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.

$$
\begin{array}{ll}
i,\ d,\ s & \text{Static Identifiers: } \in \mathit{Ident} \\
x,\ y,\ t & \text{Dynamic Identifiers: } \in \mathit{Ident} \\
f & \text{Node Identifiers: } \in \mathit{Ident} \\
\ell,\ z & \text{Integers: } \in \mathbb{N} \\
n,\ m,\ j,\ k & \text{Index variables}
\end{array}
$$

$$
\begin{array}{l}
\mathit{ind} ::= \\
\quad \mid\ a \\
\quad \mid\ a_1..a_2 \\
\quad \mid\ \overline{a_n}
\end{array}
\qquad
\begin{array}{l}
\mathit{aop} ::= \\
\quad \mid\ + \\
\quad \mid\ - \\
\quad \mid\ / \\
\quad \mid\ \times
\end{array}
$$

$$
\begin{array}{l}
v ::= \\
\quad \mid\ x \\
\quad \mid\ v[\mathit{ind}]
\end{array}
\qquad
\begin{array}{l}
e ::= \\
\quad \mid\ z \\
\quad \mid\ v \\
\quad \mid\ (\ \overline{e_n}\ ) \\
\quad \mid\ \mathit{monop}_\tau\ e \\
\quad \mid\ e_1\ \mathit{binop}_\tau\ e_2 \\
\quad \mid\ f(\ \overline{e_n}\ ) \\
\quad \mid\ f < a > (\ \overline{e_n}\ )
\end{array}
$$

$$
\begin{array}{l}
a ::= \\
\quad \mid\ i \\
\quad \mid\ z \\
\quad \mid\ a_1\ \mathit{aop}\ a_2
\end{array}
$$

$$
\begin{array}{l}
\mathit{deq} ::= \\
\quad \mid\ \overline{v_n} = e \\
\quad \mid\ \overline{v_n} := e \\
\quad \mid\ \textbf{for}\ i = a_1\ \textbf{to}\ a_2\ \textbf{do}\ \overline{\mathit{deq}_n}\ \textbf{done}
\end{array}
$$

$$
\begin{array}{l}
\mathit{node} ::= \\
\quad \mid\ \textbf{node}\, f(\ \overline{x_m : \tau_m}\ ) \to (\ \overline{y_n : \tau'_n}\ )\ \textbf{vars}\ (\ \overline{t_j : \tau''_j}\ )\ \textbf{let}\ \overline{\mathit{deq}_k}\ \textbf{tel} \\
\quad \mid\ \textbf{table}\, f(x : \tau) \to (y : \tau')[\ \overline{a_n}\ ]
\end{array}
$$

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. Ces nœuds correspondent à des fonctions du premier ordre. Les nœuds peuvent s'appeler les uns les autres mais seulement ceux qui ont été défini avant et les appels récursifs ne sont pas autorisé. En effet commes les conditionels ne sont pas autorisé, si l'on pouvais faire des appels récursif alors on aurais systématiquement une boucle infinie.

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

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

FIGURE 2 – Types et contextes en USUBA

— 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 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

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

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

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

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

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 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 **U V** 32[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

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

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

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

FIGURE 5 – Typage variables

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

Le typage des nœuds contient deux règles présenté dans la figure 10. La règle de typage des nœuds d'équations indique que toutes les équations doivent être bien typées dans le contexte de toutes les variables.

Le typage d'une table est plus subtil. En effet une table prend en entré $i1$ entiers de s bits et les considère comme b entiers de n bits en transposants la matrice de leurs représentation binaire. Ces entiers permettent de faire s accès dans la table qui contient $1 \ll i1$ entiers de $i2$ bits. On obtient alors s entiers de $i2$ bits qui sont transposé en $i2$ entiers de s bits. De plus la raison pour laquel on demande à ce que les opérations logiques soient définie sur les entiers est pour pouvoir remplacer ce nœud par une liste d'équation afin d'éviter les accès mémoires.

