Rapport de stage
Développement d'un noyau de programmation synchrone

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

25 Mars 2019

Table des matières

Présentation du laboratoire d'accueil

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 de 3 mois. 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.

Remerciement

Avant tout développement sur mon sujet de stage, j'aimerais remercier mes deux enseignants 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.

Chapitre 1

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.

Le processus de création d'un langage de programmation passe par le questionnement sur la nécessité de sa création, sur l'utilisation que l'on va pouvoir en faire, sur l'implantation optimale. Je vais développer ce dernier point, cependant il reste vaste. On va donc se concentrer sur la méthode d'exécution du langage. Toute la partie sur le langage aura été décidée en amont par mes deux encadrants.

Le but est, comme dit plus haut, de créer une machine abstraite. Avant même de s'atteler à cette tâche, on va devoir faire une recherche sur ce qui est déjà existant niveau machines abstraites : leurs fonctionnements, leurs avantages, leurs inconvénients, etc. La totalité des machines étudiées dans cet article utilisent le λ-calcul. Étant totalement ignorant sur ce langage, il m'a fallu me mettre à niveau, cela m'a donné la possibilité de retracer ma compréhension du λ-calcul afin qu'à la fin de la lecture de la première partie le lecteur puisse comprendre autant que moi ce langage. La première partie regroupe aussi toutes les recherches préliminaires effectuées dans le but de pouvoir cerner complètement le sujet du stage et de pouvoir effectuer un travail optimal.

La deuxième partie traite du travail réalisé pour créer la machine abstraite demandée à partir d'une machine déjà existante. J'y explique les contraintes créées par l'ajout de la concurrence dans une machine abstraite en prenant chaque point de la concurrence et en montrant comment on va l'appliquer sur notre machine. On commence d'abord par une concurrence basique sans partage de valeurs ensuite on ajoute un partage de valeurs via signaux. On continue avec la récursion et les types avec le filtrage qui leurs est lié. Et enfin un petit point sur les exceptions. Une partie sera consacrée au déterminisme de la machine à travers une preuve par induction.

Pour finir, je résumerai tous ce qui a été fait durant le stage jusqu'au rendu de ce rapport ainsi que les points qui sont à changer, à développer ou encore ce qu'il reste à faire. Un petit mot sera glissé par rapport aux sémantiques intermédiaires créées et leurs implantations en OCaml. Toutes ces implantations sont retrouvables sur mon git : https ://github.com/JordanIschard/StageL3.git.

Je vous souhaite une bonne lecture et n'hésitez pas à lire les articles cités dans la bibliographie si le sujet vous intéresse.

Chapitre 2

Préliminaires

La réalisation de ce stage a nécessité une montée en compétence sur le traitement formel des langages de programmation. En particulier, l'étude des articles suivants a été nécessaire au bon déroulement du stage.

- [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 abstraites permettant de réduire les termes du λ-calcul.

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 partie sera dédiée à l'article [1] qui est, lui, complètement axée sur la programmation réactive.

2.1 Conception de langages

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 *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 *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, quelle est la meilleure implantation pour une vitesse d'exécution optimale, quelle 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 *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, il peut être soit statique, soit dynamique et dans les deux cas, on peut avoir un typage fort ou un typage faible. On va définir tout 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 c'est nécessaire. 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 pareils. 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 ? Son principe 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 1 *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* $f\ 5\ 7$

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 hautes ordres et le polymorphisme. Pour éviter ça on va préférer la curryfication malgré son exécution plus lente. Une explication précise de comment palier 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 ? Il y a différentes méthodes d'exécutions :

- **Code natif :** Cette méthode optimise la vitesse d'exécution mais c'est une approche *"blood, sweat and tears"* car c'est pas du tout trivial et spécifique à un processeur ;

- **Machine Abstraite :** Cette méthode a une bonne portabilité mais génère un code avec des redondances et n'utilise pas de façon optimale la machine même si on peut écrire un optimiseur pour chaque machine ;

- **Traduire dans un langage plus haut niveau :** Cette méthode a une bonne portabilité et est accessible simplement cependant on est dépendant d'un autre langage et on cache notre langage cible à l'utilisateur

- **Interpréteur de code de la machine abstraite :** Cette méthode est un simulateur ce qui donne une bonne portabilité mais un temps d'interprétation non-négligeable

2.2 λ-calcul : sémantique et machines abstraites

Le λ-calcul est un système formel inventé par Alonzo Church dans les années 1930, qui fonde les concepts de fonction et d'application. Le λ-calcul est le modèle le plus communément accepté du calcul séquentiel. On y manipule des expressions appelées λ-expressions, où la lettre grecque λ est utilisée pour lier une variable. On va lier une variable à une λ-expression, le composant créé est nommé *abstraction*. On peut voir une *abstraction* comme une fonction qui a un paramètre. On va revoir ce point dans l'introduction de la syntaxe et de la sémantique.

