Usuba, vers une formalisation du langage

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

La date

Le contexte général

Usuba [1] est un langage de haut niveau développé pour écrire des primitives cryptographiques qui cumulent à la fois un haut débit et un temps de calcul indépendant des valeurs.

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 différents types d'unités de calcul vectoriel sont utilisées :

- Les unités AVX afin de faire des opérations arithmétiques entre des entiers 16, 32 ou 64 bits en parallèle
- Les registres usuels qui permettent de faire des opérations logiques entre 32 ou 64 bits en parallèle

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 conditionnels et les accès mémoires.

Afin d'éviter les saut conditionnels 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 conditionnels (`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

Ce langage possède déjà un compilateur (nommé *usubac*), cependant celui ci possède plusieurs défauts :

— la correction du dit compilateur n'est pas prouvé formellement, contrairement à CakeML [2] ou CompCert [3]
— 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.

Lorsque quelqu'un écrit un code, la dite personne espère comprendre ce que fait le code qui est écrit. À moins de lire le code généré, ceci nécessite de faire confiance au compilateur et de comprendre avec exactitude le code initial. Cependant un compilateur est une mécanisme complexe composé de multiples étapes de réécritures ce qui rend compliqué la possibilité de comprendre ce qu'il fait. Il devient donc quasiment impossible de savoir ce que fait exactement le compilateur afin d'avoir confiance dans le compilateur ainsi que d'obtenir une spécification claire de la sémantique. Cependant un compilateur certifié par assistant de preuve permet de remplir ces deux conditions en fournissant une spécification de la sémantique ainsi qu'une preuve vérifiable mécaniquement de la préservation de la sémantique.

La contribution proposée

Afin de commencer à palier ces problèmes, ce rapport présenteras une proposition de système de type, ainsi que 4 spécification différentes d'une sémantique de Usuba implémenté en Coq à l'aide de différentes méthodes. Cependant lors de ce travail des difficultés ont été rencontré en raison de constructions en Usuba qui se typent mal. Pour cela, nous présenterons aussi de nouvelles constructions afin d'améliorer Usuba.

Plusieurs spécification différentes de sémantique pour Usuba sont proposées afin de pouvoir à la fois clarifier certains comportement du compilateur actuel, mais aussi d'expérimenter sur différents comportement possible de certaines constructions afin de voir comment rapprocher ce langage d'un modèle plus équationel. Les spécificités et avantages des différentes sémantiques seront notamment discutés.

Les arguments en faveur de sa validité

Afin de tester la validité des sémantiques implémentés, les trois qui calculent ont été extraites de Coq vers du code OCaml afin de tester le comportement de deux primitives cryptographiques implémenté en Usuba : ACE et AES.

Afin de tester la validité du comportement de ces deux primitives pour une sémantique donné, la primitives est simulé sur un vecteur test fourni dans la spécification de la primitive afin de vérifier que le résultat est 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 a permis d'exhiber des difficultés dans le comportement existant des codes Usuba ce qui permet d'ouvrir des pistes de réflexion sur les évolutions possibles du langage. De plus les différentes implémentations de sémantiques et les discussions associés permettrons d'avoir un recul sur quelle implémentation choisir pour une implémentation d'un compilateur certifié. De plus cet effort de développement a permis de mettre en place des outils qui permettrons de faciliter une telle implémentation.

1 Syntaxe et comportement actuel de Usuba

Au court des différentes sections de ce rapport on se basera sur un même code afin d'expliquer les différentes notions mises en jeu. Il s'agira de l'implémentation en Usuba de rectangle avec un nœud pour appliquer Rectangle [4] sur une liste présenté ci dessous 1.

