

USUBA, vers une formalisation du langage

Samuel VIVIEN, sous l'encadrement de Pierre-Évariste DAGAND – IRIF

La date

Le contexte général

USUBA₁ [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'USUBA₁ 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 sur des architectures arm ou intelx64

La seconde propriété est recherchée par les développeurs de primitives cryptographiques car cela permet de diminuer le risque de fuite de données lié aux attaques par canaux cachés. En effet, si le temps d'exécution d'un code dépend de la clé secrète il est possible d'obtenir des informations sur la dite clé 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. USUBA₁ 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 USUBA₁ deux types de tableaux.

- Il y a les tableaux statique dont le contenu est connu à la compilation : il s'agit des S-Boxes, utilisées dans les primitives cryptographique. Il est possible d'accéder au contenu de ces tableaux avec une valeur arbitraire car sinon on ne pourrait écrire que 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 [2] sur comment trouver les codes les plus efficace possible pour retirer ces accès mémoire.
- Il y a aussi les tableaux dynamique dont le contenu n'est connu qu'à l'exécution. Pour ces tableaux, les seuls accès possible sont par des indices connu à la compilation. On peut 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 formellement prouvée, contrairement à CakeML [3] ou CompCert [4],
- 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'USUBA₁.

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 un mécanisme complexe composé de multiples étapes de réécritures ce qui le rend difficilement compréhensible. Il devient donc quasiment impossible de savoir ce que fait exactement le compilateur afin d'avoir confiance dans le compilateur. 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 à ces problèmes, ce rapport présentera une variante de USUBA₁ nommé USUBA₂, munie d'une proposition de système de type, ainsi que 4 spécifications différentes d'une sémantique de USUBA₂ implémenté en Coq à l'aide de différentes méthodes. Cependant lors de ce travail des difficultés ont été rencontrées en raison de constructions en USUBA₁ qui se typent mal. Pour cela, nous présenterons aussi de nouvelles constructions afin d'améliorer USUBA.

Plusieurs spécification différentes de sémantique pour USUBA₂ 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 équationnel. 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 implémentées en USUBA₂ : ACE et AES.

Afin de tester la validité du comportement de ces deux primitives pour une sémantique donnée, la primitives est simulée 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 associées permettront 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 permettront de faciliter une telle implémentation.

1 Syntaxe et comportement de Usuba₁

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 USUBA₁ de l'algorithme de chiffrement nommé Rectangle [5]. On définit dans le bloc de code suivant 1 le nœud **Rectangle** qui correspond à l'algorithme de chiffrement et le nœud **MapRectangle** qui applique l'algorithme point à point sur des listes de 64 éléments.

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[64][4])
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 USUBA₁ 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 été définis avant et les appels récursifs ne sont pas autorisés. Les récursions ne sont pas autorisées car

		i, d, s	Static Identifiers: $\in Ident$
		u, t	Dynamic Identifiers: $\in Ident$
		f	Node Identifiers: $\in Ident$
		ℓ, z	Integers: $\in \mathbb{N}$
		$n, m, j, k, g, h, q, r, x, y$	Index variables
$ind ::=$	$aop ::=$		
$\begin{array}{ l} a \\ a_1..a_2 \\ \overline{a_n} \end{array}$	$\begin{array}{ l} + \\ - \\ / \\ \times \end{array}$		
$v ::=$	$e ::=$	$deq ::=$	
$\begin{array}{ l} u \\ v[ind] \end{array}$	$\begin{array}{ l} z \\ v \\ (\overline{e_n}) \end{array}$	$\begin{array}{ l} \overline{v_n} = e \\ \overline{v_n} := e \end{array}$	
$a ::=$	$\begin{array}{ l} monop_{\tau} e \\ e_1 binop_{\tau} e_2 \\ [\overline{\ell_j}]f[\overline{z_n}](\overline{e_m}) \\ [\overline{\ell_j}]f[\overline{z_n}] < a > (\overline{e_m}) \end{array}$	$\begin{array}{ l} \textbf{for } i = a_1 \textbf{ to } a_2 \textbf{ do } \overline{deq_n} \textbf{ done} \end{array}$	
$\begin{array}{ l} i \\ z \\ a_1 aop a_2 \end{array}$		$node ::=$	
		$\begin{array}{ l} \textbf{node } f(\overline{u_m : \tau_m}) \rightarrow (\overline{u'_n : \tau'_n}) \textbf{ vars } (\overline{t_j : \tau''_j}) \textbf{ let } \overline{deq_k} \textbf{ tel} \\ \textbf{table } f(u : \tau) \rightarrow (u' : \tau')[\overline{a_n}] \end{array}$	

FIGURE 1 – AST de Usuba

le langage ne contient pas de conditionnelles. En effet, si l'on pouvait faire des appels récursif alors on aurait systématiquement une boucle infinie.