2.2.1 Le λ-calculs : syntaxe et sémantique

Les termes du λ-calcul : Il y a trois termes qui composent les λ-expressions :

1. les variables : $\{x, y...\}$ sont des λ-termes. Ce sont des variables comme on peut trouver partout dans les langages de programmations ;
2. les abstractions : $\lambda x.(v)$ est un λ-terme si x est une variable et v un λ-terme. On peut voir une abstraction comme une fonction comme dit plus haut.

   Exemple 2 *On prend la fonction toute simple f qui à y associe $y+1$. Si on revient au λ-calcul : y sera notre variable lié à λ (c'est x dans l'abstraction $\lambda x.(v)$) et $y+1$ sera la λ-expression (c'est v dans l'abstraction $\lambda x.(v)$). Quand on utilise la fonction f on va donner une valeur à y et on remplacera y par cette valeur dans $y+1$. Les λ-expressions fonctionnent de façon semblable.*

3. les applications : $(u\ v)$ est un λ-terme si u et v sont des λ-termes. L'application a un nom assez explicite, il va appliquer l'élément de droite à celui de gauche, c'est-à-dire qu'il va appliquer v à u. Cette application ce fait avec u une abstraction et v une λ-expression.

   Exemple 3 *Si on prend un exemple dans Ocaml, on a la fonction $string\_of\_int$ pour pouvoir l'utiliser on va faire $string\_of\_int$ 3. C'est exactement de la forme $(u\ v)$.*

Les règles de réduction : Maintenant que nous connaissons les termes qui composent les λ-expressions, il faut savoir comment les faire interagir entre eux. Les λ-calculs vont seulement transformer les termes, son évaluation est liée à l'application. En effet, on travaille avec des abstractions, des variables et des applications qui représentent respectivement les fonctions , les variables et les utilisations de fonctions. Quand on va donner une variable à une fonction on va l'appliquer ce qui va avoir pour impact de réduire l'expression. On évalue donc les λ-calculs par une succession de réductions des termes grâce à l'application. Il existe trois règles de **réduction générale :**

- $(\lambda X_1.M) \qquad \alpha \qquad (\lambda X_2.M[X_1 \leftarrow X_2])$ où $X_2 \notin FV(M)$
  elle sert à renommer les variables. On peut comparer cela à une réecriture d'une expression pour rendre plus lisible la lecture et moins ambigüe.
- $((\lambda X.M_1)M_2) \qquad \beta \qquad M_1[X \leftarrow M_2]$
  elle substitue une variable par un λ-terme. C'est la réduction principale, elle fait ce que j'ai expliqué plus haut avec l'*abstraction* : elle va remplacer la variable en paramètre par une λ-expression.
- $(\lambda X.(M\ X)) \qquad \eta \qquad M$ où $X \notin FV(M)$
  Cette règle traite le cas particulier d'une fonction inutile. En effet, si on a g une fonction qui à x associe $f(x)$. On aura donc $g(x) = f(x)$. La fonction g devient obsolète. Ici c'est pareil, on a $(\lambda X.(M\ X))$ quand on va appliquer une λ-expression N on va avoir $((\lambda X.(M\ X))\ N)$. Si on suit la β-réduction on va substituer X par N. On aura donc $(M\ N)$. Cela revient au même dans le cas ou on utilise l'η-réduction cependant on n'aura pas eu à faire la substitution, on gagne en rapidité.

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

On peut remarquer que dans les règles on a une forme particulière : $N[X \leftarrow M]$ avec deux λ-termes M , N et une variable X. Cette forme signifie que l'on va remplacer toutes les occurrences de X par M dans la λ-expression N. Cependant cette substitution est régie par une suite de règles qui sont les suivantes :

1. $X_1[X_1 \leftarrow M] = M$: Si on a X_1 et que l'on doit remplacer X_1 par M, on substitue X_1 par M.
2. $X_2[X_1 \leftarrow M] = X_2$ où $X_1 \neq X_2$: Si on a X_2 et que l'on doit remplacer X_1 par M, ce n'est pas la variable recherchée donc on ne fait rien.
3. $(\lambda X_1.M_1)[X_1 \leftarrow M_2] = (\lambda X_1.M_1)$
4. $(\lambda X_1.M_1)[X_2 \leftarrow M_2] = (\lambda X_3.M_1[X_1 \leftarrow X_3][X_2 \leftarrow M_2])$
   où $X_1 \neq X_2$, $X_3 \notin FV(M_2)$ et $X_3 \notin FV(M_1) \backslash X_1$
5. $(M_1\ M_2)[X \leftarrow M_3] = (M_1[X \leftarrow M_3]\ M_2[X \leftarrow M_3])$

Les deux premières règles ont été expliquées et pas les autres car on a besoin de le représenter par un exemple pour vraiment saisir le problème.

