RAPPORT TECHNIQUE EVA

Les syntaxes et la sémantique du langage de spécification
EVA

Date : 18 Novembre 2002
Auteurs : Jean Goubault-Larrecq

Titre : Les syntaxes et la sémantique du langage de spécification EVA

Rapport No. / Version : 3/ 8

TRUSTED LOGIC S.A.
5 rue du Bailliage
78000 Versailles, France
www.trusted-logic.fr

Laboratoire Spécification Vérification
CNRS UMR 8643, ENS Cachan
61, avenue du président-Wilson
94235 Cachan Cedex, France
www.lsv.ens-cachan.fr

Laboratoire Verimag
CNRS UMR 5104,
Univ. Joseph Fourier, INPG
2 av. de Vignate,
38610 Gières, France
www-verimag.imag.fr

Contents

1 Introduction

Ce document rapide a pour but de définir les syntaxes concrètes et abstraites, ainsi que la sémantique, du langage du projet EVA. Il fait suite à un document [JLM01] sur la syntaxe concrète et un autre [GL01] sur la syntaxe abstraite et sa sémantique.

2 Format d'échange

Les différents outils du projet s'échangent les informations par des fichiers textuels, dont le contenu sont des termes du premier ordre écrits dans une syntaxe lispienne. C'est le cas notamment des fichiers en `.evl` contenant les arbres syntaxiques produits par l'analyseur syntaxique `evaparse` et pris en entrée par le traducteur `evatrans`. (L'analyse syntaxique est décrite en section 3.) C'est aussi le cas des fichiers d'extension `.cpl` contenant la syntaxe abstraite EVA, sur laquelle est définie la sémantique, des protocoles traduits par l'outil `evatrans`. (Cette syntaxe abstraite est définie en section 4, la sémantique étant le sujet de la section 5.) L'architecture actuelle des outils EVA est donnée en figure 1.

Les termes du premier ordre sont soit des variables, soit des constantes entières, soit des applications.

On notera dans la suite les termes du premier ordre comme suit. Une variable, par exemple de nom "abc", sera notée '`abc`'. Un entier, par exemple 12, sera noté en décimal. Une application d'un symbole f à une liste d'arguments $t_1, \ldots, t_n$, sera notée $f(t_1, \ldots, t_n)$. En particulier, une constante (le cas $n = 0$) sera notée $f()$. Les symboles de fonctions sont des suites non vides de caractères quelconques, sauf '`"`',

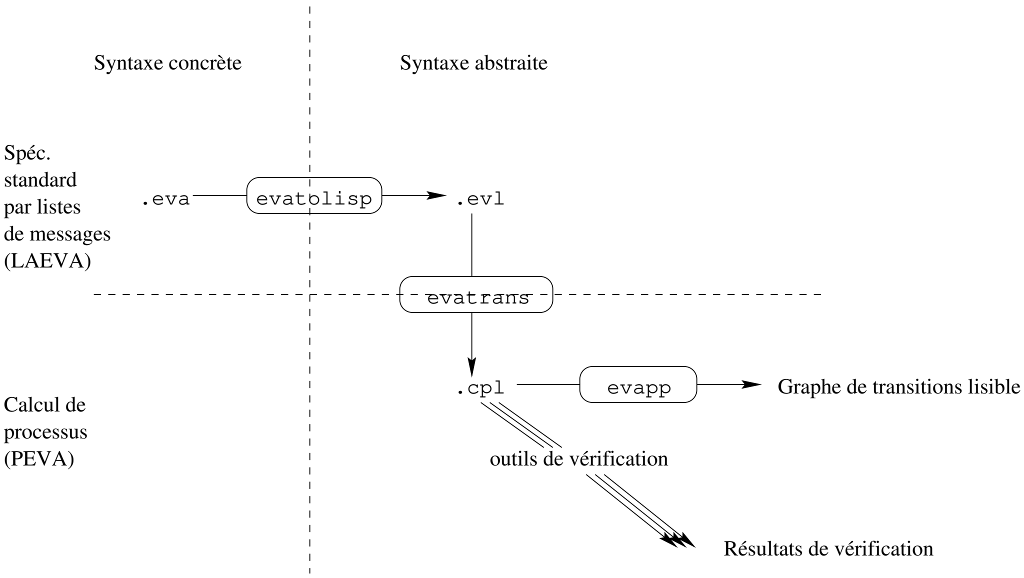

Figure 1: L'architecture EVA

l'espace, la tabulation, le retour chariot, le saut de ligne, et les parenthèses ouvrantes et fermantes. De plus, le symbole spécial '`line`' est réservé.

Un terme Lisp est soit une variable, représentée par la chaîne des caractères de son nom entre guillemets (les caractères spéciaux et les guillemets étant notées avec la notation C classique utilisant le backslash \), soit une constante entière, notée en décimal, soit une application d'un symbole f à n termes Lisp $t_1, \ldots, t_n$, notée `(`f t_1 `...` t_n`)`, soit une information de position textuelle `(line` t ℓ_1 n_1 ℓ_2 n_2 N`)`, où t est un terme Lisp, ℓ_1, n_1, ℓ_2, n_2 sont des constantes entières, et N est une chaîne de caractères. Cette dernière notation indique que le terme t correspond à une information issue du fichier de nom N (pour peu que N ne soit pas la chaîne vide, auquel cas les autres paramètres entiers sont sans signification), comprise entre les positions décrites par la ligne ℓ_1 et la colonne n_1 d'une part, et la ligne ℓ_2 et la colonne n_2 d'autre part. Les lignes sont numérotées à partir de 1, les colonnes à partir de 0.

La relation entre termes Lisp et termes du premier d'ordre est donnée par une fonction d'effacement E qui efface les appels à `line`:

$$\begin{array}{rcll} E(x) & = & x & \text{(identificateurs)} \\ E(n) & = & n & \text{(constantes entières)} \\ E((f\ t_1\ \ldots\ t_n)) & = & f(E(t_1), \ldots, E(t_n)) & \\ E((\texttt{line}\ t\ \ell_1\ n_1\ \ell_2\ n_2\ N)) & = & E(t) & \end{array}$$

La bibliothèque `lisp.cmxa` est une bibliothèque OCaml permettant de lire et d'écrire des termes Lispiens. Le type des termes lispiens est défini dans `lisp.mli` comme suit:

```
type term = VAR of string
          | INT of int
          | APP of string * term list
          | LINE of term * int * int * int * int * string;;
```

La fonction de lecture `lispread` est décrite dans `lispread.mli` comme suit:

```
exception LispNoFunction
exception LispUnexpectedFunction of string
exception LispRParen
```

```
val lispread : in_channel -> Term.term
```

Les exceptions mentionnées sont lancées en cas d'erreur de formattage du fichier lu: `LispNoFunction` lorsqu'une parenthèse ouvrante n'est pas suivie d'un symbole de fonction, `LispUnexpectedFunction` lorsqu'un symbole de fonction (passé en paramètre) a été découvert non précédé d'une parenthèse ouvrante, `LispRParen` lorsqu'une parenthèse fermante superflue a été rencontrée. Finalement, la fonction

```
lispwrite : in_channel -> Term.term -> unit
```

est fournie qui écrit un terme lispien dans un fichier.

De même, la bibliothèque `lispin.mlx` est une bibliothèque HimML remplissant le même but. Le type des termes lispiens est défini dans `term_h.ml` comme suit:

```
type lineinfo = |[ first_line : int,
                   first_column : int,
                   last_line : int,
                   last_column : int,
                   in_file : string
                   ]|;

datatype term = VAR of string
       | INT of int
       | APP of string * term list
       | LINE of term * lineinfo;
```

Le fichier `lispin_h.ml` définit ensuite les fonctions `lispread` et `lispwrite`, éponymes des fonctions ci-dessus, par:

```
type 'a inlnstream = |[ get : int -> string,
                        getline : unit -> string,
                        ... : 'a ]|;

extern val lispread : 'a inlnstream -> term;
extern val lispwrite : 'a outstream -> term -> unit;
```

Noter qu'on aurait pu aussi choisir un format XML ici. L'inter-traduction ne semble pas devoir demander beaucoup d'efforts.

3 Syntaxe concrète

Il s'agit de la syntaxe comprise par l'outil `evatolisp`, qui est un shell-script construit au-dessus du programme C `evatrans`. Cet outil a pour but d'effectuer l'analyse syntaxique de fichiers EVA, en langage LAEVA et d'extension standard `.eva`, et de ressortir un arbre syntaxique, dans une syntaxe lispienne. Ce shell-script appelle le préprocesseur C `/lib/cpp/` dans le but de fournir l'inclusion de fichiers (par `#include`) et les macros (par `#define`).

3.1 Lexèmes

id Un *identificateur* est une suite non vide de caractères alpha-numériques (de `A` à `Z`, de `a` à `z`, de `0` à `9`, ou `_`), dont le premier est une lettre (entre `A` et `Z` ou entre `a` et `z`). Les identificateurs dénotent typiquement les variables, les noms de principaux.

int Un *entier littéral* est une suite non vide de chiffres entre 0 et 9.

label Une *étiquette* est un identificateur ou un entier suivi immédiatement d'un point, sans aucun autre caractère entre les deux. Les étiquettes servent à dénoter les étapes dans un protocole.

string Une *chaîne littérale* est une suite de caractères entre guillemets `"`. Le backslask `\` est un caractère spécial, et sert à former des suites de caractères difficiles à écrire autrement: `\n` est un saut de ligne (ASCII 10), `\t` une tabulation (ASCII 9), `\r` un retour chariot (ASCII 13), `\b` un son de cloche (ASCII 7), `\f` un saut de page (ASCII 12). De plus, un backslash suivi de 1 à 3 chiffres octals dénote le code ASCII correspondant en octal; par exemple, `\012` est synonyme de `\n`. À part cela, un backslash suivi d'un caractère représente ce caractère, de sorte que `\"` dénote le caractère guillemet.

Les commentaires, dans le style de ceux de C++ ou de Java, sont introduits par les symboles `//` (commentaire se poursuivant jusqu'en bout de ligne) ou `/*` (commentaire se poursuivant jusqu'au prochain `*/`, sans imbrication). Ceci est géré par le préprocesseur C `/lib/cpp`, qui est appelé par `evatolisp`.

Les mots-clés réservés sont:

```
alias      assume   axiom    basetype  case    claim
everybody  hash     keypair  knows     secret  session
shared     switch
->         =>       ==       :=        &&      ||
*E         *A       *U       *W        *S      *B
*F         *G       *P       *Q        *N
:          (        )        ,         =       _
{          }        [        ]         %       ^
*          !        @        false     true
```

plus le symbole spécial `eof` dénotant la fin de fichier.

<table>
<tr><td>‘,’</td><td>associatif à droite</td></tr>
<tr><td>‘=>’</td><td>associatif à droite</td></tr>
<tr><td>‘||’ ‘*U’ ‘*W’ ‘*S’ ‘*B’</td><td>associatif à droite</td></tr>
<tr><td>‘&&’</td><td>associatif à droite</td></tr>
<tr><td>‘!’ ‘*E’ ‘*A’ ‘*F’ ‘*G’ ‘*P’ ‘*Q’ ‘*N’</td><td>non associatifs</td></tr>
</table>

Figure 2: Priorités

Les priorités sont données en figure 2, qui liste les opérateurs de plus forte priorité — ceux qui lient plus fort — en bas.

3.2 La grammaire concrète

La syntaxe concrète des fichiers de spécification EVA est donnée par la grammaire qui suit. On utilise la notation $[X]$ pour signifier toute chaîne reconnue par X ou la chaîne vide, X^* pour dénoter toute concaténation, possiblement vide, de chaînes reconnues par X, et X^+ pour toute concaténation non vide. La notation ‘`abc`’ dénote le littéral correspondant (ici, `abc`). Les espaces, tabulations, sauts de lignes et de pages, retours chariots sont ignorés.

<table>
<tr><td>spec</td><td>::=</td><td>id declaration* block session*
assumptions claims eof</td><td>spécification EVA</td></tr>
</table>

declaration	::=	typing-declaration	déclaration de types
	|	‘keypair’ [encryption-algorithm] id ‘,’ id [‘(’ type-list ‘)’]	déclaration de paires de clés asymétriques
	|	‘alias’ id ‘=’ term	déclaration d’alias global
	|	ids ‘: alias’ id ‘=’ term	déclaration d’alias locaux
	|	‘basetype’ id	déclaration de type de base
	|	id ‘knows’ tuple	connaissances initiales
	|	‘everybody’ ‘knows’ tuple	connaissances initiales communes
	|	‘axiom’ term ‘=’ term [quantification]	axiome global
typing-declaration	::=	ids ‘:’ type [‘shared’]	types d’identificateurs
	|	id ‘(’ type-list ‘):’ type [‘hash’]	types de fonctions
quantification	::=	‘[’ typing-declarations ‘]’	quantification universelle
typing-declarations	::=	typing-declaration (‘,’ typing-declaration)*	

ids	::=	id (‘,’ id)*	liste d’identificateurs (non vide)
type	::=	id	type non fonctionnel
type-list	::=	[non-empty-type-list]	liste de types
non-empty-type-list	::=	type (‘,’ type)*	liste de types non vide

block	::=	‘{’ message* ‘}’	bloc: protocole ou sous-protocole
message	::=	label id ‘->’ id ‘:’ term	envoi/réception de message
	|	id ‘:’ id equals-or-assign-op atomic-term-or-ciphertext	test d’égalité/affectation
	|	block	sous-protocole
	|	‘switch’ term ‘{’ case* ‘}’	conditionnelle
equals-or-assign-op	::=	‘==’ | ‘:=’	
case	::=	‘case’ term ‘:’ message*	branche de conditionnelle

term	::=	tuple	terme
tuple	::=	atomic-term-or-ciphertext (‘,’ atomic-term-or-ciphertext)*	n-uplet
term-list	::=	‘(’ [tuple] ‘)’	liste parenthésée de termes
atomic-term-or-ciphertext	::=	atomic-term	terme atomique
	|	‘{’ term ‘}_’ atomic-term [encryption-algorithm]	chiffrement
	|	‘{’ term ‘}’ encryption-algorithm ‘_’ atomic-term	
	|	‘[’ term ‘]_’ atomic-term [encryption-algorithm]	signature
	|	‘[’ term ‘]’ encryption-algorithm ‘_’ atomic-term	
	|	atomic-term ‘%’ atomic-term	schizo-notation de Lowe
	|	atomic-term-or-ciphertext ‘@’ label id	terme placé
atomic-term	::=	id term-list	appel de fonction
	|	id	identificateur
	|	‘(’ term ‘)’	
	|	‘()’	n-uplet vide
encryption-algorithm	::=	‘^’ atomic-term	invocation d’algorithme

session	::=	[label] ‘session’ [‘*’ [‘{’ [ids] ‘}’]] [principals] assignments	instantiation de session
principals	::=	‘[’ [‘^’] ids ‘]’	liste de principaux inclus/exclus
assignments	::=	assignment (‘,’ assignment)*	substitution
assignment	::=	id ‘=’ atomic-term-or-ciphertext	liaison de valeur

assumptions	::=	‘assume’ properties	hypothèses
claims	::=	‘claim’ properties	propriétés à prouver
properties	::=	property (‘,’ property)*	liste de propriétés
property	::=	id term-list	proposition atomique
	|	label id ‘@’ label	point de programme
	|	label id ‘(’ term ‘)’	origine de message
	|	‘secret (’ term ‘)’	secret de message
	|	‘true’	vrai
	|	‘false’	faux
	|	property ‘&&’ property	conjonction
	|	property ‘||’ property	disjonction
	|	property ‘=>’ property	implication
	|	‘!’ property	négation
	|	‘(’ property ‘)’	
	|	‘*E’ property	sur un chemin
	|	‘*A’ property	sur tous les chemins
	|	‘*F’ property	à un instant futur
	|	‘*G’ property	à tous les instants futurs
	|	‘*P’ property	à un instant passé
	|	‘*Q’ property	à tous les instants passés
	|	‘*N’ property	dorénavant
	|	property ‘*U’ property	until
	|	property ‘*W’ property	waiting-for (until faible)
	|	property ‘*S’ property	since
	|	property ‘*B’ property	before (since faible)

3.3 Les arbres de syntaxe concrète

Les spécifications concrètes EVA (`.eva`) obéissant à cette syntaxe sont traduites par `evatolisp` en un terme du premier ordre (écrit en syntaxe lispienne dans un fichier d’extension `.evl`).