Les tableaux permettent d'implémenter des S-BOX. Les nœuds reçoivent un tableau de n entiers de b bits qui peuvent être interprétés comme une matrice de $n \times b$ bits. Puis on interprète chacune des b colonnes comme un entier de n bits qui est utilisé pour faire un accès dans le tableau fournis. On obtient donc une matrice de $n' \times b$ bits ce qui correspond à 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 connues à la compilation et qui sont composées d'une liste d'équations. Il s'agit de sucre syntaxique afin d'écrire de façon concise 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 équationnelle d'un nœud en raison de sa nature impérative. Elle n'est donc pas supportée dans la plupart des sémantiques en raison de son incompatibilité avec d'autres fonctionnalités. Ceci n'est pas un problème car cette construction est vouée à disparaître.

Parmi les 4 sémantiques présentées dans la section 3, 2 ne sont définies que pour une liste d'équations. Leur définition commence donc par réécrire le système de déclarations en une liste d'équations en retirant le sucre syntaxique des boucles **for**. Cependant cette méthodologie est peu satisfaisante et peut être résolu avec la proposition de la section 5.3.

Les différents constructeurs d'expressions correspondent à ce que l'on peut trouver usuellement dans un langage de programmation : appel de nœuds, opérateurs binaire et unaire, tuples, constantes et variables. La syntaxe pour les appels de nœuds est plus compliquée qu'usuellement. Un tel appel dépend de deux listes d'entiers qui permettent d'expliquer au compilateur comment appliquer le nœud sur des tableaux. La liste de droite permet d'expliquer le nombre de fois qu'il faut appliquer le nœud. Cela correspond donc à une séquence de *map*. Par exemple pour l'exemple 1 le nœud **MapRectangle** fait 64 appels de nœud simple (i.e. les deux listes sont vides). Mais on peut écrire

une nouvelle version en utilisant cette syntaxe pour obtenir l'exemple 2.

Listing 2 – Rectangle appliqué à une liste

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

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 veut 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 USUBA₂ qui n'est pas implémenté dans *usubac* et qui a pour but de diminuer le nombres de boucles **for** qui doivent être écrites.

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

Les constructeurs de variables sont défini récursivement comme un identifiant ou un indigage sur une variable. Un indigage 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 $[2, 0]$.

L'implémentation actuelle de *usubac* fournis une sémantique qui n'est pas compositionnelle. En effet, 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 $[0, 1]$.

La solution que nous proposons pour résoudre ce problème est de remplacer les indigage par une séquence d'indigages 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 précédents codes USUBA₁, cependant cela permet d'avoir une sémantique compositionnelle.

2 Règles de typage

Lors de la section précédente nous avons présenté la syntaxe de USUBA₂. Cependant ce langage ne possède pas de système de type. Il est donc difficile de savoir quels programmes vont être regetés par le compilateur. Ceci est notamment une conséquence des coercions implicites qui sont très présentes en USUBA₁.

Pour cela nous définissons les types τ comme une tableau multi-dimensionnels contenant un type atomique σ . On parle de type atomique car il s'agit de la construction de base pour parler de tous les types des valeurs. Un tel type σ correspond à un entier muni d'une certaine taille *size* et d'une orientation *dir* comme indiqué dans la figure 2. Par exemple, **U V 32** correspond à un entier de 32 bits stocké verticalement et **U H 64** correspond à un entier de 64 bits stocké horizontalement. Un entier vertical correspond à le représentation usuel d'un entier et un entier horizontal de b bits est lui stocké dans b registres différents.

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

FIGURE 2 – Types et contextes en USUBA₂

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

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

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

FIGURE 3 – Inférence des classes de types

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 $\mathbf{U} \text{ } dir \text{ } 32$ où dir est une variable de direction. À partir de ces types atomiques on peut construire des tableaux pour ainsi obtenir des types tels que $\mathbf{U} \text{ } \mathbf{H} \text{ } 32[4]$ ou le type $v4$ qui correspond à $\mathbf{U} \text{ } dir \text{ } size[4]$. 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 calcul ne sont pas définies sur tous les entiers. Par exemple, les opérations arithmétiques sont définies sur les entiers 32 ou 64 bits mais pas sur les entiers 38 bits. Pour pouvoir garantir au typage que toutes les opérations utilisées sont bien définies, le langage USUBA₂ contient des classes de types ($typc$ dans la figure 2) qui permettent de spécifier sur quels types sont définies les opérations logiques, arithmétiques et de décalage. Certaines classes de types peuvent être définies 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 définit \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 définit d'abord dans la figure 4 comment une liste d'indicateurs modifie les dimension d'un tableau puis en appliquant récursivement ces règles on obtien les règles de typage des variables présenté dans la figure 5.