Listing 1 – Rectangle appliqué à une liste

```
table SubColumn (input:v4) returns (out:v4) {
    6, 5, 12, 10, 1, 14, 7, 9, 11, 0, 3, 13, 8, 15, 4, 2
}

node ShiftRows (input:u16[4]) returns (out:u16[4])
vars
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel

node Rectangle (plain:u16[4],key:const u16[26][4]) returns (cipher:u16[4])
vars
    tmp : u16[26][4]
let

    tmp[0] = plain;
    forall i in [0,24] {
      tmp[i+1] = ShiftRows( SubColumn( tmp[i] ^ key[i] ) )
    }

    cipher = tmp[25] ^ key[25]
tel

node MapRectangle(plain:u16[64][4], key:const u16[64][26][4]) returns (cipher:u16[6
vars
let
  forall i in [0, 64] {
    cipher[i] = Rectangle(plain[i], key[i])
  }
tel
```

Comme on peut le voir dans l'exemple, un programme Usuba est composé de plusieurs nœuds. Il en existe deux types : les nœuds d'équations (`ShiftRows, Rectangle` et `MapRectangle`) et les tableaux (`SubColumn`) comme indiqué dans la figure 1. Ces nœuds correspondent à des fonctions du premier ordre. Les nœuds peuvent en appeler un autre, mais seulement un de ceux qui ont étés définis avant et les appels récursifs ne sont pas autorisés. Les récursions ne sont pas autorisées car le langage ne contient pas de conditionnelles. En effet, si l'on pouvais faire des appels récursif alors on aurais systématiquement une boucle infinie.

Les tableaux permettent d'implémenter des S-BOX. Les nœuds reçoivent un tableaux de n

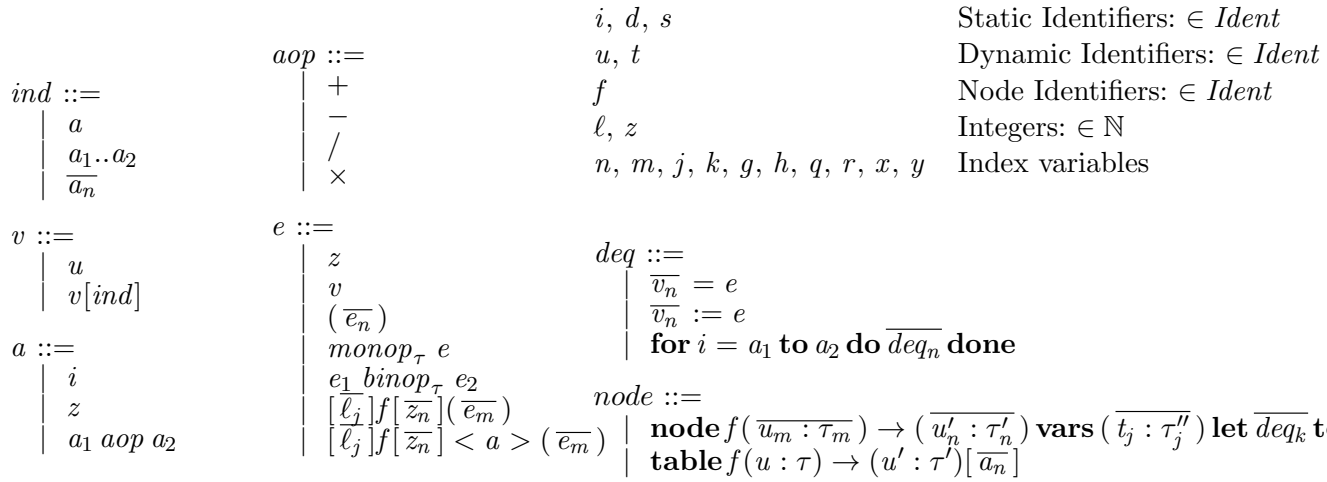

FIGURE 1 – AST de Usuba

entiers de b bits qu'ils comprennent comme b entiers de n bits. Puis ces entiers sont utilisés pour faire un accès dans le tableau fournis puis l'on transpose de nouveau la représentation binaire des entiers pour retourner n' entiers de b bits.

Les nœuds d'équations sont composés d'une liste de déclarations. Ces déclarations expliquent comment calculer la valeur des variables renvoyées à partir des variables fournis en entrée. Dans l'exemple de Rectangle, le nœud `ShiftRows` est composé de 4 déclarations qui permettent de calculer `out` à partir de `input`. Au total, il existe trois types de déclarations possibles :

- Les boucles `for` dont les deux bornes sont connu à la compilation et qui sont composé d'une liste d'équations. Il s'agit de sucre syntaxique afin d'écrire de façon conscise un grand nombre 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 équationnel 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.