Spec ::= `spec` (Declarations, Block, Sessions, Assumptions, Claims)

Declarations	::=	`declare` (Declaration*)
Declaration	::=	Typing-declaration
	|	`keypair` (Encryption-algorithm, id, id, Type)
	|	`alias` (id, Term)
	|	`local-alias` (id, Term, id*)
	|	`basetype` (id)
	|	`knows` (id, Term)
	|	`everybody-knows` (Term)
	|	`axiom` (Term, Term, Typing-declaration*)
Typing-declaration	::=	`type` (id, Type)
Type	::=	id
	|	`function` (id, id*)
	|	`one-way-function` (id, id*)
	|	`secret-function` (id, id*)
	|	`secret-one-way-function` (id, id*)

$$
\begin{array}{rcl}
\text{Block} & ::= & \texttt{block}\,(\text{Message}^*) \\
\text{Message} & ::= & \texttt{comm}\,(\text{label, id, id, Term}) \\
& | & \texttt{equals}\,(\text{id, id, Term}) \\
& | & \texttt{assign}\,(\text{id, id, Term}) \\
& | & \text{Block} \\
& | & \texttt{switch}\,(\text{Term, Case}^*) \\
\text{Case} & ::= & \texttt{case}\,(\text{Term, Message}^*) \\
\text{Term} & ::= & \text{id} \\
& | & \texttt{tuple}\,(\text{Term}^*) \\
& | & \texttt{crypt}\,(\text{Term, Term, Encryption-algorithm}) \\
& | & \texttt{sign}\,(\text{Term, Term, Encryption-algorithm}) \\
& | & \texttt{percent}\,(\text{Term, Term}) \\
& | & \texttt{locate}\,(\text{label, id, Term}) \\
& | & \texttt{apply}\,(\text{id, Term}^*) \\
\text{Encryption-algorithm} & ::= & \texttt{vanilla}\,() \\
& | & \text{Term}
\end{array}
$$

$$
\begin{array}{rcl}
\text{Sessions} & ::= & \texttt{sessions}\,(\text{Session}^*) \\
\text{Session} & ::= & \texttt{session}\,(\text{label, Principals, Assignment}^*) \\
& | & \texttt{session-repeat}\,(\text{Private, label, Principals, Assignment}^*) \\
\text{Private} & ::= & \texttt{private}\,(\text{id}^*) \\
\text{Principals} & ::= & \texttt{principals-only}\,(\text{id}^*) \\
& | & \texttt{principals-except}\,(\text{id}^*) \\
\text{Assignment} & ::= & \texttt{bind}\,(\text{id, Term})
\end{array}
$$

$$
\begin{array}{rcl}
\text{Assumptions} & ::= & \texttt{assume}\,(\text{Property}^*) \\
\text{Claims} & ::= & \texttt{claim}\,(\text{Property}^*) \\
\text{Property} & ::= & \texttt{prop}\,(\text{id, Term}^*) \\
& | & \texttt{at}\,(\text{label, id, label}) \\
& | & \texttt{owns}\,(\text{label, id, Term}) \\
& | & \texttt{secret}\,(\text{Term}) \\
& | & \texttt{and}\,(\text{Property}^*) \\
& | & \texttt{or}\,(\text{Property}^*) \\
& | & \texttt{implies}\,(\text{Property, Property}) \\
& | & \texttt{not}\,(\text{Property}) \\
& | & \texttt{E}\,(\text{Property}) \\
& | & \texttt{A}\,(\text{Property}) \\
& | & \texttt{F}\,(\text{Property}) \\
& | & \texttt{G}\,(\text{Property}) \\
& | & \texttt{P}\,(\text{Property}) \\
& | & \texttt{Q}\,(\text{Property}) \\
& | & \texttt{N}\,(\text{Property}) \\
& | & \texttt{U}\,(\text{Property, Property}) \\
& | & \texttt{W}\,(\text{Property, Property}) \\
& | & \texttt{S}\,(\text{Property, Property}) \\
& | & \texttt{B}\,(\text{Property, Property})
\end{array}
$$

3.4 Analyse syntaxique

La correspondance entre syntaxe concrète et arbres de syntaxe concrète est décrite par la série de règles de grammaires suivante, écrite dans le style de `yacc`. Cette série de règles reprend et enrichit la grammaire de la section 3.2. La notation ϵ dénote la suite vide; la virgule (par exemple dans \$1, \$3) dénote la concaténation de suites, sachant que tout terme est implicitement converti en une liste; si un terminal ou un non-terminal apparaît en position n en partie droite de règle, sa valeur est $\$n$; s'il dénote une liste (par exemple s'il apparaît sous une astérisque), alors $\$n^*$ dénote la suite de toutes les valeurs obtenus par le terminal ou le non-terminal dans la liste. La notation $(\ldots)_{x \in \$n^*}$ dénote la suite des $\ldots$ lorsque x parcourt (dans l'ordre) les éléments de la suite $$n^*$.

Les règles d'analyse syntaxique sont alors donnés sous la forme illustrée par la règle principale pour spec ci-dessous; chaque côté droit d'une règle de grammaire est associé à un calcul de \$\$, la valeur du côté gauche, sous forme d'un terme construit à partir des valeurs \$1, \$2, …, des terminaux et non-terminaux du côté droit.

spec	::=	id declaration* block session* assumptions claims eof
		\$\$ = `spec` (`declare` (\$2*), \$3, `sessions` (\$4*), \$5, \$6)

declaration	::=	typing-declaration
		\$\$ = \$1
	|	'`keypair`' opt-encryption-algorithm id '`,`' id opt-keypair-arg-type
		\$\$ = `keypair` (\$2, \$3, \$5, \$6)
	|	'`alias`' id '=' term
		\$\$ = `alias` (\$2, \$4)
	|	ids '`: alias`' id '=' term
		\$\$ = `local-alias` (\$3, \$5, \$1*)
	|	`basetype`' id
		\$\$ = `basetype` (\$2)
	|	id '`knows`' tuple
		\$\$ = (`knows` (\$1, x))$_{x\in\$3^*}$
	|	'`everybody`' '`knows`' tuple
		\$\$ = (`everybody-knows` (x))$_{x\in\$3^*}$
	|	'`axiom`' term '=' term opt-quantification
		\$\$ = `axiom` (\$2, \$4, \$5*)
typing-declaration	::=	ids '`:`' type
		\$\$ = (`type` (x, \$3))$_{x\in\$1^*}$
	|	ids '`:`' type '`shared`'
		\$\$ = (`type-shared` (x, \$3))$_{x\in\$1^*}$
	|	id '`(`' type-list '`) :`' type optional-hash-or-secret
		\$\$ = `type-shared` (\$1, f (\$5, \$3*)) where: f = `function` if $\$6 = \emptyset$; f = `one-way-function` if $\$6 = \{$'`hash`'$\}$; f = `secret-function` if $\$6 = \{$'`secret`'$\}$; f = `secret-one-way-function` if $\$6 = \{$'`hash`', '`secret`'$\}$
optional-hash-or-secret	::=	optional-hash-or-secret '`hash`'
		\$\$ = \$1 $\cup\{$'`hash`'$\}$
	|	optional-hash-or-secret '`secret`'
		\$\$ = \$1 $\cup\{$'`secret`'$\}$
	|	
		\$\$ = $\emptyset$
opt-quantification	::=	
		\$\$ = ϵ
	|	quantification
		\$\$ = \$1
quantification	::=	'`[`' typing-declarations '`]`'
		\$\$ = \$2
typing-declarations	::=	typing-declaration ('`,`' typing-declaration)*
		\$\$ = \$1, \$3*

ids	::=	id (‘,’ id)*
		$$ = $1, $3*
type	::=	id
		$$ = $1
type-list	::=	
		$$ = ϵ
	|	non-empty-type-list
		$$ = $1
non-empty-type-list	::=	type (‘,’ type)*
		$$ = $1, $3*
opt-keypair-arg-type	::=	
		$$ = ‘number’
	|	‘(’ type-list ‘)’
		$$ = one-way-function (‘number’, $2*)

block	::=	‘{’ message* ‘}’
		$$ = block ($2*)
message	::=	label id ‘->’ id ‘:’ term
		$$ = comm ($1, $2, $4, $6)
	|	id ‘:’ id ‘==’ atomic-term-or-ciphertext
		$$ = equals ($1, $3, $5)
	|	id ‘:’ id ‘:=’ atomic-term-or-ciphertext
		$$ = assign ($1, $3, $5)
	|	block
		$$ = $1
	|	‘switch’ term ‘{’ case* ‘}’
		$$ = switch ($2, $4*)
case	::=	‘case’ term ‘:’ message*
		$$ = case ($2, $4*)

term	::=	atomic-term-or-ciphertext
		$$ = $1
	|	atomic-term-or-ciphertext ‘`,`’ tuple
		$$ = `tuple` ($1, $3*)
tuple	::=	atomic-term-or-ciphertext (‘`,`’ atomic-term-or-ciphertext)*
		$$ = $1, $3*
term-list	::=	‘`()`’
		$$ = ϵ
	|	‘`(`’ tuple ‘`)`’
		$$ = $2
atomic-term-or-ciphertext	::=	atomic-term
		$$ = $1
	|	‘{’ term ‘}_’ atomic-term opt-encryption-algorithm
		$$ = `crypt` ($2, $4, $5)
	|	‘{’ term ‘}’ encryption-algorithm ‘_’ atomic-term
		$$ = `crypt` ($2, $6, $4)
	|	‘`[`’ term ‘`]`_’ atomic-term opt-encryption-algorithm
		$$ = `sign` ($2, $4, $5)
	|	‘`[`’ term ‘`]`’ encryption-algorithm ‘_’ atomic-term
		$$ = `sign` ($2, $6, $4)
	|	atomic-term ‘`%`’ atomic-term
		$$ = `percent` ($1, $3)
	|	atomic-term-or-ciphertext ‘`@`’ label id
		$$ = `locate` ($3, $4, $1)
atomic-term	::=	id term-list
		$$ = `apply` ($1, $2*)
	|	id
		$$ = $1
	|	‘`(`’ term ‘`)`’
		$$ = $2
	|	‘`()`’
		$$ = `tuple` ()
encryption-algorithm	::=	‘`^`’ atomic-term
		$$ = $2
opt-encryption-algorithm	::=	
		$$ = `vanilla` ()
	|	encryption-algorithm
		$$ = $1

session	::=	opt-label ‘`session`’ opt-principals assignments
		$$ = `session` ($1, $3, $4*)
	|	opt-label ‘`session* {`’ ids ‘`}`’ opt-principals assignments
		$$ = `session-repeat` (`private` ($3*), $1, $5, $6*)
	|	opt-label ‘`session* {}`’ opt-principals assignments
		$$ = `session-repeat` (`private` (), $1, $3, $4*)
	|	opt-label ‘`session*`’ opt-principals assignments
		$$ = `session-repeat` (`private` (), $1, $3, $4*)
opt-label	::=	
		$$ = some fresh label (intentionally left unspecified).
	|	label
		$$ = $1
opt-principals	::=	
		$$ = `principals-except` ()
	|	principals
		$$ = $1
principals	::=	‘`[`’ ids ‘`]`’
		$$ = `principals-only` ($2*)
	|	‘`[^`’ ids ‘`]`’
		$$ = `principals-except` ($2*)
assignments	::=	assignment (‘`,`’ assignment)*
		$$ = $1, $3*
assignment	::=	id ‘=’ atomic-term-or-ciphertext
		$$ = `bind` ($1, $3)

<table>
<tr><td>assumptions</td><td>::=</td><td>‘assume’ properties</td></tr>
<tr><td></td><td></td><td>$$ = assume ($2*)</td></tr>
<tr><td>claims</td><td>::=</td><td>‘claim’ properties</td></tr>
<tr><td></td><td></td><td>$$ = claim ($2*)</td></tr>
<tr><td>properties</td><td>::=</td><td>property (‘,’ property)*</td></tr>
<tr><td></td><td></td><td>$$ = $1, $3*</td></tr>
<tr><td>property</td><td>::=</td><td>id term-list</td></tr>
<tr><td></td><td></td><td>$$ = prop ($1, $2*)</td></tr>
<tr><td></td><td>|</td><td>label id ‘@’ label</td></tr>
<tr><td></td><td></td><td>$$ = at ($1, $2, $4)</td></tr>
<tr><td></td><td>|</td><td>label id ‘(’ term ‘)’</td></tr>
<tr><td></td><td></td><td>$$ = owns ($1, $2, $4)</td></tr>
<tr><td></td><td>|</td><td>‘secret (’ term ‘)’</td></tr>
<tr><td></td><td></td><td>$$ = secret ($2)</td></tr>
<tr><td></td><td>|</td><td>‘true’</td></tr>
<tr><td></td><td></td><td>$$ = and ()</td></tr>
<tr><td></td><td>|</td><td>‘false’</td></tr>
<tr><td></td><td></td><td>$$ = or ()</td></tr>
<tr><td></td><td>|</td><td>property ‘&&’ property</td></tr>
<tr><td></td><td></td><td>$$ = and ($1, $3)</td></tr>
<tr><td></td><td>|</td><td>property ‘||’ property</td></tr>
<tr><td></td><td></td><td>$$ = or ($1, $3)</td></tr>
<tr><td></td><td>|</td><td>property ‘=>’ property</td></tr>
<tr><td></td><td></td><td>$$ = implies ($1, $3)</td></tr>
<tr><td></td><td>|</td><td>‘!’ property</td></tr>
<tr><td></td><td></td><td>$$ = not ($2)</td></tr>
<tr><td></td><td>|</td><td>‘(’ property ‘)’</td></tr>
<tr><td></td><td></td><td>$$ = $2</td></tr>
<tr><td></td><td>|</td><td>‘*E’ property</td></tr>
<tr><td></td><td></td><td>$$ = E ($2)</td></tr>
<tr><td></td><td>|</td><td>‘*A’ property</td></tr>
<tr><td></td><td></td><td>$$ = A ($2)</td></tr>
<tr><td></td><td>|</td><td>‘*F’ property</td></tr>
<tr><td></td><td></td><td>$$ = F ($2)</td></tr>
<tr><td></td><td>|</td><td>‘*G’ property</td></tr>
<tr><td></td><td></td><td>$$ = G ($2)</td></tr>
<tr><td></td><td>|</td><td>‘*P’ property</td></tr>
<tr><td></td><td></td><td>$$ = P ($2)</td></tr>
<tr><td></td><td>|</td><td>‘*Q’ property</td></tr>
<tr><td></td><td></td><td>$$ = Q ($2)</td></tr>
<tr><td></td><td>|</td><td>‘*N’ property</td></tr>
<tr><td></td><td></td><td>$$ = N ($2)</td></tr>
<tr><td></td><td>|</td><td>property ‘*U’ property</td></tr>
<tr><td></td><td></td><td>$$ = U ($1, $3)</td></tr>
<tr><td></td><td>|</td><td>property ‘*W’ property</td></tr>
<tr><td></td><td></td><td>$$ = W ($1, $3)</td></tr>
<tr><td></td><td>|</td><td>property ‘*S’ property</td></tr>
<tr><td></td><td></td><td>$$ = S ($1, $3)</td></tr>
<tr><td></td><td>|</td><td>property ‘*B’ property</td></tr>
<tr><td></td><td></td><td>$$ = B ($1, $3)</td></tr>
</table>

4 Syntaxe abstraite

La syntaxe abstraite d'un protocole EVA est produite à partir d'un fichier de spécification concrète (en `.evl`) par le traducteur `evatrans`. Ce dernier fournit une traduction qui est encore une fois un terme, dans un langage qui sera appelée PEVA.

4.1 Grammaire

La grammaire de PEVA est donnée par l'ensemble des termes de type Spec, décrits comme suit.

Une spécification PEVA, d'abord, est la donnée de déclarations d'identificateurs (Types), d'alias globaux (Values), d'axiomes (Axioms), d'hypothèses (Assumptions), de formules à prouver (Claims), et d'un programme (System). Ce dernier est une composition parallèle de processus, décrits plus bas.

Spec	::=	`compiled-spec` (Types, Values, Axioms, System, Assumptions, Claims)	protocole EVA, syntaxe abstraite
Types	::=	`types` (Type-declaration*)	déclarations d'identificateurs
Type-declaration	::=	`type` (id, Type)	déclaration d'identificateur
	|	`type-shared` (id, Type)	déclaration d'identificateur partagé
Values	::=	`values` (Alias*)	déclarations d'alias
Alias	::=	`alias` (id, Term)	déclaration d'alias
Axioms	::=	`axioms` (Axiom*)	listes d'axiomes
Axiom	::=	`axiom` (Term, Term, Type, Type-declaration*)	axiome (lhs, rhs, type, variables quantifiées)
System	::=	`system` (Process*)	Le système (composition paralléle de processus)
Assumptions	::=	`assume` (Formula*)	Hypothèses
Claims	::=	`claim` (Formula*)	Formules à prouver

Type	::=	id	type d'ordre 1
	|	`function` (id, id*)	type de fonction
	|	`one-way-function` (id, id*)	type de fonction one-way
	|	`secret-function` (id, id*)	type de fonction secrète
	|	`secret-one-way-function` (id, id*)	type de fonction secrète et one-way

Un processus PEVA peut être soit un processus simple, décrit par un graphe de transitions, soit un processus en multi-session parallèle, qui est un nombre non borné de processus simples décrits par le même graphe de transitions (les *copies*), mis en parallèle. Les transitions sont données comme une liste de triplets (source, cible, action). Les actions sont décrites plus bas. Dans le cas de processus en multi-session parallèle, Privates contient la liste des identificateurs qui contiennent des valeurs possiblement différentes d'une copie à l'autre; les autres identificateurs sont partagés entre les copies. Finalement, chaque processus vient avec une liste de connaissances (Know), qui sert à faire le lien entre expressions LAEVA et identificateurs correspondants à ces expressions dans les processus PEVA.

Process	::=	`process` (id, state, Know, Transition*)	processus simple (nom, label de départ, connaissances, transitions)
	|	`repeat-process` (Privates, id, state, Know, Transition*)	processus en multi-session parallèle (vars privées, nom, départ, connaissances, transitions)
Transition	::=	`trans` (state, state, Action)	transition (source, cible, action)
state	::=	label	état visible
	|	'`%`'label	état interne
Know	::=	`knows` (As*)	connaissances
As	::=	`as` (Term, Term)	terme connu sous la forme: terme
Privates	::=	`private` (id*)	liste de variables privées

Action	::=	`new` (Pattern)	création de nonce
	|	`let` (Pattern, Term)	pattern-matching
	|	`recv` (Pattern)	réception de message
	|	`send` (Term)	émission de message
	|	`skip` ()	action nulle

Term	::=	id	variable
	|	`crypt` (Term, Term, Term)	chiffrement (algo, texte, clé)
	|	`tuple` (Term*)	n-uplet
	|	`d` (Term)	conversion `*D*` → `number`
	|	`p` (Term)	conversion `principal` → `number`
	|	`a` (Term)	conversion `*algo*` → `number`
	|	`sa` (Term)	conversion `sym_algo` → `*algo*`
	|	`aa` (Term)	conversion `asym_algo` → `*algo*`
	|	`vanilla` ()	algorithme symétrique par défaut
	|	`apply` (id, Term*)	application de fonction définie
	|	`hash-apply` (id, Term*)	application de fonction one-way
	|	`secret-apply` (id, Term*)	application de fonction secrète
	|	`hash-secret-apply` (id, Term*)	application de fonction secrète one-way
	|	`apply-pubk` (Term, id, Term*)	appl. de constructeur de clé publique (algo, constructeur, arguments)
	|	`hash-apply-pubk` (Term, id, Term*)	appl. de constructeur de clé publique one-way (algo, constructeur, arguments)
	|	`apply-privk` (Term, id, Term*)	appl. de constructeur de clé privée (algo, constructeur, arguments)
	|	`hash-apply-privk` (Term, id, Term*)	appl. de constructeur de clé privée one-way (algo, constructeur, arguments)
	|	`lambda-pubk` (Term, id)	partie publique de clé (algo, clé)
	|	`lambda-privk` (Term, id)	partie privée de clé (algo, clé)
	|	`located` (label, id, Term)	terme vu par session/principal

Pattern	::=	id	variable
	|	`exact` (Term)	constante littérale
	|	`crypt` (Term, Pattern, Term)	déchiffrement (algo, texte, clé)
	|	`apply` (id, Pattern*)	application de constructeur
	|	`secret-apply` (id, Pattern*)	application de constructeur secret
	|	`tuple` (Pattern*)	n-uplet
	|	`d` (Pattern)	extraction `number` → `*D*`
	|	`p` (Pattern)	extraction `number` → `principal`
	|	`a` (Pattern)	extraction `number` → `*algo*`
	|	`sa` (Pattern)	extraction `*algo*` → `sym_algo`
	|	`aa` (Pattern)	extraction `*algo*` → `asym_algo`
	|	`vanilla` ()	algorithme symétrique par défaut
	|	`apply-pubk` (Term, id, Pattern*)	appl. de constructeur de clé publique (algo, constructeur, arguments)
	|	`apply-privk` (Term, id, Pattern*)	appl. de constructeur de clé privée (algo, constructeur, arguments)

Form	::=	`prop` (id, Term*)	formule atomique
	|	`at` (label, id, label)	point de programme
	|	`owns` (label, id, Term)	origine de messages
	|	`secret` (Term)	secret de messages
	|	`not` (Form)	négation
	|	`and` (Form*)	conjonction
	|	`or` (Form*)	disjonction
	|	`implies` (Form, Form)	implication
	|	`E` (Form)	il existe un chemin
	|	`A` (Form)	pour tout chemin
	|	`F` (Form)	dans un futur
	|	`G` (Form)	dans tout futur
	|	`P` (Form)	dans un passé
	|	`Q` (Form)	dans tout passé
	|	`N` (Form)	dorénavant
	|	`U` (Form, Form)	until
	|	`W` (Form, Form)	until faible
	|	`S` (Form, Form)	since
	|	`B` (Form, Form)	since faible

4.2 Traduction de LAEVA en PEVA

4.2.1 Traduction des termes

On définit d'abord un jugement $\rho, \sigma, \ell, c \vdash t \Rightarrow_{\text{term}} u : \tau$, où $t, u \in$ Term, $\tau \in$ Type, ρ : id$\tilde{\to}$Type est un environnement de types, σ : id$\tilde{\to}$Term est une substitution représentant les alias actifs; ℓ : [(label $\times$ id$\tilde{\to}$Term$\tilde{\to}$Term) $\times$ $\mathbb{P}$id] sert à interpréter les termes tels qu'ils apparaissent dans les formules (**claim**, **assume**), où il est nécessaire de préciser par `locate` la session et le principal des variables non partagées, ainsi que l'ensemble des variables partagées; et c : Type $\times$ Type$\tilde{\to}$Term $\to$ Term est la table des fonctions de coercion. On note $[A]$ le type des éléments de A, en union disjointe avec un élément spécial $- \notin A$; il s'agit du type "A optionnel". La notation $A\tilde{\to}B$ désigne l'espace des fonctions (partielles) de A vers B. Le domaine de $f : A\tilde{\to}B$ est noté $\operatorname{dom} f$. La fonction vide est notée $\{\}$, et $f + g$ dénote la fonction qui à $x \in \operatorname{dom} g$ associe $g(x)$, et à $x \in \operatorname{dom} f \setminus \operatorname{dom} g$ associe $f(x)$.