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.

Parmi ces sémantiques, il y as 3 sémantiques qui calculent un résultat et une quatrième qui est une relation et qui vit dans `Prop`.

Les différences entre ces sémantiques se situent au niveau de leur définition et de leur gestion du contexte et les discussions dans les paragraphes qui vont suivre se consentrerons dessus. Cependant ces sémantiques définissent aussi le comportement du reste des constructions qui existent en USUBA nous parlerons donc d'abord de ces points communs.

3.1 Points communs des sémantiques

Tout d'abord toutes les sémantiques ont une même notion de valeur qui est un type somme entre juste un entier (pour quand on ne connaît pas le type du dit entier) et un tableau multidimensionnel qui est défini comme un triplet direction, entiers stockés dedans et liste des dimensions. Ces valeurs correspondent à un élément de type τ. Une liste de telles valeurs correspond à un élément de type $\mathcal{T}$. De plus pour les 3 sémantiques qui calcul les erreurs sont représentés par un type option où `None` correspond à une erreur.

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

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

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

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

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

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

FIGURE 6 – Règles de typage des expressions

Le second point commun entre les 3 sémantiques qui calculent est la gestion de la sémantique des nœuds. En effet, la sémantique d'un nœud est défini comme une fonction d'une liste de valeurs dans une option de liste de valeurs. On peut alors passer à la fonction d'évaluation du corp d'un nœud une liste de la sémantique de tous les nœuds défini précédemment sans créer de récursivité mutuelle entre les différentes fonctions de définition de la sémantique.

3.2 Sémantique par évaluation

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

Cette sémantique est définie de façon intuitive à partir de son nom, on évalue tout dans l'ordre. Pour cela la sémantique d'une expression est donc définie à partir d'une fonction qui à partir d'une expression prend un contexte et renvoie une option de liste de valeurs. Pour cette sémantique, un contexte est définie pour une structure (en l'occurence une liste de paire) qui associe à certains identifiants une valeur incomplète. Où une valeur incomplète est une valeur où l'on as remplacé la liste des éléments d'un tableau par une liste d'option pour pouvoir désigner les éléments par encore défini.

En effet dans le système $\{v[0] = 1; v[1] = v[0]\}$ avec v de type **U V** 32[2]. Alors entre les deux équations, seulement un des éléments du tableau est défini. On représente donc ça par un contexte qui à v associe `InR`(**V**, [`Some` 1, `None`], [1]).

À partir de cela on définie la sémantique d'une équation est définie par une fonction qui prend en entré un contexte et qui en renvoie un nouveau si la sémantique aucune erreur ne se produit. 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.

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

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

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

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

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

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

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

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

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

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 une équation sans modification = n'accepte de remplacer que des `None` par des valeurs dans le contexte alors que qu'en équation avec modification := n'accepte que de remplacer que des `Some`.

Mais cela à pour conséquence que le comportement d'un nœud dépend de l'ordre dans lequel sont écrites les équations. En effet si l'ordre déclarations n'influençais pas la sémantique, alors il ne serais pas possible d'avoir des constructions impératives tel que les déclarations :=. Les autres sémantiques ont été définie pour ne pas dépendre de l'ordre des équations afin de plus ressembler à un modèle équationel. Cependant elles ne supportent donc plus la possibilité d'écrire des équations avec modification.

Pour ce qui est d'utiliser ette sémantique pour de la preuve de programme, il est facile de définir une équivalence de programme en indiquant que deux expressions sont équivalentes si pour tous contextes elles s'évaluent en les mêmes valeurs. Et de la même façon on peut définir les équivalences d'équations, de déclaration et de nœuds. Cette sémantique possède donc l'avantage d'avoir un ordre d'évaluation clair et indiqué par la syntaxe ce qui permet de faire plus facilement des preuves de

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

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

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

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

FIGURE 9 – Typage des equations

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

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

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

FIGURE 10 – Typage d'un noeud

préservation de la sémantique pour les différentes étapes de la compilation.