3 des 4 sémantiques présenté dans la section 3 ne sont définie que pour une liste d'équations. Leur définissions commence donc par réécrire le système de déclarations en une liste d'équations en retirant le sucre syntaxique des boucles `for`.

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. La syntaxe pour les appels de nœuds est elle plus compliqué que usuellement. Un tel appel dépend de deux listes d'entiers qui permettent d'expliquer au compilateur quel foncteur utiliser afin d'appliquer le nœud sur des tableaux. La liste de droite permet d'expliquer le nombre de fois qu'il faut appliquer le nœuds. Cela correspond donc à une séquence de *map*. Par exemple pour l'exemple de Rectangle 1, on peut réécrire le nœud `MapRectangle` en une nouvelle version en utilisant cette syntaxe 2. La liste de gauche explique comment modifier l'ordre des dimensions des tableaux afin que le l'itération du *map* ne soit pas forcément appliqué sur les dimensions extérieures. L'idée étant de pouvoir appliquer un *map* mais en itérant sur d'autres dimensions que celles extérieures. Par exemple si l'on veux appliquer un nœud sur toutes les lignes ou toutes les

colonnes d'une matrice. Cette syntaxe avec les listes d'un appel de nœud est une nouveauté pour le langage Usuba qui n'est pas implémenté dans *usubac* et qui a pour but de diminuer le nombres de boucles `for` qui doivent être écrites.

Listing 2 – Rectangle appliqué à une liste

```
node MapRectangle(plain:u16[64][4], key:const u16[64][26][4]) returns (cipher:u16[6
vars
let
  cipher = Rectangle[64](plain, key)
tel
```

Plus d'explications sur ces listes sont fournis avec les règles de typages dans la section 2.

Les constructeurs de variables sont défini récursivement comme un identifiant ou 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 intervalle qui est juste du sucre syntaxique pour la liste de tous les entiers dans l'intervalle

Par exemple si l'on a 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, dans l'implémentation actuelle de *usubac*, 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.

La solution que nous proposons pour résoudre ce problème est de remplacer les indiçage par une séquence d'indiçages car cela permet d'écrire $x[0, 1 : 0]$ pour parler de $(x[0][0], x[1][0])$, et garder $x[0, 1][0]$ pour désigner $x[0]$. Cette nouvelle syntaxe fait perdre la rétro-compatibilité avec les vieux codes Usuba, cependant cela permet d'avoir une sémantique compositionnelle.

2 Règles de typage

Lors de la section précédente nous avons présenté la syntaxe de Usuba. Cependant ce langage ne contient pas de système de type. Il est donc difficile de savoir quels programmes ne vont pas être accepté par le compilateur à cause d'une erreur de typage. Ceci est notamment une conséquence des coercions implicites qui sont très présentes en Usuba.

Pour cela nous définissions les types τ comme une tableau multi-dimensionnels contenant un type atomique σ qui correspond à un entier muni d'une certaine taille *size* et d'une orientation *dir* comme indiqué dans la figure 2. Cependant la taille et l'orientation sont potentiellement non spécifié afin de permettre du polymorphisme. Par exemple le type $u32$ dans l'exemple 1 correspond au type atomique **U** *dir* 32 où *dir* est une variable de direction. Quand à lui, le type $v4$ correspond au type de tableau **U** *dir size*[4]. L'idée étant que par défaut pour un nœuds, à chaque fois qu'une direction (resp. taille) n'est pas spécifié cela correspond à un même paramètre qui sera spécifié au niveau de l'appel du nœud.

Cependant les opérations de calculs ne sont pas défini sur tous les entiers. Par exemple, les opérations arithmétiques sont défini sur les entiers 32 ou 64 bits mais pas sur les entiers 38 bits sur la plupart des CPU. Pour pouvoir garantir au typage que toutes les opérations utilisées sont bien définies, 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

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

FIGURE 2 – Types et contextes en USUBA

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

$$\frac{A \vdash \mathbf{Arith}\,\tau}{A \vdash \mathbf{Arith}\,\tau[\ell]}\ \text{ARITHL}$$

$$\frac{A \vdash \mathbf{Logic}\,\tau}{A \vdash \mathbf{Logic}\,\tau[\ell]}\ \text{LOGICL}$$

FIGURE 3 – Inférence des 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. Cependant afin de savoir exactement quelles classes de types sont définies, il faut spécifier à la compilation vers quelle architecture doit être compilé le code.