L'environnement ρ est décrit par les déclarations de type, et σ par les déclarations d''`alias`'. La composante ℓ est liée aux déclarations '`knows`', et sera commentée plus loin. De même, la composante c est liée aux déclarations '`basetype`'.

Le jugement $\rho, \sigma, \ell, c \vdash t \Rightarrow_{\text{term}} u : \tau$ décrit sous quelles conditions le terme t de la syntaxe concrète se traduit en le terme u de la syntaxe abstraite, de type τ.

$$\frac{x \in \operatorname{dom} \rho \quad x \in \operatorname{dom} \sigma}{\rho, \sigma, \ell, c \vdash x \Rightarrow_{\text{term}} \sigma(x) : \rho(x)} \tag{1}$$

$$\frac{x \in \operatorname{dom} \rho \quad x \notin \operatorname{dom} \sigma}{\rho, \sigma, -, c \vdash x \Rightarrow_{\text{term}} x : \rho(x)} \tag{2}$$

Noter que les variables n'ayant pas de valeur par σ ne sont typables que lorsque $\ell = -$. Si $\ell \neq -$, on exige que toutes les variables soient dans la portée d'un `locate` (règle (11)). Ceci est vrai sauf pour le cas de variables partagées, auquel cas la règle suivante s'applique:

$$\frac{x \in \mathit{shared} \qquad \rho, \sigma, -, c \vdash x \Rightarrow_{\text{term}} u : \tau}{\rho, \sigma, (\_, \mathit{shared}), c \vdash x \Rightarrow_{\text{term}} u : \tau} \tag{3}$$

$$\frac{\rho, \sigma, \ell, c \vdash t \Rightarrow_{\text{term}} t' : \texttt{number} \quad \rho, \sigma, \ell, c \vdash k \Rightarrow_{\text{term}} k' : \texttt{number} \quad \rho, \sigma, \ell, c \vdash a \Rightarrow_{\text{term}} a' : \texttt{number}}{\rho, \sigma, \ell, c \vdash \texttt{crypt}(t, k, a) \Rightarrow_{\text{term}} \texttt{crypt}(a', t', k') : \texttt{number}} \tag{4}$$

Noter que l'ordre des arguments est différent en LAEVA et PEVA. En PEVA, l'algorithme a' vient en premier: pour toutes les fonctions de nature cryptographique, il est convenu que l'algorithme de chiffrement vienne toujours en premier argument en PEVA.

$$\frac{\rho, \sigma, \ell, c \vdash t \Rightarrow_{\text{term}} t' : \texttt{number} \quad \rho, \sigma, \ell, c \vdash k \Rightarrow_{\text{term}} k' : \texttt{number} \quad \rho, \sigma, \ell, c \vdash a \Rightarrow_{\text{term}} a' : \texttt{number}}{\rho, \sigma, \ell, c \vdash \texttt{sign}(t, k, a) \Rightarrow_{\text{term}} \texttt{crypt}(a', t', k') : \texttt{number}} \tag{5}$$

Noter que toute signature est traduite sous forme d'un chiffrement en PEVA.

$$\frac{\rho, \sigma, \ell, c \vdash t_1 \Rightarrow_{\text{term}} u_1 : \texttt{number} \ \ldots \ \rho, \sigma, \ell, c \vdash t_n \Rightarrow_{\text{term}} u_n : \texttt{number}}{\rho, \sigma, \ell, c \vdash \texttt{tuple}(t_1, \ldots, t_n) \Rightarrow_{\text{term}} \texttt{tuple}(u_1, \ldots, u_n) : \texttt{number}} \tag{6}$$

$$\frac{}{\rho, \sigma, \ell, c \vdash \texttt{vanilla}() \Rightarrow_{\text{term}} \texttt{vanilla}() : \texttt{sym\_algo}} \tag{7}$$

Les termes $\texttt{lambda-pubk}(\ldots)$ et $\texttt{lambda-privk}(\ldots)$ désignent des parties publiques et privées respectivement de clés asymétriques. Elles ont des types *fonctionnels*, c'est-à-dire de la forme $\texttt{function}(\tau_{res}, \tau_1, \ldots, \tau_n)$ ou $\texttt{one-way-function}(\tau_{res}, \tau_1, \ldots, \tau_n)$.

$$\frac{\rho, \sigma, \ell, c \vdash a \Rightarrow_{\text{term}} a' : \texttt{*algo*} \quad \rho, \sigma, \ell, c \vdash f \Rightarrow_{\text{term}} f' : \tau \quad f' \in \text{id} \quad \tau \text{ type fonctionnel}}{\rho, \sigma, \ell, c \vdash \texttt{lambda-pubk}(a, f) \Rightarrow_{\text{term}} \texttt{lambda-pubk}(a', f') : \tau} \tag{8}$$

$$\frac{\rho, \sigma, \ell, c \vdash a \Rightarrow_{\text{term}} a' : \texttt{*algo*} \quad \rho, \sigma, \ell, c \vdash f \Rightarrow_{\text{term}} f' : \tau \quad f' \in \text{id} \quad \tau \text{ type fonctionnel}}{\rho, \sigma, \ell, c \vdash \texttt{lambda-privk}(a, f) \Rightarrow_{\text{term}} \texttt{lambda-privk}(a', f') : \tau} \tag{9}$$

L'application est définie à l'aide d'une fonction auxiliaire (partielle) $apply_{h,s}$, où h (hash) et s (secret) sont deux booléens:

$$\begin{array}{rcll}
apply_{\bot,\bot}(f, t_1, \ldots, t_n) & = & \texttt{apply}(f, t_1, \ldots, t_n) & f \in \text{id} \\
apply_{\top,\bot}(f, t_1, \ldots, t_n) & = & \texttt{hash-apply}(f, t_1, \ldots, t_n) & f \in \text{id} \\
apply_{\bot,\top}(f, t_1, \ldots, t_n) & = & \texttt{secret-apply}(f, t_1, \ldots, t_n) & f \in \text{id} \\
apply_{\top,\top}(f, t_1, \ldots, t_n) & = & \texttt{hash-secret-apply}(f, t_1, \ldots, t_n) & f \in \text{id} \\
apply_{\bot,s}(\texttt{lambda-pubk}(a, f), t_1, \ldots, t_n) & = & \texttt{apply-pubk}(a, f, t_1, \ldots, t_n) & \\
apply_{\top,s}(\texttt{lambda-pubk}(a, f), t_1, \ldots, t_n) & = & \texttt{hash-apply-pubk}(a, f, t_1, \ldots, t_n) & \\
apply_{\bot,s}(\texttt{lambda-privk}(a, f), t_1, \ldots, t_n) & = & \texttt{apply-privk}(a, f, t_1, \ldots, t_n) & \\
apply_{\top,s}(\texttt{lambda-privk}(a, f), t_1, \ldots, t_n) & = & \texttt{hash-apply-privk}(a, f, t_1, \ldots, t_n) &
\end{array}$$

Dans la règle suivante (en particulier), une prémisse de la forme $u = apply_{\bot,\bot}(f', u_1, \ldots, u_n)$ signifie que le côté droit est défini et égal au côté gauche. On pose:

$$\begin{array}{rll}
fun_flags & = \{ & \texttt{function} \mapsto (\bot, \bot), \\
& & \texttt{one-way-function} \mapsto (\top, \bot), \\
& & \texttt{secret-function} \mapsto (\bot, \top), \\
& & \texttt{secret-one-way-function} \mapsto (\top, \top)\}
\end{array}$$

$$\frac{\begin{array}{c}\rho, \sigma, \ell, c \vdash f \Rightarrow_{\text{term}} f' : fun(\tau_{res}, \tau_1, \ldots, \tau_n) \qquad fun \in \text{dom}\, fun\_flags \\ \rho, \sigma, \ell, c \vdash t_1 \Rightarrow_{\text{term}} u_1 : \tau_1 \ \ldots \ \rho, \sigma, \ell, c \vdash t_n \Rightarrow_{\text{term}} u_n : \tau_n \\ u = apply_{fun\_flags(fun)}(f', u_1, \ldots, u_n)\end{array}}{\rho, \sigma, \ell, c \vdash \texttt{apply}(f, t_1, \ldots, t_n) \Rightarrow_{\text{term}} u : \tau_{res}} \tag{10}$$

Les termes de la forme $\texttt{locate}(session, principal, t)$ sont traités par le jugement (11). Il utilise un jugement auxiliaire $known \vdash t \Rightarrow_{loc} u$, qui traduit par quel terme u le terme t est interprété dans le contexte où les termes s de $\text{dom}\, known$ doivent être vus comme $known(s)$, où $known : \text{Term} \tilde{\to} \text{Term}$.

$$\frac{\rho, \sigma, -, c \vdash t \Rightarrow_{\text{term}} u : \tau \quad (p, s) \in \operatorname{dom} loc \quad loc(p, s) \vdash u \Rightarrow_{loc} v}{\rho, \sigma, (loc, \_), c \vdash \texttt{locate}(s, p, t) \Rightarrow_{\text{term}} \texttt{located}(s, p, v) : \tau} \tag{11}$$

$$\frac{t \in \operatorname{dom} known}{known \vdash t \Rightarrow_{loc} known(t)} \tag{12}$$

$$\frac{f(t_1, \ldots, t_n) \notin \operatorname{dom} known \quad known \vdash t_1 \Rightarrow_{loc} u_1 \ \ldots \ known \vdash t_n \Rightarrow_{loc} u_n}{known \vdash f(t_1, \ldots, t_n) \Rightarrow_{loc} f(u_1, \ldots, u_n)} \tag{13}$$

Finalement, on dispose de coercions automatiques[1]

$$\frac{\rho, \sigma, \ell, c \vdash t \Rightarrow_{\text{term}} u : \tau \qquad (\tau, \tau') \in \operatorname{dom} c \quad v = c(\tau, \tau')(u)}{\rho, \sigma, \ell, c \vdash t \Rightarrow_{\text{term}} v : \tau'} \tag{14}$$

On notera qu'il n'existe pas de jugement de la forme $\rho, \sigma, \ell, c \vdash \texttt{percent}(\ldots) \Rightarrow_{\text{term}} u : \tau$: les termes utilisant la notation '`%`' sont illégaux. Les pourcents sont traités par un autre mécanisme (voir plus loin).

4.2.2 Déclarations

Les déclarations d s'élaborent en un environnement de types ρ : id$\tilde{\to}$Type, un ensemble $shared$: $\mathbb{P}$id d'identificateurs partagés, une substitution globale σ : id$\tilde{\to}$Term, des substitutions locales à chaque rôle θ : id$\tilde{\to}$id$\tilde{\to}$Term, des déclarations de connaissances globales $known$: Term$\tilde{\to}$Term, des déclarations de connaissances locales κ : id$\tilde{\to}$Term$\tilde{\to}$Term, une par rôle, un ensemble $\alpha \in \mathbb{P}$(Term $\times$ Term $\times$ Type $\times$ (id$\tilde{\to}$Type)) d'axiomes (un axiome est formé de deux termes s et t, d'un type τ, et d'un environnement $x_1 : \tau_1, \ldots, x_n : \tau_n$, et signifie $\forall x_1 : \tau_1, \ldots, x_n : \tau_n \cdot s = t : \tau$), et une table de coercions c : Type $\times$ Type$\tilde{\to}$Term $\to$ Term.

On utilisera les abréviations suivantes pour les composantes θ et κ. On note $\{p_i \mapsto x \mapsto \tau | 1 \leq i \leq n\}$ la fonction currifiée $\{p_i \mapsto \{x \mapsto \tau\} | 1 \leq i \leq n\}$. La notation $+$ est adaptée aux composantes θ en définissant $\theta + \theta'$ comme étant la fonction qui à $p \in \operatorname{dom} \theta \setminus \operatorname{dom} \theta'$ associe $\theta(p)$, à $p \in \operatorname{dom} \theta' \setminus \operatorname{dom} \theta$ associe $\theta'(p)$, et à $p \in \operatorname{dom} \theta \cap \operatorname{dom} \theta'$ associe $\theta(p) + \theta'(p)$.

On note S le nonuplet $(\rho, shared, \sigma, \theta, known, \kappa, \alpha, c, \upsilon)$, où υ est un ensemble d'id auxiliaire, qui servira à éviter la circularité dans les définitions d'alias. On notera aussi "ρ of S", "σ of S", etc., la composante correspondante de S. De façon symétrique à ces projections, les injections "ρ in S", "σ in S", etc., dénoteront respectivement $(\rho, \emptyset, \{\}, \{\}, \{\}, \{\}, \{\}, \{\}, \emptyset)$, $(\{\}, \emptyset, \sigma, \{\}, \{\}, \{\}, \{\}, \emptyset)$, etc. La notation $+$ est étendue aux octuplets composante par composante, de sorte que, par exemple, $S + (\rho \text{ in } S)$, où (le premier) S est $(\rho_1, shared_1, \sigma_1, \theta_1, known_1, \kappa_1, \alpha_1, c_1, \upsilon_1)$, est $(\rho_1 + \rho, shared_1, \sigma_1, \theta_1, known_1, \kappa_1, \alpha_1, c_1, \upsilon_1)$. On abrégera ceci le plus souvent en $S + \rho$, laissant "in S" implicite. Ces conventions sont aussi celles de la sémantique de Standard ML ([MTH90], section 4.3). On écrira aussi $S + shared = (\rho_1, shared_1 \cup shared, \sigma_1, \theta_1, known_1, \kappa_1, \alpha_1, c_1, \upsilon_1)$, le symbole $+$ abrégeant ici l'union sur la composante $shared$.

On définit un jugement de la forme $S \vdash d \Rightarrow_{\text{decl}} S'$, qui définit comment une déclaration d modifie le contexte S en le contexte S'.

$$\frac{x \notin \operatorname{dom}(\rho \text{ of } S) \quad \rho = \{x \mapsto \tau\}}{S \vdash \texttt{type}(x, \tau) \Rightarrow_{\text{decl}} S + \rho} \tag{15}$$

$$\frac{x \notin \operatorname{dom}(\rho \text{ of } S) \quad \rho = \{x \mapsto \tau\} \quad shared = \{x\}}{S \vdash \texttt{type} - \texttt{shared}(x, \tau) \Rightarrow_{\text{decl}} S + \rho + shared} \tag{16}$$

[1]La règle (14) définit des coercions non-déterministes dès qu'il existe plusieurs chaînes de coercions d'un type τ vers un type τ'. Par exemple, si on déclare `basetype key`, ce qui déclare automatiquement deux coercions `*key_of_D*` : `*D*` $\to$ `key` et `*number_of_key*` : `key` $\to$ `number` (cf. règle (21)), et comme on a aussi une coercion `d` : `*D*` $\to$ `number` (cf. règle (83)), alors dès que $\rho, \sigma, \ell, c \vdash t \Rightarrow_{\text{term}} u$: `*D*`, on pourra dériver à la fois $\rho, \sigma, \ell, c \vdash t \Rightarrow_{\text{term}}$ `*number_of_key*(*key_of_D*(`u`))` : `number` et $\rho, \sigma, \ell, c \vdash t \Rightarrow_{\text{term}}$ `d(`u`)` : `number`. L'outil `evatrans` préfère en réalité toujours les chaînes de coercions les plus courtes, et seule la dernière coercion sera dérivée.

