celles qui n'ont pas été redéfinies par un motif le long du chemin. En présence d'états de choix commis, on doit de plus vérifier que les mêmes choix seront effectués même après avoir diminué les valeurs des variables de mono, par exemple en demandant que les gardes suivant l'état de choix commis soient constantes — sauf possiblement la dernière garde.

Posons $F = (M, q_0, \mathit{sync\text{-}vars}, \lightning)$, $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$. Alors $\texttt{used\_antitonically}(F, def)$ est le plus grand ensemble tel que :

- pour tout état $q \in Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}$, et tout ensemble fini $\mathit{mono} \neq \emptyset$ de variables flexibles, posons :
  - $\delta(q) = (pat_1, e_1, q_1), \ldots, (pat_n, e_n, q_n)$;
  - $\mathit{mono}_i = \begin{cases} \mathit{mono} & \text{si } pat_i = \epsilon \\ \mathit{mono} \setminus \texttt{pat\_vars}(pat_i) & \text{sinon} \end{cases}$;
  - $\mathcal{M}_i = [x \mapsto \{\texttt{MONO}, \texttt{ANTI}\} | x \in def] \dagger [X \mapsto \{\texttt{MONO}\} | X \in \mathit{mono}_i]$.

  alors :
  - si $\texttt{ANTI} \notin \texttt{monotony}(\mathcal{M}_i, e_i)$ pour un i, $1 \leq i \leq n$, alors $(q, \mathit{mono}) \notin \texttt{used\_antitonically}(F, def)$;
  - si $(q_i, \mathit{mono}_i) \notin \texttt{used\_antitonically}(F, def)$, alors $(q, \mathit{mono}) \notin \texttt{used\_antitonically}(F, def)$;
  - si $q \in Q^*$ et $\texttt{MONO} \notin \texttt{monotony}(\mathcal{M}_i, e_i)$ pour un i, $1 \leq i \leq n-1$, alors $(q, \mathit{mono}) \notin \texttt{used\_antitonically}(F, def)$.

Noter que l'on teste que toute variable flexible est utilisée de façon antitone par les gardes de transitions accessibles, mais l'on n'a posé aucune condition sur les expressions e_i intervenant dans les motifs $\textit{cmp-pat} = i, var, [\textit{cmp-op}_1, e_1, \ldots, \textit{cmp-op}_n, e_n]$. La raison en est que par définition aucune variable flexible n'est libre dans les e_i, voir section 3.2.

Test d'inutilité du backtracking. Le prédicat $\texttt{no\_backtrack\_needed}(F, q, i, \rho)$ retourne **1** si, lorsque l'on a pu franchir la $i$ième transition sortant de q dans F, fournissant un environnement ρ, alors c'est qu'elle *doit* être franchie pour fournir un itinéraire minimal. L'idée est que, entre deux itinéraires, l'un franchissant la transition i maintenant, l'autre attendant un événement ultérieur du log, c'est toujours le premier qui est minimal. La difficulté est qu'on ne sait pas si le fait de franchir la transition numéro i mène nécessairement à un itinéraire : il se peut qu'aucun thread ayant franchi cette transition n'accepte jamais. On va donc vérifier que, s'il existe un itinéraire (de domaine $i_k < i_{k+1} \ldots < i_q$ avec $i_k > i$) ne franchissant pas la transition i, alors il en existe aussi un qui la franchit (plus précisément, de domaine $i < i_{k+1} < \ldots < i_q$). On demande donc que l'on puisse remplacer les valeurs des variables obtenues lors d'un test de motif sur un événement ultérieur $S_{i_{k+1}}$ dans le log S, par les valeurs obtenues lors du même test de motif sur l'événement courant S_i. Ceci réussit si d'une part toutes les variables rigides dont on aura jamais besoin dans la suite de l'exécution du thread (calculé par `live_trans`) sont déjà définies (dans le domaine de ρ), et si d'autre part toutes les variables flexibles liées par le motif pat de la transition i sont liées à des champs monotones, et les gardes dépendant de ces variables flexibles sont antitones (testé par `used_antitonically`).