À partir des types τ on définie $\mathcal{T}$ comme une liste de types τ afin de pouvoir représenter le type d'une expression.

À partir de cette syntaxe des types, on peut désormais définir les règles de typage des variables. Pour cela on définie d'abord dans la figure 4 comment une liste d'indiçages 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 les règles de typage des 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 τ, mais que 2 règles (MONOP et BINOP) ne sont définies que sur un type unique τ.

La règle la plus compliqué parmi les différentes expressions est la règle de typage d'un appel de nœud. En effet, lors d'un appel de nœuds il se passe deux choses qui rendent cette règle de typage particulièrement compliqué :

- Premièrement, le nœud peut être appelé plusieurs fois sur différents morceaux du tableau. L'idée est que chaque argument est un tableau dont on extrait **prod** $[\overline{\ell'_h}]$ sous tableaux et que le nœud est appelé sur chaque série de tableaux. Cependant les dimensions sur lesquels ont itère pour générer les sous tableaux ne correspondent pas forcément aux dimensions extérieures car cela permet de pouvoir appliquer un nœud aux colonnes ou aux lignes d'une matrice suivant le besoin.
- La seconde difficulté au moment de l'appel de nœud est la coercion des arguments vers le type voulu par la fonction. 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. Une coercion entre deux types n'est possible que si les deux types sont équivalent pour la notion d'équivalence présenté dans la figure 7.

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

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

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

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

FIGURE 4 – Typage des indiçages

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

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

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

FIGURE 5 – Typage des variables

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). Il s'agit de la règle BOOL.

— Deux listes de types sont identiques si elle contiennent le même nombre d'entiers de chaque tailles et orientations dans le même ordre.

Cependant ces règles de typage font perdre de l'expressivité à USUBA car elles ne permettent pas de typer certaines opérations actuellement utilisées dans des codes USUBA. Par exemple si l'on a x de type $\mathbf{U}\ \mathbf{V}\ 32[2]$ alors $x + (x[0], x[1])$, n'est pas typable.

Pour palier à ce problème nous introduisons dans le langage USUBA deux nouvelles constructions : les constructeurs de tableaux et les coercions explicites. Les constructeurs de tableaux on pour but de pouvoir permettre de gérer de nombreux soucis en permettant de créer des tableaux plutôt que des multiplets qui ont pour type une liste de τ. Cependant cela ne permet toujours pas à gérer tous les cas. C'est pour ça que l'on introduit les coercions explicites. Cela nous donnes deux nouvelles règles de typage présentées dans la figure 8.

Une fois que l'on sait comment typer les expressions on peut désormais vérifier que les déclarations sont bien typées. Pour cela on vérifie que les équations préservent bien le type et que les boucles contiennent que des sous déclarations cohérentes comme indiqué dans la figure 9.

Le typage des nœuds contient deux règles présentées 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

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

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

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

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

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

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

FIGURE 6 – Typage des expressions

toutes les variables.

Le typage d'une table est plus subtil. En effet une table prend en entrée $i1$ entiers de s bits et les considère comme s entiers de $i1$ bits en transposant 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 laquelle 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 sémantique voulu dans un assistant de preuve. L'assistant de preuve choisi pour cette formalisation est Coq en raison de sa compatibilité avec OCAML car *usubac* est écrit dans ce langage.

Au total 4 sémantiques différentes ont été implémentées. Parmi celle ci, 3 d'entre elles calculent un résultat et la quatrième est une relation qui vit dans `Prop`.

Les différences entre ces sémantiques se situent au niveau de leur définition et de leur gestion du contexte. Les discussions dans les paragraphes qui suivent se concentrerons sur ces différences mais nous présenterons tout d'abord leurs points commun.

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

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

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

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

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

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

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

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

Figure 7 – Équivalence 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 – Typage des expressions, partie 2

3.1 Points communs des sémantiques

Tout d'abord toutes les sémantiques ont une même notion de valeur qui est un type somme entre 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 calculent, les erreurs sont représentées par un type option où `None` correspond à une erreur.

Le second point commun entre ces 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 corps 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.

