Usuba, vers une formalisation du langage

Samuel Vivien, sous l'encadrement de Pierre-Évariste Dagand – IRIF

La date

Le contexte général

Usuba est un langage de haut niveau pour pour écrire des primitives cryptographiques qui cumulent à la fois un haut débit et un calcul en temps constant.

La nécessité de la première propriété est évidente et la particularité d'Usuba réside dans son implémentation. L'idée est d'exploiter au maximum les unités de calcul vectoriel des processeurs afin d'augmenter la quantité de calculs effectués en parallèles. Pour cela plusieurs types de calcul vectoriel sont utilisées :

— Les unités AVX afin de faire des opérations arithmétiques entre des entiers 16, 32 ou 64 bits
— Les registres usuelles qui permettent de faire des opérations logiques entre 32 ou 64 bits en parallèles

La seconde propriété est recherché par les développeurs de primitives cryptographiques car cela permet de diminuer le risque de fuite de données lié aux attaques par écoute. En effet si le temps d'exécution d'un code dépend du message chiffré il est possible d'obtenir des informations sur le dit message à partir du temps d'exécution. Dans un code assembleur, les deux principaux facteurs qui font varier le temps d'exécution en fonction des valeurs sont les saut conditionels et les accès mémoires.

Afin d'éviter les saut conditionels dans le code généré, la solution la plus simple est de les interdire dans le code initial. Usuba n'est donc pas un langage turing-complet car il n'est pas possible d'écrire des conditionels (`if`) ou des boucles dynamiques (`while`).

Le problème des accès mémoire est un problème très étudié et dont il existe des solutions. Pour résoudre ce problème, il existe en Usuba deux types de tableaux.

— Il y a les tableaux statique dont le contenue est connu à la compilation : il s'agit des S-Box utilisé dans les primitives cryptographiques. Il est possible d'accéder dans ces tableaux avec une valeur arbitraire car sinon on pourrais seulement écrire des constantes. Pour éviter que les accès dans ces tableaux soient des accès mémoire ils sont remplacé à la compilation par un calcul arithmétique. Il existe de nombreuses recherches sur comment trouver les codes les plus efficace possible pour retirer ces accès mémoire.
— Il y a aussi les tableaux dynamique dont le contenue n'est connu que à l'exécution. Pour ces tableaux, les seuls accès possible sont par des indices connu à la compilation. On peux donc remplacer ces tableaux par une liste de variables ce qui évite les accès mémoire.

Le problème étudié

Cependant le compilateur de Usuba (nommé *usubac*) possède plusieurs défauts :

— le compilateur n'est pas certifié
— le compilateur n'inclut pas de typeur seulement des tentative de vérification au court des différentes passes

— et il n'existe pas de spécification de la sémantique d'Usuba.

À moins de lire le code généré, ceci nécessite de faire confiance au compilateur et de comprendre avec exactitude le code fourni. Or, avoir un compilateur certifié et une spécification claire de la sémantique permet aux développeurs de plus facilement remplir ces conditions.

La contribution proposée

Afin de commencer à palier ces problèmes, ce rapport présenteras un début de système de type, ainsi que 4 spécification différentes d'une sémantique de Usuba implémenté en Coq à l'aide de différentes méthodes.

L'idée derrière ces sémantiques est à la fois de clarifier certains comportement de Usuba avec le compilateur actuel, mais aussi d'étudier des évolutions possible du langage afin de plus se rapprocher d'un modèle équationel. Les spécificités et avantages des différentes sémantiques seront notamment discutés et comparés.

Les arguments en faveur de sa validité

Afin de tester la validité des sémantiques implémentés, deux d'entre elles ont été extraites de Coq vers du code OCaml afin de tester le comportement de deux primitives cryptographiques implémenté en Usuba : ACE et AES. Ces deux programmes ont été testé sur un vecteur test afin de vérifier que le résultat de l'évaluation soit bien celui attendu.

Le bilan et les perspectives

La contribution finale est loin de l'objectif initial d'implémenter un compilateur certifié. Cependant ce travail as permis d'exhiber les difficultés dans le compotement existant des codes Usuba ce qui permet d'ouvrir des pistes de réflexion sur les évolutions possibles du langage. De plus les différentes implémentation de sémantique et les discussions associés permettrons d'avoir un recul quel implémentation choisir pour une implémentation certifié d'un compilateur. De plus cet effort de développement ont permis de mettre en place des outils qui permettrons de faciliter une implémentation future d'un compilateur certifié.