Certaines règles, dont la suivante, doivent créer des identificateurs z frais; informellement, "frais" signifie non présents dans la spécification LAEVA d'entrée, et différents de tous les identificateurs frais créés précédemment[2]. On ne précisera pas plus formellement la notion, ce qui compliquerait inutilement les règles.

$$\frac{\begin{array}{c} x, y \notin \operatorname{dom}\sigma \quad x \neq y \quad \rho \text{ of } S, \sigma \text{ of } S, -, c \text{ of } S \vdash a \Rightarrow_{\text{term}} a' : \texttt{*algo*} \quad z \in \text{id frais} \\ \rho = \{z \mapsto \texttt{*D*}, x \mapsto \texttt{number}, y \mapsto \texttt{number}\} \\ \sigma = \{x \mapsto \texttt{hash-apply-pubk}(a', \texttt{**asymk**}, z), \\ y \mapsto \texttt{hash-apply-privk}(a', \texttt{**asymk**}, z)\} \end{array}}{S \vdash \texttt{keypair}(a, x, y, \texttt{number}) \Rightarrow_{\text{decl}} S + \rho + \sigma} \tag{17}$$

$$\frac{\begin{array}{c} x, y \notin \operatorname{dom}\sigma \quad x \neq y \quad \rho \text{ of } S, \sigma \text{ of } S, -, c \text{ of } S \vdash a \Rightarrow_{\text{term}} a' : \texttt{*algo*} \quad z \in \text{id frais} \\ \tau = \texttt{one-way-function}(\texttt{number}, \tau_1, \ldots, \tau_n) \\ \mathit{shared} = \{z\} \\ \rho = \{z \mapsto \tau, x \mapsto \tau, y \mapsto \tau\} \\ \sigma = \{x \mapsto \texttt{lambda-pubk}(a', z), y \mapsto \texttt{lambda-privk}(a', z)\} \end{array}}{S \vdash \texttt{keypair}(a, x, y, \tau) \Rightarrow_{\text{decl}} S + \rho + \mathit{shared} + \sigma} \tag{18}$$

On notera que lorsque le type d'une paire de clés `keypair` est une fonction (règle (18)), alors l'identificateur interne z, donc les deux clés créées, sont partagées: elles ont la même valeur d'un principal à l'autre. Par contre, lorsqu'il s'agit de simple paires de clés, de type `number` (règle (17), alors les clés ne sont pas partagées: deux principaux différents peuvent très bien avoir des valeurs différentes pour ces clés.

On note $\text{fv}(u)$ l'ensemble des variables libres du terme u. L'ensemble υ of S collectionne toutes les variables apparaissant libres dans un terme déjà mentionné. On vérifie que x n'est pas dans υ, ce qui interdit tout alias circulaire.

$$\frac{\begin{array}{c} x \notin \operatorname{dom}(\sigma \text{ of } S) \cup \bigcup_{r \in \operatorname{dom}(\theta \text{ of } S)} \operatorname{dom}((\theta \text{ of } S)(r)) \quad x \notin \upsilon \\ x \in \operatorname{dom}(\rho \text{ of } S) \Longrightarrow \tau = (\rho \text{ of } S)(x) \\ \rho \text{ of } S, \sigma \text{ of } S, -, c \text{ of } S \vdash t \Rightarrow_{\text{term}} u : \tau \quad x \notin \text{fv}(u) \\ \rho = \{x \mapsto \tau\} \quad \sigma = \{x \mapsto u\} \quad \upsilon = \text{fv}(u) \end{array}}{S \vdash \texttt{alias}(x, t) \Rightarrow_{\text{decl}} S + \rho + \sigma + \upsilon} \tag{19}$$

$$\frac{\begin{array}{c} x \notin \operatorname{dom}(\sigma \text{ of } S) \cup \bigcup_{r \in \operatorname{dom}(\theta \text{ of } S)} \operatorname{dom}((\theta \text{ of } S)(r)) \quad x \notin \upsilon \\ x \in \operatorname{dom}(\rho \text{ of } S) \Longrightarrow \tau = (\rho \text{ of } S)(x) \\ \rho \text{ of } S, \sigma \text{ of } S, -, c \text{ of } S \vdash t \Rightarrow_{\text{term}} u : \tau \quad x \notin \text{fv}(u) \\ (\rho \text{ of } S)(p_1) = \texttt{principal}, \quad \ldots, \quad (\rho \text{ of } S)(p_n) = \texttt{principal} \\ \rho = \{x \mapsto \tau\} \quad \theta = \{p_i \mapsto x \mapsto u | 1 \leq i \leq n\} \quad \upsilon = \text{fv}(u) \end{array}}{S \vdash \texttt{local-alias}(x, t, p_1, \ldots, p_n) \Rightarrow_{\text{decl}} S + \rho + \theta + \upsilon} \tag{20}$$

Les déclarations `basetype` déclarent de nouvelles coercions, par exemple déclarer `key` comme base type crée deux nouvelles coercions `*key_of_D*` : `*D*` $\to$ `key` et `*number_of_key*` : `key` $\to$ `number`. Si w et w' sont deux mots, alors $w.w'$ est leur concaténation. On note $\pi_1(A)$ l'ensemble des x tels que $(x, y) \in A$. On notera que les deux coercions d et n sont déclarées comme connues par tout le monde (dans known).

$$\frac{\begin{array}{c} t \in \text{id} \quad t \neq \texttt{number} \quad t \notin \pi_1(\operatorname{dom} c) \\ d = *.t.\texttt{\_of\_D*} \quad n = \texttt{*number\_of\_}.t.* \\ d, n \notin \operatorname{dom}(\sigma \text{ of } S) \quad \rho = \{d \mapsto \texttt{function}(t, \texttt{*D*}), n \mapsto \texttt{function}(\texttt{number}, t)\} \\ c = \{(\texttt{*D*}, t) \mapsto (u \mapsto \texttt{apply}(d, u)), (t, \texttt{number}) \mapsto (u \mapsto \texttt{apply}(n, u))\} \\ \mathit{known} = \{d \mapsto d, n \mapsto n\} \end{array}}{S \vdash \texttt{basetype}(t) \Rightarrow_{\text{decl}} S + \rho + c + \mathit{known}} \tag{21}$$

[2] `evatrans` crée ces identificateurs de la forme '_xn', ou '_kn', ou '_kp$n$', ou '_sw$n$', ou en concaténant à un identificateur LAEVA un tiret '−' et un entier n. Formellement, ce ne sont pas des id.

$$\frac{(\rho \text{ of } S)(p) = \texttt{principal} \quad \rho \text{ of } S, \sigma \text{ of } S, -, c \text{ of } S \vdash t \Rightarrow_{\text{term}} u : \tau \qquad \kappa = \{p \mapsto u \mapsto u\} \quad \upsilon = \{p\} \cup \text{fv}(u)}{S \vdash \texttt{knows}(p, t) \Rightarrow_{\text{decl}} S + \kappa + \upsilon} \tag{22}$$

$$\frac{\rho \text{ of } S, \sigma \text{ of } S, -, c \text{ of } S \vdash t \Rightarrow_{\text{term}} u : \tau \qquad known = \{u \mapsto u\} \quad \upsilon = \text{fv}(u)}{S \vdash \texttt{everybody-knows}(t) \Rightarrow_{\text{decl}} (S + known + \upsilon)} \tag{23}$$

$$\frac{\begin{array}{c}\rho \text{ of } S + \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\}, \sigma \text{ of } S, -, c \text{ of } S \vdash s \Rightarrow_{\text{term}} u : \tau \\ \rho \text{ of } S + \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\}, \sigma \text{ of } S, -, c \text{ of } S \vdash t \Rightarrow_{\text{term}} v : \tau \\ \alpha = \{(u, v, \tau, \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\})\} \quad \upsilon = \text{fv}(u) \cup \text{fv}(v)\end{array}}{S \vdash \texttt{axiom}(s, t, \texttt{type}(x_1, \tau_1), \ldots, \texttt{type}(x_n, \tau_n)) \Rightarrow_{\text{decl}} S + \alpha + \upsilon} \tag{24}$$

4.2.3 Composition de messages

Le jugement $\rho, known, unk, c \vdash t \Rightarrow_{\text{comp}} al, t', known'$, où ρ : id$\tilde{\to}$Type, $known, known'$: Term$\tilde{\to}$Term, unk : $\mathbb{P}$(id), c : Type $\times$ Type$\tilde{\to}$Term $\to$ Term, $t, t' \in$ Term, $al \in$ Action*, décrit comment l'on peut fabriquer un message t dans un environnement de types ρ et un état de connaissances $known$. Le résultat est un message t', une liste d'actions al (typiquement des créations de nonces par `new`), et un nouvel état de connaissances $known'$. L'ensemble d'identificateurs unk représente un ensemble de variables qui, si jamais elles sont inconnues, ne doivent pas être créées par `new`.

$$\frac{t \in \text{dom}\, known}{\rho, known, unk, c \vdash t \Rightarrow_{\text{comp}} \epsilon, known(t), known} \tag{25}$$

Intuitivement, toute variable non connue est compilée comme la création d'un nonce (si son type est `number`), ou d'une paire de clés (si son type est `*D*`; ce sera le cas pour les z des règles (17) et (18)). Ceci est vrai à l'exception des variables présentes dans unk, qui représentent des valeurs bien précises mais inconnues par ailleurs.

En réalité, la règle pour compiler une variable inconnue x et pas dans unk est à la fois plus simple et plus générale: on la compile en utilisant `new` dès qu'il existe une coercion de `*D*` (le type des données fraîches) vers le type de x.

$$\frac{x \notin \text{dom}\, known \quad x \notin unk \quad \tau = \rho(x) \quad z \in \text{id frais} \quad \{z \mapsto \texttt{*D*}\}, \{\}, -, c \vdash z \Rightarrow_{\text{term}} u : \tau}{\rho, known, unk, c \vdash x \Rightarrow_{\text{comp}} \texttt{new}(z), u, known + \{x \mapsto u\}} \tag{26}$$

$$\frac{\begin{array}{c} f(t_1, \ldots, t_n) \notin \text{dom}\, known \\ f \notin \{\texttt{apply-pubk}, \texttt{apply-privk}, \texttt{hash-apply-pubk}, \texttt{hash-apply-privk}\} \\ \rho, known, unk, c \vdash t_1 \Rightarrow_{\text{comp}} al_1, t'_1, known_1 \\ \ldots \\ \rho, known_{n-1}, unk, c \vdash t_n \Rightarrow_{\text{comp}} al_n, t'_n, known_n \end{array}}{\rho, known, unk, c \vdash f(t_1, \ldots, t_n) \Rightarrow_{\text{comp}} al_1. \ldots .al_n, f(t'_1, \ldots, t'_n), known_n} \tag{27}$$

Le cas des symboles `apply-pubk`, `apply-privk`, `hash-apply-pubk`, `hash-apply-privk` doit être traité à part.

$$\frac{\begin{array}{c} \texttt{apply-pubk}(a, f, t_1, \ldots, t_n) \notin \text{dom}\, known \\ \rho, known, unk, c \vdash \texttt{lambda-pubk}(a, f) \Rightarrow_{\text{comp}} al_0, t'_0, known_0 \\ \rho, known_0, unk, c \vdash t_1 \Rightarrow_{\text{comp}} al_1, t'_1, known_1 \\ \ldots \\ \rho, known_{n-1}, unk, c \vdash t_n \Rightarrow_{\text{comp}} al_n, t'_n, known_n \\ u = apply_{\perp,\perp}(t'_0, t'_1, \ldots, t'_n) \end{array}}{\rho, known, unk, c \vdash f(t_1, \ldots, t_n) \Rightarrow_{\text{comp}} al_0.al_1. \ldots .al_n, u, known_n} \tag{28}$$

$$\frac{\begin{array}{c}\texttt{apply-privk}(a, f, t_1, \ldots, t_n) \notin \operatorname{dom} known \\ \rho, known, unk, c \vdash \texttt{lambda-privk}(a, f) \Rightarrow_{\text{comp}} al_0, t'_0, known_0 \\ \rho, known_0, unk, c \vdash t_1 \Rightarrow_{\text{comp}} al_1, t'_1, known_1 \\ \ldots \\ \rho, known_{n-1}, unk, c \vdash t_n \Rightarrow_{\text{comp}} al_n, t'_n, known_n \\ u = apply_{\bot,\top}(t'_0, t'_1, \ldots, t'_n)\end{array}}{\rho, known, unk, c \vdash f(t_1, \ldots, t_n) \Rightarrow_{\text{comp}} al_0.al_1.\ldots.al_n, u, known_n} \tag{29}$$

$$\frac{\begin{array}{c}\texttt{hash-apply-pubk}(a, f, t_1, \ldots, t_n) \notin \operatorname{dom} known \\ \rho, known, unk, c \vdash \texttt{lambda-pubk}(a, f) \Rightarrow_{\text{comp}} al_0, t'_0, known_0 \\ \rho, known_0, unk, c \vdash t_1 \Rightarrow_{\text{comp}} al_1, t'_1, known_1 \\ \ldots \\ \rho, known_{n-1}, unk, c \vdash t_n \Rightarrow_{\text{comp}} al_n, t'_n, known_n u = apply_{\top,\bot}(t'_0, t'_1, \ldots, t'_n)\end{array}}{\rho, known, unk, c \vdash f(t_1, \ldots, t_n) \Rightarrow_{\text{comp}} al_0.al_1.\ldots.al_n, u, known_n} \tag{30}$$

$$\frac{\begin{array}{c}\texttt{hash-apply-privk}(a, f, t_1, \ldots, t_n) \notin \operatorname{dom} known \\ \rho, known, unk, c \vdash \texttt{lambda-privk}(a, f) \Rightarrow_{\text{comp}} al_0, t'_0, known_0 \\ \rho, known_0, unk, c \vdash t_1 \Rightarrow_{\text{comp}} al_1, t'_1, known_1 \\ \ldots \\ \rho, known_{n-1}, unk, c \vdash t_n \Rightarrow_{\text{comp}} al_n, t'_n, known_n u = apply_{\top,\top}(t'_0, t'_1, \ldots, t'_n)\end{array}}{\rho, known, unk, c \vdash f(t_1, \ldots, t_n) \Rightarrow_{\text{comp}} al_0.al_1.\ldots.al_n, u, known_n} \tag{31}$$

4.2.4 Décomposition de message

On définit maintenant un jugement $\rho, known \vdash t \Rightarrow_{\text{dec}} pat, known'$ qui définit comment un terme t attendu par un principal sera compilé comme un pattern pat. On rappelle le jugement $known \vdash t \Rightarrow_{loc} u$ défini par les règles (12) et (13).

D'abord, un terme que l'on peut fabriquer est compilé en un pattern `exact`.

$$\frac{known \vdash t \Rightarrow_{loc} u}{\rho, known \vdash t \Rightarrow_{\text{dec}} \texttt{exact}(u), known} \tag{32}$$

Notons $known \vdash t \not\Rightarrow_{loc}$ le jugement signifiant que $known \vdash t \Rightarrow_{loc} u$ est faux pour tout u.

$$\frac{x \in \text{id} \quad z \in \text{id frais} \quad known \vdash x \not\Rightarrow_{loc}}{\rho, known \vdash x \Rightarrow_{\text{dec}} z, known + \{x \mapsto z\}} \tag{33}$$

Le déchiffrement correspond au pattern-matching des termes de la forme $\texttt{crypt}(\ldots)$. On distingue deux cas, selon que l'algorithme de chiffrement a est symétrique (règle (34)) ou asymétrique (règle (35)).

$$\frac{\begin{array}{c}known \vdash \texttt{crypt}(a, t, k) \not\Rightarrow_{loc} \quad known \vdash a \Rightarrow_{loc} a' \quad a' = \texttt{a}(\texttt{sa}(a_0)) \\ known \vdash k \Rightarrow_{loc} k' \quad \rho, known \vdash t \Rightarrow_{\text{dec}} pat, known'\end{array}}{\rho, known \vdash \texttt{crypt}(a, t, k) \Rightarrow_{\text{dec}} \texttt{crypt}(a', pat, k'), known'} \tag{34}$$

Pour le déchiffrement asymétrique, on pose :

$$\begin{aligned} Inv = \{ \; & \texttt{hash-apply-pubk} \mapsto \texttt{hash-apply-privk}, \\ & \texttt{hash-apply-privk} \mapsto \texttt{hash-apply-pubk}, \\ & \texttt{apply-pubk} \mapsto \texttt{apply-privk}, \\ & \texttt{apply-privk} \mapsto \texttt{apply-pubk}\} \end{aligned}$$

$$\frac{\begin{array}{c} known \vdash \texttt{crypt}(a,t,k) \not\Rightarrow_{loc} \quad known \vdash a \Rightarrow_{loc} a' \quad a' = \mathtt{a}(\mathtt{aa}(a_0)) \\ known \vdash k \Rightarrow_{loc} k' \quad k' = f(a', t_1, \ldots, t_n) \\ f \in \mathrm{dom}\, Inv \quad f' = Inv(f) \quad k^{-1} = f'(a', t_1, \ldots, t_n) \\ \rho, known \vdash t \Rightarrow_{\mathrm{dec}} pat, known' \end{array}}{\rho, known \vdash \texttt{crypt}(a,t,k) \Rightarrow_{\mathrm{dec}} \texttt{crypt}(a', pat, k^{-1}), known'} \tag{35}$$

On notera que le premier argument de f dans k' doit être exactement le même algorithme de chiffrement a' que celui utilisé pour le chiffrement (le premier argument de `crypt`).

$$\frac{\begin{array}{c} known \vdash \texttt{apply}(f, t_1, \ldots, t_n) \not\Rightarrow_{loc} \quad f \in \mathrm{id} \\ \rho, known \vdash t_1 \Rightarrow_{\mathrm{dec}} pat_1, known_1 \quad \ldots \quad \rho, known_{n-1} \vdash t_n \Rightarrow_{\mathrm{dec}} pat_n, known_n \end{array}}{\rho, known \vdash \texttt{apply}(f, t_1, \ldots, t_n) \Rightarrow_{\mathrm{dec}} \texttt{apply}(f, pat_1, \ldots, pat_n), known_n} \tag{36}$$

$$\frac{\begin{array}{c} known \vdash \texttt{secret-apply}(f, t_1, \ldots, t_n) \not\Rightarrow_{loc} \quad f \in \mathrm{id} \\ \rho, known \vdash t_1 \Rightarrow_{\mathrm{dec}} pat_1, known_1 \quad \ldots \quad \rho, known_{n-1} \vdash t_n \Rightarrow_{\mathrm{dec}} pat_n, known_n \end{array}}{\rho, known \vdash \texttt{secret-apply}(f, t_1, \ldots, t_n) \Rightarrow_{\mathrm{dec}} \texttt{secret-apply}(f, pat_1, \ldots, pat_n), known_n} \tag{37}$$

$$\frac{\begin{array}{c} known \vdash \texttt{apply-pubk}(f, t_1, \ldots, t_n) \not\Rightarrow_{loc} \quad known \vdash a \Rightarrow_{loc} a' \quad f \in \mathrm{id} \\ \rho, known \vdash t_1 \Rightarrow_{\mathrm{dec}} pat_1, known_1 \quad \ldots \quad \rho, known_{n-1} \vdash t_n \Rightarrow_{\mathrm{dec}} pat_n, known_n \end{array}}{\rho, known \vdash \texttt{apply-pubk}(a, f, t_1, \ldots, t_n) \Rightarrow_{\mathrm{dec}} \texttt{apply-pubk}(a, f, pat_1, \ldots, pat_n), known_n} \tag{38}$$

$$\frac{\begin{array}{c} known \vdash \texttt{apply-privk}(f, t_1, \ldots, t_n) \not\Rightarrow_{loc} \quad known \vdash a \Rightarrow_{loc} a' \quad f \in \mathrm{id} \\ \rho, known \vdash t_1 \Rightarrow_{\mathrm{dec}} pat_1, known_1 \quad \ldots \quad \rho, known_{n-1} \vdash t_n \Rightarrow_{\mathrm{dec}} pat_n, known_n \end{array}}{\rho, known \vdash \texttt{apply-privk}(a, f, t_1, \ldots, t_n) \Rightarrow_{\mathrm{dec}} \texttt{apply-privk}(a, f, pat_1, \ldots, pat_n), known_n} \tag{39}$$

On note que l'on ne peut pas faire de pattern-matching sur des messages commençant par `hash-apply`, `hash-secret-apply`, `hash-apply-pubk`, `hash-apply-privk` (on ne peut pas inverser les fonctions `hash`). On interdit aussi le pattern-matching sur `lambda-pubk` et `lambda-privk` (ce qui permettrait de récupérer la partie privée à partir de sa partie publique et réciproquement). Les cas `crypt`, `apply`, `secret-apply`, `apply-pubk`, `apply-privk` sont aussi exclus, car traités dans les règles ci-dessus.

$$\frac{\begin{array}{c} known \vdash f(t_1, \ldots, t_n) \not\Rightarrow_{loc} \\ f \notin \{\texttt{hash-apply}, \texttt{hash-secret-apply}, \texttt{hash-apply-pubk}, \texttt{hash-apply-privk}, \\ \texttt{crypt}, \texttt{apply}, \texttt{secret-apply}, \texttt{apply-pubk}, \texttt{apply-privk}, \\ \texttt{lambda-pubk}, \texttt{lambda-privk}\} \\ z \in \text{id frais} \end{array}}{\rho, known \vdash f(t_1, \ldots, t_n) \Rightarrow_{\mathrm{dec}} z, known + \{f(t_1, \ldots, t_n) \mapsto z\}} \tag{40}$$

4.2.5 Compilation des patterns

Il existe deux façon de compiler l'attente d'un message sur un canal, correspondant à un pattern pat. La première façon, qui est la façon par défaut pour `evatrans`, est de compiler simplement l'action $\texttt{recv}(pat)$. Ceci est la règle (41).

$$\frac{}{\vdash pat \Rightarrow_{\mathrm{recv}} \texttt{recv}(pat)} \tag{41}$$

Le jugement ici est de la forme $\vdash pat \Rightarrow_{\mathrm{recv}} al$, où $pat \in$ Pattern, et $al \in$ Action*.