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

$$\frac{\Gamma, P, A \vdash_E e : \mathcal{T} \quad \Gamma \vdash_V \overline{v_n} : \mathcal{T}}{\Gamma, P, A \vdash_D \overline{v_n} := e} \text{ EqnT}$$

$$\frac{\Gamma, P, A \vdash_E e : \mathcal{T} \quad \Gamma \vdash_V \overline{v_n} : \mathcal{T}}{\Gamma, P, A \vdash_D \overline{v_n} = e} \text{ EqnF}$$

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

Figure 9 – Typage des équations

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

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

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

Figure 10 – Typage d'un nœud

3.2 Sémantique par évaluation

La première sémantique implémenté pour Usuba est une sémantique par évaluatation.

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

Par exemple dans le système $\{v[0] = 1; v[1] = v[0]\}$ avec v de type **U V** 32[2], 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 par une fonction qui prend en entrée un contexte et qui en renvoie un nouveau si la sémantique ne produit aucune erreur. Pour modifier le contexte, cette fonction évalue l'expression puis utilise la valeur obtenue ainsi que la liste de variables pour mettre à jour 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, une équation sans modification = n'accepte de remplacer que des `None` par des valeurs dans le contexte alors qu'une équation avec modification := n'accepte de remplacer que des `Some`.

Mais cela à pour conséquence que le comportement d'un nœud dépend de l'ordre des équations. Certaines permutations de l'ordre préservent la sémantique, cependant toutes les permutations ne sont pas équivalentes. Ceci est une conséquence immédiate de l'existence d'équations avec modification. Les autres sémantiques ont été définies pour ne pas dépendre de l'ordre des équations afin de plus ressembler à un modèle équationnel. Cependant elles ne supportent donc plus la possibilité d'écrire des équations avec modification.

Pour ce qui est d'utiliser cette sémantique dans des preuves de préservation de la sémantique, il est facile de définir une équivalence de programme. Pour cela on peut définir que deux expressions sont équivalentes si pour tous contextes elles s'évaluent en les mêmes valeurs. On obtient alors une relation d'équivalence congruente qui indique que si deux expressions sont équivalentes alors elles ont la même sémantique. Et de la même façon on peut définir les équivalences d'équations, de déclaration et de nœuds. De plus, cette sémantique possède l'avantage d'avoir un ordre d'évaluation clair ce qui permet de faire plus facilement des preuves de préservation de la sémantique pour les différentes étapes de la compilation.

Désormais nous allons présenter les 3 autres sémantiques qui ont été définies 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 relationnelle

Cette sémantique est définie à partir de relations. Par exemple, la sémantique d'une expression est définie par une relation entre une expression, un contexte et une valeur. Ceci permet de représenter de façon plus concise les erreurs car si l'évaluation d'une expression dans un certain contexte plante, alors, pour la dite expression, la relation n'associe aucune valeur au contexte.

La spécificité de cette sémantique se situe au niveau de la gestion du contexte. La sémantique d'une équation est définie comme 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.

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

Cependant cette formule est relativement arbitraire car il en existe plusieurs variante qui sont intéressantes de considérer. En effet l'unicité de l'existence 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 résoudre ce problème, une autre possibilité est 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 abordé 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 commutativité et l'associativité du "et" logique).
— Elle ne garantie pas l'unicité des définitions contrairement 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 troisième 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 a 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. Malgré ce défaut, l'aspect relationnelle 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 présentées ici car elle fait appel a ! de nombreuses notions et preuves afin de pouvoir être définie.

Cette sémantique est une sémantique par appel par nom. En effet, plutôt que tout calculer jusqu'au résultat, l'évaluation regarde quelles variables doivent être connues puis remonte dans les calculs afin de pouvoir obtenir le résultat. Par exemple, si nous avons un nœud qui retourne `x`, on va donc chercher dans quel équation 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 s'il est possible de définir une fonction divergente alors il est possible d'exhiber une preuve de faux.