$$
\begin{array}{ll}
ind ::= & \\
\quad | \ a & \\
\quad | \ a_1..a_2 & \\
\quad | \ \overline{a_n} &
\end{array}
\qquad
\begin{array}{ll}
aop ::= & \\
\quad | \ + & \\
\quad | \ - & \\
\quad | \ / & \\
\quad | \ \times &
\end{array}
$$

$$
\begin{array}{ll}
v ::= & \\
\quad | \ x & \\
\quad | \ v[ind] &
\end{array}
\qquad
\begin{array}{ll}
e ::= & \\
\quad | \ z & \\
\quad | \ v & \\
\quad | \ (\,\overline{e_n}\,) & \\
\quad | \ monop_\tau \ e & \\
\quad | \ e_1 \ binop_\tau \ e_2 & \\
\quad | \ f(\,\overline{e_n}\,) & \\
\quad | \ f < a > (\,\overline{e_n}\,) &
\end{array}
$$

$$
\begin{array}{ll}
a ::= & \\
\quad | \ i & \\
\quad | \ z & \\
\quad | \ a_1 \ aop \ a_2 &
\end{array}
$$

i, d, s	Static Identifiers: $\in$ *Ident*
x, y, t	Dynamic Identifiers: $\in$ *Ident*
f	Node Identifiers: $\in$ *Ident*
ℓ, z	Integers: $\in \mathbb{N}$
n, m, j, k	Index variables

$$
\begin{array}{ll}
deq ::= & \\
\quad | \ \overline{v_n} = e & \\
\quad | \ \overline{v_n} := e & \\
\quad | \ \textbf{for}\ i = a_1\ \textbf{to}\ a_2\ \textbf{do}\ \overline{deq_n}\ \textbf{done} &
\end{array}
$$

$$
\begin{array}{ll}
node ::= & \\
\quad | \ \textbf{node}\, f(\,\overline{x_m : \tau_m}\,) \to (\,\overline{y_n : \tau'_n}\,)\ \textbf{vars}\ (\,\overline{t_j : \tau''_j}\,)\ \textbf{let}\ \overline{deq_k}\ \textbf{tel} & \\
\quad | \ \textbf{table}\, f(x : \tau) \to (y : \tau')[\,\overline{a_n}\,] &
\end{array}
$$

FIGURE 1 – AST de Usuba

1 Syntaxe et comportement actuel de Usuba

Un programme en USUBA est composé de plusieurs nœuds. Il en exists deux types : les nœuds d'équations et les tableaux comme indiqué dans la figure 1.

Les tableaux permettent d'implémenter des S-BOX. Ces nœuds ne sont pas particulièrement intéressants et peuvent être considérés comme des boites noires dans la suite de ce rapport.

Les nœuds d'équations sont composé d'une liste de déclarations. Ces équations expliquent comment calculer la valeur des variables renvoyé à partir des variables fournis en entrée. Il existe trois types de déclaration possibles :

— Les boucles `for` dont les deux bornes sont connu à la compilation. Il s'agit de sucre syntaxique afin d'écrire de façon conscise un grand nombre d'équations. 3 des 4 sémantiques présenté dans la section 3 commencent par retirer ce sucre syntaxique afin de directement gérer une liste d'équations

— Les équations de définition ($\overline{v_n} = e$) qui définissent les variables à partir de la valeur calculé par l'expression e.

— Les équations de modification ($\overline{v_n} := e$) qui modifient les valeurs des variables dans l'environnement. Cette construction n'est pas compatible avec une vision équationel d'un nœud en raison de sa nature impérative. Elle n'est donc pas supporté dans la plupart des sémantiques en raison de son incompatibilité avec d'autres fonctionnalités gérés par ces sémantiques. Ceci n'est pas un problème car cette construction est voué à disparaître.

Les différents constructeurs d'expressions correspondent à ce que l'on peut trouver usuellement dans un langage de programmation : appel de nœuds, opérateurs binaires et unaire, tuples, constantes et variables.

Les constructeurs de variables sont quand à eux un peu compliqués. Il peut s'agir soit d'un identifiant ou d'un indiçage sur une variable. Un indiçage peut être :

— Un indice i qui permet de projeter un tableau sur l'un de ses éléments

— Une liste d'entiers qui permet de générer un nouveau tableau en modifiant une dimension

— Un interval qui est juste du sucre syntaxique pour la liste de tous les entiers dans l'interval

Par exemple si l'on as un identifiant x qui contient un tableau de 3 entiers 32 bits $[0, 1, 2]$. Alors la construction $x[2, 0]$ s'évalue en un tableau de 2 entiers $[2, 0]$.

