

Rapport de stage

Développement d'un noyau de programmation synchrone

Jordan Ischard
3ème année de licence Informatique
Université d'Orléans

25 Mars 2019

Table des matières

1	Présentation de l'équipe	3
2	Introduction	4
3	Préliminaires	5
3.1	λ -calcul et machines abstraites	5
3.2	Machines abstraites	9
3.3	Programmation réactive	15
4	Langage fonctionnel réactif	16
4.1	Machine réactive pure	16
4.1.1	Description informelle du langage	16
4.1.2	Sémantique de la machine abstraite	19
4.2	Le partage des valeurs dans la machine	22
4.2.1	Description informelle du langage	22
4.2.2	Sémantique de la machine abstraite	23
4.3	La gestion des erreurs	27
5	Conclusion	27
6	Annexes	28
6.1	Les Exemples des machines étudiées	28
6.1.1	Exemple de fonctionnement de la machine CK	28
6.1.2	Exemple de fonctionnement de la machine CEK	29
6.1.3	Exemple de fonctionnement de la machine SECD	30
6.2	Les différentes versions faite pour rendre la machine SECD concurrente	31
6.2.1	1ère-2ème version des règles de la machine SECD Concurrente	31
6.2.2	3ème version des règles de la machine SECD Concurrente	33
6.2.3	4ème version des règles de la machine SECD Concurrente	36
7	Bibliographie	41

1 Présentation de l'équipe

Remerciement Avant tout développement sur mon sujet de stage, j'aimerais remercier mes 2 professeurs qui m'ont encadré, pour m'avoir permis de faire ce stage de recherche et de m'avoir aidé tout le long de celui-ci. J'ai beaucoup appris grâce à eux. Je remercie donc Madame Bousdira et Monsieur Dabrowski pour tout.

Le Laboratoire d'Informatique Fondamentale d'Orléans (LIFO) est un laboratoire de l'Université d'Orléans et de l'INSA Centre-Val de Loire.

Les recherches menées au LIFO concernent la science informatique et les STIC. Elles vont de l'algorithmique au traitement des langues naturelles, de l'apprentissage au parallélisme massif, de la vérification et la certification à la sécurité des systèmes, du Big Data aux systèmes embarqués. Le laboratoire est structuré en cinq équipes :

- Contraintes et Apprentissage (CA)
- Graphes, Algorithmes et Modèles de Calcul (GAMoC)
- Langages, Modèles et Vérification (LMV)
- Pamda
- Sécurité des Données et des Systèmes (SDS)

Afin d'offrir une autre approche du laboratoire et de promouvoir la coopération entre équipes, les thématiques transversales suivantes ont été définies :

- Masse de données et calcul haute performance
- Modélisation et algorithmique
- Sécurité et sûreté

J'ai eu l'occasion de travailler avec une partie de l'équipe LMV, dans l'optique d'un stage rémunéré de 3 mois ; voici la description de l'équipe LMV.

L'objectif de l'équipe LMV est de contribuer à l'amélioration de la compréhension des problèmes de sûreté et de sécurité des systèmes informatiques. Des logiques «ordres partiels» aux langages de programmation usuels, les membres de l'équipe travaillent sur ces questions à différents niveaux d'abstraction et selon différents points de vue tout en cherchant à comprendre les relations fondamentales entre ces différentes approches. L'équipe est structurée autour de deux axes : la correction de programmes et la vérification de systèmes.

- Le premier axe s'intéresse au développement de techniques liées aussi bien à la vérification de propriétés spécifiques qu'à la satisfaction de propriétés fonctionnelles quelconques. Dans les deux cas les propriétés peuvent être assurées par construction ou a posteriori (vérification déductive).
- Le second axe repose d'une part sur l'étude de techniques à base de systèmes de réécriture comme par exemple les problèmes d'accessibilité dans les systèmes de réécriture et d'autre part sur l'étude des logiques dites «ordres partiels» et leur utilisation dans le cadre du développement d'outils de vérification efficaces.

2 Introduction

Le stage a pour intitulé *Programmation réactive synchrone, implantation d'une machine virtuelle*. Il se place dans la thématique *Sémantique des systèmes concurrents*. L'objectif de ce stage est de réaliser l'implantation d'une machine virtuelle (type JVM) destinée à exécuter un langage réactif synchrone purement fonctionnel encadré par deux maîtres de conférence : Mme Bousdira et Mr Dabrowski.

Pour cela une recherche préliminaire sur des machines existantes sera faite en premier lieu pour se mettre à niveau. Ensuite une recherche sur la programmation réactive servira à comprendre les enjeux du sujet et à cerner les différentes contraintes que la réactivité apporte. Les recherches étant faite on pourra s'atteler à la création de notre propre machine concurrente. Pour finir, on pourra conclure par un résumé le travail effectué, des choix qui pourrait être améliorés, des voies qui n'ont pas été choisies et ce qu'il reste à faire.

3 Préliminaires

Ayant un niveau de 3ème année de licence informatique, j'ai beaucoup de lacunes par rapport au travail demandé, il m'a fallu me mettre à niveau. Pour cela, trois articles ont été étudiés :

- [1] expliquant le fonctionnement du ReactiveML un langage de programmation réactif
- [2] développant toute la réflexion que l'on doit avoir pour créer un langage de programmation
- [3] expliquant le fonctionnement des machines avec les λ -calculs

Pour mieux structurer ma démarche, je vais diviser mes recherches en deux sous parties. La première sera liée à mes recherches "préliminaires" qui ne sont pas liées directement à la programmation réactive. Elle regroupera l'article [2] et [3]. La seconde sera dédiée à l'article [1] qui est lui complètement axé sur la programmation réactive.

3.1 λ -calcul et machines abstraites

Créer un langage de programmation Créer un langage est un processus complexe, il faut savoir se poser les bonnes questions. Mon but est ici de vous montrer une partie du processus de réflexion pour comprendre les problèmes que soulève la création d'un langage. Je vais m'appuyer sur les travaux de Xavier Leroy sur le *ZINC*.

Pourquoi ? La création d'un langage de programmation doit venir d'un besoin de celui-ci, par exemple quand on veut des critères spécifiques. Pour le *ZINC*, La portabilité du langage ML sur micro-ordinateur ainsi que l'utilisation pour la pédagogie ont été soulevés comme problèmes des implantations déjà existantes. La nécessité de la création d'un nouveau langage devient donc flagrante. D'ailleurs *ZINC* veut dire ZINC Is Not Caml, il pointe le principal langage qui a tous les problèmes exposés plus haut pour bien montrer qu'il ne va pas les faire.

Comment ? Le plus dur reste à faire. Maintenant on doit savoir ce que l'on veut dans notre langage, comment on pourra l'utiliser, qu'elle est la meilleure implantation pour une vitesse d'exécution optimale, qu'elle sera la méthode d'exécution.

On va aborder quelques points soulevés dans l'article[2]. Je vous conseille sa lecture si le sujet vous intéresse car il est bien détaillé et assez accessible pour un novice comme moi.

- Veut-on que notre langage soit utilisé pour de petits problèmes ou au contraire pour des problèmes de tous types de tailles. Si c'est le cas il faut pouvoir simplifier la vie à notre utilisateur en l'aidant à structurer son programme. Par exemple en Caml on a les **structures**, en java on a les **classes**, en C on a les **headers**, etc ... Pour le *ZINC*, la création de modules est possible avec un principe proche du C. De plus, le principe de module est le seul qui permet de combiner une compilation séparée de chaque module avec un typage fort statique.

Petit point sur le typage, le typage peut être soit statique, soit dynamique et pour chacune d'entre elles, on peut avoir un typage fort ou un typage faible. On va définir tous ça :

- typage **statique** : on vérifie, avant exécution, tout le code. Exemple : Caml
- typage **dynamique** : on vérifie au fur et à mesure le code au moment où l'on a la nécessité. Exemple : Python
- typage **fort** : il faut que tous les types correspondent entre-eux quand on les associe. Exemple : Caml
- typage **faible** : il peut y avoir des associations entre deux types pas tout à fait pareil. Exemple : C, on peut faire une égalité entre un pointeur et un entier il va juste prévenir mais pas interdire.

-

- Veut-on des fonctions n-aires ou utiliser la curryfication ? Déjà qu'est-ce que la curryfication ? La curryfication est la transformation d'une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction sur le reste des arguments.

Exemple en Caml : La version curryfiée de $let f = fun(x, y) \rightarrow x + y$ in $f(5, 7)$ est $let f = fun x \rightarrow fun y \rightarrow x + y$ in $f\ 5\ 7$

Le *ZINC* n'a pas de fonctions n-aires. Malgré la facilité et l'efficacité que l'on peut voir dans les fonctions n-aires, un problème se pose. Il est dur de prévoir le comportement avec les fonctions de hauts ordres et le polymorphisme. Pour éviter ça on va préférer la curryfication malgré son exécution plus lente. Une explication précise de comment palier à sa vitesse d'exécution est décrite dans l'article[2] que je vous conseille une nouvelle fois vivement.

- Quelle méthode d'exécution veut-on utiliser ?

Les λ -calculs

Les bases : Le λ -calcul est un système formel inventé par Alonzo Church dans les années 1930, qui fonde les concepts de fonction et d'application. On y manipule des expressions appelées λ -expressions, où la lettre grecque λ est utilisée pour lier une variable. Il y a trois termes qui composent les λ -expressions :

1. les variables : x, y, \dots sont des λ -termes
2. les applications : $u v$ est un λ -terme si u et v sont des λ -termes, on peut voir l'application comme ceci : si u est une fonction et v un argument, alors $u v$ est le résultat de l'application à v de la fonction u ;
3. les abstractions : $\lambda x.v$ est un λ -terme si x est une variable et v un λ -terme, on peut voir l'abstraction comme ceci : $\lambda x.v$ peut être interprétée comme la fonction qui à x , associe v , où v contient en général des occurrences de x .

La réduction : Maintenant que nous connaissons les termes qui composent les λ -expressions, il faut savoir comment les faire interagir entre eux. Les λ -calculs fonctionnent par réductions des termes grâce à l'application. Il existe trois règles de **réduction générale** :

- $(\lambda X_1.M) \alpha (\lambda X_1.M[X_1 \leftarrow X_2])$ où $X_2 \notin FV(M)$: elle sert à renommer les variables
- $((\lambda X_1.M_1)M_2) \beta M_1[X \leftarrow M_2]$: elle substitue une variable par un λ -terme
- $(\lambda X.(M X)) \eta M$ où $X \notin FV(M)$: elle représente le cas où g est une fonction qui à x associe $f(x)$ telle que f est une fonction alors autant utiliser directement la fonction f .

La réduction générale $\mathbf{n} = \alpha \cup \beta \cup \eta$.

La β -réduction est plus complexe et a ses propres règles :

- $X_1[X_1 \leftarrow M] = M$
- $X_2[X_1 \leftarrow M] = X_2$ où $X_1 \neq X_2$
- $(\lambda X_1.M_1)[X_1 \leftarrow M_2] = (\lambda X_1.M_1)$
- $(\lambda X_1.M_1)[X_2 \leftarrow M_2] = (\lambda X_3.M_1[X_1 \leftarrow X_3][X_2 \leftarrow M_2])$
où $X_1 \neq X_2, X_3 \notin FV(M_2)$ et $X_3 \notin FV(M_1) \setminus X_1$
- $(M_1 M_2)[X \leftarrow M_3] = (M_1[X \leftarrow M_3] M_2[X \leftarrow M_3])$

Simplification : Afin d'alléger l'écriture en enlevant des parenthèses, il y a des règles de priorité qui sont les suivantes :

- Application associative à gauche : $M_1 M_2 M_3 = ((M_1 M_2)M_3)$
- Application prioritaire par rapport à l'abstraction : $\lambda X.M_1 M_2 = \lambda X.(M_1 M_2)$
- Les abstractions consécutives peuvent être regroupées : $\lambda XYZ.M = (\lambda X.(\lambda Y.(\lambda Z.M)))$

Version écriture lourde

$$\begin{aligned} & ((\lambda x.((\lambda z.z) x)) (\lambda x.x)) \\ \rightarrow_n^\alpha & ((\lambda x.((\lambda z.z) x)) (\lambda y.y)) \\ \rightarrow_n^\eta & ((\lambda z.z) (\lambda y.y)) \\ \rightarrow_n^\beta & (\lambda y.y) \end{aligned}$$

Version écriture allégée

$$\begin{aligned} & \lambda x.(\lambda z.z) x \ \underline{\lambda x.x} \\ \rightarrow_n^\alpha & \lambda x.(\lambda z.z) x \ \underline{\lambda y.y} \\ \rightarrow_n^\eta & (\lambda z.z) \underline{\lambda y.y} \\ \rightarrow_n^\beta & \underline{\lambda y.y} \end{aligned}$$

Forme normale Une expression est une forme normale si on ne peut pas réduire l'expression via une β ou η réduction.

Théorème de la forme normale : Si on peut réduire L tels que $L =_n M$ et $L =_n N$ et que N et M sont en forme normale alors $M = N$ à n renommages près.

Théorème de Church-Rosser (pour $=_n$) : Si on a $M =_n N$, alors il existe un L' tels que $M \rightarrow_n L'$ et $N \rightarrow_n L'$.

Certaines lambda calcul expressions n'ont pas de forme normale comme : $((\lambda x.xx) (\lambda x.xx))$. D'autres en ont une mais on peut rentrer dans une boucle infinie de réduction si on choisit la mauvaise réduction.

Le problème se pose lorsqu'on évalue un argument de la fonction qui n'est jamais utilisé. Pour palier à ce problème, on utilise la stratégie d'appliquer toujours les β et η réductions les plus à gauche. Ces règles sont les suivantes :

- $M \rightarrow_{\bar{n}} N$ si $M \beta N$
- $M \rightarrow_{\bar{n}} N$ si $M \eta N$
- $(\lambda X.M) \rightarrow_{\bar{n}} (\lambda X.N)$

- $(M\ N) \longrightarrow_{\bar{n}} (M'\ N)$ si $M \longrightarrow_{\bar{n}} M'$
et $\forall L, (M\ N) \beta L$ impossible et $(M\ N) \eta L$ impossible
- $(M\ N) \longrightarrow_{\bar{n}} (M\ N')$ si $N \longrightarrow_{\bar{n}} N'$
et M est une forme normale
et $\forall L, (M\ N) \beta L$ impossible et $(M\ N) \eta L$ impossible

Cette solution est sûre mais reste peut utilisée car elle est assez lente.

Pour évaluer un langage, on utilise ce qui s'appelle une **stratégie d'évaluation**. Cette stratégie explique quand les arguments d'une fonction est évalué. Je vous présente les 2 stratégies qui nous intéressent :

1. **l'appel par nom** : pour une fonction f donnée, on évalue chaque arguments quand on en a besoin, c'est-à-dire que l'on évalue pas les arguments avant l'appel de la fonction. Cette stratégie est pratique quand on a des arguments non utilisés et quand on veut travailler avec des listes infinies. Mais par contre si tous les arguments sont utilisés cette stratégie est plus lente que celle présentée après car on doit réévaluer les arguments à chaque fois.

Cette stratégie fait partie du groupe des stratégie d'évaluation non stricte, c'est-à-dire qu'il n'évalue pas forcément la fonction en entier.

Exemple : Pour une fonction fst tels que pour deux arguments x et y on retourne le 1er. Si on a : $fst(3+4, 5/5)$ on va évalué $3+4$. Ce qui donne $fst(7, 5/5)$ et on retourne 7 on n'évalue pas $5/5$.

2. **l'appel par valeurs** : pour une fonction g donné, on évalue ces arguments avant d'évalué la fonction. Cette stratégie fait partie du groupe des stratégie d'évaluation stricte, c'est-à-dire qu'il évalue forcément la fonction en entier.

Exemple : Pour une fonction fst tels que pour deux arguments x et y on retourne le 1er. Si on a : $fst(3+4, 5/5)$ on va évalué $3+4$. Ce qui donne $fst(7, 5/5)$, on évalue $5/5$. On arrive à $fst(7, 1)$ et on retourne 7.

La preuve de la compréhension des λ -calculs a été faite à travers son implantation en OCaml. Cependant cette implantation reste simplifiée et une version plus complète sera présentée dans la suite.

L'appel par valeur est plus facile à mettre en oeuvre dans un langage. On va voir cela dans le langage ISWIM qui est un langage qui utilise l'appel par valeur.

ISWIM ISWIM est un langage impératif à noyau fonctionnel ; de fait, c'est une syntaxe lisible du λ -calcul à laquelle sont ajoutés des variables mutables et des définitions. Grâce au λ -calcul, ISWIM comporte des fonctions d'ordre supérieur et une portée lexicale des variables. Le but est de décrire des concepts en fonction d'autres concepts. Ce langage a fortement influencé les autres langages qui l'ont suivi, principalement dans la programmation fonctionnelle.

ISWIM a une grammaire étendue de la grammaire des λ -calcul.

$M, N, L, K =$

les termes des λ -calculs :

| X

| $(\lambda X.M)$

| $(M N)$

les nouveaux termes :

| b : une constante

| $(o^n M \dots N)$ avec o^n les fonctions primitives d'arité n

On définit une valeur telle que :

$V, U, W =$

| b

| X

| $(\lambda X.M)$

Les règles de β -réductions sont les mêmes que celles pour les λ -calculs avec 2 ajouts qui sont les suivants :

— $b[X \leftarrow M] = b$

— $(o^n M_1 \dots M_n)[X \leftarrow M] = (o^n M_1[X \leftarrow M] \dots M_n[X \leftarrow M])$

La β -réduction est la même qu'en λ -calcul mais à la condition que la réduction soit faite avec une valeur V .

- $((\lambda X.M) V) \beta_v M[X \leftarrow V]$

Cette restriction permet une sorte d'ordre dans les calculs.

Cependant l' η et l' α réduction ne sont plus vue comme telles. En effet l' η -réduction n'est pas utilisée car plus très utile et contraignante à programmer. L' α -réduction sera utilisée pour rechercher une équivalence entre deux termes, on renommera d'ailleurs l'équivalence en α -équivalence.

Une réduction a été rajoutée pour gérer les opérateurs : c'est la δ -réduction. Ce qui nous donne une nouvelle **n**-réduction tels que **n**-réduction = $\beta_v \cup \delta$

Un exemple d'implantation a été codé en Ocaml.

3.2 Machines abstraites

CC Machine : CC vient des termes **Control string** et **Context** qui représente respectivement :

- la partie du λ -calcul que l'on traite
- la partie du λ -calcul que l'on met en attente

Cette séparation permet d'appliquer l'appel par valeur simplement. En effet à chaque fois que l'on a une application on évalue le terme de gauche puis le terme de droit. Le fait de se concentrer sur une sous-expression est possible grâce au contexte qui prends la partie générale de l'expression pour la mettre en attente. Cette machine reprend la sémantique du langage ISWIM.

Pour ne pas perdre la position de la sous-expression que l'on traite dans l'expression générale, on utilise un **trou** qui est représenté par \square dans notre expression.

Basiquement la machine va évaluer l'expression comme ceci :

- Quand on a une application $(M N)$:
 - la machine va évaluer M et garder dans le contexte $(\square N)$.
 - elle va remettre la sous-expression M évalué, soit $M \rightarrow_{cc} V$, dans l'application ce qui donnera $(V N)$
 - elle va évaluer N et garder dans le contexte $(V \square)$
 - elle va remettre la sous-expression N évalué, soit $N \rightarrow_{cc} U$, dans l'application ce qui donnera $(V U)$
 - elle va évaluer $(V U)$
- Quand on a une abstraction $((\lambda X, M) N)$:
 - c'est une application, sachant que $V = (\lambda X, M)$ on a $(V N)$. Si on utilise ce que l'on a dit plus haut on aura $V U$ avec $V = (\lambda X, M)$. C'est une β -réduction, on retombe sur un cas simple.
- Quand on a une opération $(o^n M \dots N)$: c'est le même principe que l'application
 - la machine va évaluer M et garder $(o^n \square \dots N)$.
 - elle va remettre la sous-expression M évalué, soit $M \rightarrow_{cc} V$, dans l'opération ce qui donnera $(o^n V \dots N)$
 - etc jusqu'à avoir $(o^n V \dots U)$ pour pouvoir évaluer l'opération

Les règles définies pour cette machine sont les suivantes :

1. $\langle (M N), E \rangle \mapsto_{cc} \langle M, E[(\square N)] \rangle$ si $M \notin V$
2. $\langle (V_1 N), E \rangle \mapsto_{cc} \langle N, E[(V_1 \square)] \rangle$ si $N \notin V$
3. $\langle ((\lambda X.M) V), E \rangle \mapsto_{cc} \langle M[X \leftarrow V], E \rangle$
4. $\langle V, E[(U \square)] \rangle \mapsto_{cc} \langle (U V), E \rangle$
5. $\langle V, E[(\square N)] \rangle \mapsto_{cc} \langle (V N), E \rangle$
6. $\langle (o^n V_1 \dots V_i M N \dots), E \rangle \mapsto_{cc} \langle M, E[(o^n V_1 \dots V_i \square N \dots)] \rangle$ si $M \notin V$
7. $\langle (o^n b_1 \dots b_n), E \rangle \mapsto_{cc} \langle V, E \rangle$ avec $V = \delta(o^n, b_1 \dots b_n)$
8. $\langle V, E[(o^n V_1 \dots V_i \square N \dots)] \rangle \mapsto_{cc} \langle (o^n V_1 \dots V_i V N \dots), E \rangle$

La machine peut s'arrêter dans 3 états différents :

- on a une **constante** tels que $\langle M, \square \rangle \rightarrow_{cc} \langle b, \square \rangle$;
- on a une **fonction** tels que $\langle M, \square \rangle \rightarrow_{cc} \langle \lambda X.N, \square \rangle$;
- on a un **état inconnu** soit une **erreur**.