Désormais nous allons présenter 3 autres sémantiques qui ont été défini dans le but de pouvoir interpréter des programmes sans dépendre de l'ordre dans lequel les équations des nœuds sont écrites. On considère donc désormais que toutes les équations sont des équations sans modification car les redéfinitions sont incompatibles avec une sémantique purement équationelle.

3.3 Sémantique relationelle

La première des ces 3 sémantiques est définie à partir de relations. Celle ci permet d'indiqué qu'une expression met en relation un contexte et une valeur si l'évaluation réussi. Cela permet de représenter de façon encore plus concise les erreurs car si l'évaluation d'une expression dans un certain contexte plante, alors l'expression ne met le contexte en relation avec aucune valeur.

La spécificité de cette sémantique se situe au niveau de la gestion du contexte. La sémantique d'une équation est la vérification 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 des valeurs compatibles. Le contexte quand à lui est définie au niveau de la sémantique globale d'un nœud d'équations.

$$\forall\ names_{in}\ values_{in}\ names_{out}\ values_{out},\ \exists!\ ctxt,$$
$$valid\_equations_{ctxt}\ eqns \rightarrow$$
$$names_{in} \mapsto_{ctxt} values_{in} \rightarrow$$
$$\text{map fst } ctxt = names_{in} +\!\!+ names_{out} +\!\!+ temps \rightarrow$$
$$names_{out} \mapsto_{ctxt} values_{out} \rightarrow$$
$$values_{in} \mapsto_{\textbf{node } f(names_{in})\rightarrow(names_{out}) \textbf{ vars } temps \textbf{ let } eqns \textbf{ tel}} values_{out}$$

Cependant cette formule est relativement arbitraire car elle il existe plusieurs variante qui sont intéressantes de considérer. En effet l'unicité de l'existance du contexte n'est pas forcément une nécessité. Par exemple pour si on prend le système $\{y = 0 * x\}$ où x est une variable temporaire et y une variable renvoyé. Alors la valeur de y est indépendante de la valeur de x. Il peut donc être décidé lors du choix de la sémantique que l'unicité de la valeur de x est inutile et que seulement l'unicité de la valeur renvoyé est nécessaire. Cependant stocker une preuve de l'unicité directement dans la sémantique nécessite que les différentes passes de réécriture de code dans le compilateur doivent prouver la préservation de l'unicité. Ce qui peut être difficile dans le cas modification de l'ensemble des variables. Pour ressoudre ce problème, une autre possibilité serais de ne pas demander la moindre unicité directement dans la sémantique, mais seulement avoir le typeur qui prouve l'unicité et les différentes passes prouvent seulement la préservation de l'ensemble des résultats possibles.

En dehors du point aborder ci-dessus, cette sémantique possèdes plusieurs autres choses qu'il est important de remarquer :

— Elle est indépendante de l'ordre des équations, cela découle de la communativité et l'associativité du "et" logique.

— Elle ne garantie pas l'unicité des définitions contrairements aux précédentes, seulement la cohérence de ces définitions pour un contexte donné. Par exemple le système $\{y = x; y = x\}$ est parfaitement valide pour cette sémantique. De plus le système $\{x = 0 \times x\}$ l'est aussi.

— Elle ne calcule pas.

Le trosième point est le plus problématique. En effet cette sémantique ne calcule pas de contexte valide mais toute preuve qu'un programme est valide doit contenir tous les contextes de tous les nœuds. Pour cela, si l'on veux utiliser cette sémantique pour construire un compilateur certifié, alors si le typeur as pour but de garantir que l'évaluation d'un nœud ne plante pas il faut que la preuve de correction de celui ci calcule un contexte valide. Il faudrait donc définir une autre sémantique en plus de celle ci pour pouvoir prouver la correction d'un typeur. Cependant l'aspect relationelle de cette sémantique rend probablement plus facile les preuves de correction des autres passes d'un compilateur.