Cependant si on prend désormais un identifiant x qui contient un tableau de 2 tableaux de

$$
\begin{array}{l}
dir ::= \\
\quad \mid \mathbf{V} \\
\quad \mid \mathbf{H} \\
\quad \mid d
\end{array}
\qquad
\begin{array}{l}
size ::= \\
\quad \mid s \\
\quad \mid z
\end{array}
\qquad
\begin{array}{l}
\sigma ::= \\
\quad \mid \mathbf{U}\ dir\ size \\
\\
\tau ::= \\
\quad \mid \sigma \\
\quad \mid \tau[a] \\
\\
\mathcal{T} ::= \\
\quad \mid \overline{\tau_n}
\end{array}
\qquad
\begin{array}{l}
typc ::= \\
\quad \mid \mathbf{Arith}\ \tau \\
\quad \mid \mathbf{Logic}\ \tau \\
\quad \mid \mathbf{Shift}\ \tau\ a_2 \\
\\
A ::= \\
\quad \mid \overline{typc_n}
\end{array}
\qquad
\begin{array}{l}
P ::= \\
\quad \mid \mathbf{nil} \\
\quad \mid P \leftarrow node \\
\\
\Gamma ::= \\
\quad \mid \overline{x_n : \tau_n}
\end{array}
$$

FIGURE 2 – Types et contextes en USUBA

2 entiers $[[0,1],[2,3]]$. Alors la construction $x[0,1][0]$ est du sucre syntaxique pour $(x[0][0], x[1][0])$ qui s'évalue en $[0,2]$. Cependant si l'on modifie le contexte avec l'équation $y = x[0,1]$, alors $y[0]$ s'évalue en $[0,1]$. L'implémentation actuelle de usubac fournis donc une sémantique qui n'est pas compositionnelle.

Afin de résoudre ce problème, s'idée serait de faire évoluer la syntaxe de USUBA afin de pouvoir écrire les deux. Pour cela, l'idée est de s'inspirer de *numpy* (une librairie de python) et de définir une syntaxe pour effectuer des indiçage sur plusieurs dimension de façon simultanés en les séparant par un point-virgule.

Par exemple cette nouvelle syntaxe permet d'écrire $x[0,1;0]$ pour parler de $(x[0][0], x[1][0])$ et $x[0,1][0]$ désigne désormais $x[0]$. Cependant, cette nouvelle syntaxe fait perdre la rétro-compatibilité mais permet d'avoir une sémantique compositionnelle.

2 Règles de typage

Maintenant que la section précédente as résolue le soucis lié à aux accès dans les tableaux qui avaient une sémantique non compositionnelle. Cependant la sémantique actuelle de USUBA contient une autre difficulté dont cette section va essayer de s'occuper. Le problème des coercions implicites est encore présent et les paragraphes suivants ont pour but d'essayer de clarifier dans quelles situations est ce qu'une telle coercion devrait avoir lieu.

Pour cela nous définissions les types τ comme une tableau multi-dimensionnels contenant un type atomique σ qui correspond à un entier avec une certaine taille *size* et orientation *dir* comme indiqué dans la figure 2. À partir de ces types on définie le type d'une expression $\mathcal{T}$ comme une liste de types τ.

De plus, afin de pouvoir définir des nœuds polymorphique, le langage USUBA contient des classes de types *typc* qui permettent de spécifier sur quels types sont définies les opérations logiques, arithmétique et de décalage. Certaines classes de types peuvent être définie sur un tableau à l'aide du foncteur de liste et si la classes est bien définie sur le type des éléments du tableau comme indiqué dans la figure 3.

À partir de cette syntaxe des types, on peut désormais définir les règles de typages des variables. Pour cela on défini d'abord dans la figure 4 comment une liste d'indiçage modifie les dimension d'un tableau puis en appliquant récursivement ces règles ont obtient les règles de typage des variables présenté dans la figure 5.

À partir du typage des variables ont peut construire le typage est expressions présenté dans la figure 6. Une particularité notable de ces règles de typage est que le type d'une expression est représenté comme une liste de types et mais que 2 règles ne sont définies que sur les tableaux d'entiers MONOP et BINOP.