Lorsque l'option `-d` est fournie à `evatrans`, ce jugement est défini de sorte à compiler une liste d'actions de reconnaissance de patterns élémentaires, qui sont des variables, ou de la forme exact(t), ou de la forme $f(x_1, \ldots, x_n)$, où les x_i ne sont plus des patterns généraux mais des variables. Le jugement est alors défini par les règles suivantes. On note x_{pat} une variable fraîche si pat n'est pas une variable, et la variable pat elle-même sinon.

$$\frac{x_{pat} \vdash pat \Rightarrow_{\mathrm{brk}} al}{\vdash pat \Rightarrow_{\mathrm{recv}} al.\mathtt{recv}(x_{pat})} \tag{42}$$

Le jugement auxiliaire $x \vdash pat \Rightarrow_{\mathrm{brk}} al$ est défini par les règles suivantes.

$$\frac{x_{pat} \vdash pat \Rightarrow_{\mathrm{brk}} al}{x \vdash \mathtt{crypt}(a, pat, k) \Rightarrow_{\mathrm{brk}} al.\mathtt{let}(\mathtt{crypt}(a, x_{pat}, k), x)} \tag{43}$$

Noter dans les règles suivantes qu'on effectue le pattern-matching de gauche à droite. C'est un choix arbitraire qui se retrouve dans la sémantique (section 5).

$$\frac{x_{pat_1} \vdash pat_1 \Rightarrow_{\mathrm{brk}} al_1 \quad \ldots \quad x_{pat_n} \vdash pat_n \Rightarrow_{\mathrm{brk}} al_n}{\begin{array}{l} x \vdash \mathtt{apply}(f, pat_1, \ldots, pat_n) \Rightarrow_{\mathrm{brk}} \\ al_1. \ldots .al_n.\mathtt{let}(\mathtt{apply}(f, x_{pat_1}, \ldots, x_{pat_n}), x) \end{array}} \tag{44}$$

$$\frac{x_{pat_1} \vdash pat_1 \Rightarrow_{\mathrm{brk}} al_1 \quad \ldots \quad x_{pat_n} \vdash pat_n \Rightarrow_{\mathrm{brk}} al_n}{\begin{array}{l} x \vdash \mathtt{secret\text{-}apply}(f, pat_1, \ldots, pat_n) \Rightarrow_{\mathrm{brk}} \\ al_1. \ldots .al_n.\mathtt{let}(\mathtt{secret\text{-}apply}(f, x_{pat_1}, \ldots, x_{pat_n}), x) \end{array}} \tag{45}$$

$$\frac{x_{pat_1} \vdash pat_1 \Rightarrow_{\mathrm{brk}} al_1 \quad \ldots \quad x_{pat_n} \vdash pat_n \Rightarrow_{\mathrm{brk}} al_n}{\begin{array}{l} x \vdash \mathtt{apply\text{-}pubk}(a, f, pat_1, \ldots, pat_n) \Rightarrow_{\mathrm{brk}} \\ al_1. \ldots .al_n.\mathtt{let}(\mathtt{apply\text{-}pubk}(a, f, x_{pat_1}, \ldots, x_{pat_n}), x) \end{array}} \tag{46}$$

$$\frac{x_{pat_1} \vdash pat_1 \Rightarrow_{\mathrm{brk}} al_1 \quad \ldots \quad x_{pat_n} \vdash pat_n \Rightarrow_{\mathrm{brk}} al_n}{\begin{array}{l} x \vdash \mathtt{apply\text{-}privk}(a, f, pat_1, \ldots, pat_n) \Rightarrow_{\mathrm{brk}} \\ al_1. \ldots .al_n.\mathtt{let}(\mathtt{apply\text{-}privk}(a, f, x_{pat_1}, \ldots, x_{pat_n}), x) \end{array}} \tag{47}$$

$$\frac{}{x \vdash \mathtt{exact}(t) \Rightarrow_{\mathrm{brk}} \mathtt{let}(\mathtt{exact}(t), x)} \tag{48}$$

On note ici que $\mathtt{let}(\mathtt{exact}(t), u)$ est juste le test d'égalité $t{==}u$.

$$\frac{\begin{array}{c} f \notin \{\mathtt{crypt}, \mathtt{apply}, \mathtt{secret\text{-}apply}, \mathtt{apply\text{-}pubk}, \mathtt{apply\text{-}privk}, \mathtt{exact}\} \\ x_{pat_1} \vdash pat_1 \Rightarrow_{\mathrm{brk}} al_1 \quad \ldots \\ x_{pat_n} \vdash pat_n \Rightarrow_{\mathrm{brk}} al_n \end{array}}{\begin{array}{l} x \vdash f(pat_1, \ldots, pat_n) \Rightarrow_{\mathrm{brk}} \\ al_1. \ldots .al_n.\mathtt{let}(f(x_{pat_1}, \ldots, x_{pat_n}), x) \end{array}} \tag{49}$$

$$\frac{}{x \vdash x \Rightarrow_{\mathrm{brk}} \epsilon} \tag{50}$$

4.2.6 Compilation des messages

On utilise ici un jugement $R, S, Unk, \gamma, src, tgt, ls \vdash m \Rightarrow_{\mathrm{code}} \gamma', S', Unk', ls'$, où $R \in \mathbb{P}(\mathrm{id})$ est l'ensemble des noms de rôles participant au protocole, S et S' sont des contextes comme décrits en section 4.2.2, $Unk, Unk' : \mathtt{id} \tilde{\rightarrow} \mathbb{P}(\mathrm{id})$ (à chaque principal associent les identificateurs qui ne doivent pas être créés par un `new` par ce principal), $src, tgt \in$ state, $ls, ls' \in \mathbb{P}(\mathrm{state})$, $m \in$ Message, $\gamma, \gamma' \in \mathtt{id} \tilde{\rightarrow} \mathbb{P}(\mathrm{state} \times \mathrm{state} \times \mathrm{Action})$ est une famille de graphes indexés par des noms de principaux. Les graphes sont des ensembles de transitions. De plus, tous les états de γ sont dans ls'.

D'abord, on traite des envois de messages (fonction `comm`). Pour traiter de la notation '`%`', on pose:

$$\begin{array}{rcl} split_1(\mathtt{percent}(s, t)) & = & s \\ split_2(\mathtt{percent}(s, t)) & = & t \\ split_i(x) & = & x \qquad (1 \leq i \leq 2) \\ split_i(f(t_1, \ldots, t_n)) & = & f(split_i(t_1), \ldots, split_i(t_n)) \qquad (1 \leq i \leq 2) \end{array}$$

Le prédicat $pure(t)$, quant à lui, est vrai si et seulement si t ne contient pas de sous-terme commençant par `percent`.

On note $[g(a)]$ la fonction $g(a)$ si $a \in \mathrm{dom}\, g$, la fonction vide $\{\}$ sinon. On note d'autre part, pour toute liste $al = a_1.\ldots.a_n$ d'actions ($n \geq 1$), et tout graphe G, $G + (src \xrightarrow{al} tgt)$ tout graphe dont les transitions sont celles de γ plus les (st_{i-1}, a_i, st_i), $1 \leq i \leq n$, où $st_0 = src$, $st_n = tgt$, et les st_i, $1 \leq i < n$, sont des états dans state $\setminus$ label, distincts deux à deux et frais (distincts de tout autre label apparaissant dans G ou un autre graphe). Lorsque $n = 0$, on étend cette notation pour dénoter $G + (src \stackrel{\texttt{skip()}}{\longrightarrow} tgt)$. On note enfin $\gamma + \{p_1 \mapsto (src_1 \xrightarrow{al_1} tgt_1), \ldots, p_n \mapsto (src_n \xrightarrow{al_n} tgt_n)\}$ la fonction qui à $p \in \mathrm{dom}\,\gamma \setminus \{p_1, \ldots, p_n\}$ associe $\gamma(p)$, à p_i associe $[\gamma(p_i)] + (src_i \xrightarrow{al_i} tgt_i)$ si $1 \leq i \leq n$.

$$\frac{\begin{array}{c} lab \in \text{label} \setminus ls \quad s \neq r \quad s, r \in \mathrm{dom}\, R \\ t_1 = split_1(m) \quad t_2 = split_2(m) \quad pure(t_1) \quad pure(t_2) \\ \rho \text{ of } S, (\sigma \text{ of } S) + [(\theta \text{ of } S)(s)], -, c \text{ of } S \vdash t_1 \Rightarrow_{\text{term}} u_1 : \texttt{number} \\ \rho \text{ of } S, [(known \text{ of } S) + (\kappa \text{ of } S)(s)], [Unk(s)], c \text{ of } S \vdash u_1 \Rightarrow_{\text{comp}} al_1, v_1, known_1 \\ \rho \text{ of } S, (\sigma \text{ of } S) + [(\theta \text{ of } S)(r)], -, c \text{ of } S \vdash t_2 \Rightarrow_{\text{term}} u_2 : \texttt{number} \\ \rho \text{ of } S, [(known \text{ of } S) + (\kappa \text{ of } S)(r)] \vdash u_2 \Rightarrow_{\text{dec}} pat, known_2 \\ Unk' = Unk + \{r \mapsto [Unk(r)] \cup \mathrm{fv}(u_2)\} \\ x_{pat} \vdash pat \Rightarrow_{\text{brk}} al_2 \\ \kappa = \{s \mapsto known_1, r \mapsto known_2\} \\ \gamma' = \gamma + \{s \mapsto (src \stackrel{al_1.\texttt{send}(v_1)}{\longrightarrow} tgt), r \mapsto (src \xrightarrow{al_2} tgt)\} + \{p \mapsto (src \stackrel{\texttt{skip()}}{\longrightarrow} tgt) | p \in R \setminus \{s, r\}\} \end{array}}{R, S, Unk, \gamma, src, tgt, ls \vdash \texttt{comm}(lab, s, r, m) \Rightarrow_{\text{code}} \gamma', S + \kappa, Unk', ls \cup \{lab\}} \tag{51}$$

On note que lab n'est pas en réalité utilisé dans cette règle. En pratique, `evatrans` a une politique de création de labels qui fait que src vaudra toujours lab ici.

Dans la règle (51), il n'est pas spécifié qu'il faille vérifier que les axiomes α of S impliquent que $u_1 = u_2$. Le traducteur `evatrans` émet cependant un warning s'il n'arrive pas à prouver cette égalité, à l'aide d'un prouveur incomplet mais terminant en temps polynomial.

$$\frac{\begin{array}{c} R, S, Unk, \gamma, src, lab_1, ls \vdash m_1 \Rightarrow_{\text{code}} \gamma_1, S_1, Unk_1, ls_1 \quad \ldots \\ R, S_{n-1}, Unk_{n-1}, \gamma_{n-1}, lab_{n-1}, tgt, ls_{n-1} \vdash m_n \Rightarrow_{\text{code}} \gamma_n, S_n, Unk_n, ls_n \\ lab_1, \ldots, lab_{n-1} \in \text{label frais} \end{array}}{R, S, Unk, \gamma, src, tgt, ls \vdash \texttt{block}(m_1, \ldots, m_n) \Rightarrow_{\text{code}} \gamma_n, S_n, Unk_n, ls_n} \tag{52}$$

$$\frac{\begin{array}{c} p \in \mathrm{dom}\, R \\ \rho \text{ of } S, (\sigma \text{ of } S) + [(\theta \text{ of } S)(p)], -, c \text{ of } S \vdash x \Rightarrow_{\text{term}} s : \texttt{number} \\ \rho \text{ of } S, (\sigma \text{ of } S) + [(\theta \text{ of } S)(p)], -, c \text{ of } S \vdash t \Rightarrow_{\text{term}} u : \texttt{number} \\ \rho \text{ of } S, [(known \text{ of } S) + (\kappa \text{ of } S)(s)], [Unk(s)], c \text{ of } S \vdash u \Rightarrow_{\text{comp}} al, v, known \\ \kappa = \{p \mapsto known\} \\ al' = al.\texttt{let}(\texttt{exact}(s), u) \\ \gamma' = \gamma + \{p \mapsto (src \xrightarrow{al'} tgt)\} + \{p' \mapsto (src \stackrel{\texttt{skip()}}{\longrightarrow} tgt) | p' \in R \setminus \{p\}\} \end{array}}{R, S, Unk, \gamma, src, tgt, ls \vdash \texttt{equals}(p, x, t) \Rightarrow_{\text{code}} \gamma', S + \kappa, Unk, ls} \tag{53}$$

$$\frac{\begin{array}{c} p \in \mathrm{dom}\, R \\ z \in \text{id frais} \\ \rho \text{ of } S, (\sigma \text{ of } S) + [(\theta \text{ of } S)(p)], -, c \text{ of } S \vdash t \Rightarrow_{\text{term}} u : \texttt{number} \\ \rho \text{ of } S, [(known \text{ of } S) + (\kappa \text{ of } S)(s)], [Unk(s)], c \text{ of } S \vdash u \Rightarrow_{\text{comp}} al, v, known \\ \kappa = \{p \mapsto known + \{x \mapsto z\}\} \\ al' = al.\texttt{let}(z, u) \\ \gamma' = \gamma + \{p \mapsto (src \xrightarrow{al'} tgt)\} + \{p' \mapsto (src \stackrel{\texttt{skip()}}{\longrightarrow} tgt) | p' \in R \setminus \{p\}\} \end{array}}{R, S, Unk, \gamma, src, tgt, ls \vdash \texttt{assign}(p, x, t) \Rightarrow_{\text{code}} \gamma', S + \kappa, Unk, ls} \tag{54}$$

$$\frac{\begin{array}{c}
z \in \text{id frais} \\
\begin{array}{lll}
R_0 = \{ & p \mapsto (known, al.\mathtt{let}(z, v)) \\
\quad\mid & \exists u \cdot \\
& \rho \text{ of } S, (\sigma \text{ of } S) + [(\theta \text{ of } S)(p)], -, c \text{ of } S \vdash t \Rightarrow_{\text{term}} u : \mathtt{number} \\
& \rho \text{ of } S, [(known \text{ of } S) + (\kappa \text{ of } S)(p)], [Unk(p)], c \text{ of } S \vdash u \Rightarrow_{\text{comp}} al, v, known \}
\end{array} \\
c_1 = \mathtt{case}(t_1, u_{11}, \ldots, u_{1m_1}) \quad \ldots \\
c_n = \mathtt{case}(t_n, u_{n1}, \ldots, u_{nm_n}) \\
lab, lab_1, \ldots, lab_n \in \text{label frais} \\
\begin{array}{lll}
\gamma_0 = \gamma & + & \{p \mapsto (src \xrightarrow{al} lab) \\
& & \mid p \in \operatorname{dom} R_0, (known, al) = R_0(p)\} \\
& + & \{p \mapsto (lab \xrightarrow{\mathtt{let(exact(}v\mathtt{)},z\mathtt{)}} lab_i) \\
& & \mid p \in \operatorname{dom} R_0, 1 \leq i \leq n, (known, al) = R_0(p), known \vdash t_i \Rightarrow_{loc} v\}
\end{array} \\
\kappa = \{p \mapsto known \mid p \in \operatorname{dom} R_0, (known, al) = R_0(p)\} \quad S_0 = S + \kappa \\
\operatorname{dom} R_0, S_0, Unk, \gamma_0, lab_1, tgt, ls \vdash \mathtt{block}(u_{11}, \ldots, u_{1m_1}) \Rightarrow_{\text{code}} \gamma_1, S_1, Unk_1, ls_1 \\
\ldots \\
\operatorname{dom} R_0, S_{n-1}, Unk_{n-1}, \gamma_{n-1}, lab_n, tgt, ls_{n-1} \vdash \mathtt{block}(u_{n1}, \ldots, u_{nm_n}) \Rightarrow_{\text{code}} \gamma_n, S_n, Unk_n, ls_n \\
\gamma' = \gamma_n + \{p \mapsto (src \xrightarrow{\mathtt{skip()}} tgt) \mid p \in R \setminus \operatorname{dom} R_0\}
\end{array}}{R, S, Unk, \gamma, src, tgt, ls \vdash \mathtt{switch}(t, c_1, \ldots, c_n) \Rightarrow_{\text{code}} \gamma', S_n, Unk_n, ls_n} \tag{55}$$