Exemple 4 *Prenons un exemple de de programme Ocaml :*

$let\ g\ x = let\ x = x + 1\ in\ x + 3;;$

On a x comme seule variable cependant il n'a pas la même valeur dans l'entièreté du programme. Si on fait g 4, on aura les variables x soulignées suivantes : let g $\underline{x}$ *= let x =* $\underline{x}$ *+ 1 in x + 3;; égales à 4 et seulement elle. Les autres sont définies par le deuxième let est sont donc "protégées" par celui-ci.*

Dans les λ-calcul c'est la même chose. On prend l'expression $((\lambda x.((\lambda x.x)\ x))\ y)$*. Si on fait une β réduction sur* $\underline{((\lambda x.((\lambda x.x)\ x))\ y)}$ *on va avoir* $((\lambda z.z)\ y)$*. Pour éviter la confusion ce que l'on va faire c'est renommer avant pour éviter les conflits. C'est ce que l'on fait dans la règle 4.*

Exemple 5 *L'α-réduction :* $(\lambda x.(x\ y))$*, On va renommer x par a.* $(\lambda x.(x\ y)) \to_\alpha (\lambda a.(x\ y)[x \leftarrow a]) \iff (\lambda a.(a\ y))$

Exemple 6 *La β-réduction :* $((\lambda x.(x\ y))\ f)$*, On va substituer x par f.* $((\lambda x.(x\ y))\ f) \to_\beta ((x\ y)[x \leftarrow f]) \iff (f\ y)$

Exemple 7 *L'eta-réduction :* $(\lambda x.((\lambda z.(z\ z))\ x)$*, On va garder* $(\lambda z.(z\ z))$ *.* $(\lambda x.((\lambda z.(z\ z))\ x) \to_\eta \lambda z.(z\ z)$

Exemple 8 *La* $\boldsymbol{n}$*-réduction : On va prendre la λ-expression suivante :* $((\lambda f.\lambda g.\lambda x.(f\ x\ (g\ x))\ (\lambda x.\lambda y.x))\ (\lambda x.\lambda y.x))$*. Cette exemple est assez lourd visuellement mais il faut d'abord comprendre avec avant d'alléger la syntaxe. Pour aider on va indicer les parenthèses pour s'y retrouver.*

$(^1(^2\lambda f.\lambda g.\lambda x.(^3(^4f\ x)^4\ (^5g\ x)^5)^3\ (^6\lambda x.\lambda y.x)^6)^2\ \underline{(^7\lambda x.\lambda y.x)^7})^1$

Dans un premier temps, on a renommé x et y grâce à α-réduction pour ne pas se mélanger

$\to^\alpha_{\boldsymbol{n}}\ (^1(^2\lambda f.\lambda g.\lambda x.(^3(^4f\ x)^4\ (^5g\ x)^5)^3\ \underline{(^6\lambda x.\lambda y.x)^6})^2\ (^7\lambda a.\lambda b.a)^7)^1$

On a fait de même avec le second terme

$\to^\alpha_{\boldsymbol{n}}\ (^1\underline{(^2\lambda f.\lambda g.\lambda x.(^3(^4f\ x)^4\ (^5g\ x)^5)^3\ (^6\lambda c.\lambda d.c)^6)^2}\ (^7\lambda a.\lambda b.a)^7)^1$

On a subsitué f par $(\lambda c.\lambda d.c)$

$\to^\beta_{\boldsymbol{n}}\ \underline{(^1(^2\lambda g.\lambda x.(^3(^4(^6\lambda c.\lambda d.c)^6\ x)^4\ (^5g\ x)^5)^3)^2\ (^7\lambda a.\lambda b.a)^7)^1}$

On a subsitué g par $(\lambda a.\lambda b.a)$

$\to^\beta_{\boldsymbol{n}}\ \lambda x.(^3(^4(^6\lambda c.\lambda d.c)^6\ x)^4\ \underline{(^5(^7\lambda a.\lambda b.a)^7\ x)^5})^3$