Exemple de fonctionnement de la machine CC : Voici un exemple de la machine CC pour l'expression : $((\lambda f. \lambda x. f x) \lambda y. (+ y y)) \ulcorner 1 \urcorner$.

CC : $\langle (((\lambda f. \lambda x. f x) \lambda y. (+ y y)) \ulcorner 1 \urcorner), \square \rangle$	> (3)
> (1)	CC : $\langle ((\lambda y. (+ y y)) x)[x \leftarrow \ulcorner 1 \urcorner], \square \rangle$
CC : $\langle ((\lambda f. \lambda x. f x) \lambda y. (+ y y)), [\square \ulcorner 1 \urcorner] \rangle$	CC : $\langle ((\lambda y. (+ y y)) \ulcorner 1 \urcorner), \square \rangle$
> (3)	> (3)
CC : $\langle (\lambda x. f x)[f \leftarrow \lambda y. (+ y y)], [\square \ulcorner 1 \urcorner] \rangle$	CC : $\langle (+ y y)[y \leftarrow \ulcorner 1 \urcorner], \square \rangle$
CC : $\langle (\lambda x. (\lambda y. (+ y y)) x), [\square \ulcorner 1 \urcorner] \rangle$	CC : $\langle (+ \ulcorner 1 \urcorner \ulcorner 1 \urcorner), \square \rangle$
> (5)	> (7)
CC : $\langle ((\lambda x. (\lambda y. (+ y y)) x) \ulcorner 1 \urcorner), \square \rangle$	CC : $\langle \ulcorner 2 \urcorner, \square \rangle$

SCC Machine : La machine SCC est une simplification de la machine CC. En effet, la machine CC exploite uniquement les informations de la chaîne de contrôle (**Control string**). D'ailleurs le S de SCC est un acronyme de **Simplified**. Du coup on combine certaines règles en exploitant les informations du contexte (**Context**).

Par exemple dans la machine CC, on peut réunir :

$$(5)_{cc} \langle V, E[(\square N)] \rangle \mapsto_{cc} \langle (V N), E \rangle$$

$$(2)_{cc} \langle (V N), E \rangle \mapsto_{cc} \langle N, E[(V \square)] \rangle$$

$$\rightarrow \text{sont combinées dans la règle } (4)_{scc} \langle V, E[(\square N)] \rangle \mapsto_{scc} \langle N, E[(V \square)] \rangle$$

$$(4)_{cc} \langle V, E[(U \square)] \rangle \mapsto_{cc} \langle (U V), E \rangle$$

$$(3)_{cc} \langle ((\lambda X.M) V), E \rangle \mapsto_{cc} \langle M[X \leftarrow V], E \rangle$$

$$\rightarrow \text{sont combinées dans la règle } (3)_{scc} \langle V, E[(\lambda X.M) \square] \rangle \mapsto_{scc} \langle M[X \leftarrow V], E \rangle$$

$$(8)_{cc} \langle V, E[(o^n V_1 \dots V_i \square N \dots)] \rangle \mapsto_{cc} \langle (o^n V_1 \dots V_i V N \dots), E \rangle$$

$$(7)_{cc} \langle (o^n b_1 \dots b_n), E \rangle \mapsto_{cc} \langle V, E \rangle \text{ avec } V = \delta(o^n, b_1 \dots b_n)$$

$$\rightarrow \text{sont combinées dans la règle } (5)_{scc} \langle b, E[(o^n, b_1, \dots b_i, \square)] \rangle \mapsto_{scc} \langle V, E \rangle \text{ avec } \delta(o^n, b_1, \dots b_i, b) = V$$

Le fonctionnement reste le même que la machine CC dans le principe et la façon de fonctionner. Le but de cette machine est d'utiliser toute les informations que l'on peut extraire d'un état de la machine pour rendre le fonctionnement plus rapide.

Les règles qui définissent la machine SCC sont les suivantes :

1. $\langle (M N), E \rangle \mapsto_{scc} \langle M, E[(\square N)] \rangle$
2. $\langle (o^n M N \dots), E \rangle \mapsto_{scc} \langle M, E[(o^n \square N \dots)] \rangle$
3. $\langle V, E[(\lambda X.M) \square] \rangle \mapsto_{scc} \langle M[X \leftarrow V], E \rangle$
4. $\langle V, E[(\square N)] \rangle \mapsto_{scc} \langle N, E[(V \square)] \rangle$
5. $\langle b, E[(o^n, b_1, \dots b_i, \square)] \rangle \mapsto_{scc} \langle V, E \rangle \text{ avec } \delta(o^n, b_1, \dots b_i, b) = V$
6. $\langle V, E[(o^n, V_1, \dots V_i, \square, N L)] \rangle \mapsto_{scc} \langle N, E[(o^n, V_1, \dots V_i, V, \square, L)] \rangle$

De même que pour la machine CC, la machine SCC peut s'arrêter dans 3 états différents :

→ on a une **constante** **b** telle que $\langle M, \square \rangle \mapsto_{scc} \langle b, \square \rangle$;

→ on a une **fonction** telle que $\langle M, \square \rangle \mapsto_{scc} \langle \lambda X.N, \square \rangle$;

→ on a un **état inconnu** soit une **erreur**.

Exemple de fonctionnement de la machine SCC : Voici un exemple de la machine SCC pour l'expression : $((\lambda f.\lambda x.f x) \lambda y.(+ y y)) \ulcorner 1 \urcorner$.

$$SCC : \langle ((\lambda f.\lambda x.f x) \lambda y.(+ y y)) \ulcorner 1 \urcorner, \square \rangle$$

> (1)

$$SCC : \langle ((\lambda f.\lambda x.f x) \lambda y.(+ y y)), [(\square \ulcorner 1 \urcorner)] \rangle$$

> (1)

$$SCC : \langle (\lambda f.\lambda x.f x), [(\square \ulcorner 1 \urcorner), (\square (\lambda y.(+ y y)))] \rangle$$

> (4)

$$SCC : \langle (\lambda y.(+ y y)), [(\square \ulcorner 1 \urcorner), ((\lambda f.\lambda x.f x) \square)] \rangle$$

> (3)

$$SCC : \langle (\lambda x.f x)[f \leftarrow (\lambda y.(+ y y))], [(\square \ulcorner 1 \urcorner)] \rangle$$

$$SCC : \langle (\lambda x.(\lambda y.(+ y y)) x), [(\square \ulcorner 1 \urcorner)] \rangle$$

> (4)

$$SCC : \langle \ulcorner 1 \urcorner, [((\lambda x.(\lambda y.(+ y y)) x) \square)] \rangle$$

> (3)

$$SCC : \langle ((\lambda y.(+ y y)) x)[x \leftarrow \ulcorner 1 \urcorner], \square \rangle$$

$$SCC : \langle ((\lambda y.(+ y y)) \ulcorner 1 \urcorner, \square \rangle$$

> (1)

$$SCC : \langle (\lambda y.(+ y y)), [(\square \ulcorner 1 \urcorner)] \rangle$$

> (4)

$$SCC : \langle \ulcorner 1 \urcorner, [(\lambda y.(+ y y)) \square] \rangle$$

> (3)

$$SCC : \langle (+ y y)[y \leftarrow \ulcorner 1 \urcorner], \square \rangle$$

$$SCC : \langle (+ \ulcorner 1 \urcorner \ulcorner 1 \urcorner), \square \rangle$$

> (2)

$$SCC : \langle \ulcorner 1 \urcorner, (+ \square \ulcorner 1 \urcorner) \rangle$$

> (6)

$$SCC : \langle \ulcorner 1 \urcorner, (+ \ulcorner 1 \urcorner \square) \rangle$$

> (5)

$$SCC : \langle \ulcorner 2 \urcorner, \square \rangle$$

CK Machine Les machines CC et SCC cherche toujours a traiter l'élément le plus à gauche , c'est-à-dire que si l'on a une application on va en créer une intermédiaire dans le contexte avec un trou et traité la partie gauche de cette application etc jusqu'à arriver sur une valeur pour pouvoir "reconstruire", en reprenant l'application intermédiaire. C'est le style **LIFO (Last In, First Out)**. Ce qui fait que les étapes de transition dépendent directement de la forme du 1er élément et non de la structure générale.

Pour palier ce problème, la machine CK ajoute un nouvel élément le **registre de contexte d'évaluation**, nommé κ , qui permet la sauvegarde ce qu'on appelle la **continuation**, c'est-à-dire la suite des instructions qu'il lui reste à exécuter.

La machine CK va donc fonctionner avec une chaîne de contrôle **C (Control string)** comme les machines CC et SCC mais remplace le contexte par la continuation **K**.

On a $\kappa = mt$
 $| \langle fun, V, \kappa \rangle$
 $| \langle arg, N, \kappa \rangle$
 $| \langle opd, \langle V, ..., V, o^n \rangle, \langle N, ... \rangle, \kappa \rangle$

Cette continuation, au-delà de sauvegarder le reste de l'expression à traiter, va garder en mémoire ce que l'on a dedans. Basiquement elle va nous dire si on a une fonction, un argument ou une opération. Cette spécification permet d'enlever les **trous**.

Cette machine va agir en fonction de ce qui est présent dans la continuation.

- Quand on a rien :
 - Soit on a une application $(M N)$, On va traiter M et dire que N est un argument.
 - Soit on a une valeur ce qui signifie la fin du fonctionnement de la machine
- Quand on a un argument M : On aura une valeur V dans la chaîne de contrôle. On va l'évaluer et stocker dans la continuation V qui est une fonction car si M a été stocké comme un argument cela veut dire qu'initialement on avait l'application $(N M)$; N est évalué donne V donc on doit appliquer M à V ce qui veut dire que V est une fonction.
- Quand on a une fonction V : On va appliquer l'argument U qui est dans la chaîne de contrôle a V
- Quand on a une opération : on va traiter successivement chaque élément de l'opération qui sont en attente d'être traité et quand c'est fait on évalue l'opération.

Les règles qui définissent la machine CK sont les suivantes :

1. $\langle (M N), \kappa \rangle \mapsto_{ck} \langle M, \langle arg, N, \kappa \rangle \rangle$
2. $\langle V, \langle fun, (\lambda X.M), \kappa \rangle \rangle \mapsto_{ck} \langle M[X \leftarrow V], \kappa \rangle$
3. $\langle V, \langle arg, N, \kappa \rangle \rangle \mapsto_{ck} \langle N, \langle fun, V, \kappa \rangle \rangle$
4. $\langle \langle o^n M N ... \rangle, \kappa \rangle \mapsto_{ck} \langle M, \langle opd, \langle o^n \rangle, \langle N, ... \rangle, \kappa \rangle \rangle$
5. $\langle b, \langle opd, \langle b_i, ... b_1, o^n \rangle, \langle \rangle, \kappa \rangle \rangle \mapsto_{ck} \langle V, \kappa \rangle$ avec $\delta(o^n, b_1, ... b_i, b) = V$
6. $\langle V, \langle opd, \langle V', ... o^n \rangle, \langle N, L, ... \rangle, \kappa \rangle \rangle \mapsto_{ck} \langle N, \langle opd, \langle V, V', ... o^n \rangle, \langle L, ... \rangle, \kappa \rangle \rangle$

la machine CK peut s'arrêter dans 3 états différents :

- on a une **constante b** telle que $\langle M, mt \rangle \rightarrow_{ck} \langle b, mt \rangle$;
- on a une **fonction** telle que $\langle M, mt \rangle \rightarrow_{ck} \langle \lambda X.N, mt \rangle$;
- on a un **état inconnu** soit une **erreur**.

Voici une partie de exemple de la machine CK pour l'expression : $((\lambda f. \lambda x. f x) \lambda y. (+ y y)) \ulcorner 1 \urcorner$.
L'exemple complet peut être retrouvé dans l'Annexe 6.1.1.

CK : $\langle (((\lambda f. \lambda x. f x) \lambda y. (+ y y)) \ulcorner 1 \urcorner), mt \rangle$...
...	On voit comment l'opération est traitée
On a un exemple d'application sur une abstraction	CK : $\langle (+ \ulcorner 1 \urcorner \ulcorner 1 \urcorner), mt \rangle$
CK : $\langle (\lambda x. (\lambda y. (+ y y)) x), \langle arg, \ulcorner 1 \urcorner, mt \rangle \rangle$	$> (4)$
$> (3)$	CK : $\langle \ulcorner 1 \urcorner, \langle opd, \langle + \rangle, \langle \ulcorner 1 \urcorner \rangle, mt \rangle \rangle$
CK : $\langle \ulcorner 1 \urcorner, \langle fun, (\lambda x. (\lambda y. (+ y y)) x), mt \rangle \rangle$	$> (6)$
$> (2)$	CK : $\langle \ulcorner 1 \urcorner, \langle opd, \langle \ulcorner 1 \urcorner, + \rangle, \langle \rangle, mt \rangle \rangle$
CK : $\langle ((\lambda y. (+ y y)) x)[x \leftarrow \ulcorner 1 \urcorner], mt \rangle$	$> (5)$
CK : $\langle ((\lambda y. (+ y y)) \ulcorner 1 \urcorner), mt \rangle$	CK : $\langle \ulcorner 2 \urcorner, mt \rangle$

CEK Machine Pour toutes les machines vues jusqu'à présent la β -réduction est appliquée immédiatement. Cela coûte cher surtout quand l'expression devient grande. De plus, si notre substitution n'est pas une variable elle est traitée avant d'être appliquée.

Il est plus intéressant d'appliquer les substitutions quand on en a vraiment la nécessité. Pour cela, la machine CEK ajoute les fermetures et un environnement ε qui va stocker les substitutions à effectuer. Cette environnement est une fonction qui pour une variable va retourner une expression.

On a alors :

ε = une fonction $\{\langle X, c \rangle, \dots\}$ $c = \{\langle M, \varepsilon \rangle \mid FV(M) \subset dom(\varepsilon)\}$ $v = \{\langle V, \varepsilon \rangle \mid \langle V, \varepsilon \rangle \in c\}$

$\varepsilon[X \leftarrow c] = \{\langle X, c \rangle\} \cup \{\langle Y, c' \rangle \mid \langle Y, c' \rangle \in \varepsilon \text{ et } Y \neq X\}$

κ est renommé $\bar{\kappa}$ et défini par :

$\bar{\kappa} = mt$

$\mid \langle fun, v, \bar{\kappa} \rangle$

$\mid \langle arg, c, \bar{\kappa} \rangle$

$\mid \langle opd, \langle v, \dots, v, o^n \rangle, \langle c, \dots \rangle, \bar{\kappa} \rangle$

Les règles qui définissent la machine CEK sont les suivantes :

1. $\langle \langle M N \rangle, \varepsilon \rangle, \bar{\kappa} \rangle \mapsto_{cek} \langle \langle M, \varepsilon \rangle, \langle arg, \langle N, \varepsilon \rangle, \bar{\kappa} \rangle \rangle$
2. $\langle \langle X, \varepsilon \rangle, \bar{\kappa} \rangle \mapsto_{cek} \langle c, \bar{\kappa} \rangle$ avec $\varepsilon(X) = c$
3. $\langle \langle V, \varepsilon \rangle, \langle fun, \langle \langle \lambda X 1.M \rangle, \varepsilon' \rangle, \bar{\kappa} \rangle \rangle \mapsto_{cek} \langle \langle M, \varepsilon'[X 1 \leftarrow \langle V, \varepsilon \rangle] \rangle, \bar{\kappa} \rangle$ si $V \notin X$
4. $\langle \langle V, \varepsilon \rangle, \langle arg, \langle N, \varepsilon' \rangle, \bar{\kappa} \rangle \rangle \mapsto_{cek} \langle \langle N, \varepsilon' \rangle, \langle fun, \langle V, \varepsilon \rangle, \bar{\kappa} \rangle \rangle$ si $V \notin X$
5. $\langle \langle \langle o^n M N \dots \rangle, \varepsilon \rangle, \bar{\kappa} \rangle \mapsto_{cek} \langle \langle M, \varepsilon \rangle, \langle opd, \langle o^n \rangle, \langle \langle N, \varepsilon \rangle, \dots \rangle, \bar{\kappa} \rangle \rangle$
6. $\langle \langle b, \varepsilon \rangle, \langle opd, \langle \langle b_i, \varepsilon_i \rangle, \dots \langle b_1, \varepsilon_1 \rangle, o^n \rangle, \langle \rangle, \bar{\kappa} \rangle \rangle \mapsto_{cek} \langle \langle V, \emptyset \rangle, \bar{\kappa} \rangle$ avec $\delta(o^n, b_1, \dots b_i, b) = V$
7. $\langle \langle V, \varepsilon \rangle, \langle opd, \langle v', \dots o^n \rangle, \langle \langle N, \varepsilon' \rangle, c, \dots \rangle, \bar{\kappa} \rangle \rangle \mapsto_{cek} \langle \langle N, \varepsilon' \rangle, \langle opd, \langle \langle V, \varepsilon \rangle, v', \dots o^n \rangle, \langle c, \dots \rangle, \bar{\kappa} \rangle \rangle$ si $V \notin X$

la machine CEK peut s'arrêter dans 3 états différents :

→ on a une **constante** **b** telle que $\langle \langle M, \emptyset \rangle, mt \rangle \rightarrow_{cek} \langle \langle b, \varepsilon \rangle, mt \rangle$;

→ on a une **fonction** telle que $\langle \langle M, \emptyset \rangle, mt \rangle \rightarrow_{cek} \langle \langle \lambda X.N, \varepsilon \rangle, mt \rangle$;

→ on a un **état inconnu** soit une **erreur**.

On va voir une partie d'un exemple pour voir le changement avec la machine CK. On se concentre sur l'utilisation de l'environnement.

CEK machine : $\langle \langle \langle \langle \langle \lambda f. \lambda x. f x \rangle \lambda y. (+ y y) \rangle \uparrow 1 \uparrow \rangle, \emptyset \rangle, mt \rangle$

...

On a deux parties qui appliquent une abstraction.

CEK : $\langle \langle \langle \lambda y. (+ y y) \rangle, \emptyset \rangle, \langle fun, \langle \langle \lambda f. \lambda x. f x \rangle, \emptyset \rangle, \langle arg, \langle \uparrow 1 \uparrow, \emptyset \rangle, mt \rangle \rangle \rangle$

> (3)

CEK : $\langle \langle \langle \lambda x. f x \rangle, \emptyset [f \leftarrow \langle \langle \lambda y. (+ y y) \rangle, \emptyset \rangle] \rangle, \langle arg, \langle \uparrow 1 \uparrow, \emptyset \rangle, mt \rangle \rangle$

CEK : $\langle \langle \langle \lambda x. f x \rangle, \{ \langle f, \langle \langle \lambda y. (+ y y) \rangle, \emptyset \rangle \} \rangle, \langle arg, \langle \uparrow 1 \uparrow, \emptyset \rangle, mt \rangle \rangle$

> (4)

CEK : $\langle \langle \uparrow 1 \uparrow, \emptyset \rangle, \langle fun, \langle \langle \lambda x. f x \rangle, \{ \langle f, \langle \langle \lambda y. (+ y y) \rangle, \emptyset \rangle \} \rangle, mt \rangle \rangle$

> (3)

CEK : $\langle \langle \langle f x \rangle, \{ \langle f, \langle \langle \lambda y. (+ y y) \rangle, \emptyset \rangle \} [x \leftarrow \langle \uparrow 1 \uparrow, \emptyset \rangle] \rangle, mt \rangle$

CEK : $\langle \langle \langle f x \rangle, \{ \langle f, \langle \langle \lambda y. (+ y y) \rangle, \emptyset \rangle \}, \langle x, \langle \uparrow 1 \uparrow, \emptyset \rangle \rangle \rangle, mt \rangle$

> (1)

...

On voit comment la substitution s'applique sur cette partie.

CEK : $\langle \langle \langle f, \{ \langle f, \langle \langle \lambda y. (+ y y) \rangle, \emptyset \rangle \}, \langle x, \langle \uparrow 1 \uparrow, \emptyset \rangle \rangle \rangle, \langle arg, \langle x, \{ \langle f, \langle \langle \lambda y. (+ y y) \rangle, \emptyset \rangle \}, \langle x, \langle \uparrow 1 \uparrow, \emptyset \rangle \rangle \rangle \rangle, mt \rangle \rangle$

> (2)

CEK : $\langle \langle \langle \lambda y. (+ y y) \rangle, \emptyset \rangle, \langle arg, \langle x, \{ \langle f, \langle \langle \lambda y. (+ y y) \rangle, \emptyset \rangle \}, \langle x, \langle \uparrow 1 \uparrow, \emptyset \rangle \rangle \rangle \rangle, mt \rangle \rangle$

...