Cependant, convaincre Coq que des fonctions mutuellement 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 décroît strictement à chaque appel récursif. Cependant cette technique possède plusieurs limitation :

— Cela modifie le code extrait.
— Il est difficile de garantir que l'on a mis suffisamment 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 (Generalized Algebraic Data Type) 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 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'existence d'un prédicat d'accessibilité. Or, pour prouver l'existence d'un tel précidat il faut que notre fonction termine bien sur les arguments fournis. Afin de résoudre ce problème, l'évaluation d'un nœud commence par vérifier si l'évaluation va terminer ou non. Puis, si le vérificateur de terminaison l'accepte, le système de déclarations 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 déclarations en une liste d'équations. Puis, un tri topologique est effectué sur ces équations pour obtenir la garantie qu'il n'existe pas de cycles. Ce tri est effectué sur le graphe orienté obtenu en regardant si une équation dépend du calcul d'une autre. En effet, si on prend deux équations tel que la première définie une variable `x` qui est utilisé par la seconde, la seconde équation dépend de la première. En raison de l'existence des tableaux en Usuba et la possibilité de les définir en plusieurs fois, les dépendances sont complexes et expliquées plus en détail dans la section 4.

Cette sémantique possède deux grosses limitations :

— La sémantique est particulièrement lourde à définir car afin de prouver l'existence 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 a 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 à 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 tout de même 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 sensée être plus légère que la précédente tout en étant encore une sémantique qui calcule indépendamment de l'ordre des équations.

L'idée derrière celle ci est que chaque équation est évaluée 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 a parcouru 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 a 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 a interdit les équations de modification.

De plus, on est convaincu que cette sémantique est un sous ensemble stricte de la sémantique relationnelle de la section 3.3. L'inclusion est sensé être vrai mais cela n'a pas été vérifier dans Coq. Pour ce qui est du stricte il existe des exemples simple de codes accepté par seulement une seule des deux sémantiques. Par exemple, le système $\{y = x; y = y\}$ (où x est fourni en entrée) n'est pas valide pour notre nouvelle sémantique mais l'est pour la sémantique relationnelle. Ceci vient du fait que la sémantique relationnelle accepte les duplicata de définition. De plus la nouvelle sémantique calcule un plus petit point fixe mais l'existence d'un plus petit point fixe sans duplicata de définition ne garanti pas l'absence d'erreur. Par exemple, le système $\{y = 0 \times y\}$ est accepté seulement par la sémantique relationnelle.

Même si cette sémantique, tout comme la sémantique par tri topologique, garanti que toute valeur est défini au plus une fois elle ne garanti pas que toutes les variables intermédiaires sont bien défini. Contrairement 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. Cependant ceci est surtout un détail d'implémentation pour les deux sémantiques et cette différence peut être modifiée.

4 Tri topologique sur les équations

Nous allons désormais revenir sur la définition du tri topologique sur les équations calculé dans la sémantique par tri topologique 3.4. Ce tri est important car il fait aussi parti des étapes nécessaire à l'implémentation d'un typeur qui vérifierait que le système d'équations est bien fondé. Car tout compilateur aura besoin de produire un code ordonné en sortie et donc de pouvoir d'ordonner les calculs.

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] : on remarque que l'équation 3 nécessite les deux autres équations.

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

Pour parler de cela on définit la notion de chemin dans un identifiant comme une liste d'entiers. De 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 intervalle, 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 définit le chemin c de v alors on a $x \prec y$.

Cependant cette propriété n'est pas encore suffisante. En effet si une définition définit 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 existe 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

À partir de cette relation d'ordre il est donc possible de calculer un tri topologique entre les équations. S'il existe un cycle alors les définitions sont mutuellement dépendantes ce qui ne devrait pas avoir lieu. L'absence de cycle garanti que toute évaluation des équations peut terminer ce qui permet pour la sémantique par tri topologique d'exhiber un prédicat d'accessibilité.

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 indiçages sur des variables et pas sur des expressions. Ceci est une conséquence direct de l'absence de système de type clair qui permet de savoir comment les structures de tableaux 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 l'intérêt présenté ci dessus, une telle modification rendrais la sémantique non compositionnelle. En effet, $v[1]$ n'aurais pas le même comportement suivant si l'indiçage a lieu dans une variable ou sur l'expression v. La différence est que, si on indice sur l'expression v alors il faut que tout v soit défini et pas seulement la partie utilisée.