La règle la plus notable parmi les différentes expressions est la règle de typage d'un appel

$\boxed{A \vdash \overline{typc_n}}$

$$\frac{A \vdash \mathbf{Arith}\,\tau}{A \vdash \mathbf{Arith}\,\tau[\ell]}\ \textsc{ArithL}$$

$$\frac{A \vdash \mathbf{Logic}\,\tau}{A \vdash \mathbf{Logic}\,\tau[\ell]}\ \textsc{LogicL}$$

FIGURE 3 – Inférence des type-class

$\boxed{\tau_1 - [\,\overline{ind_m}\,] \to \tau_2}$

$$\frac{\begin{array}{c}\sigma\,\overline{[d_n]} - [\,\overline{ind_m}\,] \to \sigma\,\overline{[d'_k]} \\ \vdash 0 \leqslant a < \ell\end{array}}{\sigma[\ell]\,\overline{[d_n]} - [a\ ;\ \overline{ind_m}\,] \to \sigma\,\overline{[d'_k]}}\ \textsc{Index}$$

$$\frac{\begin{array}{c}\sigma\,\overline{[d_n]} - [\,\overline{ind_m}\,] \to \sigma\,\overline{[d'_k]} \\ \vdash 0 \leqslant a_1 < \ell \\ \vdash 0 \leqslant a_2 < \ell\end{array}}{\sigma[\ell]\,\overline{[d_n]} - [a_1..a_2\ ;\ \overline{ind_m}\,] \to \sigma[abs(a_1\ -\ a_2)\ +\ 1]\,\overline{[d'_k]}}\ \textsc{Range}$$

$$\frac{\begin{array}{c}\sigma\,\overline{[d_n]} - [\,\overline{ind_m}\,] \to \sigma\,\overline{[d'_k]} \\ \vdash 0 \leqslant\ \overline{a_j}\ < \ell\end{array}}{\sigma[\ell]\,\overline{[d_n]} - [\overline{a_j}\ ;\ \overline{ind_m}\,] \to \sigma[\mathbf{len}\,\overline{a_j}]\,\overline{[d'_k]}}\ \textsc{Slice}$$

FIGURE 4 – Typages indiçages

de nœud. En effet il y as à ce moment là une coercion de type des arguments. Il s'agit d'une fonctionnalités très utilisé en USUBA car elle permet notamment de changer un tableau de 64 éléments en deux tableaux de 32 parmi d'autres fonctionnalités. Afin de pouvoir décider quand une telle coercion est possible, nous défissons une notion d'équivalence entre deux listes de types dont les règles sont dans la figure 7.

Ces règles peuvent sembler obscures cependant l'intuition derrière est relativement simple et peut être résumé en seulement deux règles :

— Les entier 1 bit sont les mêmes pour toute représentation mémoire (verticale ou horizontale).
— Deux listes de types sont identiques si elle contiennent le même nombre d'entier de chaque taille et orientation et dans le même ordre.

Cependant ces règles de typage ne permettent pas de typer certaines opérations qui sont actuellement utilisé dans des codes USUBA. Par exemple si l'on as x de type $u32[2]$ alors $x + (x[0], x[1])$, n'est pas typable. Pour palier à ce problèmes nous introduisonts dans le langage USUBA deux nouvelles constructions : les constructeurs de tableaux et les coercions. Les constructeurs de tableau on pour but de pouvoir permettre de gérer de nombreux soucis en permettant de créer des tableau plutôt que des objets avec un type abstrait et fluctuant comme les tuples dans l'implémentation actuelle. Cependant cela ne permet pas de gérer tous les cas et c'est pour ça que l'on introduit une notion de coercion afin de ne pas perdre en expressivité. Cela nous donnes deux nouvelles règles de typage présentées dans la figure 8.

Une fois que l'ont sait comment typer les expressions on peut désormais vérifier que les déclarations sont bien typées. Pour cela les règles de la figure 9 indiquent qu'il faut vérifier que les équations font

$\boxed{\Gamma \vdash_V v : \tau}$

$$\frac{\Gamma \vdash_I x : \tau \in \Gamma}{\Gamma \vdash_V x : \tau} \quad \textsc{Ident}$$

$$\frac{\begin{array}{c}\Gamma \vdash_V v : \tau_1 \\ \tau_1 - [\,\overline{ind_n}\,] \to \tau_2\end{array}}{\Gamma \vdash_V v\,[\,\overline{ind_n}\,] : \tau_2} \quad \textsc{Indexing}$$

FIGURE 5 – Typage variables

le lien entre deux listes de types équivalentes et que pour les boucles toutes les sous déclarations sont bien typées.

3 Sémantiques

Afin de fournir une spécification du langage USUBA dans le but d'implémenter un compilateur certifié par assistant de preuve il faut implémenter la dite sémantique dans un assistant de preuve. L'assistant de preuve choisi pour cette formalisation est Coq en raison de sa compatibilité avec OCAML car le compilateur existant est écrit dans ce langage.

Afin de pouvoir comparer différentes techniques 4 sémantique différentes ont été implémenté. Nous allons donc désormais discuter des avantages et inconvénients des différentes sémantiques.

3.1 Sémantique par évaluation

La première sémantique implémenté pour USUBA est une sémantique par évalutation.