À partir du typage des variables, on 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 deux règles (MONOP et BINOP) ne sont définies que sur un tableau et non pas une liste de tableaux.

La règle la plus compliquée 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 :

- Premièrement, le nœud peut être appelé plusieurs fois sur différents morceaux du tableau.

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

$$\begin{array}{c}
\sigma[\overline{d_n}] - [\overline{ind_m}] \rightarrow \sigma[\overline{d'_k}] \\
\vdash 0 \leq a < \ell \\
\hline
\sigma[\ell][\overline{d_n}] - [a; \overline{ind_m}] \rightarrow \sigma[\overline{d'_k}] \quad \text{INDEX} \\
\\
\sigma[\overline{d_n}] - [\overline{ind_m}] \rightarrow \sigma[\overline{d'_k}] \\
\vdash 0 \leq a_1 < \ell \\
\vdash 0 \leq a_2 < \ell \\
\hline
\sigma[\ell][\overline{d_n}] - [a_1..a_2; \overline{ind_m}] \rightarrow \sigma[abs(a_1 - a_2) + 1][\overline{d'_k}] \quad \text{RANGE} \\
\\
\sigma[\overline{d_n}] - [\overline{ind_m}] \rightarrow \sigma[\overline{d'_k}] \\
\vdash 0 \leq \overline{a_j} < \ell \\
\hline
\sigma[\ell][\overline{d_n}] - [\overline{a_j}; \overline{ind_m}] \rightarrow \sigma[\mathbf{len} \overline{a_j}][\overline{d'_k}] \quad \text{SLICE}
\end{array}$$

FIGURE 4 – Typage des indiçages

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

$$\begin{array}{c}
\frac{\Gamma \vdash_I u : \tau \in \Gamma}{\Gamma \vdash_V u : \tau} \quad \text{IDENT} \\
\\
\frac{\Gamma \vdash_V v : \tau_1 \quad \tau_1 - [\overline{ind_n}] \rightarrow \tau_2}{\Gamma \vdash_V v [\overline{ind_n}] : \tau_2} \quad \text{INDEXING}
\end{array}$$

FIGURE 5 – Typage des variables

L'idée est que chaque argument est un tableau dont on extrait $\mathbf{prod} [\overline{\ell'_h}]^1$ sous tableaux et que le nœud est appelé sur chaque série de tableaux. Cependant les dimensions sur lesquels on 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 utilisées en USUBA₁ 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 présentée 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 entiers 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 taille et orientation 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 USUBA₁. Par exemple si l'on

1. Notation pour parler du produit des éléments d'une liste.

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

$$\begin{array}{c}
\frac{\Gamma \vdash_V v : \tau}{\Gamma, P, A \vdash_E v : \tau} \text{ VAR} \\
\\
\frac{\Gamma, P, A \vdash_E e_1 : \tau \quad \Gamma, P, A \vdash_E e_2 : \tau \quad A \vdash \mathbf{ClassOf} \text{ binop } \tau}{\Gamma, P, A \vdash_E e_1 \text{ binop}_\tau e_2 : \tau} \text{ BINOP} \\
\\
\frac{\Gamma, P, A \vdash_E e : \tau \quad A \vdash \mathbf{ClassOf} \text{ monop } \tau}{\Gamma, P, A \vdash_E \text{ monop}_\tau e : \tau} \text{ MONOP} \\
\\
\frac{\Gamma, P, A \vdash_E \overline{e_n} : \overline{\mathcal{T}_n}}{\Gamma, P, A \vdash_E (\overline{e_n}) : \overline{\mathcal{T}_n}} \text{ TUPLE} \\
\\
\frac{P \vdash f : \forall \overline{d_n}, \forall \overline{s_m}, \overline{\text{typc}_j} \Rightarrow \mathcal{T}_1 \rightarrow \mathcal{T}_2 \quad \Gamma, P, A \vdash_E (\overline{e_n}) : \mathcal{T}'_1 \quad A \vdash \overline{\text{typc}_j}[\overline{d_n} \leftarrow \overline{d'_n} ; \overline{s_m} \leftarrow \overline{s'_m}] \quad \mathcal{T}_1[\overline{d_n} \leftarrow \overline{d'_n} ; \overline{s_m} \leftarrow \overline{s'_m}] = \overline{\sigma_x[\ell_g][z_{x,q}]} \quad \mathcal{T}_2[\overline{d_n} \leftarrow \overline{d'_n} ; \overline{s_m} \leftarrow \overline{s'_m}] = \overline{\sigma'_y[\ell_g][z'_{y,r}]} \quad \mathcal{T}'_1 \cong \overline{\sigma_x[\ell_g][\ell'_h][z_{x,q}]} }{\Gamma, P, A \vdash_E [\overline{\ell_g}]f[\overline{\ell'_h}](\overline{e_n}) : \overline{\sigma'_y[\ell_g][\ell'_h][z'_{y,r}]}} \text{ FUN}
\end{array}$$