On forme le jugement $S \vdash m \Rightarrow_{\text{msgs}} start, \gamma, S'$, où $m \in$ Message, $\gamma \in \mathtt{id} \tilde{\rightarrow} \mathbb{P}(\text{state} \times \text{state} \times \text{Action})$ et $start \in$ state est l'état de départ de tous les graphes. On définit la fonction $roles(m)$, qui calcule l'ensemble des noms de rôles utilisés dans le protocole m:

$$\begin{array}{rcl}
roles(\mathtt{comm}(lab, s, r, t)) & = & \{s, r\} \\
roles(\mathtt{equals}(p, x, t)) & = & \{p\} \\
roles(\mathtt{assign}(p, x, t)) & = & \{p\} \\
roles(\mathtt{block}(t_1, \ldots, t_n)) & = & roles(t_1) \cup \ldots \cup roles(t_n) \\
roles(\mathtt{switch}(t, c_1, \ldots, c_n)) & = & roles(c_1) \cup \ldots \cup roles(c_n) \\
roles(\mathtt{case}(t, t_1, \ldots, t_n)) & = & roles(t_1) \cup \ldots \cup roles(t_n)
\end{array}$$

$$\frac{roles(m), S, \{\}, \{\}, start, \mathtt{end.}, \{\mathtt{end.}\} \vdash m \Rightarrow_{\text{code}} \gamma, S', ls'}{S \vdash m \Rightarrow_{\text{msgs}} start, \gamma, S'} \tag{56}$$

4.2.7 Déclarations de sessions

On définit un jugement $\rho, \sigma, c \vdash ss \Rightarrow_{\text{sess}} \varsigma$, où ρ : id$\tilde{\rightarrow}$Type, σ : id$\tilde{\rightarrow}$Term, c : Type $\times$ Type$\tilde{\rightarrow}$Term $\rightarrow$ Term, $ss \in$ Sessions, ς : label$\tilde{\rightarrow}[\mathbb{P}(\text{id})] \times (\text{id}\tilde{\rightarrow}\text{Term}) \times \mathbb{P}(\text{id})$, qui précompile les déclarations de sessions: ς associe à chaque identificateur de session un triplet formé d'un ensemble optionnel d'identificateurs (les variables privées, déclarées par `private`, des multi-sessions parallèles, si présent; si absent, c'est une mono-session), d'une substitution (les affectations de termes aux variables), et d'un ensemble d'identificateurs (l'ensemble des rôles présents dans la session).

$$\frac{\begin{array}{c}
\varsigma_0 = \{\} \\
\rho, \sigma, \varsigma_0, c \vdash s_1 \Rightarrow_{\text{ses}} \varsigma_1 \quad \ldots \\
\rho, \sigma, \varsigma_{n-1}, c \vdash s_n \Rightarrow_{\text{ses}} \varsigma_n
\end{array}}{\rho, \sigma, c \vdash \mathtt{sessions}(s_1, \ldots, s_n) \Rightarrow_{\text{sess}} \varsigma_n} \tag{57}$$

On utilise le jugement auxiliaire $\rho, \sigma, \varsigma, c \vdash s \Rightarrow_{\text{ses}} \varsigma'$, où $s \in$ Session. Ce jugement est lui-même défini en fonction de deux autres jugements auxiliaires $\rho, \sigma, \sigma', c \vdash b_1, \ldots, b_n \Rightarrow_{\text{bind}} \sigma''$ et $\vdash ps \Rightarrow_{\text{princ}} prs$.

$$\frac{\begin{array}{c} lab \in \text{label} \setminus \operatorname{dom} \varsigma \\ \vdash ps \Rightarrow_{\text{princ}} prs \\ \rho, \sigma, \{\}, c \vdash b_1, \ldots, b_n \Rightarrow_{\text{bind}} \sigma' \end{array}}{\rho, \sigma, \varsigma, c \vdash \texttt{session}(lab, ps, b_1, \ldots, b_n) \Rightarrow_{\text{ses}} \varsigma + \{lab \mapsto (-, \sigma', prs)\}} \tag{58}$$

Rappelons que l'on note $-$ un champ option absent.

$$\frac{\begin{array}{c} x_1, \ldots, x_n \in \text{id} \\ pr = \texttt{private}(x_1, \ldots, x_m) \\ lab \in \text{label} \setminus \operatorname{dom} \varsigma \\ \vdash ps \Rightarrow_{\text{princ}} prs \\ \rho, \sigma, \{\}, c \vdash b_1, \ldots, b_n \Rightarrow_{\text{bind}} \sigma' \end{array}}{\rho, \sigma, \varsigma, c \vdash \texttt{session-repeat}(pr, lab, ps, b_1, \ldots, b_n) \Rightarrow_{\text{ses}} \varsigma + \{lab \mapsto (\{x_1, \ldots, x_m\}, \sigma', prs)\}} \tag{59}$$

$$\frac{x_1, \ldots, x_n \in \text{id}}{\vdash \texttt{principals-only}(x_1, \ldots, x_n) \Rightarrow_{\text{princ}} \{x_1, \ldots, x_n\}} \tag{60}$$

$$\frac{x_1, \ldots, x_n \in \text{id}}{\vdash \texttt{principals-except}(x_1, \ldots, x_n) \Rightarrow_{\text{princ}} \text{id} \setminus \{x_1, \ldots, x_n\}} \tag{61}$$

$$\frac{}{\rho, \sigma, \sigma', c \vdash \epsilon \Rightarrow_{\text{bind}} \sigma'} \tag{62}$$

$$\frac{\begin{array}{c} x \in \operatorname{dom} \rho \setminus (\operatorname{dom} \sigma \cup \operatorname{dom} \sigma') \\ \rho, \sigma, -, c \vdash t \Rightarrow_{\text{term}} u : \rho(x) \\ u \in \text{id} \setminus \operatorname{dom} \sigma' \\ \rho, \sigma, \sigma' + \{x \mapsto u\}, c \vdash b_2, \ldots, b_n \Rightarrow_{\text{bind}} \sigma'' \end{array}}{\rho, \sigma, \sigma', c \vdash \texttt{bind}(x, t), b_2, \ldots, b_n \Rightarrow_{\text{bind}} \sigma''} \tag{63}$$

On note que u est contraint à être une variable, et non n'importe quel terme.

4.2.8 Formules

On définit un jugement $\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F'$, où ρ : id$\tilde{\to}$Type, σ : id$\tilde{\to}$Term, loc : label $\times$ id$\tilde{\to}$Term$\tilde{\to}$Term, $shared$: $\mathbb{P}$id, c : Type $\times$ Type$\tilde{\to}$Term $\to$ Term, $\tilde{\gamma}$: label $\times$ id$\tilde{\to}\mathbb{P}$(state $\times$ state $\times$ Action), et F, F' : Form.

On note que $\ell = (loc, shared) \neq -$ dans les jugements $\_ \vdash \_ \Rightarrow_{\text{term}} \_ : \_$ ici. Ceci impose que les termes t_i soient équipés d'annotations `locate` permettant de savoir, au moins pour toute variable libre de t_i, dans quel principal elle doit être consultée.

$$\frac{P \in \text{id} \quad \rho, \sigma, (loc, shared), c \vdash t_1 \Rightarrow_{\text{term}} u_1 : \texttt{number} \quad \ldots \quad \rho, \sigma, (loc, shared), c \vdash t_n \Rightarrow_{\text{term}} u_n : \texttt{number}}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{prop}(P, t_1, \ldots, t_n) \Rightarrow_{\text{form}} \texttt{prop}(P, u_1, \ldots, u_n)} \tag{64}$$

Dans la règle suivante, $\tilde{\gamma}(s, p)$ est un graphe G. On définit $\operatorname{states} G$ comme étant l'ensemble $\{src, tgt | (src \xrightarrow{a} tgt) \in G\}$ des sommets de G.

$$\frac{(s, p) \in \operatorname{dom} loc \quad (s, p) \in \operatorname{dom} \tilde{\gamma} \quad lab \in \operatorname{states}(\tilde{\gamma}(s, p)) \cup \{\texttt{end.}\}}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{at}(s, p, lab) \Rightarrow_{\text{form}} \texttt{at}(s, p, lab)} \tag{65}$$

$$\frac{(s, p) \in \operatorname{dom} loc \quad \rho, \sigma, (loc, shared), c \vdash t \Rightarrow_{\text{term}} u : \texttt{number}}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{owns}(s, p, t) \Rightarrow_{\text{form}} \texttt{owns}(s, p, u)} \tag{66}$$

$$\frac{\rho, \sigma, (loc, shared), c \vdash t \Rightarrow_{\text{term}} u : \texttt{number}}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{secret}(t) \Rightarrow_{\text{form}} \texttt{secret}(s, p, u)} \tag{67}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F' \quad \rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash G \Rightarrow_{\text{form}} G'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{implies}(F, G) \Rightarrow_{\text{form}} \texttt{implies}(F', G')} \tag{68}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F_1 \Rightarrow_{\text{form}} F_1' \quad \ldots \quad \rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F_n \Rightarrow_{\text{form}} F_n'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{and}(F_1, \ldots, F_n) \Rightarrow_{\text{form}} \texttt{and}(F_1', \ldots, F_n')} \tag{69}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F_1 \Rightarrow_{\text{form}} F_1' \quad \ldots \quad \rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F_n \Rightarrow_{\text{form}} F_n'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{or}(F_1, \ldots, F_n) \Rightarrow_{\text{form}} \texttt{or}(F_1', \ldots, F_n')} \tag{70}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{not}(F) \Rightarrow_{\text{form}} \texttt{not}(F')} \tag{71}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{E}(F) \Rightarrow_{\text{form}} \texttt{E}(F')} \tag{72}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{A}(F) \Rightarrow_{\text{form}} \texttt{A}(F')} \tag{73}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{F}(F) \Rightarrow_{\text{form}} \texttt{F}(F')} \tag{74}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{G}(F) \Rightarrow_{\text{form}} \texttt{G}(F')} \tag{75}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{P}(F) \Rightarrow_{\text{form}} \texttt{P}(F')} \tag{76}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{Q}(F) \Rightarrow_{\text{form}} \texttt{Q}(F')} \tag{77}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{N}(F) \Rightarrow_{\text{form}} \texttt{N}(F')} \tag{78}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F' \quad \rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash G \Rightarrow_{\text{form}} G'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{U}(F, G) \Rightarrow_{\text{form}} \texttt{U}(F', G')} \tag{79}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F' \quad \rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash G \Rightarrow_{\text{form}} G'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{W}(F, G) \Rightarrow_{\text{form}} \texttt{W}(F', G')} \tag{80}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F' \quad \rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash G \Rightarrow_{\text{form}} G'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{S}(F, G) \Rightarrow_{\text{form}} \texttt{S}(F', G')} \tag{81}$$

$$\frac{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash F \Rightarrow_{\text{form}} F' \quad \rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash G \Rightarrow_{\text{form}} G'}{\rho, \sigma, loc, shared, c, \tilde{\gamma} \vdash \texttt{B}(F, G) \Rightarrow_{\text{form}} \texttt{B}(F', G')} \tag{82}$$

4.2.9 Spécifications

Si σ est une substitution, et t est un terme, on note $t\sigma$ le résultat de l'application de la substitution σ à t:

$$\begin{array}{rcl} x\sigma & = & x \quad \text{si } x \in \mathrm{id} \setminus \mathrm{dom}\,\sigma \\ x\sigma & = & \sigma(x) \quad \text{si } x \in \mathrm{dom}\,\sigma \\ f(t_1, \dots, t_n)\sigma & = & f(t_1\sigma, \dots, t_n\sigma) \end{array}$$

Noter que la substitution s'effectue aussi à travers les patterns. Ceci n'est correct que si la substitution σ remplace des variables par des variables. Ceci est assuré par la règle (63).

On étend cette notation: si G est un graphe ($G = \gamma(p)$ dans la règle (83) ci-dessous), $G\sigma$ est le graphe dont les transitions sont les $src \xrightarrow{t\sigma} tgt$, lorsque $(src \xrightarrow{t} tgt)$ parcourt les transitions de G. De même, si $known$: Term$\tilde{\to}$Term ($known = \big((known \text{ of } S) + (\kappa \text{ of } S)(p)\big)$ dans la règle (83) ci-dessous), $known\sigma$ est la fonction qui à $t \in \mathrm{dom}\, known$ associe $known(t)\sigma$.

$$\frac{\begin{array}{c} \rho_0 = \{\mathtt{**asymk**} \mapsto \mathtt{one\text{-}way\text{-}function}(\mathtt{*D*}, \mathtt{number})\} \\ \begin{array}{rll} c_0 & = \{ & (\mathtt{*D*}, \mathtt{number}) \mapsto (u \mapsto \mathtt{d}(u)), \quad (\mathtt{principal}, \mathtt{number}) \mapsto (u \mapsto \mathtt{p}(u)), \\ & & (\mathtt{*algo*}, \mathtt{number}) \mapsto (u \mapsto \mathtt{a}(u)), \quad (\mathtt{sym\_algo}, \mathtt{*algo*}) \mapsto (u \mapsto \mathtt{sa}(u)), \\ & & (\mathtt{asym\_algo}, \mathtt{*algo*}) \mapsto (u \mapsto \mathtt{aa}(u)) \ \} \end{array} \\ S_0 = \rho_0 \text{ in } S + c_0 \text{ in } S \\ ds = \mathtt{declare}(d_1, \dots, d_n) \\ S_0 \vdash d_1 \Rightarrow_{\mathrm{decl}} S_1 \\ \dots \\ S_{n-1} \vdash d_n \Rightarrow_{\mathrm{decl}} S_n \\ S_n \vdash m \Rightarrow_{\mathrm{msgs}} start, \gamma, S \\ \rho \text{ of } S, \sigma \text{ of } S, c \text{ of } S \vdash ss \Rightarrow_{\mathrm{sess}} \varsigma \\ \tilde{\gamma} = \{(s,p) \mapsto \gamma(p)\sigma | s \in \mathrm{dom}\,\varsigma, (privs, \sigma, prs) = \varsigma(s), p \in \mathrm{dom}\,\gamma \cap prs\} \\ \begin{array}{rcl} loc & = & \{(s,p) \mapsto \big((known \text{ of } S) + (\kappa \text{ of } S)(p)\big)\sigma | s \in \mathrm{dom}\,\varsigma, \\ & & \quad (privs, \sigma, prs) = \varsigma(s), p \in \mathrm{dom}(\kappa \text{ of } S) \cap prs\} \\ shared & = & shared \text{ of } S \\ rp & = & \{(s,p) \mapsto privs | s \in \mathrm{dom}\,\varsigma, \\ & & \quad (privs, \sigma, prs) = \varsigma(s), privs \neq -, p \in \mathrm{dom}\,\gamma \cap prs\} \end{array} \\ as = \mathtt{assume}(F_1, \dots, F_p) \\ \rho \text{ of } S, \sigma \text{ of } S, loc, shared, c \text{ of } S, \tilde{\gamma} \vdash F_1 \Rightarrow_{\mathrm{form}} F'_1 \\ \dots \\ \rho \text{ of } S, \sigma \text{ of } S, loc, shared, c \text{ of } S, \tilde{\gamma} \vdash F_p \Rightarrow_{\mathrm{form}} F'_p \\ as' = \mathtt{assume}(F'_1, \dots, F'_p) \\ cs = \mathtt{claim}(G_1, \dots, G_q) \\ \rho \text{ of } S, \sigma \text{ of } S, loc, shared, c \text{ of } S, \tilde{\gamma} \vdash G_1 \Rightarrow_{\mathrm{form}} G'_1 \\ \dots \\ \rho \text{ of } S, \sigma \text{ of } S, loc, shared, c \text{ of } S, \tilde{\gamma} \vdash G_q \Rightarrow_{\mathrm{form}} G'_q \\ cs' = \mathtt{claim}(G'_1, \dots, G'_q) \end{array}}{\vdash \mathtt{spec}(ds, m, ss, as, cs) \Rightarrow_{\mathrm{spec}} \rho \text{ of } S, \sigma \text{ of } S, \alpha \text{ of } S, start, \tilde{\gamma}, loc, shared, rp, as', cs'} \tag{83}$$

Pour tout ρ : id$\tilde{\to}$Type, on définit de façon sous-déterminée un terme Types(ρ) tel que Types(ρ) est de la forme $\mathtt{types}(\mathtt{type}(x_1, \tau_1), \dots, \mathtt{type}(x_n, \tau_n))$, avec $\rho = \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\}$, x_1, …, x_n étant disjointes deux à deux.

De même, pour tout σ : id$\tilde{\to}$Term, on définit Values(σ) comme étant n'importe quel terme $\mathtt{values}(\mathtt{alias}(x_1, t_1), \dots, \mathtt{alias}(x_n, t_n))$ tel que $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$, x_1, …, x_n étant disjointes deux à deux.

Pour tout $\alpha \in \mathbb{P}$(Term $\times$ Term $\times$ Type $\times$ (id$\tilde{\to}$Type)), on pose Axioms(α) n'importe quel terme de la forme $\mathtt{axioms}(\mathtt{axiom}(s_1, t_1, \tau_1, l_1), \dots, \mathtt{axiom}(s_n, t_n, \tau_n, l_n))$, où $\alpha = \{(s_1, t_1, \tau_1, \rho_1), \dots, (s_n, t_n, \tau_n, \rho_n)\}$, Types($\rho_1$) = $\mathtt{types}(l_1)$, …, Types(ρ_n) = $\mathtt{types}(l_n)$. (Où l'on a supposé que l_1, …, l_n étaient des listes de Type-declaration.)

Pour tout $known$: Term$\tilde{\to}$Term, Know($known$) est un terme de la forme $\texttt{knows}(\texttt{as}(s_1, t_1), \ldots, \texttt{as}(s_n, t_n))$, où $known = \{s_1 \mapsto t_1, \ldots, s_n \mapsto t_n\}$.

Pour tout graphe $G \in \mathbb{P}$(state × state × Action), Trans(G) est toute liste de la forme $\texttt{trans}(src_1, tgt_1, a_1), \ldots, \texttt{trans}(src_n, tgt_n, a_n)$, où $G = \{(src_1, tgt_1, a_1), \ldots, (src_n, tgt_n, a_n)$.