CEK : $\langle \langle \uparrow 2 \uparrow, \emptyset \rangle, mt \rangle$

L'exemple complet peut être retrouvé dans l'Annexe 6.1.2.

SECD Machine La différence entre la machine CEK et SECD se situe dans la façon dont le contexte est sauvegardé pendant que les sous-expressions sont évaluées.

En effet, dans la machine SECD le contexte est créé par un appel de fonction, quand tout est stocké dans \widehat{D} pour laisser un espace de travail. Par contre pour la machine CEK, le contexte est créé quand on évalue une application ou un argument indépendamment de la complexité de celui-ci.

Dans les langages tels que Java, Pascal ou encore C la façon de faire de la machine SECD est plus naturelle. Par contre dans les langages λ -calculs, Scheme ou encore ML c'est la façon de faire de la machine CEK qui est la plus naturelle.

La machine SECD est composée d'une pile pour les valeurs (\widehat{S}), d'un environnement ($\widehat{\varepsilon}$) pour lier les variables X à une valeur \widehat{V} , d'une chaîne de contrôle (\widehat{C}) et d'un dépôt (\widehat{D}). Les différentes définitions de ces éléments sont les suivantes :

$$\begin{aligned}\widehat{S} &= \epsilon \\ &| \widehat{V} \widehat{S} \\ \widehat{\varepsilon} &= \text{une fonction } \{\langle X, \widehat{V} \rangle, \dots\} \\ \widehat{C} &= \epsilon \\ &| b \widehat{C} \\ &| X \widehat{C} \\ &| ap \widehat{C} \\ &| prim_{o^n} \widehat{C} \\ &| \langle X, \widehat{C} \rangle \widehat{C} \\ \widehat{D} &= \epsilon \\ &| \langle \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D} \rangle \\ \widehat{V} &= b \\ &| \langle \langle X, \widehat{C} \rangle, \widehat{\varepsilon} \rangle\end{aligned}$$

Une autre spécificité de la machine SECD vient de sa propre grammaire. En effet la machine SECD convertit la λ -expression par son propre langage. Le langage fonctionne avec des éléments simple comme des constantes, des variables et des fonctions et des commandes qui permettent de savoir ce que doit faire la machine. Voici les règles de conversion :

$$\begin{aligned}[b]_{secd} &= b \\ [X]_{secd} &= X \\ [(M_1 M_2)]_{secd} &= [M_1]_{secd} [M_2]_{secd} \text{ ap} \\ [(o^n M_1 \dots M_n)]_{secd} &= [M_1]_{secd} \dots [M_n]_{secd} \text{ prim}_{o^n} \\ [(\lambda X.M)]_{secd} &= \langle X, [M]_{secd} \rangle\end{aligned}$$

Cette machine doit être la plus simple à comprendre dans son fonctionnement :

- Si elle a une constante, elle la stocke dans la pile. On peut dire qu'elle la garde en attente de l'utiliser pour une commande.
- Si elle a une variable, elle prend sa substitution dans l'environnement et la stocke dans la pile comme une constante
- Si elle a une abstraction, elle va créer ce qu'on appelle une fermeture. C'est-à-dire que l'on va lier l'abstraction avec l'environnement. Ce processus va permettre de mettre en attente l'évaluation de l'expression présent dans l'abstraction. elle stocke la fermeture dans la pile comme une constante.
- Si elle a une commande *ap*, on effectue une application sur les deux éléments de tête de la pile. Cette application va mettre en attente l'expression principale pour se concentrer sur une sous-expression. Pour cela, la machine va faire une sauvegarde d'elle-même avant et la stocker dans le dépôt.
- Si elle a une commande *prim_{oⁿ}*, on effectue l'opération sur les n premiers éléments de la pile.
- Si elle a rien :
 - elle a une sauvegarde dans le dépôt, elle va reprendre cette sauvegarde.
 - elle n'a pas de sauvegarde, elle a finit son travail.

Les règles qui définissent la machine SECD sont les suivantes :

1. $\langle \widehat{S}, \widehat{\varepsilon}, b \widehat{C}, \widehat{D} \rangle \mapsto_{secd} \langle b \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D} \rangle$
2. $\langle \widehat{S}, \widehat{\varepsilon}, X \widehat{C}, \widehat{D} \rangle \mapsto_{secd} \langle \widehat{V} \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D} \rangle$ où $\widehat{V} = \varepsilon(X)$
3. $\langle \widehat{S}, \widehat{\varepsilon}, \langle X, C' \rangle \widehat{C}, \widehat{D} \rangle \mapsto_{secd} \langle \langle \langle X, C' \rangle, \varepsilon \rangle \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D} \rangle$
4. $\langle \widehat{V} \langle \langle X, C' \rangle, \varepsilon' \rangle \widehat{S}, \widehat{\varepsilon}, ap \widehat{C}, \widehat{D} \rangle \mapsto_{secd} \langle \varepsilon, \varepsilon' [X \leftarrow \widehat{V}], C', \langle \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D} \rangle \rangle$
5. $\langle \widehat{V} \widehat{S}, \widehat{\varepsilon}, \emptyset, \langle \widehat{S}', \widehat{\varepsilon}', \widehat{C}', \widehat{D}' \rangle \rangle \mapsto_{secd} \langle \widehat{V} \widehat{S}', \widehat{\varepsilon}', \widehat{C}', \widehat{D}' \rangle$
6. $\langle b_1 \dots b_n \widehat{S}, \widehat{\varepsilon}, prim_{o^n} \widehat{C}, \widehat{D} \rangle \mapsto_{secd} \langle \widehat{V} \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D} \rangle$ où $\widehat{V} = \delta(o^n, b_1, \dots, b_n)$

la machine SECD peut s'arrêter dans 3 états différents :

- on a une **constante** **b** telle que $\langle \varepsilon, \emptyset, [M]_{secd}, \varepsilon \rangle \rightarrow_{secd} \langle b, \widehat{\varepsilon}, \varepsilon, \varepsilon \rangle$;
- on a une **fonction** telle que $\langle \varepsilon, \emptyset, [M]_{secd}, \varepsilon \rangle \rightarrow_{secd} \langle \langle \langle X, \widehat{C} \rangle, \widehat{\varepsilon}' \rangle, \widehat{\varepsilon}, \varepsilon, \varepsilon \rangle$;
- on a un **état inconnu** soit une **erreur**.

Voyons sur un exemple les avantages de cette machine. Un exemple étant assez long, on va s'intéresser à certaines parties. L'exemple complet est disponible dans l'Annexe 6.1.3.

On va tester la machine sur la λ -expression suivante : $((\lambda f. \lambda x. f \ x) \ \lambda y. (+ \ y \ y)) \ulcorner 1 \urcorner$.

On passe la conversion, utile mais pas intéressante.

Conversion : $[(((\lambda f. \lambda x. f \ x) \ \lambda y. (+ \ y \ y)) \ulcorner 1 \urcorner)]_{secd}$

...

Conversion : $\langle f, \langle x, f \ x \ ap \rangle \rangle \langle y, y \ y \ prim_+ \rangle \ ap \ulcorner 1 \urcorner \ ap$

SECD Machine : $\langle \varepsilon, \emptyset, \langle f, \langle x, f \ x \ ap \rangle \rangle \langle y, y \ y \ prim_+ \rangle \ ap \ulcorner 1 \urcorner \ ap, \varepsilon \rangle$

...

On voit comment l'application est traitée

SECD : $\langle \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \langle \langle f, \langle x, f \ x \ ap \rangle \rangle, \emptyset \rangle, \emptyset, ap \ulcorner 1 \urcorner \ ap, \varepsilon \rangle$

> (4)

SECD : $\langle \varepsilon, \emptyset [f \leftarrow \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle], \langle x, f \ x \ ap \rangle, \langle \varepsilon, \emptyset, \ulcorner 1 \urcorner \ ap, \varepsilon \rangle \rangle$

SECD : $\langle \varepsilon, \{ \langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \} \rangle, \langle x, f \ x \ ap \rangle, \langle \varepsilon, \emptyset, \ulcorner 1 \urcorner \ ap, \varepsilon \rangle \rangle$

> (3)

On voit l'intérêt de la sauvegarde.

SECD : $\langle \langle \langle x, f \ x \ ap \rangle, \{ \langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \} \rangle, \{ f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \}, \emptyset, \langle \varepsilon, \emptyset, \ulcorner 1 \urcorner \ ap, \varepsilon \rangle \rangle$

> (5)

SECD : $\langle \langle \langle x, f \ x \ ap \rangle, \{ \langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \} \rangle, \emptyset, \ulcorner 1 \urcorner \ ap, \varepsilon \rangle$

...

On voit comment est gérée l'opération

SECD : $\langle \ulcorner 1 \urcorner \ulcorner 1 \urcorner, \{ \langle y, \ulcorner 1 \urcorner \rangle \}, prim_+, \langle \varepsilon, \{ \langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \}, \langle x, \ulcorner 1 \urcorner \rangle \}, \emptyset, \langle \varepsilon, \emptyset, \varepsilon, \varepsilon \rangle \rangle$

> (6)

SECD : $\langle \ulcorner 2 \urcorner, \{ \langle y, \ulcorner 1 \urcorner \rangle \}, \emptyset, \langle \varepsilon, \{ \langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \}, \langle x, \ulcorner 1 \urcorner \rangle \}, \emptyset, \langle \varepsilon, \emptyset, \varepsilon, \varepsilon \rangle \rangle$

> (5)

SECD : $\langle \ulcorner 2 \urcorner, \{ \langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \}, \langle x, \ulcorner 1 \urcorner \rangle \}, \emptyset, \langle \varepsilon, \emptyset, \varepsilon, \varepsilon \rangle \rangle$

> (5)

SECD : $\langle \ulcorner 2 \urcorner, \emptyset, \varepsilon, \varepsilon \rangle$

3.3 Programmation réactive

Le cas du Réactive ML Modèle de programmation \rightarrow concurrence coopérative

L'analyse est découpé en 2 sous analyse :

- **statique** : système de type et d'effet
- **réactive** : détecter les erreurs de concurrence

Ordonnancement coopératif* \rightarrow *chaque processus va régulièrement "laisser la main aux autres"*

Ordonnancement préemptif \rightarrow *le système va "donner un temps de parole" à chacun*

Points fort :

- *implémentation séquentielle efficace*
- *pas de problème de parallélisme*

Points faible :

- *responsabilité de la réactivité au dev*

Le modèle réactif synchrone définit une notion de temps logique qui est une succession d'instant.

Un programme est réactif si son exécution fait progresser les instants logique.

Exemple

1. let process clock timer s =
2. let time = ref(Unix.gettimeofday()) in
3. loop
4. let time' = Unix.gettimeofday() in
5. if time' -. !time >= timer
6. then(emit s(); time := time')
7. end

Le problème ici est que le contenu de la boucle peut s'effectuer instantanément alors qu'il faudrait attendre un instant logique pour l'exécuter. Par conséquent, on doit ajouter une **pause entre les lignes 6 et 7**.

Une **condition suffisante** pour qu'un **processus récursif soit réactif** est qu'il y ait **toujours un instant logique entre l'instanciation du processus et l'appel récursif**.

Le reste de l'article rentre beaucoup plus dans les détails avec l'analyse syntaxique qui prend en compte la réactivité du langage.

4 Langage fonctionnel réactif

Le but est de réutiliser une des machines étudiées et de la rendre réactive. Les premières machines sont trop simple pour être utilisées. Restent la machine CEK et la machine SECD. Cependant il y a une contrainte qui va réussir à les départager. En effet, il nous faut une machine qui fait un appel par valeur, c'est-à-dire que les paramètres seront évalués avant la fonction. La machine SECD remplit ce critère donc c'est celle que nous utiliserons comme base.

Mes encadrants m'ont donné des ajouts à faire par petites parties afin de structurer mon avancée. Je vais donc redonner les ajouts pour chaque partie ainsi que l'explication de leurs implantations.

4.1 Machine réactive pure

4.1.1 Description informelle du langage

Avant tout développement sur l'implantation il faut pouvoir bien cerner le principe de la réactivité. Le but est de créer un "dialogue" entre plusieurs parties de la machine.

En arrondissant les angles, on peut voir ça comme une discussion. Je m'explique, la discussion c'est notre machine, on a une personne que l'on va nommer Monsieur X qui représente notre programme principal. On peut dire qu'il initie la discussion. Monsieur X va introduire des personnes dans la discussion, ces personnes sont nos parties de programme indépendantes. Monsieur X va commencer à parler puis laissera la parole quand il aura fini de parler ou ne saura pas quoi répondre et ainsi de suite. Pendant qu'une personne parle, il peut donner une information qui sera utilisée par un autre et donc qui se remettra dans la discussion. Ces informations seront représentées par des signaux. Quand tout le monde a fini de parler, c'est la fin de la discussion ce qui représente la fin de la machine.

Un point qu'il faut bien comprendre est que l'absence d'une information (d'un signal) est aussi importante que sa présence. Un exemple naïf, si vous demandez à quelqu'un s'il dort une réponse donne une information aussi importante qu'une absence de réponse.

Un autre point délicat est la fin d'un instant logique. Pour savoir si on nous répond ou pas, on attend un instant et on en déduit que l'on ne nous répondra pas. Pour la machine c'est pareil. L'instant courant est lorsque tout le monde discute, quand plus personne ne parle c'est la fin de l'instant courant. On passe à l'instant suivant et ceux qui attendaient une information spécifique vont exhiber un autre argument et la discussion recommence jusqu'à ce que plus personne n'ait d'argument.

Le principe étant expliqué, nous pouvons rentrer dans les détails. L'idée est d'avoir plusieurs machine SECD qui se parlent donc il faut créer une structure plus large qui peut les stocker. On part cependant d'une seule chaîne de contrôle donc il faut pouvoir créer ces "mini-machines". Ces "mini-machines" sont ce que l'ont va appeler des **threads**.

Un Thread

Qu'est-ce qu'un thread ? Un thread est l'équivalent d'un processus, il a sa propre pile d'exécution mais peut récupérer des informations sur une mémoire partagée. En simplifiant, si on fait un lien avec la machine SECD, on peut dire que chaque thread est une machine SECD et que toutes ces machines SECD vont communiquer entre elles.

Quelle forme doit avoir un thread ? Un thread est, comme dit plus haut, une machine SECD en soi donc elle va prendre cette forme c'est-à-dire que l'on va avoir $T = \langle S, E, C, D \rangle$. Pour l'instant c'est tout ce qui nous est nécessaire dans le thread.

Comment les stocker ? Pour les stocker, on va se demander si l'on veut un ordre ou pas dans notre stockage. Il faut noter un point important pour faire une machine fonctionnelle, il vaut mieux avoir une machine déterministe.

Machine déterministe : Une machine est déterministe si pour une entrée donnée, on a une seule sortie possible et un unique "chemin" possible dans la machine, c'est-à-dire une unique suite de transitions possibles.

Pour éviter de la rendre non déterministe, on va opter pour une file d'attente TL tels que $\forall tl \in TL : tl = T$.

Un Signal

Qu'est-ce qu'un signal ? Un signal est une information que l'on transmet entre chaque thread. Pour les bases, tout ce qui va nous intéresser est la présence ou l'absence d'un signal. C'est grâce à cela qu'un thread va pouvoir agir en conséquence d'un autre.

Quelle forme doit avoir un thread ? Dans un premier temps un signal sera constitué de deux informations :

1. Est-il initialisé ?
2. Est-il émis ?

La forme d'un signal dans notre machine est développée dans la partie **L'initialisation d'un signal**.

Comment les stocker ? Comme pour les threads, on doit se poser la question de la structure que l'on veut mais aussi de si l'on veut un ordre ou non. Plusieurs possibilités ont été explorées et sont directement liées ,là aussi, on développe ce point dans la partie **L'initialisation d'un signal**. Mais dans tous les cas on va devoir créer une liste qui va stocker au moins le fait qu'il est émis, il faudra donc forcément un élément que l'on va nommer *SI* qui sera une liste prévue pour stocker les signaux.

Le Spawn

Comment créer un thread ? Pour créer ce que l'on va appeler **des threads**, on va devoir prendre dans la chaîne de contrôle une partie de celle-ci. Pour cela on va requérir à la commande *Spawn*. Elle va nous servir à délimiter la partie à prendre. Deux versions de cette commande existent, la première peut être retrouvé dans la plupart des versions de la machine et la seconde a été proposée par un de mes encadrants et elle est présente dans la dernière version.

1. *bspawn C espawn* : la machine prend *Spawn C* et le convertit en deux délimiteurs qui servent à entourer la partie de la chaîne de contrôle que l'on veut prendre. Le gros défaut de cette version bien qu'utilisable est sa forme. On crée une structure que doit reconnaître la machine alors que la machine SECD attend soit des éléments simples soit une commande.
2. $\langle X, C \rangle$ *spawn* : ici la machine convertit le *Spawn C* en une abstraction et une commande *spawn*. L'abstraction permet d'encapsuler la partie de la chaîne de contrôle que l'on veut. Grâce à cela la machine n'essaie pas de l'évaluer. *spawn* est juste une commande, donc plus logique dans le fonctionnement de la machine. Elle permet de comprendre que la fermeture présent dans la pile est ce que l'on veut pour créer notre thread.

L'initialisation d'un signal

Comment créer un signal ? Pour créer un signal, on va devoir utiliser une commande *Signal s in t* avec un signal *s* et une expression *t*. Cette commande a été modifiée dans les dernières versions. Voici une énumération du travail fait autour de cette commande.

1. *Signal s in t* : l'identifiant du signal est donné et va être initialisé seulement pour l'expression *t*. Cette version est la plus complexe car il faut stocker énormément d'informations. En effet, il faut stocker l'initialisation du signal pour le thread courant mais aussi pour une partie précise de la chaîne de contrôle du thread courant. Pour cela, deux versions ont été créées :
 - (a) On utilise le principe de la sauvegarde. Simplement expliqué, quand la machine SECD applique une valeur sur une abstraction, elle fait une sauvegarde d'elle-même pour mettre en pause le travail sur la partie principale et se mettre à travailler sur une partie secondaire. Si on reprend le même principe, on peut stocker l'initialisation du signal dans l'environnement après l'avoir sauvegardé. L'élément *SI* nommé plus haut sera donc de la forme $SI = \{s, \dots\}$ telle que *s* représente un signal émis.

Exemple : Si on prend $\langle s, C' \rangle$ la version convertie de *Signal s in t* on a :

$$\langle S, E, \langle s, C' \rangle C, D, SI \rangle \longrightarrow \langle \emptyset, \emptyset [init \leftarrow s], C', \langle S, E, C, D \rangle, SI \rangle$$

Le problème de cette version est qu'il provoque un stockage double des signaux dans la machine à cause des commandes à venir. En effet, on va garder l'initialisation dans l'environnement et l'émission dans l'élément *SI*. Ce n'est pas optimal.

- (b) On utilise la sauvegarde mais différemment. En effet on va faire une sauvegarde comme pour la première version cependant on va pas mettre l'initialisation dans l'environnement mais directement là où l'on voudra stocker plus tard via un booléen. On va créer une variante de la sauvegarde de la machine SECD ,dans le dépôt, de la forme $\langle s, \langle S, E, C, D \rangle \rangle$. Ce qui va nous permettre de savoir quand est-ce que l'on sort de l'expression pour laquelle on avait initialisé le signal. La structure *SI* ici prend une autre forme, on a $SI = \{s, init, emit\}$ telle que un signal *s*, un booléen représentant l'initialisation *init* et un booléen représentant l'émission du signal *emit*.

Exemple : Si on prend $\langle s, C' \rangle$ la version convertie de *Signal s in t* on a :

$$\langle S, E, \langle s, C' \rangle C, D, SI \rangle \longrightarrow \langle \emptyset, \emptyset, C', \langle s, \langle S, E, C, D \rangle \rangle, SI \langle s, true, false \rangle \rangle$$

Cette version est assez efficace mais reste lourde dans la machine niveau du stockage et la façon dont cette commande est convertie crée le même problème que pour la première version du *Spawn* : on est pas censé avoir de structure à traiter dans la machine.

2. *Init s* : l'identifiant du signal est donné et va être initialisé pour tous les éléments de la machine. Cette version va drastiquement simplifier le fonctionnement de la machine car on a plus besoin de le limiter l'action de l'initialisation à une partie de la chaîne de contrôle ni à un thread en particulier. Simplement, on va mettre le signal dans le stockage des signaux. Ce fait sera nécessaire et suffisant pour savoir qu'il est initialisé. La structure de *SI* va donc être de la forme $\langle s, emit \rangle$ telle qu'un signal *s* et un booléen représentant l'émission du signal *emit*.