Il serais 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 génèrent des erreurs à l'exécution. Par exemple, si l'on peut écrire $\{y[1] = (x/y)[0]\}$ et si ce calcul n'est pas simplifié au cours de la compilation le code généré calculera $x[1]/y[1]$ avec $y[1]$ non défini. Or si $y[1]$ est nul, le code assembleur associé générera une erreur de division par zéro.

5.2 Élaboration de types

Les règles de typages présenté dans la section 2 font que de nombreux codes ne typent plus. Ceci est une conséquence du fait que de nombreuses coercions implicites ne sont plus valide. Une

alternative pour pouvoir récupérer toutes les opérations qui ne sont désormais plus expressibles est de rajouter une étape d'élaboration de type dans le compilateur afin de pouvoir continuer à interpréter tous ces codes. Ceci permettrais de rajouter des coercions explicites dans le code afin d'avoir selon l'utilisateur des coercions implicites alors que la sémantique n'accepte que des coercions explicites.

Il y a deux opérations qui ne sont plus possible à cause des règles de typages proposé parmi les 28 exemples valide de code écrits en Usuba. Ces opérations sont les coercions au niveau des équations et les opérations binaires entre deux types différents.

Pour ce qui est des coercions implicites pour les équations, ce problème se résout facilement en remplaçant toutes les occurrences de $v = e$ en $v = e$ **into** (**typeof** v).

Pour ce qui est des opérations binaires, 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 types. L'idée de serais d'utiliser l'élaboration de type pour calculer un type $\tau_1 \wedge \tau_2$ à partir des types τ_1 et τ_2 de x et y. Mais seulement dans les cas où τ_1 et τ_2 sont des tableaux avec le même nombre d'éléments et que les éléments sont du même type. Puis de transformer le code en $(x$ **into** $\tau_1 \wedge \tau_2) + (y$ **into** $\tau_1 \wedge \tau_2)$.

De point de vue de l'utilisateur, cela donnerais l'impression qu'il y aurais une nouvelle règle Binop qui serait celle présenté dans la figure 11, alors qu'en réalité on utilise juste une coercion.

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

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

Figure 11 – Nouvelle règle de typage des opérateurs binaires

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 unidimensionnel associé $type_{elements}[nb_{elements}]$
3. Renvoyer le type du tableau unidimensionnel associé dans tous les cas
4. Préserver toutes les dimensions extérieures identiques et aplatir à partir de la première dimension différente
5. Préserver toutes les dimensions intérieures identiques et aplatir à partir de la première dimension différente
6. Préserver autant de dimensions intérieures et extérieures que possible et aplatir 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, les deux occurrences d'une telle opération dans les codes existant se font entre un tableau unidimensionnel et un tableau dimensionnel. Les règles 2, 4, 5 et 6 sont donc indistinguables sur ces exemples ce qui rend toute comparaison difficile.

5.3 Boucles temporelles

Un des objectifs de ce travail est d'essayer de rendre les codes écrit en Usuba plus esthétiques. Cela passe notamment par retirer les boucles qui font très impératives et les remplacer par des structures plus équationelles. En l'occurrence, la plupart des boucles dans les codes Usuba sont

des *map* et des *fold* écrit à l'aide d'une boucle. Pour ce qui est des *map*, nous avons déjà présenté un syntaxe et un typage qui permet d'effectuer une telle opération avec un simple appel de nœud annoté.

Pour ce qui est des *fold*, l'idée est de s'inspirer de Lustre et rajouter deux constructions `fby` et `wrap` dans le langage. L'idée étant qu'un `wrap` est une boucle mais avec une notion de temporalité et l'opération `fby` permet de calculer la valeur d'une expression à l'instant précédent. Afin d'expliquer mieux comment marche ces constructions nous allons nous baser sur une réécriture du nœud `Rectangle` de l'exemple 1 mais avec ces constructions 3.