3.4 Sémantique par tri topologique

La sémantique par tri topologique est de loin la plus compliqué des 4 sémantiques présentés ici car elle fait appel as de nombreuses notion et preuves afin de pouvoir être définie.

Ces sémantique est une sémantique par appel par nom. En effet, plutôt que tout calculer jusqu'au résultat, l'évaluation regarde quel est le résultat voulu avant de remonter dans les dépendances pour les évaluer. Par exemple, si nous avons 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. En effet, l'évaluation de x dans le système $\{x = y; y = x\}$ ne termine pas. Or, toute fonction définie dans la logique de Coq doit posséder une preuve de terminaison. Cette nécessité est un pré-requis pour la cohérence car si il est possible de définir une fonction divergente alors il est possible d'exhiber une preuve de faux.

Cependant, convaincre Coq que des fonctions mutuellements récursives terminent toujours est ardu à moins d'avoir un argument strictement décroissant. Pour cela la méthode la plus courante pour prouver la terminaison est de rajouter un nouvel argument strictement décroissant.

Parmi les différentes possibilités d'arguments à rajouter, le plus courant est de rajouter un entier (que l'on nomme couramment "carburant") qui décroit strictement à chaque appel récursif. Cependant cette technique possède 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.

De plus dans le cas d'un compilateur, quand on réécrit un morceau de code, on risque de changer la quantité de carburant nécessaire pour évaluer une expression. Il faut donc réussir à garantir que cette modification est aussi accompagné d'une modification du carburant fourni afin de s'assurer que l'on ne change pas un code qui est interprété comme divergeant en un autre qui ne diverge pas ou inversement.

Pour éviter ces soucis il existe une autre possibilité : fournir un prédicat d'accessibilité. Il s'agit d'un terme dont le type dépend des arguments de notre fonction dont on veux prouver la terminaison et dont les sous-termes correspondent aux appels récursifs de la fonction. Cela permet d'éviter de ne jamais manquer de carburant si le GADT qui sert de prédicat d'accessibilité est bien défini. Utiliser un prédicat possède l'énorme intérêt que si le GADT utilisé pour définir le prédicat d'accessibilité est une proposition, alors le code extrait ne contient pas ce la prédicat d'accessibilité. On obtient donc un code OCaml plus propre et plus efficace. Cependant cette technique possède aussi ses limites car il faut être capable de prouver l'existance d'un prédicat d'accessibilité. Or, pour prouver l'existance d'un tel précidat il faut que notre fonction termine bien sur les arguments fournis. Afin de résourdre ce problème l'évaluation d'un nœud commence donc par vérifier si l'évaluation va terminer ou non. Puis, si le vérificateur de terminaison accepte le système de déclaration, le système de déclaration est utilisé pour évaluer les valeurs de renvoie.

Afin de pouvoir tester si l'évaluation va terminer ou non, on commence par réécrire notre système de déclaration en une liste d'équations. Puis un tri topologique est effectué sur ces équations pour obtenir la garanti qu'il n'existe pas de cycles. Ce tri est effectué sur le graphe orienté obtenue en regardant si une équation dépend du calcul d'une autre. 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. En raison de l'existance des tableaux en Usuba et la possibilité de les définir en plusieurs fois, une équation peut dépendre d'une variable x sans pour autant que l'on ai x qui apparaîsse dans l'équation. Une définition plus en détail de la relation de dépendance est fourni 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.
— De plus, toute modification sur le programme oblige de pouvoir garantir que la vérificateur de terminaison continue as accepter la programme fourni.

Le second problèmes rend donc cette sémantique difficilement utilisable pour prouver la correction d'un compilateur. Cependant les outils implémentés lors de son implémentation peuvent

être très utile pour un compilateur. Plus particulièrement, le vérificateur de terminaison est probablement une étape nécessaire dans un typeur pour une sémantique où l'évaluation de dépend pas de l'ordre des équations. Car pour pouvoir garantir que l'évaluation ne génère pas d'erreur il faut garantir qu'il n'est pas possible d'avoir une erreur de divergence.