Exemple : Si on prend *s Init* la version convertie de *Signal s in t* on a :

$$\langle s, S, E, Init, C, D, SI \rangle \longrightarrow \langle S, E, C, D, SI \langle s, false \rangle \rangle$$

3. *Init* : l'identifiant du signal sera retourné par la machine, l'utilisateur ne lui donne plus l'identifiant. On créera l'identifiant en incrémentant le dernier identifiant de signal de un. Le reste est identique au fonctionnement du *Init s*. Cette possibilité aura la même structure pour *SI* que la précédente à l'exception que l'on va rajouter un ordre car il faudra créer des identifiants donc on va toujours mettre en fin de la liste afin de pouvoir créer facilement un nouveau identifiant.

Exemple : Si on prends *Init* la version convertie de *Signal s in t* on a :

$$\langle S, E, Init, C, D, \emptyset \rangle \longrightarrow \langle 0, S, E, C, D, SI \langle 0, false \rangle \rangle$$

ou

$$\langle S, E, Init, C, D, SI \langle s, emit \rangle \rangle \longrightarrow \langle (s+1), S, E, C, D, SI \langle s, emit \rangle \langle (s+1), false \rangle \rangle$$

La présence d'un signal

Comment savoir si un signal est présent ? *Un signal est présent s'il est émis.*

C'est ici que la notion de temps logique va nous être utile. Via l'initialisation d'un signal on sait que l'on a un booléen pour savoir si un signal est émis donc le cas émis est simple mais si il ne l'est pas ?

Un signal est absent si il n'est pas émis durant l'instant logique.

On a besoin de déterminer la fin d'un instant logique. Les différents threads vont s'exécuter à la chaîne donc la fin d'un instant sera lorsqu'aucun thread ne s'exécute. Cependant certains vont tomber dans ce cas ou ils devront attendre l'absence d'un signal. Cette contrainte prise en compte, on redéfinit la fin d'un instant logique comme la fin ou le blocage de tous les threads de la machine.

Dans notre machine on va devoir différencier les threads en attentes de leurs tour et ceux en attente de l'émission d'un signal. Deux possibilités ont été étudiées :

1. On scinde la file d'attente de threads en deux avec d'un côté les threads en attente de leurs tour et de l'autre ceux en attente de l'émission d'un signal. On aurait $TL = \langle W, ST \rangle$ telle que :
 - *W* une file des threads qui attendent leurs tour telle que $\forall w \in W : w = T$
 - *ST* une liste des threads qui attendent un signal telle que $\forall st \in ST : st = \langle s, T \rangle$ avec un signal *s*

Le problème de cette possibilité est la nécessité de stocker quelle signal attend un thread pour chaque threads de *ST*. La deuxième possibilité résout ce problème.

2. On ne touche pas *TL* mais on va stocker les threads bloqués par un signal directement dans les informations de ce signal. Ce qui revient à modifier *SI* telle que $SI = \langle s, \langle emit, ST \rangle \rangle$ avec un signal *s*, le booléen représentant l'émission *emit* et une liste de threads *ST* qui attendent l'émission de *s*.

La différenciation des threads faite, on peut tester la présence d'un signal. La commande a eu deux versions :

1. $\langle s, C', C'' \rangle$ avec un signal *s* et deux expressions *C'* et *C''*. Si vous avez bien suivi jusque là, vous aurez remarqué que les premières versions ont toujours le même problème récurrent : c'est une structure que l'on veut mettre dans une machine qui traite que des éléments simples et des commandes. Cette forme est donc utilisable mais n'est pas dans la logique de la machine de base.
2. $s \langle \langle X, C' \rangle \langle \langle X, C'' \rangle present \rangle$ avec un signal *s*. Cette version paraît plus compliquée, je m'en vais donc vous l'expliquer. On a un signal jusque là rien de bien étonnant. Après on a deux abstractions, elles vont nous servir de protection pour nos deux possibilités de notre présence car elle empêche à la machine d'essayer de l'évaluer. On a utilisé cette même astuce pour la commande *Spawn* plus haut. Pour finir il y a le mot *present* qui sert de commande pour comprendre que l'on veut faire un test.

Récapitulatif : Si on reprends le tout, pour faire notre test de présence nous avons défini :

- la forme de la commande : $s \langle \langle X, C' \rangle \langle \langle X, C'' \rangle \text{ present avec un signal } s ;$
- la présence d'un signal : *un signal est présent si il est émis dans l'instant courant ;*
- l'absence d'un signal : *un signal est absent si il n'est pas émis durant tout l'instant courant ;*
- la fin de l'instant courant : *l'instant courant est fini quand plus aucun threads ne peut plus effectuer d'instruction ;*
- une structure contenant les threads bloqués : $SI = \langle s, \langle \text{emit}, ST \rangle \rangle$ avec un signal s , le booléen représentant l'émission *emit* et une liste de threads ST qui attendent s .

Émettre un signal Cette commande est sûrement la plus simple à mettre en place car tout a été fait en amont via les trois commandes précédentes. La forme de la commande *emit* n'a pas beaucoup bouger, seulement deux versions existent :

1. emit_s Une forme inspirée de prim_{on} . Cependant la grosse erreur est quand mettant un signal s en indice de la commande *emit* on a l'impression qu'il y a une commande *emit* par signal s or l'émission est indépendant du signal qu'il émet.
2. $s \text{ emit}$ avec un signal s . Celle-ci est privée de toute ambiguïté et s'intègre bien à la machine.

Le faite d'émettre va impliqué deux choses dans notre machine :

1. Notre booléen représentant l'émission présent dans les informations d'un signal sera mis à vraie
2. Tous les threads présent dans le tuple du signal seront mis dans la file d'attente W

4.1.2 Sémantique de la machine abstraite

Maintenant que l'on a défini les nouvelles commandes et ce qu'elles ont besoin pour fonctionner il est temps de voir à quoi ressemble notre machine avec ces ajouts. On a gardé la base de la machine SECD en changeant les noms, plus précisément en enlevant les accents circonflexes et en changeant le nom de l'environnement par E afin d'alléger l'écriture.

Soit $\langle T, TL, SI \rangle$ **avec :**

$TL =$ **une file de threads telle que** : $\forall tl \in TL \mid tl = T$ avec :

$T = \langle S, E, C, D \rangle$ **le thread courant avec :**

$V = b$

| $\langle \langle X, C' \rangle E \rangle$

| *signal*

$S = \emptyset$

| $V S$

$E = \{ \dots, \langle X, V \rangle, \dots \}$

$C = \epsilon$

| $b C$ (une constante)

| $X C$ (une variable)

| *signal* C (un signal)

| $\langle X, C' \rangle C$ (une abstraction)

| *ap* C (une application)

| $\text{prim}_{on} C$ (un opérateur)

| *spawn* C (créateur d'un nouveau thread)

| *present* C (teste la présence d'un signal)

| *init* C (initialise un signal)

| *emit* C (émet un signal)

$D = \emptyset$

| $\langle S, E, C, D \rangle$ (une sauvegarde liée à une application)

$SI =$ **une liste de signaux telle que** : $\forall si \in SI \mid si = \langle \text{signal}, \langle \text{emit}, ST \rangle \rangle$ avec :

- un booléen représentant l'émission de ce signal : *emit*
- la liste des threads bloqués par ce signal : ST avec $\forall st \in ST : st = T$

On va définir une règle pour simplifier les règles futurs :

$$\frac{T \rightarrow T'}{\langle T, TL, SI \rangle \rightarrow \langle T', TL, SI \rangle}$$

Une suite de fonctions ont été écrites pour simplifier la lecture des règles. Les voici :

$\iota(SI)$ une fonction qui prend l'identifiant du dernier signal créé, l'incrmente pour en créer un nouveau et retourne l'identifiant du signal créé avec la liste mise à jour.

Exemple :

Si on initialise pour la première fois alors $\iota(\{\}) = (0, \{\langle 0, \langle false, \{\} \rangle\})$

Sinon $\iota(\{\dots, \langle s, data \rangle\}) = (s + 1, \{\dots, \langle s, data \rangle, \langle s + 1, \langle false, \{\} \rangle\})$ avec $data = \langle emit, ST \rangle$

$SI(s)$ une fonction qui retourne le 2nd élément du couple $\langle s, data \rangle$ avec $data = \langle emit, ST \rangle$.

Exemple : $SI(s) = \langle emit, ST \rangle$

$\tau(SI)$ une fonction qui prend tous les éléments bloqués et les retourne en prenant en compte que le signal n'est pas émis, met à faux toutes les émissions et retourne le couple $\langle TL, SI \rangle$ avec une liste de threads TL et SI la liste des signaux modifiés.

Exemple : $\tau(SI) = \forall si \in SI :$

- $\langle s, \langle true, \{\} \rangle \rightarrow \langle s, \langle false, \{\} \rangle$

- $\langle s, \langle true, ST \rangle \rightarrow \langle s, \langle false, \{\} \rangle$ et

$\forall st \in ST : \langle I, \langle \langle X', C'' \rangle, E \rangle \langle \langle X, C' \rangle, E \rangle s S, E, present C, D \rangle \rightarrow \langle I, S, E, C'' C, D \rangle$ et on l'ajoute dans une liste temporaire TL .

Les éléments composant la machine étant expliqués, voici les nouvelles règles :

Partie de base de la machine SECD : Nous avons rajouté des règles mais la machine doit pouvoir traiter les mêmes informations que la machine de base donc on reprend directement les règles de la machine SECD.

Constante : On a une constante, on la déplace dans la pile.

$\langle S, E, n C, D \rangle \rightarrow_{TTS} \langle n S, E, C, D \rangle$ avec n une constante b ou un signal s

Substitution : On a une variable, on substitue la variable par sa valeur liée dans l'environnement via la fonction E .

$\langle S, E, X C, D \rangle \rightarrow_{TTS} \langle V S, E, C, D \rangle$ avec $E(X) = V$

Opération : On a un opérateur dans la chaîne de contrôle et le nombre de constante nécessaire dans la pile, via la fonction δ on retourne le résultat dans la pile.

$\langle b_n, \dots, b_1 S, E, prim_{o^n} C, D \rangle \rightarrow_{TTS} \langle V S, E, C, D \rangle$ avec $\delta(o^n b_1 \dots b_n) = V$

Abstraction : On a une abstraction, on crée une fermeture composée de l'abstraction et de l'environnement courant. On place la fermeture dans la pile.

$\langle S, E, \langle X, C' \rangle C, D \rangle \rightarrow_{TTS} \langle \langle \langle X, C' \rangle, E \rangle S, E, C, D \rangle$

Application : On a une application, on sauvegarde dans le dépôt, on remplace la chaîne de contrôle et l'environnement par ceux présent dans la fermeture et on ajoute une substitution dans cet environnement.

$\langle V \langle \langle X, C' \rangle, E' \rangle S, E, ap C, D \rangle \rightarrow_{TTS} \langle \emptyset, E'[X \leftarrow V], C', \langle S, E, C, D \rangle \rangle$

Récupération de sauvegarde : On a rien mais le dépôt comporte une sauvegarde donc on prend celle-ci.

$\langle V S, E, \epsilon, \langle S', E', C, D \rangle \rangle \rightarrow_{TTS} \langle V S', E', C, D \rangle$

Partie pour la concurrence : Voici les règles rajoutées dans le but de gérer les threads et les signaux. Ce sont les bases de la concurrence de la machine.

Création thread : On crée un nouveau thread.

$$\langle\langle\langle X, C' \rangle, E \rangle S, E, \text{spawn } C, D \rangle, TL, SI \rangle \rightarrow_{TTS} \langle\langle S, E, C, D \rangle, TL \langle S, E, C', D \rangle, SI \rangle$$

Initialisation signal : On initialise le signal via la fonction ι .

$$\langle\langle S, E, \text{init } C, D \rangle, TL, SI \rangle \rightarrow_{TTS} \langle\langle s, S, E, C, D \rangle, TL, SI' \rangle \text{ avec } \iota(SI) = (s, SI')$$

Présence d'un signal : On teste la présence d'un signal, via la fonction SI on sait qu'il est émis donc on prends le 1er choix.

$$\langle\langle\langle X', C'' \rangle, E \rangle \langle\langle X, C' \rangle, E \rangle s, S, E, \text{present } C, D \rangle, TL, SI \rangle \rightarrow_{TTS} \langle\langle S, E, C' C, D \rangle, TL, SI \rangle$$

avec $SI(s) = \langle \text{vraie}, ST \rangle$

Thread bloqué remplacé : On teste la présence d'un signal, il n'est pas émis et il y a un thread dans la file d'attente donc on met le thread courant dans la liste de threads bloqués et on prend le thread en tête de la file.

$$\langle\langle\langle X', C'' \rangle, E \rangle \langle\langle X, C' \rangle, E \rangle s, S, E, \text{present } C, D \rangle, \langle S', E', C''', D' \rangle TL, SI \rangle \rightarrow_{TTS} \langle\langle S', E', C''', D' \rangle, TL, SI' \rangle$$

avec $SI(s) = \langle \text{faux}, ST \rangle$ et $SI'(s) = \langle \text{faux}, ST \langle\langle X', C'' \rangle, E \rangle \langle\langle X, C' \rangle, E \rangle s, S, E, \text{present } C, D \rangle \rangle$

Thread bloqué non remplacé : On teste la présence d'un signal, il n'est pas émis donc on met ce thread dans la liste de threads bloqués.

$$\langle\langle\langle X', C'' \rangle, E \rangle \langle\langle X, C' \rangle, E \rangle s, S, E, C, D \rangle, \emptyset, SI \rangle \rightarrow_{TTS} \langle\langle \emptyset, \epsilon, \emptyset, \emptyset \rangle, \emptyset, SI' \rangle$$

avec $SI(s) = \langle \text{faux}, ST \rangle$ et $SI'(s) = \langle \text{faux}, ST \langle\langle X', C'' \rangle, E \rangle \langle\langle X, C' \rangle, E \rangle s, S, E, \text{present } C, D \rangle \rangle$

Récupération dans la file d'attente : On a plus rien à traiter et on a aucune sauvegarde, du coup on change le thread courant par le thread en tête de la file d'attente.

$$\langle\langle S, E, \epsilon, \emptyset \rangle, \langle S', E', C, D \rangle TL, SI \rangle \rightarrow_{TTS} \langle\langle S', E', C, D \rangle, TL, SI \rangle$$

Fin d'instant logique : On a plus rien à traiter, on a aucune sauvegarde et on a plus rien dans la file d'attente, c'est la fin d'un instant logique.

$$\langle\langle V, S, E, \epsilon, \emptyset \rangle, \emptyset, SI \rangle \rightarrow_{TTS} \langle\langle V, S, E, \epsilon, \emptyset \rangle, TL, SI' \rangle$$

avec $\tau(SI) = (TL, SI')$

Partie commune : Quand on rajoute des règles le plus gros risque est de créer des conflits avec les anciennes règles. Les conflits viennent de l'application et de la récupération de sauvegarde car *spawn* et *emit* ne retournent rien dans la pile donc il faut pouvoir continuer de faire fonctionner la machine avec ces deux cas.

Application neutre : On a une application sur rien, cela revient juste à rien faire.

$$\langle S, E, \text{ap } C, D \rangle \rightarrow_{TTS} \langle S, E, C, D \rangle$$

Récupération de sauvegarde avec pile vide : On a rien mais le dépôt comporte une sauvegarde donc on prends celle-ci.

$$\langle \emptyset, E, \epsilon, \langle S', E', C, D \rangle \rangle \rightarrow_{TTS} \langle S', E', C, D \rangle$$

la machine TTS peut s'arrêter dans 3 états différents :

- on a une **constante** telle que $\langle\langle \emptyset, \emptyset, [M]_{TTS}, \emptyset \rangle, \emptyset, \emptyset \rangle \rightarrow_{TTS} \langle\langle b, S, E, \epsilon, \emptyset \rangle, \emptyset, SI \rangle$;
- on a une **fonction** telle que $\langle\langle \emptyset, \emptyset, [M]_{TTS}, \emptyset \rangle, \emptyset, \emptyset \rangle \rightarrow_{TTS} \langle\langle \langle X, C' \rangle, E \rangle S, E, \epsilon, \emptyset \rangle, \emptyset, SI \rangle$;
- on a une **erreur**

Cette version des règles est la plus optimale car on a pris le meilleur de chaque possibilités. Cependant il faut savoir que des règles intermédiaires ont été créées, implantées et testées en OCaml. En effet je vais vous présenter 2 version mais 3 versions des règles existent et 4 machines ont été implantées. On peut retrouver ces règles en Annexe si cela vous intéresse. Cela vous permettra de voir tout le chemin parcouru durant le stage.

4.2 Le partage des valeurs dans la machine

4.2.1 Description informelle du langage

Les signaux nous permettent déjà de communiquer entre les threads par la présence ou l'absence de ceux-ci. On va monter d'un cran en ajoutant la possibilité de partager des valeurs avec les signaux. Cela va créer un semblant de mémoire partagée.

Les valeurs partagées

Quelles sont les contraintes pour accéder à ces valeurs ? On a déjà spécifié que l'on voulait que les valeurs soit lié à un signal précis. On va rajouter une contrainte supplémentaire. Chaque thread aura sa propre liste de valeurs partagées. C'est-à-dire que pour accéder à une valeur il faudra connaître le signal et le thread. Cela pose un problème non traité précédemment qui est de différencier chaque thread. On va devoir ajouter un identifiant à T , c'est-à-dire que l'on aura $T = \langle I, S, E, C, D \rangle$ avec un entier qui va représenter l'identifiant I . Un autre problème se pose par rapport à cela : l'attribution d'un identifiant.

Je m'explique, dans le cas des signaux ce n'était pas compliqué car on garde tous les signaux créés dans notre machine durant tout le processus donc on ne peut pas attribuer un identifiant qui a déjà été attribué avant. Or ici, quand un thread est fini on ne le garde pas. Même quand il est en cours, on peut le stocker dans deux endroits différents TL et ST . On va utiliser un producteur d'identifiant dans notre machine. Pour cela, on va créer un nouvel élément IP qui est un entier dans notre machine ce qui nous donne pour l'instant $\langle T, TL, SI, IP \rangle$.

Pour garder la machine concurrente fonctionnelle, on va devoir séparer les valeurs partagées en deux parties :

1. Une liste de valeurs courante : c'est là que l'on va insérer les valeurs, on ne peut pas prendre dans cette liste, on peut le voir comme une liste tampon. Cela va permettre d'éviter le problème suivant :

Un thread veut accéder à une valeur, il n'y en a pas, il laisse sa place et le thread d'après mets une valeur. Le problème vient du fait que la machine est dans le même instant logique donc le premier thread devrait pouvoir y accéder car hypothétiquement il fonctionne en même temps que le second.

2. Une liste de valeurs partagées : c'est là que l'on va pouvoir prendre les valeurs, on ne peut pas insérer dans cette liste.

Pour faire la transition entre la liste tampon et la liste de valeurs partagées on attend la fin d'un instant logique et, à la condition que le signal soit émis, on transfère ces valeurs.

Quand on prend une valeur on ne spécifie pas laquelle car on ne le sait pas. On va contraindre à prendre les valeurs dans l'ordre et une unique fois. Pour cela, on va avoir un pointeur pour chaque thread.

Comment stocker ces valeurs ? La liste des signaux est pour l'instant de la forme $SI \mid \forall si \in SI : \langle s, \langle emit, ST \rangle \rangle$. On va se contenter d'expliquer pour cette forme car une autre forme a été faite durant le stage mais n'est pas si éloigné de celle-ci. Si vous avez envie de la voir malgré tout elle se trouve en Annexe. Comme vu plus haut il nous faut deux listes :

1. CS la liste des valeurs courantes telle que $\forall cs \in CS : cs = \langle id, CL \rangle$ avec l'identifiant du thread id qui insère ces valeurs dans la liste des valeurs CL ;
2. SSI la liste des valeurs partagées telle que $\forall ssi \in SSI : ssi = \langle id, \langle CI, EL \rangle \rangle$ avec l'identifiant du thread id qui a inséré ces valeurs à l'instant précédent. On a le couple $\langle CI, EL \rangle$ qui va nous servir à itérer. EL est une liste d'identifiant qui représente la fin de l'itération dans cette liste. CI est la liste des valeurs telle que $\forall ci \in CI : ci = \langle b, IL \rangle$ avec une valeur b et une liste d'identifiant qui représente une position possible de l'itérateur IL .

$SI = \forall si \in SI : \langle s, \langle emit, ST \rangle \rangle$ devient $SI = \forall si \in SI : \langle s, \langle emit, CS, SSI, ST \rangle \rangle$

Accéder à une valeur On veut ajouter une commande *get*. Elle va servir à accéder à une valeur. Pour pouvoir y accéder, on a besoin d'un signal et d'un identifiant. On aura donc $s \ b \ get$ avec un signal s et un identifiant id . Cependant une question se pose, comment faire si on a pris toutes les valeurs ?

- On pourra lever une erreur : cela est possible si on ajoute une gestion des erreurs mais sur cette version ce n'est pas le cas. Si cela vous intéresse une version existe en Annexe avec ce principe.