On a subsitué a par x

$\to^\beta_{\boldsymbol{n}}\ \lambda x.(^3\underline{(^4(^6\lambda c.\lambda d.c)^6\ x)^4}\ x)^3$

On a subsitué c par x

$\to^\beta_{\boldsymbol{n}}\ \lambda x.(^3x\ x)^3$

Le résultat est $\lambda x.(x\ x)$*. On peut voir que l'on se perd facilement avec les parenthèse dans tous les sens. Pour éviter cela, des simplifications d'écritures existent.*

Simplification : Afin d'alléger l'écriture en enlevant des parenthèses, on a une succession de règles de priorité :

- Application associative à gauche : $M1\ M2\ M3\ = ((M1\ M2)M3)$
- Application prioritaire par rapport à l'abstraction : $\lambda X.M1\ M2\ = \lambda X.(M1\ M2)$
- Les abstractions consécutives peuvent être regroupées : $\lambda XYZ.M\ = (\lambda X.(\lambda Y.(\lambda Z.M)))$

Exemple 9 *Voici un petit exemple comparatif.*

Version écriture lourde	*Version écriture allégée*
$((\lambda x.((\lambda z.z)\ x))\ \underline{(\lambda x.x)})$	$(\lambda x.(\lambda z.z)\ x)\ \underline{\lambda x.x}$
$\to^\alpha_n\ (\underline{(\lambda x.((\lambda z.z)\ x))}\ (\lambda y.y))$	$\to^\alpha_n\ \underline{(\lambda x.(\lambda z.z)\ x)}\ \lambda y.y$
$\to^\eta_n\ \underline{((\lambda z.z)\ (\lambda y.y))}$	$\to^\eta_n\ \underline{(\lambda z.z)\ \lambda y.y}$
$\to^\beta_n\ (\lambda y.y)$	$\to^\beta_n\ \lambda y.y$

2.2.2 Forme normale et stratégie de réduction

Forme normale Comment peut-on savoir quand une expression est réduite au maximum ? On peut se dire que l'on a réduit une expression au maximum quand on ne peut plus appliquer de réduction. Or l'α-réduction est presque toujours applicable. On va se focaliser sur les deux autres réductions : La β et l'η réduction. On en ressort la règle suivante : *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.

Certaines λ-expressions n'ont pas de forme normale. On va voir cela sur un exemple.

Exemple 10 *L'expression $((\lambda x.x\ x)\ (\lambda x.x\ x))$ n'a pas de forme normale, elle va boucler indéfiniment.*

$\underline{((\lambda x.x\ x)\ (\lambda x.x\ x))}$
$\rightarrow_\beta\ \underline{((\lambda x.x\ x)\ (\lambda x.x\ x))}$
$\rightarrow_\beta\ \underline{((\lambda x.x\ x)\ (\lambda x.x\ x))}$
— ...
$\rightarrow_\beta\ \underline{((\lambda x.x\ x)\ (\lambda x.x\ x))}$

Le problème de la boucle infinie de réduction est aussi possible sur une λ-expression qui a une forme normale si on applique une "mauvaise" réduction.

Exemple 11 *Par exemple, si on prend l'expression $((\lambda x.\lambda y.y)((\lambda x.xx)\ (\lambda x.xx))\ z)$.*
Si on choisit de réduire $((\lambda x.\lambda y.y)\underline{((\lambda x.xx)\ (\lambda x.xx))}\ z)$ on va rentrer dans une boucle infinie.
Mais si on décide de $\underline{((\lambda x.\lambda y.y)((\lambda x.xx)\ (\lambda x.xx))\ z)}$ cela reviendra à $((\lambda y.y)\ z)$ ce qui donnera z.

Pour régler se problème on va utiliser **une stratégie de réduction**.

Stratégie de réduction La question de l'ordre dans laquelle il faut réduire l'expression se pose. En effet si on prend l'exemple suivant : $(\lambda x.x\ x)\ ((\lambda y.y)\ (\lambda z.z))$. On a deux possibilités de réduction :

1. $\underline{(\lambda x.x\ x)\ ((\lambda y.y)\ (\lambda z.z))} \rightarrow_\beta ((\lambda y.y)\ (\lambda z.z))\ ((\lambda y.y)\ (\lambda z.z))$
2. $(\lambda x.x\ x)\ \underline{((\lambda y.y)\ (\lambda z.z))} \rightarrow_\beta (\lambda x.x\ x)\ (\lambda z.z)$