3.5 Sémantique par point fixe

Cette dernière sémantique est sensé être plus légère que la précédente tout en étant encore une sémantique qui calcule et qui est 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. Pour cela, la fonction d'évaluation parcours la liste des équations en essayant de les évaluer. Pour chaque équation on a deux possibilités :

1. Soit on réussi à évaluer l'expression de cette équation, on modifie alors le contexte et on peut oublier l'équation.
2. Soit 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 renvoyer le contexte calculé.
2. Le nombre d'équations a strictement diminué mais il en reste : on refait une itération 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 de cette évaluation termine après un nombre d'itération d'au plus la quantité initiale d'équations. Mais on remarque que dans le cas où l'on restreint le nombre d'itération à uniquement 1, alors on retombe sur la sémantique par évaluation de la section 3.2 où l'on as interdit les équations de modification.

Cependant cette sémantique ne permet pas d'interpréter certains programmes valide pour la sémantique relationelle (section 3.3). En effets le système $\{y = y\}$ n'est pas valide pour notre nouvelle sémantique mais l'est pour la sémantique relationelle. Ceci vient du fait que la sémantique relationelle considère comme valide tous les contextes qui sont des point fixes du système de déclaration alors que celle ci calcule un plus petit point fixe (et il n'en existe pas pour le système $\{y = y\}$). Cependant l'existance d'un plus petit point fixe ne garanti pas non plus l'absence d'erreur dans l'évaluation d'un système (par exemple le système $\{y = 0 \times y\}$).

Ce plus cette sémantique, tout comme la sémantique par tri topologique garanti que toute valeur est défini au plus une fois contrairements à la sémantique relationelle. Mais cette sémantique ne garanti pas que toutes les variables intermédiaries sont bien défini contrairements à la sémantique par tri topologique où le vérificateur de terminaison s'assure que tous les tableaux ne sont jamais partiellement défini même si la définition est réparti dans plusieurs équations différentes.

4 Tri topologique sur les équations

Nous allons désormais revenir sur la définition du tri topologique sur les équations effectué dans la sémantique par tri topologique 3.4. Ce tri est important car il fait aussi parti des étapes utiles à l'implémentation d'un typeur qui vérifierais que le système d'équations est bien fondé et que l'évaluation à l'aide de la sémantique par point fixe **??** termine bien.

Afin de pouvoir effectuer un tri topologique sur les équations il faut d'abord avoir 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. Or pour pouvoir effectuer un tri sur le système $\{v[0] = 1; v[1] = v[0]\}$ où l'on a v de type **U V** 1[2] il faut avoir une notion plus précise que seulement : "l'équation $v[1] = v[0]$ utilise et défini v".

Maintenant si l'on regarde l'exemple ci dessous avec v de type **U V** 1[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 identifiant comme une liste d'entiers. Plus plus on dit que le chemin est une instanciation d'une liste d'indiçages si chaque entiers est une instanciation de l'indiçage correspondant et que les deux listes ont la même longueur.

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

On utilise cette notion de chemin pour obtenir la condition suivante : 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 a $x \prec y$.

Cependant cette propriété 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 de $x \prec y$: 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. et c_x est un préfixe de c_y ou c_y est un préfixe de c_x

5 Extensions possibles

Plusieurs extensions de Usuba et des fonctionnalités présentés ci dessus sont possibles. Dans les paragraphes suivants nous présenterons donc de tels extensions et des intérêts que celles ci ont.

5.1 Autoriser l'indiçage sur les expressions

À l'heure actuelle dans Usuba il est uniquement possible de faire des indiçage sur des variables et pas sur des expressions. Ceci est une conséquence direct de l'absence de système de type clair qui permet de s'avoir comment les structures de tableau se propagent lors différentes opérations. Cependant le nouveau système de type présenté dans la section 2 permet de résoudre ce problème. De plus autoriser une telle syntaxe d'indiçage sur une expression pourrais permettre de faire de la substitution sur les termes en rendant la syntaxe compositionnelle.