Deux graphes G et G' sont *équivalents* modulo l'état $start$, ce que l'on notera $G \equiv_{start} G'$, si et seulement s'ils vérifient exactement les mêmes formules d'état closes (cf. section 5.5) en l'état $start$. Ceci permet de remplacer un graphe G par un graphe G' plus simple mais ayant le même comportement. Par exemple, `evatrans` élimine les états inaccessibles depuis $start$, et les transitions $\texttt{skip}()$ si possible[3].

Pour tout $ps \in \mathbb{P}(\text{id})$, on définit Privates($ps$) comme étant tout terme $\texttt{private}(x_1, \ldots, x_n)$, avec $ps = \{x_1, \ldots, x_n\}$.

Pour tout $\tilde{\gamma}$: label×id$\tilde{\to}\mathbb{P}$(state×state×Action), loc : label×id$\tilde{\to}$Term$\tilde{\to}$Term, rp : label×id$\tilde{\to}\mathbb{P}$(id), et $start \in$ state, on pose System$(start, \tilde{\gamma}, loc, rp) = \texttt{system}(\pi_1, \ldots, \pi_n)$, où $\pi_1, \ldots, \pi_n$ sont n processus construits comme suit. Énumérons $\operatorname{dom}\tilde{\gamma} \cup \operatorname{dom} loc$ sous la forme $\{(s_1, p_1), \ldots, (s_n, p_n)\}$. Pour chaque i, $1 \le i \le n$, posons $N_i = s_i.p_i$, la concaténation du nom de session s_i et du nom de processus p_i. Posons $known_i = [loc(s_i, p_i)]$ (cf. section 4.2.6 pour la notation $[g(a)]$), k_i = Know($known_i$); posons encore $G_i = [\tilde{\gamma}(s_i, p_i)]$, g_i = Trans(G'_i), où G'_i est n'importe quel graphe tel que $G_i \equiv_{start} G'_i$. Alors π_i est $\texttt{process}(N_i, start, k_i, g_i)$ si $rp(s, p) = -$, et est $\texttt{repeat-process}(\text{Privates}(rp(s, p)), N_i, start, k_i, g_i)$ sinon.

Le résultat de la traduction d'une spécification LAEVA $spec$ en PEVA réussit si et seulement si on peut dériver:

$$\vdash spec \Rightarrow_{\text{spec}} \rho, \sigma, \alpha, start, \tilde{\gamma}, loc, shared, rp, as, cs$$

Ce résultat est alors le terme:

$$\texttt{compiled-spec}(\text{Types}(\rho), \text{Values}(\sigma), \text{Axioms}(\alpha), \text{System}(start, \tilde{\gamma}, loc, rp), as, cs)$$

5 Sémantique

5.1 Valeurs

Étant donné un ensemble d'axiomes $\alpha \in \mathbb{P}$(Term × Term × Type × (id$\tilde{\to}$Type)), on définit la relation d'équivalence $\approx_\alpha$ sur Term comme étant la plus petite qui vérifie tous les axiomes de α. Par vérifier les axiomes de α, on entend que $s\sigma \approx_\alpha t\sigma$ pour tout axiome $(s, t, \tau, \rho) \in \alpha$, pour toute substitution σ de domaine inclus dans $\operatorname{dom}\rho$.

On notera que cette définition ignore les types τ ainsi que ceux donnés par ρ. La restriction de $\approx_\alpha$ aux termes bien typés est identique à la relation souhaitée, qui ne quantifierait que sur les substitutions bien typées et respectant les types donnés par ρ. (Ceci dépend du fait qu'aucun de nos types n'est vide, car contenant une infinité de variables.) Cette dernière notion est un peu plus compliquée à définir, car dépendant aussi d'un environnement de typage global; d'où son abandon.

Dans la suite, au lieu de raisonner sur la structure quotient Term/$\approx_\alpha$, on raisonnera sur des termes. Toutes les constructions (sauf $FST(t)$ dans la règle (102)) seront invariantes par $\approx_\alpha$.

5.2 Formalisation de l'intrus

Le jugement $E \mapsto_\alpha t$ définit les conditions sous lesquelles un terme t (de type `number`) est déductible par l'intrus, lorsque l'intrus a acquis les termes (de type `number`) de l'ensemble de termes E. Il est paramétré par un ensemble d'axiomes α, qui est utilisé dans la règle ($\approx_\alpha$). Encore une fois, on raisonne en réalité sur des termes non typés.

[3]Formellement, l'implémentation actuelle est non conforme à la spécification (ce document). En effet, la condition d'équivalence implique qu'on ne puisse pas renommer, supprimer ou rajouter d'état visible (dans label, ne commençant pas par '`%`'), puisque le prédicat `at` permet de tester la présence en un état visible. Or `evatrans` remplace deux transitions de la forme $lab \xrightarrow{\texttt{skip}()} lab' \xrightarrow{a} lab''$ par une transition $lab \xrightarrow{a} lab''$, supprimant l'état inaccessible lab', même s'il était visible. Il est facile de corriger le code pour tenir compte de ce bug. Il n'est cependant pas clair que ce soit souhaitable, et une relation moins fine que $\equiv_{start}$ ou une logique moins finie que celle proposée est probablement ce qu'il faudrait définir.

$$\frac{}{E, t \mapsto_\alpha t}\,(Ax) \qquad \frac{E \mapsto_\alpha t \quad t \approx_\alpha t'}{E \mapsto_\alpha t'}\,(\approx_\alpha)$$

$$\frac{E \mapsto_\alpha t \quad E \mapsto_\alpha k}{E \mapsto_\alpha \texttt{crypt}(a, t, k)}\,(CryptI) \qquad \frac{E \mapsto_\alpha \texttt{crypt}(\texttt{a}(\texttt{sa}(a_0)), t, k) \quad E \mapsto_\alpha k}{E \mapsto_\alpha t}\,(CryptE/sym)$$

$$\frac{\begin{array}{c} E \mapsto_\alpha \texttt{crypt}(\texttt{a}(\texttt{aa}(a_0)), t, k) \quad k = f(\texttt{a}(\texttt{aa}(a_0)), t_1, \ldots, t_n) \\ f \in \operatorname{dom} Inv \quad f' = Inv(f) \quad k^{-1} = f'(\texttt{a}(\texttt{aa}(a_0)), t_1, \ldots, t_n) \\ E \mapsto_\alpha k^{-1} \end{array}}{E \mapsto_\alpha t}\,(CryptE/asym)$$

Dans la règle $(CryptI)$, noter que l'intrus n'a pas besoin de connaître l'algorithme de chiffrement a: on suppose que l'intrus connaît *tous* les algorithmes de chiffrement. La règle $(CryptE/sym)$ traite du déchiffrement par des algorithmes à clés symétriques, la règle $(CryptE/asym)$ du déchiffrement par algorithmes à clés asymétriques; Inv est définie en section 4.2.4. On notera que l'intrus n'a aucune règle pour déchiffrer un chiffrement mal typé, de la forme $\texttt{crypt}(a, t, k)$ où a n'est ni de la forme $\texttt{a}(\texttt{sa}(a_0))$ ni de la forme $\texttt{a}(\texttt{aa}(a_0))$; de même si a est de la seconde forme mais k n'est pas de la forme $f(a', t_1, \ldots, t_n)$ avec $f \in Inv$, ou bien est de cette forme mais avec $a \neq a'$.

$$\frac{E \mapsto_\alpha t_1 \quad \ldots \quad E \mapsto_\alpha t_n}{E \mapsto_\alpha \texttt{tuple}(t_1, \ldots, t_n)}\,(TupleI) \quad \frac{E \mapsto_\alpha \texttt{tuple}(t_1, \ldots, t_n)}{E \mapsto_\alpha t_i}\,(TupleE_i)(1 \leq i \leq n)$$

L'intrus connaît tous les algorithmes de chiffrement:

$$\frac{}{E \mapsto_\alpha \texttt{a}(a_0)}\,(AlgI)$$

Ensuite, l'intrus peut construire et détruire (récupérer un argument de) tout ce qui est formé avec `apply`, mais ne sait pas détruire les `hash-apply` ni construire les `secret-apply`. Il n'y a aucune règle sur les `hash-secret-apply`: l'intrus ne sait ni les construire ni les détruire.

$$\frac{E \mapsto_\alpha t_1 \quad \ldots \quad E \mapsto_\alpha t_n}{E \mapsto_\alpha \texttt{apply}(f, t_1, \ldots, t_n)}\,(AppI) \quad \frac{E \mapsto_\alpha \texttt{apply}(f, t_1, \ldots, t_n)}{E \mapsto_\alpha t_i}\,(AppE_i)(1 \leq i \leq n)$$

$$\frac{E \mapsto_\alpha t_1 \quad \ldots \quad E \mapsto_\alpha t_n}{E \mapsto_\alpha \texttt{hash-apply}(f, t_1, \ldots, t_n)}\,(HAppI) \quad \frac{E \mapsto_\alpha \texttt{secret-apply}(f, t_1, \ldots, t_n)}{E \mapsto_\alpha t_i}\,(SAppE_i)(1 \leq i \leq n)$$

Quant aux constructions de clés publiques et privées, elles sont presque symétriques, à ceci près que les parties privées sont analogues aux `secret-apply`: l'intrus est supposé ne pas pouvoir appliquer ni `apply-privk` ni `hash-apply-privk`.

$$\frac{E \mapsto_\alpha t_1 \quad \ldots \quad E \mapsto_\alpha t_n}{E \mapsto_\alpha \texttt{apply-pubk}(a, f, t_1, \ldots, t_n)}\,(AppPubI) \quad \frac{E \mapsto_\alpha \texttt{apply-pubk}(a, f, t_1, \ldots, t_n)}{E \mapsto_\alpha t_i}\,(AppPubE_i)(1 \leq i \leq n)$$

$$\frac{E \mapsto_\alpha t_1 \quad \ldots \quad E \mapsto_\alpha t_n}{E \mapsto_\alpha \texttt{hash-apply-pubk}(a, f, t_1, \ldots, t_n)}\,(HAppPubI) \quad \frac{E \mapsto_\alpha \texttt{apply-privk}(a, f, t_1, \ldots, t_n)}{E \mapsto_\alpha t_i \qquad (1 \leq i \leq n)}\,(AppPrivE_i)$$

Il n'y a aucune règle sur les `lambda-pubk` et les `lambda-privk`, qui ne fournissent pas des termes de type `number`.

Pour traiter du prédicats `owns`, on a un second jugement $E; \Omega \vdash A \ni_\alpha t$, où A est censé être un principal honnête ou l'intrus, t est un terme de type `number`, E est un ensemble de termes de type `number`, et Ω est une fonction qui à tout principal honnête associe un ensemble de termes, censé être l'ensemble des termes qu'il a lui-même fabriqués (à l'exclusion de ceux qu'il retransmet).

Formellement, $A \in \mathrm{P} \cup \{\mathtt{I}\}$, où l'ensemble P des principaux est défini comme l'union disjointe de id et de id $\times \mathbb{N}$, et $\mathtt{I} \notin \mathrm{P}$ désigne l'intrus, $t \in$ Term, $\Omega : \mathrm{P} \to \mathbb{P}(\text{Term})$, et l'on a les règles:

$$\frac{t \in \Omega(A)}{E; \Omega \vdash A \ni_\alpha t}\,(Creator) \qquad \frac{E; \Omega \vdash A \ni_\alpha t \quad t \approx_\alpha t'}{E; \Omega \vdash A \ni_\alpha t'}\,(\approx_\alpha) \qquad \frac{E \mapsto_\alpha t}{E; \Omega \vdash \mathtt{I} \ni_\alpha t}\,(Intruder)$$

5.3 Actions

On définit la sémantique des actions par un jugement $E; D, C, \sigma \vdash_\alpha a \Rightarrow_{\mathrm{act}} E'; D', C', \sigma'$, où $E, E' \in \mathbb{P}(\text{Term})$ (ensembles de termes connus de l'intrus), $D, D' \in \mathbb{P}(\text{Term})$ (ensembles de données, de type $*\mathtt{D}*$, déjà créées dans le passé), $C, C' \in \mathbb{P}(\text{Term})$ (ensembles de données créés par le processus), σ, σ' : id$\tilde{\to}$Term. Dans le cas d'une action `recv` ou `let`, on utilise un jugement auxiliaire $\sigma; t \vdash_\alpha pat \Rightarrow_{\mathrm{match}} \sigma'$, où σ, σ' : id$\tilde{\to}$Term, $t \in$ Term, $pat \in$ Pattern.

$$\frac{x \in \text{id}}{\sigma; t \vdash_\alpha x \Rightarrow_{\mathrm{match}} \sigma + \{x \mapsto t\}} \tag{84}$$

$$\frac{u\sigma \approx_\alpha t}{\sigma; t \vdash_\alpha \mathtt{exact}(u) \Rightarrow_{\mathrm{match}} \sigma} \tag{85}$$

$$\frac{a\sigma \approx_\alpha \mathtt{a}(\mathtt{sa}(a_0)) \quad t \approx_\alpha \mathtt{crypt}(a\sigma, t_1, k\sigma) \quad \sigma; t_1 \vdash_\alpha pat \Rightarrow_{\mathrm{match}} \sigma'}{\sigma; t \vdash_\alpha \mathtt{crypt}(a, pat, k) \Rightarrow_{\mathrm{match}} \sigma'} \tag{86}$$

$$\frac{\begin{array}{c} a\sigma \approx_\alpha \mathtt{a}(\mathtt{aa}(a_0)) \quad k\sigma \approx_\alpha f(a\sigma, s_1, \ldots, s_n) \\ f \in \operatorname{dom} Inv \quad f' = Inv(f) \quad k^{-1} = f'(a\sigma, s_1, \ldots, s_n) \\ t \approx_\alpha \mathtt{crypt}(a\sigma, t_1, k^{-1}) \\ \sigma; t_1 \vdash_\alpha pat \Rightarrow_{\mathrm{match}} \sigma' \end{array}}{\sigma; t \vdash_\alpha \mathtt{crypt}(a, pat, k) \Rightarrow_{\mathrm{match}} \sigma'} \tag{87}$$

On rappelle que Inv est définie en section 4.2.4.

Pour les patterns qui ont plusieurs sous-patterns en arguments, on effectue le pattern-matching de gauche à droite. C'est un choix arbitraire.

$$\frac{\begin{array}{c} t \approx_\alpha \mathtt{apply}(f, t_1, \ldots, t_n) \\ \sigma; t_1 \vdash_\alpha pat_1 \Rightarrow_{\mathrm{match}} \sigma_1 \\ \ldots \\ \sigma_{n-1}; t_n \vdash_\alpha pat_n \Rightarrow_{\mathrm{match}} \sigma_n \end{array}}{\sigma; t \vdash_\alpha \mathtt{apply}(f, pat_1, \ldots, pat_n) \Rightarrow_{\mathrm{match}} \sigma_n} \tag{88}$$

$$\frac{\begin{array}{c} t \approx_\alpha \mathtt{secret\text{-}apply}(f, t_1, \ldots, t_n) \\ \sigma; t_1 \vdash_\alpha pat_1 \Rightarrow_{\mathrm{match}} \sigma_1 \\ \ldots \\ \sigma_{n-1}; t_n \vdash_\alpha pat_n \Rightarrow_{\mathrm{match}} \sigma_n \end{array}}{\sigma; t \vdash_\alpha \mathtt{secret\text{-}apply}(f, pat_1, \ldots, pat_n) \Rightarrow_{\mathrm{match}} \sigma_n} \tag{89}$$

$$\frac{\begin{array}{c} t \approx_\alpha \mathtt{tuple}(t_1, \ldots, t_n) \\ \sigma; t_1 \vdash_\alpha pat_1 \Rightarrow_{\mathrm{match}} \sigma_1 \\ \ldots \\ \sigma_{n-1}; t_n \vdash_\alpha pat_n \Rightarrow_{\mathrm{match}} \sigma_n \end{array}}{\sigma; t \vdash_\alpha \mathtt{tuple}(pat_1, \ldots, pat_n) \Rightarrow_{\mathrm{match}} \sigma_n} \tag{90}$$

$$\frac{t \approx_\alpha \mathtt{d}(t_1) \quad \sigma; t_1 \vdash_\alpha pat \Rightarrow_{\mathrm{match}} \sigma'}{\sigma; t \vdash_\alpha \mathtt{d}(pat) \Rightarrow_{\mathrm{match}} \sigma'} \tag{91}$$

$$\frac{t \approx_\alpha \mathtt{p}(t_1) \quad \sigma; t_1 \vdash_\alpha pat \Rightarrow_{\text{match}} \sigma'}{\sigma; t \vdash_\alpha \mathtt{p}(pat) \Rightarrow_{\text{match}} \sigma'} \tag{92}$$

$$\frac{t \approx_\alpha \mathtt{a}(t_1) \quad \sigma; t_1 \vdash_\alpha pat \Rightarrow_{\text{match}} \sigma'}{\sigma; t \vdash_\alpha \mathtt{a}(pat) \Rightarrow_{\text{match}} \sigma'} \tag{93}$$

$$\frac{t \approx_\alpha \mathtt{sa}(t_1) \quad \sigma; t_1 \vdash_\alpha pat \Rightarrow_{\text{match}} \sigma'}{\sigma; t \vdash_\alpha \mathtt{sa}(pat) \Rightarrow_{\text{match}} \sigma'} \tag{94}$$

$$\frac{t \approx_\alpha \mathtt{aa}(t_1) \quad \sigma; t_1 \vdash_\alpha pat \Rightarrow_{\text{match}} \sigma'}{\sigma; t \vdash_\alpha \mathtt{aa}(pat) \Rightarrow_{\text{match}} \sigma'} \tag{95}$$

$$\frac{t \approx_\alpha \mathtt{vanilla}()}{\sigma; t \vdash_\alpha \mathtt{vanilla}() \Rightarrow_{\text{match}} \sigma} \tag{96}$$

$$\frac{\begin{array}{c} t \approx_\alpha \mathtt{apply\text{-}pubk}(a, f, t_1, \ldots, t_n) \\ \sigma; t_1 \vdash_\alpha pat_1 \Rightarrow_{\text{match}} \sigma_1 \\ \ldots \\ \sigma_{n-1}; t_n \vdash_\alpha pat_n \Rightarrow_{\text{match}} \sigma_n \end{array}}{\sigma; t \vdash_\alpha \mathtt{apply\text{-}pubk}(a, f, pat_1, \ldots, pat_n) \Rightarrow_{\text{match}} \sigma_n} \tag{97}$$

$$\frac{\begin{array}{c} t \approx_\alpha \mathtt{apply\text{-}privk}(a, f, t_1, \ldots, t_n) \\ \sigma; t_1 \vdash_\alpha pat_1 \Rightarrow_{\text{match}} \sigma_1 \\ \ldots \\ \sigma_{n-1}; t_n \vdash_\alpha pat_n \Rightarrow_{\text{match}} \sigma_n \end{array}}{\sigma; t \vdash_\alpha \mathtt{apply\text{-}privk}(a, f, pat_1, \ldots, pat_n) \Rightarrow_{\text{match}} \sigma_n} \tag{98}$$