FIGURE 6 – Typage des expressions

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 USUBA₂ deux nouvelles constructions : les constructeurs de tableaux et les coercions explicites. Les constructeurs de tableaux ont pour but de pouvoir permettre de gérer d'écrire des multiplats 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 donne 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 toutes les variables.

Le typage d'une table est plus subtil car il reflète la sémantique d'une table présenté dans la section 1. Une table prend en entrée i_1 entiers de s bits pour obtenir s entiers de i_1 bits. Ces entiers permettent de faire s accès dans la table qui contient $1 \ll i_1$ entiers de i_2 bits. On obtient alors s entiers de i_2 bits qui transformé en i_2 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.

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

$$\begin{array}{c}
\frac{}{\overline{\mathcal{T} \cong \mathcal{T}}} \text{ REFL} \\
\frac{\mathcal{T}_1 \cong \mathcal{T}_2}{\mathcal{T}_2 \cong \mathcal{T}_1} \text{ SYM} \\
\frac{\mathcal{T}_1 \cong \mathcal{T}_2 \quad \mathcal{T}_2 \cong \mathcal{T}_3}{\mathcal{T}_1 \cong \mathcal{T}_3} \text{ TRANS} \\
\frac{\mathcal{T}_1 \cong \mathcal{T}_2}{\tau :: \mathcal{T}_1 \cong \tau :: \mathcal{T}_2} \text{ REC} \\
\frac{}{\overline{\mathbf{U} \text{ dir } s[\overline{a'_m}] :: \mathcal{T} \cong \mathbf{U} \text{ dir } s[\mathbf{prod}[\overline{a'_m}]] :: \mathcal{T}}} \text{ SIMPLFORM} \\
\frac{}{\overline{\mathbf{U} \mathbf{V} 1[\overline{a'_m}] :: \mathcal{T} \cong \mathbf{U} \mathbf{H} 1[\overline{a'_m}] :: \mathcal{T}}} \text{ BOOL} \\
\frac{}{\overline{\mathbf{U} \text{ dir } s[\ell_1] :: \mathbf{U} \text{ dir } s[\ell_2] :: \mathcal{T} \cong \mathbf{U} \text{ dir } s[\ell_1 + \ell_2] :: \mathcal{T}}} \text{ JOIN}
\end{array}$$

FIGURE 7 – Équivalence de types

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

$$\begin{array}{c}
\frac{\Gamma, P, A \vdash_E e : \mathcal{T}_1 \quad \mathcal{T}_1 \cong \mathcal{T}_2}{\Gamma, P, A \vdash_E e \mathbf{into} \mathcal{T}_2 : \mathcal{T}_2} \text{ INTO} \\
\frac{\Gamma, P, A \vdash_E \overline{e_n} : \sigma[\overline{a_m}]}{\Gamma, P, A \vdash_E [\overline{e_n}] : \sigma[\text{len } \overline{e_n}][\overline{a_m}]} \text{ ARRAY}
\end{array}$$

FIGURE 8 – Typage des expressions, partie 2

3 Sémantiques

Afin de fournir une spécification du langage USUBA₂ 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 quatres sémantiques différentes ont été implémentées. Parmi celles-ci, trois 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 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 un entier (pour les valeurs statiques) et un tableau multidimensionnel qui est défini comme un

$$\boxed{\Gamma, P, A \vdash_D \text{deg}}$$

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

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

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

FIGURE 9 – Typage des équations

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

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

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

FIGURE 10 – Typage d'un nœud

triplet direction, entiers stockés dans la tableau 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.