La sémantique d'une expression est donc définie de façon intuive par une fonction qui prend en argument un contexte et un arbre de syntaxe et qui retourne potentiellement une valeur car les erreurs sont représentés par `None`.

À partir de cela la sémantique d'une équation est définie par une fonction qui prend en entré un contexte et une équation représentés par une paire composé d'une liste de variables et d'une expression. Cette fonction évalue l'expression puis utilise cette valeur et la liste de variables pour modifier le contexte. La sémantique d'une liste de d'équation est quand à elle définie par un itération de la sémantique d'une équation.

Cette sémantique est celle la plus proche de l'implémentation actuelle de USUBA car c'est la seule des 4 qui accepte de définir une même variable plusieurs fois. En effet chaque équation spécifie s'il s'agit de la création d'une variable ou de sa modification.

De plus cette sémantique permet de définir facilement une équivalence de programme en indiquant que deux expressions sont équivalentes si pour tous contextes elles s'évaluent en les mêmes valeurs.

Cependant dans cette sémantique, le comportement d'un nœud dépend de l'ordre dans lequel sont écrites les équations. Les autres sémantiques ont donc été définie pour ne pas dépendre de l'ordre des équations.

3.2 Sémantique relationelle

Afin de palier les limites montré précédemment une nouvelle sémantique as été définie. Cependant celle ci est relationelle, ce qui lui permet de représentés seulement les évaluations qui

$\boxed{\Gamma, P, A \vdash_E e : \mathcal{T}}$

$$\frac{\Gamma \vdash_V v : \tau}{\Gamma, P, A \vdash_E v : \tau} \quad \textsc{Var}$$

$$\frac{\begin{array}{c}\Gamma, P, A \vdash_E e_1 : \tau \\ \Gamma, P, A \vdash_E e_2 : \tau \\ A \vdash \textbf{ClassOf}\ binop\ \tau\end{array}}{\Gamma, P, A \vdash_E e_1\ binop_\tau\ e_2 : \tau} \quad \textsc{Binop}$$

$$\frac{\begin{array}{c}\Gamma, P, A \vdash_E e : \tau \\ A \vdash \textbf{ClassOf}\ monop\ \tau\end{array}}{\Gamma, P, A \vdash_E monop_\tau\ e : \tau} \quad \textsc{Monop}$$

$$\frac{\Gamma, P, A \vdash_E \overline{e_n : \mathcal{T}_n}}{\Gamma, P, A \vdash_E (\overline{e_n}) : \overline{\mathcal{T}_n}} \quad \textsc{Tuple}$$

$$\frac{\begin{array}{c}P \vdash f : \forall\, \overline{d_n}\,, \forall\, \overline{s_m}\,, \overline{typc_j} \Rightarrow \mathcal{T}_1 \to \mathcal{T}_2 \\ \Gamma, P, A \vdash_E (\overline{e_n}) : \mathcal{T}_1' \\ A \vdash typc_j[\overline{d_n \leftarrow d_n'}\ ;\ \overline{s_m \leftarrow s_m'}] \\ \mathcal{T}_1' \cong \mathcal{T}_1[\overline{d_n \leftarrow d_n'}\ ;\ \overline{s_m \leftarrow s_m'}]\end{array}}{\Gamma, P, A \vdash_E f(\overline{e_n}) : \mathcal{T}_2[\overline{d_n \leftarrow d_n'}\ ;\ \overline{s_m \leftarrow s_m'}]} \quad \textsc{Fun}$$

FIGURE 6 – Règles de typage des expressions

réussissent.

Pour la sémantique par évaluation `e` s'évalue en `val` était représenté par

```
eval_expr ctxt e = Some val
```

pour la nouvelle sémantique relationelle on représente ça par une simple relation

$$\texttt{e} \mapsto_{ctxt} \texttt{val}$$

De plus les erreurs sont représenté par une absence de relation, on n'as donc plus besoin d'avoir une valeur spécifique pour les représentés.