- On demande un paramètre supplémentaire que l'on nommerait n et qui servirait de neutre, c'est-à-dire si on a fini d'itérer, on retournera le neutre.

On a donc une commande *get* de la forme $s \ id \ n \ get$ avec un signal s , l'identifiant du thread i dont l'on veut une valeur de ces valeurs partagées et n le neutre.

Insérer une valeur Il ne manque qu'à donner la possibilité d'insérer. Pour ça une commande *put* va être nécessaire. Pour insérer, on a besoin de deux informations : le signal et l'identifiant du thread courant. On aura juste besoin de spécifier le signal, l'autre on l'aura directement. Ce qui nous donne *s put* avec un signal *s*.

Petit point à spécifier, quand on insère un élément on a pour but de le partager. Sachant cela et que l'émission d'un signal est nécessaire pour pouvoir partager ces valeurs, on va réunir les commandes *emit* et *put*.

4.2.2 Sémantique de la machine abstraite

Ces deux commandes ajoutées engrangent énormément de changements. On va redéfinir nos règles pour voir les changements provoqués.

Soit $\langle T, TL, SI, IP \rangle$ avec :

TL = une file de thread telle que : $\forall tl \in TL \mid tl = T$ avec :

$T = \langle I, S, E, C, D \rangle$ le thread courant avec :

$V = b$

| $\langle \langle X, C' \rangle E \rangle$

| *signal*

I = un entier représentant l'identifiant du thread

$S = \emptyset$

| $V S$

$E = \{ \dots, \langle X, V \rangle, \dots \}$

$C = \epsilon$

| $b C$

(une constante)

| $X C$

(une variable)

| *signal* C

(un signal)

| $\langle X, C' \rangle C$

(une abstraction)

| *ap* C

(une application)

| *prim_{on}* C

(un opérateur)

| *spawn* C

(créer d'un nouveau thread)

| *present* C

(le test de présence d'un signal)

| *init* C

(initialise un signal)

| *put* C

(insère une valeur dans un signal)

| *get* C

(prends une valeurs dans un signal)

$D = \emptyset$

| $\langle S, E, C, D \rangle$

(une sauvegarde liée à une abstraction)

SI = une liste de signaux telle que : $\forall si \in SI : si = \langle signal, \langle emit, CS, SSI, TL \rangle \rangle$ avec :

- un booléen représentant l'émission du signal : *emit*

- un identifiant de thread : I

- une liste des signaux courants telle que : $\forall cs \in CS : cs = \langle I, CL \rangle$ avec

- une liste de constantes telle que : $\forall cl \in CL : cl = b$

- la liste des signaux partagés telle que : $\forall ssi \in SSI : ssi = \langle I, \langle CI, EL \rangle \rangle$ avec

- une liste d'identifiant de threads telle que : $\forall il \in EL : il = I$

- une liste de constante avec itérateur telle que : $\forall ci \in CI : ci = \langle b, EL \rangle$

IP = un entier servant à attribuer l'identifiant à un nouveau thread

Une suite de fonctions ont été écrites afin de simplifier la lecture des règles. Les voici :

$\iota(SI)$ une fonction qui prend l'identifiant du dernier signal créé, l'incrmente pour en créer un nouveau et retourne l'identifiant du signal créé avec la liste mise à jour.

Exemple :

Si on initialise pour la première fois alors $\iota(\{\}) = (0, \{\langle 0, \langle false, \{\}, \{\}, \{\} \rangle\})$

sinon $\iota(\{\dots, \langle s, data \rangle\}) = (s + 1, \{\dots, \langle s, data \rangle, \langle s + 1, \langle false, \{\}, \{\}, \{\} \rangle\})$ avec $data = \langle emit, CS, SSI, ST \rangle$

$SI(s)$ une fonction qui retourne le 2nd élément du couple $\langle s, data \rangle$ avec $data = \langle emit, CS, SSI, ST \rangle$.

Exemple : $SI(s) = \langle emit, CS, SSI, ST \rangle$

$\tau(SI)$ une fonction qui prend la liste signaux, met les liste de valeurs courantes dans la liste des valeurs partagés si il est émis, prend en compte l'absence des signaux non émis et retourne le couple $\langle TL, SI \rangle$ avec une liste de threads TL et SI la liste des signaux modifiés

Exemple : $\tau(SI) = \forall si \in SI :$

- $\langle true, CS, SSI, \{\} \rangle \rightarrow \langle false, \{\}, CS', \{\} \rangle$ en mettant en place la possibilité d'itérer

- $\langle false, CS, SSI, ST \rangle \rightarrow \langle false, \{\}, \{\}, \{\} \rangle$ et

$\forall st \in ST : \langle I, \langle \langle X', C'' \rangle, E \rangle \langle \langle X, C' \rangle, E \rangle s S, E, present C, D \rangle \rightarrow \langle I, S, E, C'' C, D \rangle$ et on l'ajoute dans une liste temporaire TL .

$SI[(s, i) \leftarrow b]$ est une fonction qui met dans la liste de valeurs ,de s pour le thread i, b et met à vrai le booléen représentant l'émission *emit*.

$SSI(i)$ une fonction qui retourne le couple lié à un signal et un thread dans la liste des signaux partagés.

Exemple : $SSI(i) = \langle CI, IL \rangle$

$\gamma(id, \langle CI, IL \rangle)$ une fonction qui retourne la constante et décale l'itérateur lié à l'identifiant de thread *id*.

On va définir une règle afin de simplifier les règles futurs :

Dans tous les cas :

$$\frac{\langle S, E, C, D \rangle \rightarrow_{TTSI} \langle S', E', C', D' \rangle}{\langle \langle I, S, E, C, D \rangle, TL, SI, IP \rangle \rightarrow_{TTSI} \langle \langle I, S', E', C', D' \rangle, TL, SI, IP \rangle}$$

Si la règle utilisée est ni **Thread bloqué non remplacé** ni **Création thread** :

$$\frac{\langle \langle S, E, C, D \rangle, TL, SI \rangle \rightarrow_{TTSI} \langle \langle S', E', C', D' \rangle, TL', SI' \rangle}{\langle \langle I, S, E, C, D \rangle, TL, SI, IP \rangle \rightarrow_{TTSI} \langle \langle I, S', E', C', D' \rangle, TL', SI', IP \rangle}$$

Les éléments étant expliqués, voici les nouvelles règles de la machine :

Partie de base de la machine SECD : On veut garder le fonctionnement de la machine SECD de base donc il faut garder ces règles.

Constante : On a une constante, on la déplace dans la pile.

$$\langle S, E, n \ C, D \rangle \longrightarrow_{TTSI} \langle n \ S, E, C, D \rangle \text{ avec } n \text{ une constante } b \text{ ou un signal } s$$

Substitution : On a une variable, on substitue la variable par sa valeur liée dans l'environnement via la fonction E .

$$\langle S, E, X \ C, D \rangle \longrightarrow_{TTSI} \langle V \ S, E, C, D \rangle \text{ avec } E(X) = V$$

Opération : On a un opérateur et le nombre de constante nécessaire dans la pile, via la fonction δ on retourne le résultat dans la pile.

$$\langle b_n, \dots, b_1 \ S, E, \text{prim}_{o^n} \ C, D \rangle \longrightarrow_{TTSI} \langle V \ S, E, C, D \rangle \text{ avec } \delta(o^n \ b_1 \dots b_n) = V$$

Abstraction : On a une abstraction, on crée une fermeture comportant l'abstraction et l'environnement courant et on met la fermeture dans la pile.

$$\langle S, E, \langle X, C' \rangle \ C, D \rangle \longrightarrow_{TTSI} \langle \langle \langle X, C' \rangle, E \rangle \ S, E, C, D \rangle$$

Application : On a une application, on sauvegarde dans le dépôt, on remplace la chaîne de contrôle et l'environnement par ceux présent dans la fermeture et on ajoute une substitution dans le nouveau environnement.

$$\langle V \ \langle \langle X, C' \rangle, E' \rangle \ S, E, \text{ap} \ C, D \rangle \longrightarrow_{TTSI} \langle \emptyset, E'[X \leftarrow V], C', \langle S, E, C, D \rangle \rangle$$

Récupération de sauvegarde : On a rien mais le dépôt comporte une sauvegarde donc on prend celle-ci.

$$\langle V \ S, E, \epsilon, \langle S', E', C, D \rangle \rangle \longrightarrow_{TTSI} \langle V \ S', E', C, D \rangle$$

Partie pour la concurrence : Cette partie rajoute la concurrence dans notre machine.

Création thread : On crée un nouveau thread.

$$\langle \langle I, \langle \langle X, C' \rangle, E \rangle \ S, E, \text{spawn} \ C, D \rangle, TL, SI, IP \rangle \longrightarrow_{TTSI} \langle \langle I, S, E, C, D \rangle, TL \ \langle IP, S, E, C', D \rangle, SI, IP + 1 \rangle$$

Ajouter dans un signal : On ajoute une constante dans une liste de valeurs d'un signal et on met à vraie le booléen *emit*

$$\langle \langle I, s \ b \ S, E, \text{put} \ C, D \rangle, TL, SI \rangle \longrightarrow_{TTSI} \langle \langle I, S, E, C, D \rangle, TL, SI \ [(s, I) \leftarrow b] \rangle$$

Prendre une valeur partagée : On prend dans la liste de valeurs d'un signal partagé lié à un thread et on décale l'itérateur.

$$\langle \langle I, s \ b \ n \ \langle \langle X, C' \rangle, E' \rangle \ S, E, \text{get} \ C, D \rangle, TL, SI \rangle \longrightarrow_{TTSI} \langle \langle I, \emptyset, E'[X \leftarrow V], C', \langle S, E, C, D \rangle \rangle, TL, SI \rangle$$

si pour $SI(s) = \langle \text{emit}, CS, SSI \rangle$ et $SSI(b) = \langle CI, IL \rangle$ on a $I \notin IL$ alors $\gamma(SSI(b)) = V$ sinon $n = V$

Initialisation signal : On initialise le signal via la fonction ι .

$$\langle \langle I, S, E, \text{init} \ C, D \rangle, TL, SI \rangle \longrightarrow_{TTSI} \langle \langle I, s \ S, E, C, D \rangle, TL, SI' \rangle \text{ avec } \iota(SI) = (s, SI')$$

Présence du signal : On teste la présence d'un signal, via la fonction SI on sait qu'il est émis donc on prend le 1er choix.

$$\langle \langle I, \langle \langle X', C'' \rangle, E \rangle \ \langle \langle X, C' \rangle, E \rangle \ s \ S, E, \text{present} \ C, D \rangle, TL, SI \rangle \longrightarrow_{TTSI} \langle \langle I, S, E, C' \ C, D \rangle, TL, SI \rangle$$

avec $SI(s) = \langle \text{vraie}, CS, SSI, TL \rangle$

Thread bloqué remplacé : On teste la présence d'un signal, il n'est pas émis et il y a un thread dans la file d'attente donc on met le thread courant dans la liste de threads bloqués et on prend le thread en tête de la file.

$$\begin{aligned} & \langle \langle I, \langle \langle X', C'' \rangle, E \rangle \langle \langle X, C' \rangle, E \rangle s S, E, present C, D \rangle, \langle I', S', E', C''' \rangle, D' \rangle TL, SI \rangle \\ & \longrightarrow_{TTSI} \langle \langle I', S', E', C''' \rangle, D' \rangle, TL, SI' \rangle \\ & \text{avec } SI(s) = \langle faux, CS, SSI, ST \rangle \\ & \text{et } SI'(s) = \langle faux, CS, SSI, ST \langle I, \langle \langle X', C'' \rangle, E \rangle \langle \langle X, C' \rangle, E \rangle s S, E, present C, D \rangle \rangle \end{aligned}$$

Thread bloqué non remplacé : On teste la présence d'un signal, il n'est pas émis donc on met ce thread dans la liste de threads bloqués.

$$\begin{aligned} & \langle \langle I, \langle \langle X', C'' \rangle, E \rangle \langle \langle X, C' \rangle, E \rangle s S, E, C, D \rangle, \emptyset, SI, IP \rangle \longrightarrow_{TTSI} \langle \langle IP, \emptyset, \epsilon, \emptyset, \emptyset \rangle, \emptyset, SI', IP + 1 \rangle \\ & \text{avec } SI(s) = \langle faux, CS, SSI, ST \rangle \\ & \text{et } SI'(s) = \langle faux, CS, SSI, ST \langle I, \langle \langle X', C'' \rangle, E \rangle \langle \langle X, C' \rangle, E \rangle s S, E, present C, D \rangle \rangle \end{aligned}$$

Récupération dans la file d'attente : On a plus rien à traiter et on a aucune sauvegarde, du coup on change le thread courant par le thread en tête de la file d'attente.

$$\langle \langle I, S, E, \epsilon, \emptyset \rangle, \langle I', S', E', C, D \rangle TL, SI \rangle \longrightarrow_{TTSI} \langle \langle I', S', E', C, D \rangle, TL, SI \rangle$$

Fin d'instant logique : On a plus rien à traiter, on a aucune sauvegarde et on a plus rien dans la file d'attente, c'est la fin d'un instant logique.

$$\langle \langle I, S, E, \epsilon, \emptyset \rangle, \emptyset, SI \rangle \longrightarrow_{TTSI} \langle \langle I, S, E, \epsilon, \emptyset \rangle, TL, SI' \rangle \text{ avec } \tau(SI) = (SI', TL)$$

Partie commune : Quand on ajoute des règles dans une machine déjà existante, le plus délicat est de ne pas avoir de conflits dans les règles. Pour cela, on définit des règles exprès pour faire la liaison entre ce qui existait et ce que l'on ajoute.

Application neutre : On a une application sur rien, cela revient juste à rien faire.

$$\langle S, E, ap C, D \rangle \longrightarrow_{TTSI} \langle S, E, C, D \rangle$$

Récupération de sauvegarde avec pile vide : On a rien mais le dépôt comporte une sauvegarde donc on prend celle-ci.

$$\langle \emptyset, E, \epsilon, \langle S', E', C, D \rangle \rangle \longrightarrow_{TTSI} \langle S', E', C, D \rangle$$

la machine TTSI peut s'arrêter dans 3 états différents :

on a une **constante** telle que $\langle \langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset \rangle, \emptyset, \emptyset, 1 \rangle \rightarrow_{TTSI} \langle \langle I, b S, E, \epsilon, \emptyset \rangle, \emptyset, SI, IP \rangle ;$

on a une **fonction** telle que $\langle \langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset \rangle, \emptyset, \emptyset, 1 \rangle \rightarrow_{TTSI} \langle \langle I, \langle \langle X, C \rangle, E \rangle S, E, \epsilon, \emptyset \rangle, \emptyset, SI, IP \rangle ;$

on a une **erreur**

4.3 La gestion des erreurs

5 Conclusion

6 Annexes

6.1 Les Exemples des machines étudiées

6.1.1 Exemple de fonctionnement de la machine CK

Voici un exemple de fonctionnement de la machine CK :

CK machine : $\langle ((\lambda f.\lambda x.f\ x)\ \lambda y.(+ y\ y))\ \lceil 1 \rceil, mt \rangle$
 $> (1)\ \langle (M\ N), \kappa \rangle \mapsto_{ck} \langle M, \langle arg, N, \kappa \rangle \rangle$
 CK machine : $\langle ((\lambda f.\lambda x.f\ x)\ \lambda y.(+ y\ y)), \langle arg, \lceil 1 \rceil, mt \rangle \rangle$
 $> (1)\ \langle (M\ N), \kappa \rangle \mapsto_{ck} \langle M, \langle arg, N, \kappa \rangle \rangle$
 CK machine : $\langle (\lambda f.\lambda x.f\ x), \langle arg, (\lambda y.(+ y\ y)), \langle arg, \lceil 1 \rceil, mt \rangle \rangle \rangle$
 $> (3)\ \langle V, \langle arg, N, \kappa \rangle \rangle \mapsto_{ck} \langle N, \langle fun, V, \kappa \rangle \rangle$
 CK machine : $\langle (\lambda y.(+ y\ y)), \langle fun, (\lambda f.\lambda x.f\ x), \langle arg, \lceil 1 \rceil, mt \rangle \rangle \rangle$
 $> (2)\ \langle V, \langle fun, (\lambda X.M), \kappa \rangle \rangle \mapsto_{ck} \langle M[X \leftarrow V], \kappa \rangle$
 CK machine : $\langle (\lambda x.f\ x)[f \leftarrow (\lambda y.(+ y\ y))], \langle arg, \lceil 1 \rceil, mt \rangle \rangle$
 CK machine : $\langle (\lambda x.(\lambda y.(+ y\ y))\ x), \langle arg, \lceil 1 \rceil, mt \rangle \rangle$
 $> (3)\ \langle V, \langle arg, N, \kappa \rangle \rangle \mapsto_{ck} \langle N, \langle fun, V, \kappa \rangle \rangle$
 CK machine : $\langle \lceil 1 \rceil, \langle fun, (\lambda x.(\lambda y.(+ y\ y))\ x), mt \rangle \rangle$
 $> (2)\ \langle V, \langle fun, (\lambda X.M), \kappa \rangle \rangle \mapsto_{ck} \langle M[X \leftarrow V], \kappa \rangle$
 CK machine : $\langle ((\lambda y.(+ y\ y))\ x)[x \leftarrow \lceil 1 \rceil], mt \rangle$
 CK machine : $\langle ((\lambda y.(+ y\ y))\ \lceil 1 \rceil), mt \rangle$
 $> (1)\ \langle (M\ N), \kappa \rangle \mapsto_{ck} \langle M, \langle arg, N, \kappa \rangle \rangle$
 CK machine : $\langle (\lambda y.(+ y\ y)), \langle arg, \lceil 1 \rceil, mt \rangle \rangle$
 $> (3)\ \langle V, \langle arg, N, \kappa \rangle \rangle \mapsto_{ck} \langle N, \langle fun, V, \kappa \rangle \rangle$
 CK machine : $\langle \lceil 1 \rceil, \langle fun, (\lambda y.(+ y\ y)), mt \rangle \rangle$
 $> (2)\ \langle V, \langle fun, (\lambda X.M), \kappa \rangle \rangle \mapsto_{ck} \langle M[X \leftarrow V], \kappa \rangle$
 CK machine : $\langle (+ y\ y)[y \leftarrow \lceil 1 \rceil], mt \rangle$
 CK machine : $\langle (+ \lceil 1 \rceil\ \lceil 1 \rceil), mt \rangle$
 $> (4)\ \langle (o^n\ M\ N\dots), \kappa \rangle \mapsto_{ck} \langle M, \langle opd, \langle o^n \rangle, \langle N, \dots \rangle, \kappa \rangle \rangle$
 CK machine : $\langle \lceil 1 \rceil, \langle opd, \langle + \rangle, \langle \lceil 1 \rceil \rangle, mt \rangle \rangle$
 $> (6)\ \langle V, \langle opd, \langle V', \dots o^n \rangle, \langle N, L, \dots \rangle, \kappa \rangle \rangle \mapsto_{ck} \langle N, \langle opd, \langle V, V', \dots o^n \rangle, \langle L, \dots \rangle, \kappa \rangle \rangle$
 CK machine : $\langle \lceil 1 \rceil, \langle opd, \langle \lceil 1 \rceil, + \rangle, \langle \rangle, mt \rangle \rangle$
 $> (5)\ \langle b, \langle opd, \langle b_i, \dots b_1, o^n \rangle, \langle \rangle, \kappa \rangle \rangle \mapsto_{ck} \langle V, \kappa \rangle$ avec $\delta(o^n, b_1, \dots b_i, b) = V$
 CK machine : $\langle \lceil 2 \rceil, mt \rangle$

6.1.2 Exemple de fonctionnement de la machine CEK

Voici un exemple de fonctionnement de la machine CEK :

CEK machine : $\langle \langle (((\lambda f. \lambda x. f \ x) \ \lambda y. (+ \ y \ y)) \ \lceil 1 \rceil) \emptyset), mt \rangle$

$$> (1) \langle \langle (M \ N), \varepsilon \rangle, \bar{\kappa} \rangle \mapsto_{cek} \langle \langle M, \varepsilon \rangle, \langle arg, \langle N, \varepsilon \rangle, \bar{\kappa} \rangle \rangle$$

CEK machine : $\langle \langle ((\lambda f. \lambda x. f \ x) \ \lambda y. (+ \ y \ y)) \emptyset \rangle, \langle arg, \langle \lceil 1 \rceil, \emptyset \rangle, mt \rangle \rangle$

$$> (1) \langle \langle (M \ N), \varepsilon \rangle, \bar{\kappa} \rangle \mapsto_{cek} \langle \langle M, \varepsilon \rangle, \langle arg, \langle N, \varepsilon \rangle, \bar{\kappa} \rangle \rangle$$

CEK machine : $\langle\langle(\lambda f.\lambda x.f\ x),\emptyset\rangle,\langle arg,\langle(\lambda y.(+ \ y\ y)),\emptyset\rangle,\langle arg,\langle\ulcorner 1 \urcorner,\emptyset\rangle,mt\rangle\rangle\rangle$