On pose donc $\texttt{no\_backtrack\_needed}(F, q, i, \rho) = \mathbf{1}$ si et seulement si $bound \cap \mathcal{V}_{\text{flex}} \subseteq mono$, $(q', mono) \in \texttt{used\_antitonically}(F, def)$, et $\texttt{live\_trans}(F, pat, e, q') \subseteq \text{dom}\rho$, où :

- $F = (M, q_0, \textit{sync-vars}, \lightning)$, $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$, $\delta(q)[i] = (pat, e, q')$, $pat \neq \epsilon$;
- $def = \texttt{defined}(M, q_0)(q) \neq \top$;
- $mono = \texttt{monovars}(pat)$;
- $(free, bound) = \texttt{pat\_vars}(pat)$.

A noter que ce test n'est pas entièrement statique, à cause de l'utilisation de ρ. Un test entièrement statique, donc plus efficace à l'exécution, mais possiblement moins précis, consisterait à tester non pas $\texttt{live\_trans}(F, pat, e, q') \subseteq \text{dom}\rho$ mais $\texttt{live\_trans}(F, pat, e, q') \subseteq \texttt{defined}(M, q_0)(q')$.

4.5 Restrictions syntaxiques sur les formules

Il est nécessaire de restreindre les formules à des formules dites *bien formées*. Ceci a pour but non seulement d'avertir l'utilisateur en cas de formule visiblement pathologique, mais surtout de garantir la correction des optimisations de l'algorithme reposant sur une analyse statique de la formule. Par exemple, la définition de $\Box(R, \varrho, \textit{cmp-pat})$ (section 3.3) n'est correcte que modulo la restriction de la section 4.5.1.

4.5.1 Toute variable rigide utilisée doit être définie

On impose que dans toute transition $q \xrightarrow[e]{pat} q'$, toute variable rigide libre dans pat ou dans e ait nécessairement été définie auparavant. Ceci est notamment important pour la correction du prédicat $\Box(R, \varrho, pat)$ (section 3.2).

Formellement, on requiert que dans toute formule $F = (M, q_0, sync\text{-}vars, \lightning)$, $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$, pour toute transition $q \xrightarrow[e]{pat} q'$ de M telle que $\texttt{defined}(M, q_0)(q) \neq \top$:

- si $pat \neq \epsilon$, alors $free \subseteq \texttt{defined}(M, q_0)(q)$, où $(free, bound) = \texttt{pat\_vars}(pat)$;
- $\text{fv}(e) \cap \mathcal{V}_{\text{rigid}} \subseteq def$, où $def = \texttt{defined}(M, q_0)(q) \cup bound$ et $bound = \emptyset$ si $pat = \epsilon$, $(free, bound) = \texttt{pat\_vars}(pat)$ sinon.

Noter que si $\texttt{defined}(M, q_0), (q)$, q est un état inaccessible de M.

Références

[Christiansen et al., 2000] Christiansen, T., Wall, L., and Orwant, J. (2000). *Programming Perl*. O'Reilly and Associates, 3rd edition.

[Goubault-Larrecq, 2001] Goubault-Larrecq, J. (2001). *An Introduction to* log$\mathcal{W}$eaver *(v2.8)*. GIE Dyade & LSV. Disponible sur la page Web du projet DICO au LSV, `http://www.lsv.ens-cachan.fr/~goubault/DICO.html`.

[Goubault-Larrecq et al., 2002] Goubault-Larrecq, J., Pouzol, J.-P., Demri, S., Mé, L., and Carle, P. (2002). Langages de détection d'attaques par signatures. Sous-projet 3, livrable 1 du projet RNTL DICO. Version 1. 30 pages.

[Roger and Goubault-Larrecq, 2001] Roger, M. and Goubault-Larrecq, J. (2001). Log auditing through model checking. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW'01), Cape Breton, Nova Scotia, Canada, June 2001*, pages 220–236. IEEE Comp. Soc. Press.

[Spencer, 1986] Spencer, H. (1986). Regexp package. Available at `http://arglist.com/regex/`.

Index