La spécificité de cette sémantique se situe au niveau de la gestion du contexte. La sémantique d'une équation est définie par vérifier dans le contexte donné la liste de variables à gauche de l'équation et l'expression à droite de l'équation s'évaluent bien en la même valeur. Le contexte quand à lui est définie au niveau de la sémantique globale d'un nœud :

$$\begin{array}{l}\forall\, names_{in}\ values_{in}\ names_{out}\ values_{out},\ \exists!\ ctxt, \\ \quad valid\_equations_{ctxt}\ eqns \to \\ \quad names_{in} \mapsto_{ctxt} values_{in} \to \\ \quad names_{out} \mapsto_{ctxt} values_{out} \to \\ \quad (names_{in} \to_{eqns} names_{out})\ values_{in} \mapsto values_{out}\end{array}$$

Cette sémantique possède plusieurs caractéristiques notables :

$\boxed{\mathcal{T}_1 \cong \mathcal{T}_2}$

$$\frac{}{\mathcal{T} \cong \mathcal{T}}\ \textsc{Refl}$$

$$\frac{\mathcal{T}_1 \cong \mathcal{T}_2}{\mathcal{T}_2 \cong \mathcal{T}_1}\ \textsc{Sym}$$

$$\frac{\mathcal{T}_1 \cong \mathcal{T}_2 \quad \mathcal{T}_2 \cong \mathcal{T}_3}{\mathcal{T}_1 \cong \mathcal{T}_3}\ \textsc{Trans}$$

$$\frac{\mathcal{T}_1 \cong \mathcal{T}_2}{\tau :: \mathcal{T}_1 \cong \tau :: \mathcal{T}_2}\ \textsc{Rec}$$

$$\frac{}{\mathbf{U}\ dir\ s\ \overline{[a'_m]} :: \mathcal{T} \cong \mathbf{U}\ dir\ s[\mathbf{prod}\ [\ \overline{a'_m}\ ]] :: \mathcal{T}}\ \textsc{SimplForm}$$

$$\frac{}{\mathbf{U}\,\mathbf{V}\,1\,\overline{[a'_m]} :: \mathcal{T} \cong \mathbf{U}\,\mathbf{H}\,1\,\overline{[a'_m]} :: \mathcal{T}}\ \textsc{Bool}$$

$$\frac{}{\mathbf{U}\ dir\ s[\ell_1] :: \mathbf{U}\ dir\ s[\ell_2] :: \mathcal{T} \cong \mathbf{U}\ dir\ s[\ell_1 + \ell_2] :: \mathcal{T}}\ \textsc{Join}$$

Figure 7 – Equivalence de types

$\boxed{\Gamma, P, A \vdash_E e : \mathcal{T}}$

$$\frac{\Gamma, P, A \vdash_E e : \mathcal{T}_1 \quad \mathcal{T}_1 \cong \mathcal{T}_2}{\Gamma, P, A \vdash_E e\ \mathbf{into}\ \mathcal{T}_2 : \mathcal{T}_2}\ \textsc{Into}$$

$$\frac{\Gamma, P, A \vdash_E \overline{e_n} : \sigma\ \overline{[a_m]}}{\Gamma, P, A \vdash_E [\ \overline{e_n}\ ] : \sigma[len\ \overline{e_n}]\ \overline{[a_m]}}\ \textsc{Array}$$

Figure 8 – Règles de typage des expressions, partie 2

— Elle est indépendante de l'ordre des équations, cela découle de la communativité du et logique.
— Elle ne garantie pas l'unicité des définitions contrairements aux précédentes, seulement la cohérence de ces définitions pour un contexte donné.

Cependant cette sémantique possède un très grosse limitation qui complique sont utilisation dans un compilateur certifié. Pour pouvoir garantir la correction du typage on voudrait prouver que si le typeur accepte un programme, alors pour toute entré le programme renvoie une valeur de sortie. Cependant cela nécessite d'être capable pour toute entrée d'exhiber un contexte qui vérifie les bonnes hypothèses. On aurais donc besoin de définir une autre sémantique par dessus afin de pouvoir prouver la correction du typage.

Cependant l'aspect relationelle de cette sémantique rend probablement plus facile les preuves de correction des autres passes d'un compilateur.

3.3 Sémantique par tri topologique

La sémantique par tri topologique est de loin la plus compliqué des 4 sémantiques.

L'idée de cette sémantique peut être vue comme de l'appel par nom. Supossons que nous avons

$\boxed{\Gamma, P, A \vdash_D deq}$

$$\frac{\Gamma, P, A \vdash_E e : \mathcal{T} \quad \mathcal{T} \cong \mathcal{T}' \quad \Gamma \vdash_V \overline{v_n} : \mathcal{T}'}{\Gamma, P, A \vdash_D \overline{v_n} := e} \text{ EqnT}$$

$$\frac{\Gamma, P, A \vdash_E e : \mathcal{T} \quad \mathcal{T} \cong \mathcal{T}' \quad \Gamma \vdash_V \overline{v_n} : \mathcal{T}'}{\Gamma, P, A \vdash_D \overline{v_n} = e} \text{ EqnF}$$

$$\frac{\forall\, i \in [a_1, a_2].\ \Gamma, P, A \vdash_D \overline{deq_n[x \leftarrow i]}}{\Gamma, P, A \vdash_D \textbf{for}\, i = a_1\, \textbf{to}\, a_2\, \textbf{do}\, \overline{deq_n}\, \textbf{done}} \text{ Loop}$$