> (4) $\langle \langle V, \varepsilon \rangle, \langle arg, \langle N, \varepsilon' \rangle, \kappa \rangle \rangle \mapsto_{cek} \langle \langle N, \varepsilon' \rangle, \langle fun, \langle V, \varepsilon \rangle, \bar{\kappa} \rangle \rangle$ si $V \notin X$

CEK machine : $\langle\langle(\lambda y.(+ \ y \ y)), \emptyset\rangle, \langle fun, \langle(\lambda f.\lambda x.f \ x), \emptyset\rangle, \langle arg, \langle\ulcorner 1 \urcorner, \emptyset\rangle, mt\rangle\rangle\rangle$

$$> (3) \langle \langle V, \varepsilon \rangle, \langle fun, \langle (\lambda X1.M), \varepsilon' \rangle, \bar{k} \rangle \rangle \mapsto_{cek} \langle \langle M, \varepsilon'[X1 \leftarrow \langle V, \varepsilon \rangle] \rangle, \bar{k} \rangle \text{ si } V \notin X$$

CEK machine : $\langle \langle (\lambda x.f \ x), \emptyset[f \leftarrow \langle (\lambda y.(+ \ y \ y)), \emptyset] \rangle, \langle arg, \langle \ulcorner 1 \urcorner, \emptyset \rangle, mt \rangle \rangle$

CEK machine : $\langle\langle(\lambda x.f\ x), \{\langle f, \langle(\lambda y.(+ y\ y)), \emptyset\rangle\rangle\}\rangle, \langle arg, \langle\lceil 1 \rceil, \emptyset\rangle, mt\rangle\rangle$

> (4) $\langle \langle V, \varepsilon \rangle, \langle arg, \langle N, \varepsilon' \rangle, \kappa \rangle \rangle \mapsto_{cek} \langle \langle N, \varepsilon' \rangle, \langle fun, \langle V, \varepsilon \rangle, \bar{\kappa} \rangle \rangle$ si $V \notin X$

CEK machine : $\langle \langle \ulcorner 1 \urcorner, \emptyset \rangle, \langle fun, \langle (\lambda x. f \ x), \{ \langle f, \langle (\lambda y. (+ \ y \ y)), \emptyset \rangle \} \rangle \rangle, mt \rangle$

$$> (3) \langle \langle V, \varepsilon \rangle, \langle fun, \langle (\lambda X1.M), \varepsilon' \rangle, \bar{\kappa} \rangle \rangle \mapsto_{cek} \langle \langle M, \varepsilon'[X1 \leftarrow \langle V, \varepsilon \rangle] \rangle, \bar{\kappa} \rangle \text{ si } V \notin X$$

CEK machine : $\langle \langle (f\ x), \{ \langle f, \langle (\lambda y. (+\ y\ y)), \emptyset \rangle \rangle \} [x \leftarrow \langle \ulcorner 1 \urcorner, \emptyset \rangle], mt \rangle$

CEK machine : $\langle \langle (f\ x), \{ \langle f, \langle (\lambda y. (+\ y\ y)), \emptyset \rangle \rangle, \langle x, \langle \ulcorner 1 \urcorner, \emptyset \rangle \rangle \} \rangle, mt \rangle$

$$> (1) \langle \langle (M \ N), \varepsilon \rangle, \overline{\kappa} \rangle \longmapsto_{cek} \langle \langle M, \varepsilon \rangle, \langle arg, \langle N, \varepsilon \rangle, \overline{\kappa} \rangle \rangle$$

CEK machine : $\langle \langle f, \{ \langle f, \langle (\lambda y. (+ y y)), \emptyset \rangle \rangle, \langle x, \langle \ulcorner 1 \urcorner, \emptyset \rangle \rangle \} \rangle, \langle arg, \langle x, \{ \langle f, \langle (\lambda y. (+ y y)), \emptyset \rangle \rangle, \langle x, \langle \ulcorner 1 \urcorner, \emptyset \rangle \rangle \} \rangle, mt \rangle \rangle$

> (2) $\langle \langle X, \varepsilon \rangle, \bar{K} \rangle \mapsto_{cek} \langle c, \bar{K} \rangle$ avec $\varepsilon(X) = c$

CEK machine : $\langle \langle (\lambda y. (+ y y)), \emptyset \rangle, \langle arg, \langle x, \{ \langle f, \langle (\lambda y. (+ y y)), \emptyset \rangle \rangle, \langle x, \langle \ulcorner 1 \urcorner, \emptyset \rangle \rangle \} \rangle, mt \rangle$

> (4) $\langle \langle V, \varepsilon \rangle, \langle arg, \langle N, \varepsilon' \rangle, \kappa \rangle \rangle \mapsto_{cek} \langle \langle N, \varepsilon' \rangle, \langle fun, \langle V, \varepsilon \rangle, \bar{\kappa} \rangle \rangle$ si $V \notin X$

CEK machine : $\langle \langle x, \{ \langle f, \langle (\lambda y. (+ y y)), \emptyset \rangle \rangle, \langle x, \langle \ulcorner 1 \urcorner, \emptyset \rangle \rangle \} \rangle, \langle fun, \langle (\lambda y. (+ y y)), \emptyset \rangle, mt \rangle$

> (2) $\langle \langle X, \varepsilon \rangle, \bar{\kappa} \rangle \mapsto_{cek} \langle c, \bar{\kappa} \rangle$ avec $\varepsilon(X) = c$

CEK machine : $\langle \langle \ulcorner 1 \urcorner, \emptyset \rangle, \langle fun, \langle (\lambda y. (+ y y)), \emptyset \rangle, mt \rangle \rangle$

$$> (3) \langle \langle V, \varepsilon \rangle, \langle fun, \langle (\lambda X1.M), \varepsilon' \rangle, \bar{\kappa} \rangle \rangle \mapsto_{cek} \langle \langle M, \varepsilon'[X1 \leftarrow \langle V, \varepsilon \rangle] \rangle, \bar{\kappa} \rangle \text{ si } V \notin X$$

CEK machine : $\langle \langle (+ \ y \ y), \emptyset[y \leftarrow \langle \ulcorner 1 \urcorner, \emptyset \rangle] \rangle, mt \rangle$

CEK machine : $\langle \langle (+ \ y \ y), \{ \langle y, \langle \ulcorner 1 \urcorner, \emptyset \rangle \rangle \} \rangle, mt \rangle$

$$> (5) \langle \langle (o^n M N \dots), \varepsilon \rangle, \bar{\kappa} \rangle \mapsto_{cek} \langle \langle M, \varepsilon \rangle, \langle opd, \langle o^n \rangle, \langle \langle N, \varepsilon \rangle, \dots \rangle, \bar{\kappa} \rangle \rangle$$

CEK machine : $\langle \langle y, \{ \langle y, \langle \ulcorner 1 \urcorner, \emptyset \rangle \} \rangle, \langle opd, \langle + \rangle, \langle \langle y, \{ \langle y, \langle \ulcorner 1 \urcorner, \emptyset \rangle \} \rangle \rangle, mt \rangle \rangle$

> (2) $\langle \langle X, \varepsilon \rangle, \bar{\kappa} \rangle \mapsto_{cek} \langle c, \bar{\kappa} \rangle$ avec $\varepsilon(X) = c$

CEK machine : $\langle \langle \ulcorner 1 \urcorner, \emptyset \rangle, \langle opd, \langle + \rangle, \langle \langle y, \{ \langle y, \langle \ulcorner 1 \urcorner, \emptyset \rangle \} \rangle \rangle, mt \rangle \rangle$

$$> (7) \langle \langle V, \varepsilon \rangle, \langle opd, \langle v', \dots o^n \rangle, \langle \langle N, \varepsilon' \rangle, c, \dots \rangle, \bar{\kappa} \rangle \rangle \mapsto_{cek} \langle \langle N, \varepsilon' \rangle, \langle opd, \langle \langle V, \varepsilon \rangle, v', \dots o^n \rangle, \langle c, \dots \rangle, \bar{\kappa} \rangle \rangle \text{ si } V \notin X$$

CEK machine : $\langle \langle y, \{ \langle y, \langle \ulcorner 1 \urcorner, \emptyset \rangle \rangle \} \rangle, \langle opd, \langle \ulcorner 1 \urcorner + \rangle, \langle \rangle, mt \rangle \rangle$

> (2) $\langle \langle X, \varepsilon \rangle, \bar{\kappa} \rangle \mapsto_{cek} \langle c, \bar{\kappa} \rangle$ avec $\varepsilon(X) = c$

CEK machine : $\langle \langle \ulcorner 1 \urcorner, \emptyset \rangle, \langle opd, \langle \ulcorner 1 \urcorner + \rangle, \langle \rangle, mt \rangle \rangle$

> (6) $\langle \langle b, \varepsilon \rangle, \langle opd, \langle \langle b_i, \varepsilon_i \rangle, \dots \langle b_1, \varepsilon_1 \rangle, o^n \rangle, \langle \rangle, \bar{\kappa} \rangle \rangle \mapsto_{cek} \langle \langle V, \emptyset \rangle, \bar{\kappa} \rangle$ avec $\delta(o^n, b_1, \dots b_i, b) = V$

CEK machine : $\langle \langle \ulcorner 2 \urcorner, \emptyset \rangle, mt \rangle$

6.1.3 Exemple de fonctionnement de la machine SECD

Voici un exemple de fonctionnement de la machine SECD :

Conversion : $[(((\lambda f.\lambda x.f\ x)\ \lambda y.(+ y\ y))\ \lceil 1 \rceil)]_{secd}$

Conversion : $[((\lambda f.\lambda x.f\ x)\ \lambda y.(+ y\ y))]_{secd}\ [\ulcorner 1 \urcorner]_{secd}\ ap$

Conversion : $[(\lambda f. \lambda x. f \ x)]_{secd} \ [\lambda y. (+ \ y \ y)]_{secd} \ ap \ \lceil 1 \rceil \ ap$

Conversion : $\langle f, [\lambda x. (f \ x)]_{\text{secd}} \rangle \ \langle y, y \ y \ \text{prim}_+ \rangle \ \text{ap} \ \lceil 1 \rceil \ \text{ap}$

Conversion : $\langle f, \langle x, [f\ x]_{secd} \rangle \rangle \langle y, y\ y\ prim_+ \rangle\ ap\ \lceil 1 \rceil\ ap$

Conversion : $\langle f, \langle x, f \ x \ ap \rangle \rangle \ \langle y, y \ y \ prim_+ \rangle \ ap \ \ulcorner 1 \urcorner \ ap$

SECD Machine : $\langle \epsilon, \emptyset, \langle f, \langle x, f \ x \ ap \rangle \rangle \ \langle y, y \ y \ prim_+ \rangle \ ap \ulcorner 1 \urcorner \ ap, \epsilon \rangle$

$$> (3) \langle \hat{S}, \hat{\varepsilon}, \langle X, C' \rangle \hat{C}, \hat{D} \rangle \mapsto_{sec d} \langle \langle \langle X, C' \rangle, \varepsilon \rangle \hat{S}, \hat{\varepsilon}, \hat{C}, \hat{D} \rangle$$

SECD Machine : $\langle \langle \langle f, \langle x, f \ x \ ap \rangle \rangle, \emptyset \rangle, \emptyset, \langle y, y \ y \ prim_+ \rangle \ ap \ulcorner 1 \urcorner \ ap, \epsilon \rangle$

$$> (3) \langle \widehat{S}, \widehat{\varepsilon}, \langle X, C' \rangle \widehat{C}, \widehat{D} \rangle \longmapsto_{sec d} \langle \langle \langle X, C' \rangle, \varepsilon \rangle \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D} \rangle$$

SECD Machine : $\langle \langle \langle y, y \text{ } y \text{ } prim_+ \rangle, \emptyset \rangle \langle \langle f, \langle x, f \text{ } x \text{ } ap \rangle \rangle, \emptyset \rangle, \emptyset, ap \text{ } \ulcorner 1 \urcorner \text{ } ap, \epsilon \rangle$

$$> (4) \langle \hat{V} \langle \langle X, C' \rangle, \varepsilon' \rangle \hat{S}, \hat{\varepsilon}, ap \hat{C}, \hat{D} \rangle \mapsto_{secd} \langle \epsilon, \epsilon' [X \leftarrow \hat{V}], C', \langle \hat{S}, \hat{\varepsilon}, \hat{C}, \hat{D} \rangle \rangle$$
$$\text{SECD Machine} : \langle \epsilon, \emptyset [f \leftarrow \langle \langle y, y \text{ } y \text{ } \textit{prim}_+ \rangle, \emptyset \rangle], \langle x, f \text{ } x \text{ } \textit{ap} \rangle, \langle \epsilon, \emptyset, \lceil 1 \rceil \text{ } \textit{ap}, \epsilon \rangle \rangle$$

SECD Machine : $\langle \epsilon, \{ \langle f, \langle \langle y, y \text{ } y \text{ } prim_+ \rangle, \emptyset \rangle \rangle, \langle x, f \text{ } x \text{ } ap \rangle, \langle \epsilon, \emptyset, \lceil 1 \rceil \text{ } ap, \epsilon \rangle \rangle$

$$> (3) \langle \hat{S}, \hat{\varepsilon}, \langle X, C' \rangle \hat{C}, \hat{D} \rangle \mapsto_{sec d} \langle \langle \langle X, C' \rangle, \varepsilon \rangle \hat{S}, \hat{\varepsilon}, \hat{C}, \hat{D} \rangle$$

SECD Machine : $\langle \langle \langle x, f \ x \ ap \rangle, \{ \langle f, \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \} \rangle, \{ f, \langle y, y \ y \ prim_+ \rangle, \emptyset \}, \emptyset, \langle \epsilon, \emptyset, \lceil 1 \rceil \ ap, \epsilon \rangle \rangle$

$$> (5) \langle \widehat{V} \widehat{S}, \widehat{\varepsilon}, \emptyset, \langle \widehat{S}', \widehat{\varepsilon}', \widehat{C}', \widehat{D} \rangle \rangle \mapsto_{sec d} \langle \widehat{V} \widehat{S}', \widehat{\varepsilon}', \widehat{C}', \widehat{D} \rangle$$

SECD Machine : $\langle \langle \langle x, f \ x \ ap \rangle, \{ \langle f, \langle y, y \ y \ prim_+ \rangle, \emptyset \} \rangle \rangle, \emptyset, \lceil 1 \rceil \ ap, \epsilon \rangle$

$$> (1) \langle \hat{S}, \hat{\varepsilon}, b \hat{C}, \hat{D} \rangle \mapsto_{\text{secd}} \langle b \hat{S}, \hat{\varepsilon}, \hat{C}, \hat{D} \rangle$$

SECD Machine : $\langle \ulcorner 1 \urcorner \langle \langle x, f \ x \ ap \rangle, \{ \langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \} \rangle, \emptyset, ap, \epsilon \rangle$

$$> (4) \langle \hat{V} \mid \langle \langle X, C' \rangle, \varepsilon' \rangle \mid \hat{S}, \hat{\varepsilon}, ap \hat{C}, \hat{D} \rangle \mapsto_{secd} \langle \epsilon, \varepsilon' [X \leftarrow \hat{V}], C', \langle \hat{S}, \hat{\varepsilon}, \hat{C}, \hat{D} \rangle \rangle$$

SECD Machine : $\langle \epsilon, \{ \langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \} [x \leftarrow \ulcorner 1 \urcorner], f \ x \ ap, \langle \epsilon, \emptyset, \epsilon, \epsilon \rangle \rangle$

SECD Machine : $\langle \epsilon, \{ \langle f, \langle \langle y, y \text{ } prim_+ \rangle, \emptyset \rangle \rangle, \langle x, \lceil 1 \rceil \rangle \}, f \text{ } x \text{ } ap, \langle \epsilon, \emptyset, \epsilon, \epsilon \rangle \rangle$

$$> (2) \langle \hat{S}, \hat{\varepsilon}, X \hat{C}, \hat{D} \rangle \mapsto_{secd} \langle \hat{V} \hat{S}, \hat{\varepsilon}, \hat{C}, \hat{D} \rangle \text{ où } \hat{V} = \varepsilon(X)$$

SECD Machine : $\langle \langle y, y \text{ } y \text{ } prim_+ \rangle, \emptyset \rangle, \{ \langle f, \langle y, y \text{ } y \text{ } prim_+ \rangle, \emptyset \rangle, \langle x, \lceil 1 \rceil \rangle \}, x \text{ } ap, \langle \epsilon, \emptyset, \epsilon, \epsilon \rangle$

$$> (2) \langle \hat{S}, \hat{\varepsilon}, X \hat{C}, \hat{D} \rangle \mapsto_{sec d} \langle \hat{V} \hat{S}, \hat{\varepsilon}, v \hat{C}, \hat{D} \rangle \text{ où } \hat{V} = \varepsilon(X)$$

SECD Machine : $\langle \ulcorner 1 \urcorner \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle, \{ \langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \rangle, \langle x, \ulcorner 1 \urcorner \rangle \}, ap, \langle \epsilon, \emptyset, \epsilon, \epsilon \rangle \rangle$

$$> (4) \langle \hat{V} \langle \langle X, C' \rangle, \varepsilon' \rangle \hat{S}, \hat{\varepsilon}, ap \hat{C}, \hat{D} \rangle \mapsto_{secd} \langle \epsilon, \varepsilon' [X \leftarrow \hat{V}], C', \langle \hat{S}, \hat{\varepsilon}, \hat{C}, \hat{D} \rangle \rangle$$

SECD Machine : $\langle \epsilon, \emptyset[y \leftarrow \ulcorner 1 \urcorner], y \ y \ prim_+, \langle \epsilon, \{ \langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \}, \langle x, \ulcorner 1 \urcorner \rangle \}, \emptyset, \langle \epsilon, \emptyset, \epsilon, \epsilon \rangle \rangle$

SECD Machine : $\langle \epsilon, \{\langle y, \lceil 1 \rceil\} \rangle, y \ y \ prim_+, \langle \epsilon, \{\langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle\}, \langle x, \lceil 1 \rceil \rangle\}, \emptyset, \langle \epsilon, \emptyset, \epsilon, \epsilon \rangle \rangle$

$$> (2) \langle \widehat{S}, \widehat{\varepsilon}, X \widehat{C}, \widehat{D} \rangle \longmapsto_{sec d} \langle \widehat{V} \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D} \rangle \text{ où } \widehat{V} = \varepsilon(X)$$

SECD Machine : $\langle \lceil 1 \rceil, \{ \langle y, \lceil 1 \rceil \rangle \}, y \text{ prim}_+, \langle \epsilon, \{ \langle f, \langle \langle y, y \text{ prim}_+ \rangle, \emptyset \rangle \}, \langle x, \lceil 1 \rceil \rangle \}, \emptyset, \langle \epsilon, \emptyset, \epsilon, \epsilon \rangle \rangle$

$$> (2) \langle \widehat{S}, \widehat{\varepsilon}, X \widehat{C}, \widehat{D} \rangle \longmapsto_{sec d} \langle \widehat{V} \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D} \rangle \text{ où } \widehat{V} = \varepsilon(X)$$

SECD Machine : $\langle \lceil 1 \rceil \lceil 1 \rceil, \{ \langle y, \lceil 1 \rceil \}, prim_+, \langle \epsilon, \{ \langle f, \langle \langle y, y \ y \ prim_+ \rangle, \emptyset \rangle \}, \langle x, \lceil 1 \rceil \rangle, \emptyset, \langle \epsilon, \emptyset, \epsilon, \epsilon \rangle \rangle$

$$> (6) \langle b_1 \dots b_n \hat{S}, \hat{\varepsilon}, \text{prim}_{o^n} \hat{C}, \hat{D} \rangle \mapsto_{\text{secd}} \langle \hat{V} \hat{S}, \hat{\varepsilon}, \hat{C}, \hat{D} \rangle \text{ où } \hat{V} = \delta(o^n, b_1, \dots, b_n)$$
$$\text{SECD Machine : } \langle \lceil 2 \rceil, \{ \langle y, \lceil 1 \rceil \} \}, \emptyset, \langle \epsilon, \{ \langle f, \langle \langle y, y \text{ prim}_+ \rangle, \emptyset \rangle \}, \langle x, \lceil 1 \rceil \} \}, \emptyset, \langle \epsilon, \emptyset, \epsilon, \epsilon \rangle \rangle$$
$$> (5) \langle \widehat{V} \widehat{S}, \widehat{\varepsilon}, \emptyset, \langle \widehat{S}', \widehat{\varepsilon}', \widehat{C}', \widehat{D} \rangle \rangle \mapsto_{sec d} \langle \widehat{V} \widehat{S}', \widehat{\varepsilon}', \widehat{C}', \widehat{D} \rangle$$

SECD Machine : $\langle \lceil 2 \rceil, \{ \langle f, \langle \langle y, y \text{ } y \text{ } prim_+ \rangle, \emptyset \rangle \rangle, \langle x, \lceil 1 \rceil \rangle \}, \emptyset, \langle \epsilon, \emptyset, \epsilon, \epsilon \rangle \rangle$

$$> (5) \langle \widehat{V} \widehat{S}, \widehat{\varepsilon}, \emptyset, \langle \widehat{S}', \widehat{\varepsilon}', \widehat{C}', \widehat{D} \rangle \rangle \mapsto_{secd} \langle \widehat{V} \widehat{S}', \widehat{\varepsilon}', \widehat{C}', \widehat{D} \rangle$$

SECD Machine : $\langle \ulcorner 2 \urcorner, \emptyset, \epsilon, \epsilon \rangle$

6.2 Les différentes versions faite pour rendre la machine SECD concurrente

Cette partie énumère les différentes versions créées depuis le début du stage jusqu'à ce jour. Elle paraissent compliqué à comprendre à cause de leurs format qui est bien trop lourd mais elle garde le même principe que celle exprimé plus haut dans le rapport. Il ne faut donc pas avoir peur de chercher à comprendre.