Listing 3 – Rectangle appliqué à une liste

```
node Rectangle (plain:u16[4],key:const u16[26][4]) returns (cipher:u16[4])
vars
    tmp : u16[26][4]
let

  chipher = wrap i in [0, 24] vars tmp:u16[4] return tmp ^ key[25]
    let
      tmp = ShiftRows( SubColumn( (plain fby tmp) ^ key[i] ) )
    tel
tel
```

L'idée du `wrap` est d'exécuter l'ensemble des équations qui lui sont fournis à chaque instant afin de calculer un nouveau contexte. Puis renvoie la valeur de l'expression fourni au dernier instant (ici à l'instant 24). La construction `fby` permet elle d'interagir avec le temps. À la première itération (ici l'instant 0) la valeur renvoyé et celle de l'expression à gauche. Puis lors des instants suivant, la valeur renvoyé est celle de l'expression de droite calculé dans le contexte de l'instant précédant.

Cela permet donc d'implémenter facilement des *fold* et ainsi de pouvoir remplacer dans les codes Usuba la plupart des boucles qui resterais après avoir utilisé la syntaxe pour écrire des *map*.

5.4 Conditions et arguments statiques

Une autre amélioration de Usuba serait de pouvoir rajouter des conditionnelles et la possibilité de faire des fonctions récursives. Cela peut sembler contradictoire avec les propos tenus lors de l'introduction mais il existe un moyen de rajouter des conditions dans Usuba. En effet, il n'est pas possible de faire des conditions qui dépendent des valeurs pour des raisons de sécurité. Cependant il est possible de rajouter des conditionnelles qui dépendent de paramètres statiques connu à la compilation. De plus afin de pouvoir utiliser pleinement cette fonctionnalité en Usuba il faudrait pouvoir fournir à l'appel de nœud un argument statique connu à la compilation.

Il serait donc possible avec de tels fonctionnalités d'écrire une fonction récursive qui calcule un terme de la suite de Fibonacci.

Listing 4 – Fibonacci

```
node Fibo<i>() returns (val:u64)
let
  val = if i > 2
        then Fibo<i-1>() + Fibo<i-2>()
        else 1
tel
```

6 Conclusion

Dans ce rapport nous avons vu que Usuba est à l'heure actuelle un langage incomplet dont il manque une spécification de sa sémantique et que celle qui peut peut être inféré à partir de l'implémentation du compilateur est non compositionnelle. Cependant nous avons aussi vu différentes méthodes afin d'implémenter en Coq une spécification d'une sémantique pour Usuba ainsi que plusieurs constructions qui ont pour but d'améliorer la propreté des codes Usuba ainsi que de clarifier la sémantique en retirant des ambiguïtés. De plus un système de types a aussi été proposé afin de pouvoir dans le futur intégrer un typeur à *usubac* afin d'aider les utilisateurs à avoir des messages d'erreur plus clair et prédictibles.

7 Annexe

Le code Coq pour spécifier les différentes sémantiques de la section 3 est disponible sur le répositoire : `https://github.com/samsa1/usuba_coq`.

Ce répositoire contient notamment :

— Un spécification en Coq de l'AST d'Usuba : src/usuba_AST.v
— La sémantique par évaluation : src/usuba_sem.v
— La sémantique relationnelle : src/relation_semantic.v
— La sémantique par tri topologique : src/topo_sort/topo_sem.v
— La sémantique par point fixe : src/subst_semantic.v
— Des preuves de correction d'optimisations : src/normalization
— Deux exemples de primitives : src/examples/ace_bitslice.v et src/examples/aes.v
— L'extracteur (src/extract.v) et le script de test test.sh

Références

[1] Darius Mercadier et al. "Usuba : Optimizing & Trustworthy Bitslicing Compiler". In : *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*. WPMVP'18. Vienna, Austria : Association for Computing Machinery, 2018. isbn : 9781450356466. doi : 10.1145/3178433.3178437. url : https://doi.org/10.1145/3178433.3178437.

[2] Ramana Kumar et al. "CakeML : a verified implementation of ML". In : *Principles of Programming Languages (POPL)*. ACM, 2014. doi : 10.1145/2535838.2535841.

[3] Xavier Leroy. "Formal verification of a realistic compiler". In : *Communications of the ACM* 52.7 (2009). doi : 10.1145/1538788.1538814.

[4] Wentao Zhang et al. "RECTANGLE : a bit-slice lightweight block cipher suitable for multiple platforms". In : *Sci. China Inf. Sci.* 58.12 (2015), p. 1-15. doi : 10.1007/s11432-015-5459-7. url : https://doi.org/10.1007/s11432-015-5459-7.