3.2 Sémantique par évaluation

La première sémantique implémentée pour USUBA₂ est une sémantique par évaluation. Cette sémantique est définie de façon intuitive : on évalue toutes les équations dans l'ordre du programme. 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 pas encore défini.

Par exemple dans le système $\{v[0] = 1; v[1] = v[0]\}$ avec v de type $\mathbf{U} \mathbf{V} 32[2]$, entre les deux

équations seulement un des éléments du tableau est défini. On représente donc ça par un contexte qui à v associe $\text{InR}(\mathbf{V}, [\text{Some } 1, \text{None}], [1])$.

À partir de cela on définit la sémantique d'une équation par une fonction qui prend en entrée un contexte et qui en renvoie une version modifiée 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 d'équations est quant à elle définie par itération de la sémantique d'une équation.

Cette sémantique est celle la plus proche de l'implémentation actuelle de USUBA_1 car c'est la seule des 4 qui accepte de définir une même variable plusieurs fois. L'idée étant, qu'une équation sans modification = rajoute des valeurs dans le contexte (i.e. remplace des **None** par une valeur) alors alors qu'une équation avec modification $:=$ modifie le contexte (i.e. remplace des **Some** par d'autres valeurs).

Mais cela a 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 pas la possibilité d'écrire des équations mutable.

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 énoncer 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 déclarations 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 équationnelle.

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 génère une erreur, 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 que la liste de variables et l'expression associés s'évaluent bien en des valeurs compatibles pour le contexte fourni. Le contexte quand à lui est définie au niveau de la sémantique globale d'un nœud.

$$\begin{aligned}
& \forall \text{names}_{in} \text{values}_{in} \text{names}_{out} \text{values}_{out}, \exists! \text{ctxt}, \\
& \text{valid_equations}_{\text{ctxt}} \text{eqns} \rightarrow \\
& \text{names}_{in} \mapsto_{\text{ctxt}} \text{values}_{in} \rightarrow \\
& \text{map fst ctxt} = \text{names}_{in} ++ \text{names}_{out} ++ \text{temps} \rightarrow \\
& \text{names}_{out} \mapsto_{\text{ctxt}} \text{values}_{out} \rightarrow \\
& \text{values}_{in} \mapsto \mathbf{node} \ f(\text{names}_{in}) \rightarrow (\text{names}_{out}) \ \mathbf{vars} \ \text{temps} \ \mathbf{let} \ \text{eqns} \ \mathbf{tel} \ \text{values}_{out}
\end{aligned}$$

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, si on prend le système $\{y = 0 * x\}$ où x est une variable temporaire et y une variable renvoyée. 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 possède 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. Cependant toute preuve qu'un programme est valide doit contenir tous les contextes de tous les nœuds. Donc si un typeur veut garantir formellement que l'évaluation d'un nœud ne plante pas, il faut une preuve de correction qui calcule un contexte valide. Ceci complique donc l'utilisation de cette sémantique dans un compilateur certifié. 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 compliquées 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 en 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 \mathbf{x} , on va donc chercher dans quel équation est définie \mathbf{x} . Puis l'on évalue l'expression associé afin d'obtenir la valeur de \mathbf{x} , mais cela nécessite potentiellement de connaître la valeur de \mathbf{y} . On continue donc récursivement en calculant la valeur de \mathbf{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 limitations :

- 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 veut 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 la famille inductive qui sert de prédicat d'accessibilité est bien définie. Utiliser un prédicat possède l'énorme intérêt que si la famille inductive utilisée 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 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 prédicat 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 renvoi.

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 définit une variable x qui est utilisée 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. Si il s'agit d'un prérequis pour la sémantique.

Le second problème 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 ne 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 connaît pas encore l'ordre dans lequel il faut le faire. Pour cela, la fonction d'évaluation parcourt la liste des équations en essayant de les évaluer. Pour chaque équation on a deux possibilités :

1. Soit on réussit à évaluer l'expression de cette équation, on modifie alors le contexte et on peut oublier l'équation.
2. Soit on ne réussit 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 lieu :

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 vérifié dans Coq. Cependant, il existe des exemples simple de codes accepté par seulement la sémantique relationnelle. 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 définies. 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 calculées dans la sémantique par tri topologique section 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 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'indigages si chaque entiers est une instanciation de l'indigage 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 définit 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 à la sémantique par tri topologique d'exhiber un prédicat d'accessibilité.