6.2.1 1ère-2ème version des règles de la machine SECD Concurrente

Cette version ajoute les prémisses de la concurrence dans la machine SECD avec la possibilité de créer des threads d'initialiser des signaux, les émettre où encore de tester la présence d'un signal. Cette version est un condensé de 2 versions.

Soit $\langle S, E, C, D, W, ST, SI \rangle$ *avec :*

$$\begin{aligned}
V &= b \\
&| \langle \langle X, C \rangle, E \rangle \\
S &= \epsilon \\
&| V S \\
&| Remp S \\
E &= \text{une fonction } \{ \langle X, V \rangle, \dots \} \\
C &= \epsilon \\
&| b C \\
&| X C \\
&| ap C \\
&| prim_{o^n} C \\
&| \langle X, C \rangle C \\
&| bspawn C \\
&| espawn C \\
&| \langle s, C', C'' \rangle C \\
&| \langle s, C' \rangle C \\
&| emit_s C \\
D &= \epsilon \\
&| \langle S, E, C, D \rangle \\
W &= \{ D, \dots \} \\
ST &= \{ \dots, \langle s, D \rangle, \dots \} \\
SI &= \{ s, \dots \}
\end{aligned}$$

Les nouvelles règles sont les suivantes :

Partie de base de la machine SECD

Constante : On a une constante, on la déplace dans la pile.

$$\langle S, E, b C, D, W, ST, SI \rangle \mapsto_{secdv1c} \langle b S, E, C, D, W, ST, SI \rangle$$

Substitution : On a une variable, on prends la substitution dans l'environnement et on la mets dans la pile.

$$\langle S, E, X C, D, W, ST, SI \rangle \mapsto_{secdv1-2} \langle V S, E, C, D, W, ST, SI \rangle \text{ où } V = E(X)$$

Opération : On a un opérateur et le nombre de constante nécessaire dans la pile, via la fonction δ et in retourne le résultat dans la pile.

$$\langle b_1 \dots b_n S, E, prim_{o^n} C, D, W, ST, SI \rangle \mapsto_{secdv1-2} \langle V S, E, C, D, W, ST, SI \rangle \text{ où } V = \delta(o^n, b_1, \dots b_n)$$

Abstraction : On a une abstraction, on crée une fermeture comportant l'abstraction et l'environnement courant et on mets la fermeture dans la pile

$$\langle S, E, \langle X, C' \rangle C, D, W, ST, SI \rangle \mapsto_{secdv1-2} \langle \langle \langle X, C' \rangle, E \rangle S, E, C, D, W, ST, SI \rangle$$

Application : On a une application, donc on la sauvegarde dans le dépôt, on ajoute une substitution et on remplace la chaîne de contrôle et l'environnement par ceux présent dans la fermeture.

$$\langle V \langle \langle X, C' \rangle, E' \rangle S, E, ap C, D, W, ST, SI \rangle \mapsto_{secdv1-2} \langle \epsilon, E'[X \leftarrow V], C', \langle S, E, C, D \rangle, W, ST, SI \rangle$$

Récupération de sauvegarde : On a rien mais le dépôt comporte une sauvegarde donc on prends celle-ci.

$$\langle V S, E, \epsilon, \langle S, E, C, D \rangle, W, ST, SI \rangle \mapsto_{secdv1-2} \langle V S', E, C', D, W, ST, SI \rangle$$

Partie pour la concurrence

Création thread : On veut créer un nouveau thread

$$\langle S, E, bspawn C' espawn C, D, W, ST, SI \rangle \mapsto_{secdv1-2} \langle Remp S, E, C, D, W \langle S, E, C', D \rangle, ST, SI \rangle$$

Initialisation signal : On initialise le signal

$$\langle S, E, \langle s, C' \rangle C, D, W, ST, SI \rangle \mapsto_{secdv1-2} \langle \epsilon, E[init \leftarrow s], C', \langle S, E, C, D \rangle, W, ST, SI \rangle$$

Présence d'un signal : On teste la présence d'un signal et il l'est donc on prends la 1ère option.

$$\langle S, E, \langle s, C', C'' \rangle C, D, W, ST, SI \rangle \mapsto_{secdv1-2} \langle S, E, C' C, D, W, ST, SI \rangle$$

si $s \in SI$ et $s \in E$

Thread bloqué remplacé : On teste la présence d'un signal et il ne l'est pas donc on le remplace par le thread en tête de la file d'attente.

$$\langle S, E, \langle s, C', C'' \rangle C, D, \langle S', E', C''', D' \rangle W, ST, SI \rangle \mapsto_{secdv1-2} \langle S', E', C''', D', W, ST \langle S, E, \langle s, C', C'' \rangle C, D \rangle, SI \rangle$$

si $s \notin SI$ et $S \in E$

Thread bloqué non remplacé : On teste la présence d'un signal et il ne l'est pas et la file est vide mest juste le thread courant dans la liste de threads bloqués.

$$\langle S, E, \langle s, C', C'' \rangle C, D, \emptyset, ST, SI \rangle \mapsto_{secdv1-2} \langle \emptyset, \emptyset, \epsilon, \emptyset, \emptyset, ST \langle S, E, \langle s, C', C'' \rangle C, D \rangle, SI \rangle$$

si $s \notin SI$ et $s \in E$

Émettre : on émet un signal

$$\langle S, E, emit_s C, D, W, ST, SI \rangle \mapsto_{secdv1-2} \langle S, E, C, D, W', ST', SI \rangle$$

avec $W' = W \cup$ tout les éléments de ST qui attendent l'émission de s et
avec $ST' = ST \setminus$ tout les éléments de ST qui attendent l'émission de s

Récupération dans la file d'attente : On a plus rien à traiter et on a aucune sauvegarde, du coup on change de thread courant par le thread en tête de la file d'attente.

$$\langle S, E, \epsilon, \emptyset, \langle S', E', C, D \rangle W, ST, SI \rangle \mapsto_{secdv1-2} \langle S', E', C, D, W, ST, SI \rangle$$

Fin d'instant logique : On a plus rien à traiter et on a plus rien dans la file d'attente. C'est la fin de l'instant logique

$$\langle S, E, \epsilon, \emptyset, \emptyset, ST, SI \rangle \mapsto_{secdv1-2} \langle S, E, \emptyset, \emptyset, W, \emptyset, \emptyset, \emptyset \rangle$$

avec $W =$ tout les éléments de ST qui prennent leurs 2nd choix

Partie commune

Application neutre droite : on a une application avec un neutre dans la pile donc on l'enlève

$$\langle V Remp S, E, ap C, D, W, ST, SI \rangle \mapsto_{secdv1-2} \langle V S, E, C, D, W, ST, SI \rangle$$

Application neutre gauche : on a une application avec un neutre dans la pile donc on l'enlève

$$\langle Remp V S, E, ap C, D, W, ST, SI \rangle \mapsto_{secdv1-2} \langle V S, E, C, D, W, ST, SI \rangle$$

la machine SECD version 1 peut s'arrêter dans 4 états différents :

- on a une **constante b** tels que $\langle \epsilon, \emptyset, [M]_{secdv1-2}, \emptyset, \emptyset, \emptyset, \epsilon \rangle \rightarrow_{secdv1-2} \langle b, E, \epsilon, \emptyset, \emptyset, SI, \epsilon \rangle$;
- on a une **abstraction function** tels que $\langle \epsilon, \emptyset, [M]_{secdv1-2}, \emptyset, \emptyset, \emptyset, \epsilon \rangle \rightarrow_{secdv1-2} \langle \langle \langle X, C \rangle, E' \rangle, E, \epsilon, \emptyset, \emptyset, SI, \epsilon \rangle$;
- on a un **remplacement** tels que $\langle \epsilon, \emptyset, [M]_{secdv1-2}, \emptyset, \emptyset, \emptyset, \epsilon \rangle \rightarrow_{secdv1-2} \langle Remp, E, \epsilon, \emptyset, \emptyset, SI, \epsilon \rangle$;
- on a un **état inconnu** soit une **erreur**.

6.2.2 3ème version des règles de la machine SECD Concurrente

Cette version ajoute le contrôle des erreurs sans propagation via l'ajout d'un gestionnaire d'erreur dans la machine. Il commence à y avoir beaucoup d'éléments dans notre machine donc on va en rassembler. On va définir TL un couple qui regroupe W et ST , c'est-à-dire $TL = \langle W, ST \rangle$.

Soit $\langle S, E, C, D, TL, SI, H \rangle$ *avec :*

$$\begin{aligned}
V &= b \\
&| \langle \langle X, C \rangle, E \rangle \\
&| erreur_e \\
S &= \epsilon \\
&| V S \\
&| Remp S \\
E &= \text{une fonction } \{ \langle X, V \rangle, \dots \} \\
C &= \epsilon \\
&| b C \\
&| X C \\
&| ap C \\
&| prim_{o^n} C \\
&| \langle X, C \rangle C \\
&| bspawn C \\
&| espawn C \\
&| \langle s, C', C'' \rangle C \\
&| \langle s, C' \rangle C \\
&| emit_s C \\
&| throw_e C \\
&| \langle e, \langle C', \langle X, C'' \rangle \rangle \rangle C \\
D &= \epsilon \\
&| \langle S, E, C, D \rangle \\
TL &= \langle W, ST \rangle \text{ avec} \\
&- W = \{ D, \dots \} \\
&- ST = \{ \dots, \langle s, D \rangle, \dots \} \\
SI &= \{ s, \dots \} \\
H &= \epsilon \\
&| \langle e, \langle S, E, C, D, TL, SI, H \rangle \rangle
\end{aligned}$$

Les nouvelles règles sont les suivantes :

Partie de base de la machine SECD

Constante : On a une constante, on la déplace dans la pile.

$$\langle S, E, b C, D, TL, SI, H \rangle \mapsto_{secdv3} \langle b S, E, C, TL, SI, H \rangle$$

Substitution : On a une abstraction, on crée une fermeture avec celle-ci et l'environnement courant et on la place dans la pile.

$$\langle S, E, X C, D, TL, SI, H \rangle \mapsto_{secdv3} \langle V S, E, C, D, TL, SI, H \rangle$$

où $V = E(X)$

Opération : On a un opérateur et le nombre de constante nécessaire dans la pile, via la fonction δ on retourne le résultat dans la pile.

$$\langle b_1 \dots b_n S, E, prim_{o^n} C, D, TL, SI, H \rangle \mapsto_{secdv3} \langle V S, E, C, D, TL, SI, H \rangle$$

où $V = \delta(o^n, b_1, \dots, b_n)$

Abstraction : On a une abstraction, on créer une fermeture comportant l'abstraction et l'environnement courant et on mets la fermeture dans la pile.

$$\langle S, E, \langle X, C' \rangle C, D, TL, SI, H \rangle \mapsto_{secdv3} \langle \langle X, C' \rangle, E \rangle S, E, C, D, TL, SI, H \rangle$$

Application : On a une application, donc on sauvegarde dans le dépôt, on ajoute une substitution et on remplace la chaîne de contrôle et l'environnement par ceux présent dans la fermeture.

$$\langle V \langle \langle X, C' \rangle, E' \rangle S, E, ap C, D, TL, SI, H \rangle \mapsto_{secdv3} \langle \epsilon, E'[X \leftarrow V], C', \langle S, E, C, D \rangle, TL, SI, H \rangle$$

Récupération de sauvegarde : On a rien mais le dépôt comporte une sauvegarde donc on prends celle-ci.

$$\langle V S, E, \epsilon, \langle S', E', C, D \rangle, TL, SI, H \rangle \mapsto_{secdv3} \langle V S', E', C, D, TL, SI, H \rangle$$

Partie pour les erreurs

Erreur : On a une erreur, on la déplace en tête de la pile.

$$\langle S, E, throw_e C, D, TL, SI, H \rangle \mapsto_{secdv3} \langle erreur_e S, E, C, D, TL, SI, H \rangle$$

Traitée erreur via gestionnaire d'erreur : On a plus rien mais on a une erreur levé dans la pile du coup on regarde si le gestionnaire d'erreur gère celle-ci ; oui du coup prend la sauvegarde.

$$\langle S, E, throw_e C, D, TL, SI, \langle e, \langle S', E', \langle X, C'' \rangle C', D', TL', SI', H \rangle \rangle \rangle \mapsto_{secdv3} \langle S', E', \langle X, C'' \rangle throw_e C', D', TL', SI', H \rangle$$

Traitement erreur récursif : On a plus rien mais on a une erreur levé dans la pile du coup on regarde si le gestionnaire d'erreur gère celle-ci mais non du coup on regarde pour le gestionnaire sauvegardé.

$$\langle S, E, throw_e C, D, TL, SI, \langle e', \langle S', E', \langle X, C'' \rangle C', D', TL', SI', H \rangle \rangle \rangle \mapsto_{secdv3} \langle S, E, throw_e C, D, TL, SI, H \rangle$$

Erreur non traitée : On a plus rien mais on a une erreur levé dans la pile du coup on arrête la machine en vidant tout sauf la pile.

$$\langle S, E, throw_e C, D, TL, SI, \emptyset \rangle \mapsto_{secdv3} \langle S, E, throw_e, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

Création d'un gestionnaire d'erreur : On a un try...catch donc on test avec la chaîne de contrôle du try et on sauvegarde catch dans le gestionnaire d'erreur.

$$\langle S, E, \langle e, \langle C', \langle X, C'' \rangle \rangle \rangle C, D, TL, SI, H \rangle \mapsto_{secdv3} \langle S, E, C' C, D, TL, SI, \langle e, \langle S, E, \langle X, C'' \rangle C, D, TL, SI, H \rangle \rangle \rangle$$

Partie pour la concurrence

Création thread : On veut créer un nouveau thread.

$$\langle S, E, bspawn C' spawn C, D, \langle W, ST \rangle, SI, H \rangle \mapsto_{secdv3} \langle Remp S, E, C, D, \langle W \langle S, E, C', D \rangle, ST \rangle, SI, H \rangle$$

Initialisation signal : On initialise le signal.

$$\langle S, E, \langle s, C' \rangle C, D, TL, SI, H \rangle \mapsto_{secdv3} \langle \epsilon, E [init \leftarrow s], C', \langle S, E, C, D \rangle, TL, SI, H \rangle$$

Présence signal : On teste la présence d'un signal, on sait qu'il est émit donc on prends le 1er choix.

$$\langle S, E, \langle s, C', C'' \rangle C, D, TL, SI, H \rangle \mapsto_{secdv3} \langle S, E, C' C, D, TL, SI, H \rangle$$

si $s \in SI$ et $s \in E$

Thread bloqué remplacé : On teste la présence d'un signal, on sait qu'il n'est pas émit et il y a un thread dans la file d'attente donc on mets ce thread dans la liste de threads bloqués et on prends le thread en tête de la file.

$$\langle S, E, \langle s, C', C'' \rangle C, D, \langle \langle S', E', C''', D' \rangle W, ST \rangle, SI, H \rangle \mapsto_{secdv3} \langle S', E', C''', D', \langle W, ST \langle s, \langle S, E, \langle s, C', C'' \rangle C, D \rangle \rangle \rangle, SI, H \rangle \text{ si } s \notin SI \text{ et } s \in E$$

Thread bloqué non remplacé : On teste la présence d'un signal, on sait qu'il n'est pas émit donc on mets ce thread dans la liste de threads bloqués.

$$\langle S, E, \langle s, C', C'' \rangle C, D, \langle \emptyset, ST \rangle, SI, H \rangle \mapsto_{secdv3} \langle \epsilon, \emptyset, \epsilon, \emptyset, \langle \emptyset, ST \rangle \langle s, \langle S, E, \langle s, C', C'' \rangle C, D \rangle \rangle, SI, H \rangle$$

si $s \notin SI$ et $s \in E$

Émission : On émet un signal donc on mets dans la file d'attente tous les threads attendant le signal.

$$\langle S, E, emit_s C, D, \langle W, ST \rangle, SI, H \rangle \mapsto_{secdv3} \langle S, E, C, D, \langle W', ST' \rangle, SI, H \rangle$$

avec $W' = W \cup$ tout les éléments de ST qui attendent l'émission de s et
avec $ST' = ST \setminus$ tout les éléments de ST qui attendent l'émission de s

Récupération dans la file d'attente : On a plus rien à traité et on a aucune sauvegarde, du coup on change le thread courant par le thread en tête de la file d'attente.

$$\langle S, E, \epsilon, \emptyset, \langle \langle S', E', C', D' \rangle W, ST \rangle, SI, H \rangle \mapsto_{secdv3} \langle S', E', C', D', \langle W, ST \rangle, SI, H \rangle$$

Fin d'instant logique : On a plus rien à traiter, on a aucune sauvegarde et on a plus rien dans la file d'attente, c'est la fin d'un instant logique.

$$\langle S, E, \epsilon, \emptyset, \langle \emptyset, ST \rangle, SI, H \rangle \mapsto_{secdv3} \langle S, E, \epsilon, \emptyset, \langle W, \emptyset \rangle, \emptyset, H \rangle$$

avec $W =$ tout les éléments de ST qui prennent leurs 2nd choix

Partie commune

Application neutre droite : On a une application avec un *Remp* à droite donc on enlève juste le *Remp*.

$$\langle V \text{ Remp } S, E, ap C, D, TL, SI, H \rangle \mapsto_{secdv3} \langle V S, E, C, D, TL, SI, H \rangle$$

Application neutre gauche : On a une application avec un *Remp* à gauche donc on enlève juste le *Remp*.

$$\langle \text{Remp } V S, E, ap C, D, TL, SI, H \rangle \mapsto_{secdv3} \langle V S, E, C, D, TL, SI, H \rangle$$

la machine SECD version 3 peut s'arrêter dans 4 états différents :

$$\longrightarrow \text{on a une constante } b \text{ tels que } \langle \emptyset, \emptyset, [M]_{secdv3}, \emptyset, \langle \emptyset, \emptyset \rangle, \emptyset, \emptyset \rangle \rightarrow_{secdv3} \langle b, E, \epsilon, \emptyset, \langle \emptyset, \emptyset \rangle, SI, H \rangle ;$$

$$\longrightarrow \text{on a une abstraction function tels que } \langle \epsilon, \emptyset, [M]_{secdv3}, \emptyset, \langle \emptyset, \emptyset \rangle, \emptyset, \emptyset \rangle \rightarrow_{secdv3} \langle \langle \langle X, C \rangle, E' \rangle, E, \epsilon, \emptyset, \langle \emptyset, \emptyset \rangle, SI, H \rangle ;$$

$$\longrightarrow \text{on a un remplacement tels que } \langle \epsilon, \emptyset, [M]_{secdv3}, \emptyset, \langle \emptyset, \emptyset \rangle, \emptyset, \emptyset \rangle \rightarrow_{secdv3} \langle \text{Remp}, E, \epsilon, \emptyset, \langle \emptyset, \emptyset \rangle, SI, H \rangle ;$$

$$\longrightarrow \text{on a une erreur } e \text{ tels que } \langle \epsilon, \emptyset, [M]_{secdv3}, \emptyset, \langle \emptyset, \emptyset \rangle, \emptyset, \emptyset \rangle \rightarrow_{secdv3} \langle erreur_e, E, \epsilon, \emptyset, \langle \emptyset, \emptyset \rangle, SI, H \rangle.$$

6.2.3 4ème version des règles de la machine SECD Concurrente

La machine ayant beaucoup évolué depuis le début du travail, je vais redéfinir chaque élément de la machine pour pouvoir mieux comprendre les nouvelles règles.

Une suite de fonctions ont été écrite pour simplifier la lecture des règles. Les voici :

$\rho(l, v, s, i)$ = la fonction qui pour une liste des signaux l , une valeur v , un signal s et un identifiant du thread courant donnés, renvoie la liste l' avec v ajouté à la liste des valeurs du signal s pour le thread i .