FIGURE 9 – Typage des equations

$\boxed{P \vdash f : \forall\, \overline{d_n}\,, \forall\, \overline{s_m}\,, A \Rightarrow \mathcal{T}_1 \to \mathcal{T}_2}$

$$\frac{\overline{x_m : \tau_m} + \overline{y_n : \tau'_n} + \overline{t_j : \tau''_j}, P, A \vdash_D \overline{deq_k} \quad node = \textbf{node}\, f\, (\overline{x_m : \tau_m}) \to (\overline{y_n : \tau'_n})\, \textbf{vars}\, (\overline{t_j : \tau''_j})\, \textbf{let}\, \overline{deq_k}\, \textbf{tel}}{P \leftarrow node \vdash f : \forall\, \overline{d_n}\,, \forall\, \overline{s_m}\,, A \Rightarrow \overline{\tau_m} \to \overline{\tau'_n}} \text{ Node}$$

$$\frac{\vdash 0 \leqslant \overline{z_n} < 1 \ll i_2 \quad \textbf{len}\, \overline{z_n} = 1 \ll i_1 \quad node = \textbf{table}\, f (x : \mathbf{U}\, d\, s[i_1]) \to (y : \mathbf{U}\, d\, s[i_2])[\,\overline{z_n}\,]}{P \leftarrow node \vdash f : \forall\, d, \forall\, s, \textbf{Logic}\, (\mathbf{U}\, d\, s) \Rightarrow \mathbf{U}\, dir\, s[i_1] \to \mathbf{U}\, dir\, s[i_2]} \text{ Table}$$

FIGURE 10 – Typage d'un noeud

un nœud qui retourne `x`, on va donc chercher dans quel equation est définie `x`. Puis l'on évalue l'expression associé afin d'obtenir la valeur de `x`, mais cela nécessite potentiellement de connaître la valeur de `y`. On continue donc récursivement en calculant la valeur de `y` et ainsi de suite.

Cependant une telle évaluation n'est pas garantie de terminer et toute fonction définie dans la logique de Coq doit posséder une preuve de terminaison. En effet perdre cette garantie permet de prouver faux.

Il existe donc une méthode pour prouver la terminaison d'une fonction : rajouter un argument strictement décroissant. La méthode simple est de fournir un carburant (ie : un entier naturel qui diminue à chaque appel récursif). Cependant cette techniques possèdes plusieurs limitation :

— Cela modifie le code extrait.
— Il est difficile de garantir que l'on as mis suffisament de carburant pour n'en manquer que dans les boucles infinies.

Pour éviter ces soucis il existe une autre possibilité : fournir un prédicat d'accessibilité. Cela permet d'éviter de gérer les cas où l'on manque de carburant car cela n'arrive jamais. De plus le code extrait ne contient pas ce prédicat car les termes dans `Prop` ne sont pas extraits. Cependant cette technique possède aussi ses limites car il faut être capable de prouver l'existance d'un prédicat d'accessibilité. Il faut donc que lors de l'évaluation d'un nœud une telle sémantique vérifie si l'évaluation va terminer. Puis, seulement la vérification a accepté, évaluer le corps du nœud.

Il faut aussi prouver la correction de la vérification afin de pouvoir obtenir un prédicat d'accessibilité si le vérificateur de terminaison accepte le nœud.

Pour cela nous avons définie et certifié un tri topologique sur une liste d'équation. L'idée étant que si nous avons deux équations tel que l'une définie une variable `x` qui est utilisé par la seconde. Alors la seconde équation dépend de la première. À partir de là nous sommes avons définie un graphe de dépendance sur les équations, on peut donc effecter un tri topologique sur ce graphe afin de vérifier qu'il s'agit bien d'un graphe acyclique.

En réalité la définition du graphe est plus compliqué que ci-dessus car Usuba contient des tableaux. Une définition plus exacte de ce graphe est fournis dans la section 4.

Cette sémantique possède deux grosses limitations :

— La sémantique est particulièrement lourde à définir car afin de prouver l'existance d'un prédicat d'accessibilité il faut être capable de calculer le graphe (et prouver des propriétés dessus) puis prouver que si on a un tri topologique alors on as un prédicat d'accessibilité ce qui a nécessité plus de 5k lignes de Coq.

— Toute modification sur le programme oblige de pouvoir garantir que la vérificateur de terminaison continue as accepter la programme fourni.