5 Extensions possibles

Étant donné que la sémantique de **USUBA**₁ est non compositionnelle des modifications doivent être apporté au langage. Certaines de ces modifications ont déjà été présenté précédemment et d'autres extensions de **USUBA**₁ sont possibles et ont été partiellement étudiées. Dans les paragraphes suivants nous présenterons donc de telles extensions et leur intérêt.

5.1 Autoriser l'indigage sur les expressions

À l'heure actuelle dans USUBA₁ il est uniquement possible de faire des indigages sur des variables et pas sur des expressions. Ceci est une conséquence direct de l'absence de système de type ne qui permet pas de savoir comment les structures de tableaux se propagent lors de 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'indigage sur une expression pourrait 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'aurait pas le même comportement suivant si l'indigage 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 serait 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 définies 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 présentées dans la section 2 font que de nombreux codes écrit en USUBA₁ ne typent plus. Ceci est une conséquence du fait que de nombreuses coercions implicites ne sont plus valides. 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 permettrait 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 proposées parmi les 28 exemples valides de code écrits en USUBA₁. 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 peut 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 serait 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** $\tau_1 \wedge \tau_2)$.

Du point de vue de l'utilisateur, cela donnerait l'impression qu'il y aurait une nouvelle règle BINOP (figure 11), alors qu'en réalité on utilise juste une coercion.

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

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

$$\frac{\begin{array}{c} \Gamma, P, A \vdash_E e_1 : \tau_1 \\ \Gamma, P, A \vdash_E e_2 : \tau_2 \\ A \vdash \mathbf{ClassOf} \textit{binop} (\tau_1 \wedge \tau_2) \end{array}}{\Gamma, P, A \vdash_E e_1 \textit{binop}_{\tau_1 \wedge \tau_2} e_2 : \tau_1 \wedge \tau_2} \text{BINOP}$$

FIGURE 11 – Nouvelle règle de typage des opérateurs binaires

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 intéressante que 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 écrits en USUBA plus esthétiques. Cela passe notamment par retirer les boucles qui induisent un style impératif et les remplacer par des structures plus équationnelles. En l'occurrence, la plupart des boucles dans les codes USUBA₁ sont des *map* et des *fold* écrits à l'aide d'une boucle. Pour ce qui est des *map*, nous avons déjà présenté une 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 [6] et rajouter deux constructions **fb**y et **w**rap dans le langage. L'idée étant qu'un **w**rap est une boucle mais avec une notion de temporalité et l'opération **fb**y permet de calculer la valeur d'une expression à l'instant précédent. Afin d'expliquer mieux comment fonctionnent ces constructions nous allons nous baser sur une réécriture du nœud **Rectangle** de l'exemple 1 mais avec ces constructions (exemple 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
  cipher = 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 **w**rap 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 fournie au dernier instant (ici à l'instant 24). La construction **fb**y permet elle d'interagir avec le temps. À la première itération (ici, l'instant 0) la valeur renvoyée est celle de l'expression à gauche. Puis lors des instants suivants, la valeur renvoyée est celle de l'expression de droite calculée dans le contexte de l'instant précédent.

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

5.4 Conditions et arguments statiques

Une autre amélioration de USUBA₁ 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 connus 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 par expansion statique du circuit.

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 USUBA₁ est à l'heure actuelle un langage incomplet dont il manque une spécification de sa sémantique et que celle qui peut être inférée 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 propriété des codes USUBA₁ 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 clairs et prévisibles.

7 Annexe

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

Ce référentiel contient notamment :

- Une spécification en Coq de l'AST d'USUBA₂ : `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] Joan BOYAR et René PERALTA. “A depth-16 circuit for the AES S-box.” In : t. 2011. Jan. 2011, p. 332. ISBN : 978-3-642-30435-4. DOI : 10.1007/978-3-642-30436-1_24.
- [3] Ramana KUMAR et al. “CakeML : a verified implementation of ML”. In : *Principles of Programming Languages (POPL)*. ACM, 2014. DOI : 10.1145/2535838.2535841.
- [4] Xavier LEROY. “Formal verification of a realistic compiler”. In : *Communications of the ACM* 52.7 (2009). DOI : 10.1145/1538788.1538814.
- [5] 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>.
- [6] Timothy BOURKE et al. “A formally verified compiler for Lustre”. In : *ACM SIGPLAN Notices* 52 (juin 2017), p. 586-601. DOI : 10.1145/3140587.3062358.