On passe maintenant aux actions elles-mêmes. En premier, `new` crée une nouvelle donnée (de type `*D*`, normalement). Cette donnée étant créée par le principal courant, on l'ajoute aussi à C.

$$\frac{t \notin D \quad \sigma; t \vdash_\alpha pat \Rightarrow_{\text{match}} \sigma'}{E; D, C, \sigma \vdash_\alpha \mathtt{new}(pat) \Rightarrow_{\text{act}} E; D \cup \{t\}, C \cup \{t\}, \sigma'} \tag{99}$$

$$\frac{\sigma; t \vdash_\alpha pat \Rightarrow_{\text{match}} \sigma'}{E; D, C, \sigma \vdash_\alpha \mathtt{let}(pat, t) \Rightarrow_{\text{act}} E; D, C, \sigma'} \tag{100}$$

La réception de message opère en acceptant n'importe quel message que l'intrus peut fabriquer:

$$\frac{E \mid\!\!\mapsto_\alpha t \quad \sigma; t \vdash_\alpha pat \Rightarrow_{\text{match}} \sigma'}{E; D, C, \sigma \vdash_\alpha \mathtt{recv}(pat) \Rightarrow_{\text{act}} E; D, C, \sigma'} \tag{101}$$

L'envoi de message consiste à ajouter le message qui est la valeur de t dans σ, soit $t\sigma$, à l'ensemble des messages E connus par l'intrus. On ajoute aussi à C tous les messages créés par le principal courant à l'aide du terme t. Par convention, il s'agit des valeurs de tous les sous-termes non variables u de t dans σ. On note $FST(t)$ l'ensemble de tous les sous-termes non variables de t. (Noter que cette notion n'est *pas* définie modulo $\approx_\alpha$.)

$$\frac{}{E; D, C, \sigma \vdash_\alpha \mathtt{send}(t) \Rightarrow_{\text{act}} E \cup \{t\sigma\}; D, C \cup \{u\sigma | u \in FST(t)\}, \sigma} \tag{102}$$

$$\frac{}{E; D, C, \sigma \vdash_\alpha \mathtt{skip}() \Rightarrow_{\text{act}} E; D, C, \sigma} \tag{103}$$

5.4 Processus

Étant donnés $\tilde{\gamma}$: label $\times$ id$\tilde{\to}\mathbb{P}$(state $\times$ state $\times$ Action), rp : label $\times$ id$\tilde{\to}\mathbb{P}(\text{id})$, et $start \in$ state, on définit l'ensemble des principaux actifs comme étant l'ensemble des $s.p$, où $(s,p) \in \operatorname{dom}\tilde{\gamma}$ tels que $rp(s,p) = -$ (pour les rôles en mono-session), union l'ensemble des $(s.p, n)$, où $(s,p) \in \operatorname{dom}\tilde{\gamma}$ tels que $rp(s,p) \neq -$, et $n \in \mathbb{N}$ (rôles en multi-session parallèle). En tout état de cause, les principaux sont dans l'ensemble $\mathrm{P} = \mathrm{id} \cup \mathrm{id} \times \mathbb{N}$.

Une configuration globale $\mathfrak{G}$ du système est un quintuplet $(E; D, \Omega, \Sigma, pc)$, où $E \in \mathbb{P}(\text{Term})$ est l'état de connaissance de l'intrus, $D \in \mathbb{P}(\text{Term})$ est l'ensemble des données déjà créées, Ω : P $\to \mathbb{P}(\text{Term})$ envoie chaque principal vers l'ensemble des termes qu'il a fabriqués, Σ : P $\to$ id$\tilde{\to}$Term envoie chaque principal vers un environnement σ contenant les valeurs de ses variables, et pc : P $\to$ state donne le point de programme courant de chaque principal.

Le système global évolue via la relation de transition $\tilde{\gamma}, rp \vdash_\alpha \mathfrak{G} \Rightarrow_{\text{exec}} \mathfrak{G}'$. On a d'abord le cas des principaux en mono-session:

$$\frac{\begin{array}{c}(p,s) \in \operatorname{dom}\tilde{\gamma} \quad (p,s) \in \operatorname{dom} rp \Longrightarrow rp(p,s) = - \\ (pc \xrightarrow{a} pc') \in \tilde{\gamma}(p,s) \\ E; D, \Omega(p.s), \Sigma(p.s) \vdash_\alpha a \Rightarrow_{\text{act}} E'; D', C', \sigma' \\ \Omega' = \Omega + \{p.s \mapsto C'\} \quad \Sigma' = \Sigma + \{p.s \mapsto \sigma'\}\end{array}}{\tilde{\gamma}, rp \vdash_\alpha E; D, \Omega, \Sigma, pc \Rightarrow_{\text{exec}} E'; D', \Omega', \Sigma', pc'} \tag{104}$$

Les principaux en multi-session évoluent comme une mise en parallèle d'un nombre infini de copies $(p.s, n)$, $n \in \mathbb{N}$, du processus $p.s$:

$$\frac{\begin{array}{c}(p,s) \in \operatorname{dom}\tilde{\gamma} \quad (p,s) \in \operatorname{dom} rp \text{ et } rp(p,s) \neq - \quad n \in \mathbb{N} \\ (pc \xrightarrow{a} pc') \in \tilde{\gamma}(p,s) \\ E; D, \Omega(p.s,n), \Sigma(p.s,n) \vdash_\alpha a \Rightarrow_{\text{act}} E'; D', C', \sigma' \\ \Omega' = \Omega + \{(p.s,n) \mapsto C'\} \quad \Sigma' = \Sigma + \{(p.s,n) \mapsto \sigma'\}\end{array}}{\tilde{\gamma}, rp \vdash_\alpha E; D, \Omega, \Sigma, pc \Rightarrow_{\text{exec}} E'; D', \Omega', \Sigma', pc'} \tag{105}$$

Une configuration *initiale* est telle que les états locaux de tous les processus sont à $start$. On impose de plus que la connaissance de l'intrus initiale E, ainsi que l'ensemble des termes déjà créés, contiennent un ensemble infini D_0 de variables (dans id). On suppose de plus que id $\setminus D_0$ est lui aussi infini. Cette astuce permet de modéliser la création de nonces et de clés par l'intrus, et remplace une règle qui énoncerait que $E \models_\alpha t$ pour tout $t \in D_0$. On impose de plus qu'aucun principal n'ait initialement créé de donnée ($\Omega(A) = \emptyset$) et que toute paire de copies d'un même principal en multi-session parallèle ait les mêmes valeurs pour leurs variables non privées.

$$\begin{array}{rcl} Init_{start,rp}(E; D, \Omega, \Sigma, pc) & \Leftrightarrow & E \supseteq D_0 \\ & \text{et} & D \supseteq D_0 \\ & \text{et} & \forall A \in \mathrm{P} \cdot (pc(A) = start \text{ et } \Omega(A) = \emptyset) \\ & \text{et} & \forall (s,p) \in \operatorname{dom} rp \cap \operatorname{dom}\Sigma \cdot rp(s,p) \neq - \Longrightarrow \\ & & \qquad \forall n, n' \in \mathbb{N} \cdot \forall x \in \operatorname{dom}\Sigma(s.p,n) \setminus rp(s,p) \cdot \\ & & \qquad\quad x \in \operatorname{dom}\Sigma(s.p,n') \text{ et } \Sigma(s.p,n)(x) = \Sigma(s.p,n')(x) \end{array}$$

Ainsi, $start, \tilde{\gamma}, rp$ définissent un automate dont les états sont les configurations globales $\mathfrak{G}$, les transitions de $\mathfrak{G}$ vers $\mathfrak{G}'$ sont données par les instances dérivables de $\tilde{\gamma}, rp \vdash_\alpha \mathfrak{G} \Rightarrow_{\text{exec}} \mathfrak{G}'$, et les états initiaux sont ceux vérifiant $Init_{start,rp}$.

On note cet automate $\mathcal{A}_{start,\tilde{\gamma},rp}$.

5.5 Formules

Un *chemin* $\bar{\mathfrak{G}}$ dans un automate $\mathcal{A}$, partant de l'état $\mathfrak{G}$, est une suite infinie $\mathfrak{G}_0$, $\mathfrak{G}_1$, ..., $\mathfrak{G}_k$, ..., de configurations telles que $\mathfrak{G}_0 = \mathfrak{G}$, et pour tout $i \in \mathbb{N}$, soit $\mathfrak{G}_i = \mathfrak{G}_{i+1}$ soit il existe une transition de $\mathfrak{G}_i$

vers $\mathfrak{G}_{i+1}$ dans $\mathcal{A}$. On note $\bar{\mathfrak{G}}(k) = \mathfrak{G}_k$, et $\bar{\mathfrak{G}}(\geq k)$ le suffixe $\mathfrak{G}_k, \mathfrak{G}_{k+1}, \ldots, \mathfrak{G}_i, \ldots$ On remarque que l'on autorise le bégaiement ($\mathfrak{G}_i = \mathfrak{G}_{i+1}$) sur les chemins, et même le bégaiement infini (pour tout $i \geq k$, $\mathfrak{G}_i = \mathfrak{G}_k$). De la sorte, il existe toujours un chemin partant de n'importe quel état.

Une formule F est évaluée sur un chemin $\bar{\mathfrak{G}}$ à une position $k \in \mathbb{N}$. La logique d'EVA est, comme on le constatera, une variante de NCTL* [LS95]. Il ne lui manque que les opérateurs "next-state" et "previous-state", et elle a une sémantique invariante par bégaiement par construction, ce qui en fait un lointain cousin de TLA [Lam94].

Pour tout $(s,p) \in \operatorname{dom} rp$, on note $(s.p)_{rp}$ le processus $s.p$ si $rp(s,p) = -$, et $(s.p, 0)$ sinon. Dans ce dernier cas, $(s.p)_{rp}$ dénote donc la copie numéro 0 du processus en multi-session parallèle $s.p$.

L'*interprétation* I est une application de id vers $\mathbb{P}(\text{Term}^*)$, qui sert à interpréter les symboles de prédicats (premiers arguments de `prop`). On suppose que I envoie '`equals`' vers l'ensemble $\{(t,t) | t \in \text{Term}\}$. (C'est la relation d'égalité.) Les interprétations des autres prédicats sont réservées, et pourront être définies ultérieurement.

Les termes t sont interprétés par la fonction $[\![t]\!]\, \Sigma\; rp$ définie comme suit:

$$[\![\mathtt{located}(s,p,t)]\!]\, \Sigma\; rp \quad = \quad t\sigma, \text{ avec } \sigma = \Sigma((s.p)_{rp}) \tag{106}$$

$$[\![f(t_1, \ldots, t_n)]\!]\, \Sigma\; rp \quad = \quad f([\![t_1]\!]\, \Sigma\; rp, \ldots, [\![t_n]\!]\, \Sigma\; rp) \tag{107}$$

où $f \neq \mathtt{located}$. Cette fonction est partielle. Notamment $[\![x]\!]\, \Sigma rp$ n'est pas définie lorsque x est une variable, et $[\![\mathtt{located}(s,p,t)]\!]\, \Sigma rp$ n'est pas définie lorsque $(s.p)_{rp} \notin \operatorname{dom} \Sigma$. Les règles de typage assurent cependant que $[\![t]\!]\, \Sigma rp$ sera toujours définie dans les cas où elle sera utilisée.

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{prop}(P, t_1, \ldots, t_n) \quad \Leftrightarrow \quad ([\![t_1]\!]\, (\Sigma \text{ of } \bar{\mathfrak{G}}(k))rp, \ldots, [\![t_n]\!]\, (\Sigma \text{ of } \bar{\mathfrak{G}}(k))rp) \in I(P) \tag{108}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{at}(s, p, lab) \quad \Leftrightarrow \quad pc \text{ of } \bar{\mathfrak{G}}(k)((s.p)_{rp}) = lab \tag{109}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{owns}(s, p, t) \quad \Leftrightarrow \quad \forall A \in \mathtt{P} \cup \{\mathtt{I}\} \cdot (E \text{ of } \bar{\mathfrak{G}}(k); \Omega \text{ of } \bar{\mathfrak{G}}(k) \vdash A \ni_\alpha [\![t]\!]\, (\Sigma \text{ of } \bar{\mathfrak{G}}(k))rp \Longrightarrow A = (s.p)_{rp}) \tag{110}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{secret}(t) \quad \Leftrightarrow \quad E \text{ of } \bar{\mathfrak{G}}(k) \not\vdash_\alpha [\![t]\!]\, (\Sigma \text{ of } \bar{\mathfrak{G}}(k))rp \tag{111}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{implies}(F, G) \quad \Leftrightarrow \quad (\bar{\mathfrak{G}}, k; I, rp \models_\alpha F) \Longrightarrow (\bar{\mathfrak{G}}, k; I, rp \models_\alpha G) \tag{112}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{and}(F_1, \ldots, F_n) \quad \Leftrightarrow \quad (\bar{\mathfrak{G}}, k; I, rp \models_\alpha F_1) \text{ et} \ldots \text{ et } (\bar{\mathfrak{G}}, k; I, rp \models_\alpha F_n) \tag{113}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{or}(F_1, \ldots, F_n) \quad \Leftrightarrow \quad (\bar{\mathfrak{G}}, k; I, rp \models_\alpha F_1) \text{ ou} \ldots \text{ ou } (\bar{\mathfrak{G}}, k; I, rp \models_\alpha F_n) \tag{114}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{not}(F) \quad \Leftrightarrow \quad \bar{\mathfrak{G}}, k; I, rp \not\models_\alpha F \tag{115}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{E}(F) \quad \Leftrightarrow \quad \exists \bar{\mathfrak{G}}' \cdot (\forall i \leq k \cdot \bar{\mathfrak{G}}'(i) = \bar{\mathfrak{G}}(i)) \text{ et } \bar{\mathfrak{G}}', k; I, rp \models_\alpha F \tag{116}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{A}(F) \quad \Leftrightarrow \quad \forall \bar{\mathfrak{G}}' \cdot (\forall i \leq k \cdot \bar{\mathfrak{G}}'(i) = \bar{\mathfrak{G}}(i)) \Longrightarrow \bar{\mathfrak{G}}', k; I, rp \models_\alpha F \tag{117}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{F}(F) \quad \Leftrightarrow \quad \exists k' \geq k \cdot \bar{\mathfrak{G}}, k'; I, rp \models_\alpha F \tag{118}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{G}(F) \quad \Leftrightarrow \quad \forall k' \geq k \cdot \bar{\mathfrak{G}}, k'; I, rp \models_\alpha F \tag{119}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{P}(F) \quad \Leftrightarrow \quad \exists k' \leq k \cdot \bar{\mathfrak{G}}, k'; I, rp \models_\alpha F \tag{120}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{Q}(F) \quad \Leftrightarrow \quad \forall k' \leq k \cdot \bar{\mathfrak{G}}, k'; I, rp \models_\alpha F \tag{121}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{N}(F) \quad \Leftrightarrow \quad \bar{\mathfrak{G}}(\geq k), 0; I, rp \models_\alpha F \tag{122}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{U}(F, G) \quad \Leftrightarrow \quad \exists k' \geq k \cdot (\forall i \cdot k \leq i < k' \Longrightarrow \bar{\mathfrak{G}}, i; I, rp \models_\alpha F) \text{ et } \bar{\mathfrak{G}}, k'; I, rp \models_\alpha G \tag{123}$$

$$\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{W}(F, G) \quad \Leftrightarrow \quad \forall k' \geq k \cdot (\forall i \cdot k \leq i < k' \Longrightarrow \bar{\mathfrak{G}}, i; I, rp \not\models_\alpha G) \tag{124}$$

$$\begin{aligned}
& && \Longrightarrow (\forall i \cdot k \leq i < k' \Longrightarrow \bar{\mathfrak{G}}, i; I, rp \models_\alpha F) \\
\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{S}(F, G) & \Leftrightarrow && \exists k' \leq k \cdot \qquad (125) \\
& && (\forall i \cdot k' < i \leq k \Longrightarrow \bar{\mathfrak{G}}, i; I, rp \models_\alpha F) \\
& && \text{et } \bar{\mathfrak{G}}, k'; I, rp \models_\alpha G \\
\bar{\mathfrak{G}}, k; I, rp \models_\alpha \mathtt{B}(F, G) & \Leftrightarrow && \forall k' \leq k \cdot \qquad (126) \\
& && (\forall i \cdot k' < i \leq k \Longrightarrow \bar{\mathfrak{G}}, i; I, rp \not\models_\alpha G) \\
& && \Longrightarrow (\forall i \cdot k' < i \leq k \Longrightarrow \bar{\mathfrak{G}}, i; I, rp \models_\alpha F)
\end{aligned}$$

5.6 Validité

Finalement, étant donné une spécification LAEVA $spec$ telle que:

$$\vdash spec \Rightarrow_{\text{spec}} \rho, \sigma, \alpha, start, \tilde{\gamma}, loc, shared, rp, \mathtt{assume}(F_1, \ldots, F_m), \mathtt{claim}(G_1, \ldots, G_n)$$

on dit que $spec$ est *valide* si et seulement si, définissant $\mathcal{A}_{start,\tilde{\gamma},rp}$ et $Init_{start,rp}$ comme en section 5.4, alors pour tout chemin $\bar{\mathfrak{G}}$ partant d'un état quelconque $\mathfrak{G}$ satisfaisant $Init_{start,rp}$ et tel que

$$\bar{\mathfrak{G}}, 0; I, rp \models_\alpha F_i$$

pour tout i, $1 \leq i \leq m$, alors

$$\bar{\mathfrak{G}}, 0; I, rp \models_\alpha G_i$$

pour tout i, $1 \leq i \leq n$.

References

[GL01] Jean Goubault-Larrecq. Langage de spécification de protocoles de EVA: syntaxe abstraite et sémantique. Technical Report EVA-2, LSV/CNRS UMR 8643 & ENS Cachan, July 2001.

[JLM01] Florent Jacquemard and Daniel Le Métayer. Langage de spécification de protocoles de EVA: syntaxe concrète. Technical Report EVA-1, Trusted Logic S.A., July 2001. Version 3.14.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[LS95] François Laroussinie and Philippe Schnoebelen. A hierarchy of temporal logics with past. *Theoretical Computer Science*, 148(2):303–324, 1995.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.