Malgré ce bénéfice, une telle modification rendrais la sémantique non compositionnelle. En effet $v[1]$ n'aurais pas le même comportement suivant si l'indiçage as lieu sur une variable v ou une expression. Cette différence est que si on indice sur une expression alors il faut que tout v soit défini

et pas seulement la partie utilisée. Il est possible de définir une sémantique compositionnelle pour pouvoir avoir le même comportement pour les deux interprétations de la syntaxe en autorisant de manipuler des valeurs non définies. Cependant si on autorise les opérations à manipuler des valeurs potentiellement non défini alors le compilateur risque de générer des codes qui plantent. En effet, on pourrais alors écrire $\{y[1] = (x/y)[0]\}$ et si ce calcul n'est pas simplifié au cours de la compilation le code généré calculeras $x[1]/y[1]$ avec $y[1]$ non défini. Or si $y[1]$ est nul, le code assembleur associé générera une erreur de division par zéro.

5.2 Augmenter les types possibles pour les opérateurs binaires

Un autre ajout à Usuba possible serais d'augementer le nombre de codes typables en autorisant les opérations binaires entre deux types différents mais compatibles. Actuellement on ne peux pas calculer $x + y$ où x est de type **U V** 32[2][3] et y de type **U V** 32[6] d'après le système de type présenté précédemment. Cependant parmi les 28 exemples valide de Usuba pour le compilateur actuel, une telle opération est utilisé 2 fois.

L'idée serais donc de transformer la règles Binop de la figure 6 en celle de la figure 11. Où $\tau_1 \wedge \tau_2$ est défini uniquement pour deux tableaux multidimensionnels ayant le même type atomique et le même nombre d'éléments.

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

$$\frac{\begin{array}{c}\Gamma, P, A \vdash_E e_1 : \tau_1 \\ \Gamma, P, A \vdash_E e_2 : \tau_2 \\ A \vdash \textbf{ClassOf}\ binop\ (\tau_1 \wedge \tau_2)\end{array}}{\Gamma, P, A \vdash_E (e_1\ \textbf{into}\ \tau_1 \wedge \tau_2)\ binop_{\tau_1 \wedge \tau_2}\ (e_2\ \textbf{into}\ \tau_1 \wedge \tau_2) : \tau_1 \wedge \tau_2}\ \textsc{Binop}$$

Figure 11 – Nouvelle règle de typage des opérateurs binaires

Une telle règle permet donc de typer correctement l'exemple précédent. Cependant il existe plusieurs définitions possibles pour ce PGCD sur deux types de tableaux :

1. Si les deux types sont compatibles, renvoyer le premier
2. Si les deux types sont identiques en renvoyer un sinon renvoyer le type du tableau unidimensionel associé $type_{elements}[nb_{elements}]$
3. Renvoyer le type du tableau unidimensionel associé dans tous les cas
4. Préserver toutes les dimensions extérieures identiques et applatir à partir de la première dimension différente
5. Préserver toutes les dimensions intérieures identiques et applatir à partir de la première dimension différente
6. Préserver autant de dimensions intérieures et extérieures que possible et applatir le milieu

À part la troisième règle qui semble particulièrement arbitraire il est difficile de choisir parmi les autres si l'une est plus intéressantes les autres. Or, toutes les occurences d'une telle opération dans les codes existant se font entre un tableau unidimensionel et un tableau bidimensionel. Les règles 2, 4, 5 et 6 sont donc indistinguables sur ces exemples ce qui rend toute comparaison difficile.

Cependant comme il fallais choisir une règle lors de l'implémentation des sémantique, c'est la règle 4 qui as été choisi. Une opération entre **U V** 32[2][3][2][2] et **U V** 32[2][6][2] renvoie donc un **U V** 32[2][12].

6 Conclusion