Exemple : $\rho(\{..., \langle s, \{..., \langle id, valeur \rangle, ... \rangle, emit \rangle, ... \}, v, s, id) = \{..., \langle s, \{..., \langle id, valeur \ v \rangle, ... \rangle, emit \rangle, ... \}$

$\gamma(l, s, i, i')$ = la fonction qui pour une liste de valeurs partagées classés par signaux et par thread l , un signal s , l'identifiant du thread courant et l'identifiant du thread auxquels on veut accéder, renvoie soit un couple la liste avec l'itérateur déplacé et la valeur ou une exception si on ne peut plus donner de nouvelles valeurs

Example :

$$\gamma(\langle \dots, \langle s, \{ \dots, \langle id, \{ \dots, \langle b, \{ \dots, id', \dots \} \rangle, \langle n, \{ \dots \} \rangle, \dots \rangle, \{ \dots \} \rangle, \dots \rangle, \dots \rangle, s, id, id' \rangle = \langle b, \{ \dots, \langle s, \{ \dots, \langle id, \{ \dots, \langle b, \{ \dots \} \rangle, \langle n, \{ \dots, id' \} \rangle, \dots \rangle, \{ \dots \} \rangle, \dots \rangle, \dots \rangle \rangle$$
$$\gamma(\{\dots, \langle s, \{\dots, \langle id, \{\dots, \langle b, \{\dots, id', \dots \rangle\}, \{\dots\} \rangle, \dots \rangle, \dots \rangle, s, id, id' \rangle = \langle b, \{\dots, \langle s, \{\dots, \langle id, \{\dots, \langle b, \{\dots\} \rangle\}, \{\dots, id' \rangle\}, \dots \rangle, \dots \rangle\}$$
$$\gamma(\{..., \langle s, \{..., \langle id, valeurs, \{..., id', ... \rangle, ... \rangle, ... \rangle, s, id, id' \rangle) = throw\ erreur_e$$

$\iota(l, s, i)$ = la fonction qui pour une liste de signaux courant l, un signal s, renvoie une liste des signaux courant avec le signal s initialisé.

Example : $\iota(\{\dots\}, s) = \{\dots, \langle s, \{\}, false \rangle\}$

$\beta(l, s)$ = la fonction qui pour une liste de signal courant l et un signal s donnés, renvoie le booléen émit.

Exemple :

$$\beta(\{..., \langle s, \{...\}, vraie \rangle, ...\}, s) = vraie$$
$$\beta(\{..., \langle s, \{...\}, faux \rangle, ...\}, s) = faux$$

$\varepsilon(l, s)$ = la fonction qui pour une liste de signaux courant l et un signal s donnés, renvoie la liste avec le booléen représentant l'émission du signal à vraie.

Exemple :

$$\varepsilon(\{..., \langle s, \{...\}, faux \rangle, ...\}, s) = \{..., \langle s, \{...\}, vraie \rangle, ...\}$$

$\alpha(\langle CS, SSI \rangle)$ = la fonction qui pour la liste des signaux courant et la liste des signaux partagées données, renvoie la liste des signaux courant vidées de ses listes de valeurs et le booléen représentant l'émission mis à nulle ainsi que la liste des signaux partagées remplacer par les listes de valeurs de la liste des signaux courants qui sont émit.

Exemple :

$$\alpha(\langle CS, SSI \rangle) =$$

SSI vidé

$\forall x \in CS$ tels que $\langle s, \{..., \langle id, \{..., b, ... \} \}, ... \rangle, true \rangle$, on ajoute x dans SSI

$\forall x \in CS$ tels que $\langle s, \{..., \langle id, \{..., b, ... \} \}, ... \rangle, emit \rangle$ on le remplace par $\langle s, \{..., \langle id, \{ \} \}, ... \rangle, faux \rangle$

Soit $\langle I, S, E, C, D, TL, SI, H, IP \rangle$ **avec** :

$V = b$

| $\langle \langle X, C' \rangle E \rangle$

| $erreur_e$

$I =$ un entier représentant l'identifiant du thread

$S = \emptyset$

| VS

| $signal\ S$

| $throw\ S$

$E = \{..., \langle X, V \rangle, ...\}$

$C = \epsilon$

| $b\ C$ (une constante)

| $X\ C$ (une variable)

| $signal\ C$ (un signal)

| $\langle X, C' \rangle\ C$ (une abstraction)

| $ap\ C$ (une application)

| $prim_{o^n}\ C$ (un opérateur)

| $bspawn\ C$ (début d'un nouveau thread)

| $espawn\ C$ (fin d'un nouveau thread)

| $\langle C', C'' \rangle\ C$ (le test de présence d'un signal)

| $emit\ C$ (émet un signal)

| $init\ C$ (initialise un signal pour une chaîne de contrôle donné)

| $put\ C$ (insère une valeur dans la liste de valeurs d'un signal)

| $get\ C$ (prends une valeurs dans la liste de valeurs d'un signal)

| $erreur_e\ C$ (une erreur)

| $throw\ C$ (lève une erreur)

| $\langle C', \langle X, C'' \rangle \rangle\ C$ (un gestionnaire d'erreur)

$TL = \langle W, ST \rangle$

$W = \{..., \langle I, S, E, C, D \rangle, ...\}$ (liste des threads en attente)

$ST = \{..., \langle s, \langle I, S, E, C, D \rangle \rangle, ...\}$ (liste des threads en attente d'un signal)

$SI = \langle CS, SSI \rangle$

$CS = \{..., \langle s, \{..., \langle id, \{..., b, ... \} \rangle, ... \rangle, emit \rangle, ...\}$ (liste des signaux courants)

on va découper cette élément pour mieux en comprendre le sens :

- $\{..., *, ... \}$ Une liste.

- $\langle s, \{..., **, ... \} \rangle, emit \rangle$

Une liste composée de trinôme comportant le identifiant du signal, une sous-liste et un booléen exprimant l'émission de ce signal.

- $\langle id, \{..., b, ... \} \rangle$

Une sous-liste composée d'un trinôme comportant l'identifiant du thread et une liste de valeur.

$SSI = \{..., \langle s, \{..., \langle id, \{..., \langle b, \{..., id', ... \} \rangle, ... \rangle, \{..., id'', ... \} \rangle, ... \rangle, ... \rangle, ... \}$ (liste des signaux partagés)

comme pour CS on va découper cette élément pour pouvoir le comprendre :

- $\{..., *, ... \}$ Une liste.

- $\langle s, \{..., **, ... \} \rangle$

Une liste composée d'un couple comportant un identifiant de signal et d'une sous-liste

- $\langle id, \{..., *, *, ..., ... \} \rangle, \{..., id'', ... \}$

Une sous-liste composée d'un trinôme comportant un identifiant d'un thread, d'un liste et d'une sous-sous-liste d'identifiant de thread représentant la liste des threads ayant fini leurs parcours de la sous-sous-liste.

- $\langle b, \{..., id', ... \} \rangle$

Une sous-sous-liste composée d'un couple comportant une valeur et une liste d'identifiant de threads qui représente un pointeur

$D = \emptyset$

| $\langle S, E, C, D \rangle$ (une sauvegarde liée à une abstraction)

$H = \emptyset$

| $\langle e \langle I, S, E, \langle X, C' \rangle C, D, TL, SI, H, IP \rangle \rangle$

$IP =$ un entier servant à attribuer l'identifiant à un nouveau thread

Les éléments étant expliqués, voici les nouvelles règles de la machine :

Partie de base de la machine SECD

Constante : On a une constante, on la déplace dans la pile.

$$\langle I, S, E, b \ C, D, TL, SI, H, IP \rangle \longrightarrow_{secdv4} \langle I, b \ S, E, C, D, TL, SI, H, IP \rangle$$

Substitution : On a une abstraction, on créer une fermeture avec celle-ci et l'environnement courant et on la place dans la pile.

$$\langle I, S, E, X \ C, D, TL, SI, H, IP \rangle \longrightarrow_{secdv4} \langle I, V \ S, E, C, D, TL, SI, H, IP \rangle$$

avec $E(X) = V$

Opération : On a un opérateur et le nombre de constante nécessaire dans la pile, via la fonction δ on retourne le résultat dans la pile.

$$\langle I, b_n, \dots, b_1 \ S, E, prim_{o^n} \ C, D, TL, SI, H, IP \rangle \longrightarrow_{secdv4} \langle I, V \ S, E, C, D, TL, SI, H, IP \rangle$$

avec $\delta(o^n \ b_1 \dots b_n) = V$

Abstraction : On a une abstraction, on créer une fermeture comportant l'abstraction et l'environnement courant et on mets la fermeture dans la pile.

$$\langle I, S, E, \langle X, C' \rangle \ C, D, TL, SI, H, IP \rangle \longrightarrow_{secdv4} \langle I, \langle \langle X, C' \rangle, E \rangle \ S, E, C, D, TL, SI, H, IP \rangle$$

Application : On a une application, donc on sauvegarde dans le dépôt, on ajoute une substitution et on remplace la chaîne de contrôle et l'environnement par ceux présent dans la fermeture.

$$\langle I, V \ \langle \langle X, C' \rangle, E' \rangle \ S, E, ap \ C, D, TL, SI, H, IP \rangle \longrightarrow_{secdv4} \langle I, \epsilon, E'[X \leftarrow V], C', \langle S, E, C, D \rangle, TL, SI, H, IP \rangle$$

Récupération de sauvegarde : On a rien mais le dépôt comporte une sauvegarde donc on prends celle-ci.

$$\langle I, V \ S, E, \epsilon, \langle S', E', C, D \rangle, TL, SI, H, IP \rangle \longrightarrow_{secdv4} \langle I, V \ S', E', C, D, TL, SI, H, IP \rangle$$

Partie pour les erreurs

Erreur : On a une erreur, on la déplace en tête de la pile.

$$\langle I, S, E, erreur_e \ C, D, TL, SI, H, IP \rangle \longrightarrow_{secdv4} \langle I, erreur_e \ S, E, C, D, TL, SI, H, IP \rangle$$

Lever erreur : On a un throw, on le déplace en tête de la pile.

$$\langle I, S, E, throw \ C, D, TL, SI, H, IP \rangle \longrightarrow_{secdv4} \langle I, throw \ S, E, C, D, TL, SI, H, IP \rangle$$

Opération sur erreur : On a l'opérateur qui traite cette erreur donc on mets le résultat de la fonction δ dans la pile.

$$\langle I, throw \ erreur_e \ S, E, prim_{o^{1e}} \ C, D, TL, SI, H, IP \rangle \longrightarrow_{secdv4} \langle I, V \ S, E, C, D, TL, SI, H, IP \rangle$$

avec $\delta(o^{1e} \ erreur_e) = V$

Propagation : On a un un élément excepté l'opérateur qui traite cette erreur donc on propage l'erreur.

$$\langle I, throw \ erreur_e \ S, E, M \ C, D, TL, SI, H, IP \rangle \longrightarrow_{secdv4} \langle I, throw \ erreur_e \ S, E, C, D, TL, SI, H, IP \rangle$$

avec $M = \text{un élément de } C \setminus prim_{o^{1e}}$

Traitée erreur via gestionnaire d'erreur : On a plus rien mais on a une erreur levé dans la pile du coup on regarde si le gestionnaire d'erreur gère celle-ci ; oui du coup prend la sauvegarde.

$$\langle I, throw \ erreur_e \ S, E, \epsilon, D, TL, SI, \langle e, \langle I', S', E', \langle X, C'' \rangle C', D', TL', SI', H, IP' \rangle \rangle, IP \rangle \longrightarrow_{secdv4} \langle I', \emptyset, E'[X \leftarrow erreur_e], C'', \langle S', E', C', D' \rangle, TL', SI', H, IP' \rangle$$

Erreur non traitée : On a plus rien mais on a une erreur levé dans la pile du coup on arrête la machine en vidant tout sauf l'erreur

$$\langle I, S, E, throw_e \ C, D, TL, SI, \emptyset, IP \rangle \longmapsto_{secdv4} \langle I, S, E, throw_e, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, IP \rangle$$

Traitement erreur récursif : On a plus rien mais on a une erreur levé dans la pile du coup on regarde si le gestionnaire d'erreur gère celle-ci mais non du coup on regarde pour le gestionnaire sauvegardé.

$$\langle I, throw \ erreur_e \ S, E, \epsilon, D, TL, SI, \langle e', \langle I', S', E', \langle X, C'' \rangle C', D', TL', SI', H, IP' \rangle \rangle, IP \rangle \longrightarrow_{secdv4} \langle I, throw \ erreur_e \ S, E, \epsilon, D, TL, SI, H, IP \rangle$$

Création d'un gestionnaire d'erreur : On a un try...catch donc on test avec la chaîne de contrôle du try et on sauvegarde catch dans le gestionnaire d'erreur.

$$\langle I, erreur_e S, E, \langle C', \langle X, C'' \rangle \rangle C, D, TL, SI, H, IP \rangle \\ \rightarrow_{secdv4} \langle I, S, E, C' C, D, TL, SI, \langle e, \langle I, erreur_e S, E, \langle X, C'' \rangle C, D, TL, SI, H, IP \rangle \rangle, IP \rangle$$

Partie pour la concurrence

Création thread : On veut créer un nouveau thread.

$$\langle I, S, E, bspawn C' espawn C, D, \langle W, ST \rangle, SI, H, IP \rangle \\ \rightarrow_{secdv4} \langle I, S, E, C, D, \langle W \langle IP, S, E, C', D \rangle, ST \rangle, SI, H, IP + 1 \rangle$$

Signal : On a un signal, on le déplace dans la pile.

$$\langle I, S, E, signal C, D, TL, SI, H, IP \rangle \rightarrow_{secdv4} \langle I, signal S, E, C, D, TL, SI, H, IP \rangle$$

Ajouter dans un signal : On ajoute une constante dans une liste de valeurs d'un signal via la fonction ρ

$$\langle I, s b S, E, put C, D, TL, \langle CS, SSI \rangle, H, IP \rangle \rightarrow_{secdv4} \langle I, S, E, C, D, TL, \langle CS', SSI \rangle, H, IP \rangle$$

avec $CS' = \rho(CS, b, s, I)$ et s initialisé

Prendre une valeur partagée (possible) : On prends dans la liste de valeurs d'un signal partagé lié à un identifiant une constante via la fonction γ .

$$\langle I, s b \langle \langle X, C' \rangle, E' \rangle S, E, get C, D, TL, \langle CS, SSI \rangle, H, IP \rangle \\ \rightarrow_{secdv4} \langle I, \emptyset, E'[X \leftarrow V], C', \langle S, E, C, D \rangle, TL, \langle CS, SSI' \rangle, H, IP \rangle$$

avec $\gamma(SSI, s, I, b) = \langle V, SSI' \rangle$ si il reste une valeur à prendre et s partagé

Prendre une valeur partagée (impossible) : On prends dans la liste de valeurs d'un signal partagé lié à un identifiant une constante via la fonction γ mais on a déjà tout pris donc on lève une erreur.

$$\langle I, s b \langle \langle X, C' \rangle, E' \rangle S, E, get C, D, TL, \langle CS, SSI \rangle, H, IP \rangle \\ \rightarrow_{secdv4} \langle I, throw erreur_e S, E, C, D, TL, \langle CS, SSI' \rangle, H, IP \rangle$$

avec $\gamma(SSI, s, I, b) = throw erreur_e$ si il reste aucune valeur à prendre et s partagé

Initialisation signal : On initialise le signal via la fonction ι .

$$\langle I, s S, E, init C, D, TL, \langle CS, SSI \rangle, H, IP \rangle \rightarrow_{secdv4} \langle I, S, E, C, D, TL, \langle CS', SSI \rangle, H, IP \rangle$$

avec $\iota(CS, s) = CS'$

Présence signal : On teste la présence d'un signal, via la fonction β on sait qu'il est émit donc on prends le 1er choix.

$$\langle I, s S, E, \langle C', C'' \rangle C, D, TL, \langle CS, SSI \rangle, H, IP \rangle \rightarrow_{secdv4} \langle I, S, E, C' C, D, TL, \langle CS, SSI \rangle, H, IP \rangle$$

avec $\beta(CS, s) = vraie$

Thread bloqué remplacé : On teste la présence d'un signal, via la fonction β on sait qu'il n'est pas émit et il y a un thread dans la file d'attente donc on mets ce thread dans la liste de threads bloqués et on prends le thread en tête de la file.

$$\langle I, s S, E, \langle C', C'' \rangle C, D, \langle \langle I', S', E', C''', D' \rangle W, ST \rangle, \langle CS, SSI \rangle, H, IP \rangle \\ \rightarrow_{secdv4} \langle I', S', E', C''', D', \langle W, ST \langle s, \langle I, s S, E, \langle C', C'' \rangle C, D \rangle \rangle \rangle, \langle CS, SSI \rangle, H, IP \rangle$$

avec $\beta(CS, s) = faux$

Thread bloqué non remplacé : On teste la présence d'un signal, via la fonction β on sait qu'il n'est pas émit donc on mets ce thread dans la liste de threads bloqués.

$$\langle I, s S, E, \langle C', C'' \rangle C, D, \langle \emptyset, ST \rangle, \langle CS, SSI \rangle, H, IP \rangle \\ \rightarrow_{secdv4} \langle IP, \emptyset, \emptyset, \epsilon, \emptyset, \langle W, ST \langle s, \langle I, s S, E, \langle C', C'' \rangle C, D \rangle \rangle \rangle, \langle CS, SSI \rangle, H, IP + 1 \rangle$$

avec $\beta(CS, s) = faux$

Émission : On émet un signal donc on mets dans la file d'attente tous les threads attendant le signal.

$\langle I, s, S, E, emit, C, D, TL, \langle CS, SSI \rangle, H, IP \rangle \xrightarrow{secdv4} \langle I, Unit, S, E, C, D, TL', \langle CS', SSI \rangle, H, IP \rangle$
avec $\varepsilon(CS, s) = CS'$ et $TL' = \langle W', ST' \rangle$ et $TL = \langle W, ST \rangle$:

$W' = W \cup$ les éléments de ST qui attendent le signal s

$ST' = ST \setminus$ les éléments de ST qui attendent le signal s

Récupération dans la file d'attente : On a plus rien à traité et on a aucune sauvegarde, du coup on change le thread courant par le thread en tête de la file d'attente.

$\langle I, V, S, E, \epsilon, \emptyset, \langle \langle I', S', E', C, D \rangle W, ST \rangle, SI, H, IP \rangle \xrightarrow{secdv4} \langle I', V, S', E', C, D, \langle W, ST \rangle, SI, H, IP \rangle$

Fin d'instant logique : On a plus rien à traiter, on a aucune sauvegarde et on a plus rien dans la file d'attente, c'est la fin d'un instant logique.

$\langle I, V, S, E, \epsilon, \emptyset, \langle \emptyset, ST \rangle, SI, H, IP \rangle \xrightarrow{secdv4} \langle I, V, S, E, \epsilon, \emptyset, \langle W, \emptyset \rangle, SI', H, IP \rangle$

avec $W = ST$ avec tous ces éléments qui prennent en compte l'absence de l'émission du signal attendu et $\alpha(SI) = SI'$

Partie commune

Application neutre : On a une application sur rien, cela revient juste à rien faire.

$\langle I, S, E, ap, C, D, TL, SI, H, IP \rangle \xrightarrow{secdv4} \langle I, S, E, C, D, TL, SI, H, IP \rangle$

la machine SECD version 4 peut s'arrêter dans 3 états différents :

on a une **constante b** tels que $\langle 0, \emptyset, \emptyset, \emptyset, [M]_{secdv4}, \langle \emptyset, \emptyset \rangle, \langle \emptyset, \emptyset \rangle, \emptyset, \emptyset, 1 \rangle$
 $\xrightarrow{secdv4} \langle I, b, S, E, \epsilon, \emptyset, \langle \emptyset, \emptyset \rangle, SI, H, IP \rangle$;

on a une **abstraction function** tels que $\langle 0, \emptyset, \emptyset, [M]_{secdv4}, \emptyset, \langle \emptyset, \emptyset \rangle, \langle \emptyset, \emptyset \rangle, \emptyset, \emptyset, 1 \rangle$
 $\xrightarrow{secdv4} \langle I, \langle \langle X, C \rangle, E' \rangle, S, E, \epsilon, \emptyset, \langle \emptyset, \emptyset \rangle, SI, H, IP \rangle$;

on a une **erreur e** tels que $\langle 0, \emptyset, \emptyset, [M]_{secdv4}, \emptyset, \langle \emptyset, \emptyset \rangle, \langle \emptyset, \emptyset \rangle, \emptyset, \emptyset, 1 \rangle$
 $\xrightarrow{secdv4} \langle I, erreur_e, S, E, \epsilon, \emptyset, \langle \emptyset, \emptyset \rangle, SI, H, IP \rangle$;

7 Bibliographie

- [1] *Réactivité des systèmes coopératifs : le cas Réactive ML* de Louis Mandrel et Cédric Pasteur
- [2] *The ZINC experiment : an economical implementation of the ML language* de Xavier Leroy
- [3] *Programming Languages And Lambda Calculi* de Mathias Felleisen et Matthew Flatt
- [4] <https://www.irif.fr/~carton/Enseignement/Complexite/MasterInfo/Cours/turing.html>
- [5] https://fr.wikipedia.org/wiki/Automate_fini_déterministe
- [6] https://fr.wikipedia.org/wiki/Algorithme_déterministe