Cette sémantique est donc peut utilisable en partique dans un compilateur et raison de sa difficulté à supporter des réécritures. Cependant les outils implémenté lors de sa mise en place sont très intéressants intellectuellement et sont nécessaire pour prouver la terminaison de toute évaluation indépendante de l'ordre. Le dernier point est important car c'est une étape nécessaire d'un typeur si l'on veux qu'il puisse garantir qu'un programme ne s'évalue pas en une erreur.

3.4 Sémantique par point fixe

Cette dernière sémantique est sensé être plus légère que la précédente tout en étant une sémantique qui calcule et indépendante de l'ordre d'évaluation.

L'idée derrière celle ci est que chaque équation peut être évalué exactement une fois mais l'on ne connais pas encore l'ordre dans lequel il faut le faire. On parcours donc la liste des équations en essayant de les évaluer. Pour chaque équation on a deux cas :

1. On réussi à évaluer l'expression de cette équation, on modifie donc le contexte et on peut oublié l'équation.
2. On ne réussi pas à évaluer l'expression, on garde donc l'équation pour plus tard.

Une fois que l'on as parcourus toutes les équations plusieurs cas de figure peuvent avoir lieux :

1. Il reste plus aucune équation : on a donc fini et on peut renvoyé le contexte calculé.
2. Le nombre d'équations a strictement diminué mais il en reste : on recommence avec le contexte obtenue et les équations qui restent.
3. On as gardé le même nombre non nul d'équations : on renvoie une erreur car on n'arrive pas à conclure.

Le nombre d'équations diminuant strictement à chaque appel récursif cette évaluation termine après au plus nombre d'équations initial itérations.

Cette sémantique possède plusieurs charactéristiques intéressantes :

— Elle est bien indépendante de l'ordre des équations

— Si on la restreint à un seul parcours sur la liste d'équations, on retrouve sur la sémantique par évaluation de la section 3.1

Cette méthode essaye de calculer un plus petit point fixe (d'où le nom de la sémantique), cependant l'existance d'un point fixe ne garanti pas qu'un programme ne génère pas une erreur.

En effet, voici deux exemples d'équations possèdant un point fixe mais dont l'évaluation par la sémantique renvoie une erreur :

$$\{x = x\}$$
$$\{y = 0 \times y\}$$

Il s'agit d'une différence notable avec la sémantique de la section 3.2 qui elle accepte tous les point fixes.

4 Tri topologique sur les équations

Afin de pouvoir effectué un tri topologique sur les équations il faut générer un graphe de dépendances entre les équations. Pour cela nous allons poser la relation $x \prec y$ qui signifie que l'équation numéro x dépend de l'équation numéro y. Une telle dépendance arrive si l'équation numéro y utilise une valeur définie dans l'équation numéro x.

Comme notre langage possède des tableaux, une variable est composé de deux informations : un identifiant et une liste d'indices.

Précédemment nous avons indiqué que seul l'identifiant était utilisé. Cependant cela ne permet pas de faire un tri sur le programme suivant où v est de type $b[2]$

$$\begin{aligned}\{v[0] &= 1\\ v[1] &= v[0]\}\end{aligned}$$

alors qu'il sensé être valide malgrè le fait que la seconde équation nécessite une sous composante de v et définie une sous composante de v.

Maintenant si l'on regarde l'exemple ci dessous où on prend v de type $b[5]$:

$$\begin{aligned}\{v[0,1] &= (0,1)\\ v[3] &= 3\\ v[2,4] &= v[1,3]\}\end{aligned}$$

on remarque que l'équation 3 nécessite les deux autres équations. Pour parler de celà on définie la notion de chemin dans un indentifiant comme une liste d'entiers. On dit que ce chemin est une instanciation d'une liste d'indices si chaque entiers est une instanciation de l'indice correspondant et que les deux listes ont la même longueur.

— Pour un indice seul, l'entier associé est son unique instanciation.
— Pour un interval, les entiers dedans sont ses instanciations.
— Pour une liste d'entiers, les entiers de la liste sont ses instanciations.

On observe donc que si l'équation numéro y utilise le chemin c dans un identifiant v et que l'équation numéro x définie le chemin c de v alors on doit avoir $x \prec y$. Cependant cette propriétés n'est pas encore suffisante.

En effet si une définition définie v et qu'un autre utilise $v[0]$ alors la second dépend de la première.

On pose donc la définition suivante : $x \prec y$ si il existes deux chemins c_x et c_y et un identifiant v tel que

1. L'équation numéro x définie le chemin c_x de l'identifiant v
2. L'équation numéro y utilise le chemin c_y de l'identifiant v
3. c_x est un préfixe de c_y ou c_y est un préfixe de c_x