Pour palier à ce problème on va définir des règles qui vont donner un ordre précis de réduction à faire. L'idée va être d'appliquer la réduction la plus large d'abord puis la plus à gauche. Cela va permettre d'éviter l'évaluation de sous-expressions inutiles. Voici les règles qui régissent la stratégie de réduction :

- $M \longrightarrow_{\bar{n}} N$ si $M\ \beta\ N$: N est une réduction de M si on peut β réduire M en N.
- $M \longrightarrow_{\bar{n}} N$ si $M\ \eta\ N$: N est une réduction de M si on peut η réduire M en N.
- $(\lambda X.M) \longrightarrow_{\bar{n}} (\lambda X.N)$: $(\lambda X.N)$ est une réduction de $(\lambda X.M)$ si on a $M\ \beta\ N$ ou $M\ \eta\ N$.
- $(M\ N) \longrightarrow_{\bar{n}} (M'\ N)$ si $M \longrightarrow_{\bar{n}} M'$ et $\forall\ L$, $(M\ N)\ \beta\ L$ impossible et $(M\ N)\ \eta\ L$ impossible : $(M'\ N)$ est une réduction de $(M\ N)$ si il existe une réduction pour M et si on ne peut pas appliquer une réduction sur $(M\ N)$.
- $(M\ N) \longrightarrow_{\bar{n}} (M\ N')$ 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 : $(M\ N')$ est une réduction de $(M\ N')$ si M est en forme normale, si il existe une réduction pour N et si on ne peut pas appliquer une réduction sur $(M\ N)$.

Cette suite de règles permet d'être sûr d'avoir une forme normale. Cependant elle ne suit pas forcément le chemin optimal cela créer donc une lenteur.

Exemple 12 *Si on prend l'expression suivante : $(\lambda x.(x\ x))\ ((\lambda y.y)\ (\lambda z.z))$. On va faire deux développements, un qui suit les règles de la stratégie de réduction et l'autre que l'on va faire par nous-même en regardant ce qui nous avantages le plus.*

Version qui suit la stratégie de réduction

$\underline{(\lambda x.(x\ x))\ ((\lambda y.y)\ (\lambda z.z))}$
$\rightarrow^\beta_n\ \underline{((\lambda y.y)\ (\lambda z.z))}\ ((\lambda y.y)\ (\lambda z.z))$
$\rightarrow^\beta_n\ \underline{(\lambda z.z)\ ((\lambda y.y)\ (\lambda z.z))}$
$\rightarrow^\beta_n\ \underline{((\lambda y.y)\ (\lambda z.z))}$
$\rightarrow^\beta_n\ (\lambda z.z)$

Version personnelle

$(\lambda x.(x\ x))\ \underline{((\lambda y.y)\ (\lambda z.z))}$
$\rightarrow^\beta_n\ \underline{(\lambda x.(x\ x))\ (\lambda z.z)}$
$\rightarrow^\beta_n\ \underline{(\lambda z.z)\ (\lambda z.z)}$
$\rightarrow^\beta_n\ (\lambda z.z)$

J'ai résussi à atteindre la forme normale plus vite sans suivre la stratégie de réduction.

Le faite de pouvoir atteindre le même résultat avec deux suites de réductions différentes vient de la propriété du diamant.

Théorème (Diamond Property pour $\twoheadrightarrow_n$) : Si $L \twoheadrightarrow_n M$ et $L \twoheadrightarrow_n N$ alors il existe une expression L' telle que $M \twoheadrightarrow_n L'$ et $N \twoheadrightarrow_n L'$.

Stratégie d'évaluation Pour évaluer un langage, on utilise ce qui s'appelle une **stratégie d'évaluation**, ici on parlait de stratégie de réduction. Cette stratégie explique quand les arguments d'une fonction sont évalués. Je vous présente les 2 stratégies qui nous intéressent :

1. **l'appel par nom :** pour une fonction f donnée, on évalue chaque argument quand on en a besoin, c'est-à-dire que l'on n'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 elle est plus lente que celle présentée après car on doit réévaluer les arguments à chaque fois. Elle fait partie du groupe des stratégies d'évaluations non strictes, c'est-à-dire qu'il n'évalue pas forcément la fonction en entier.

   Exemple 13 *Pour une fonction fst telle 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 donnée, on évalue ses arguments avant d'évaluer la fonction.
   Cette stratégie fait partie du groupe des stratégies d'évaluations strictes, c'est-à-dire qu'il évalue forcément la fonction en entier.

   Exemple 14 *Pour une fonction fst telle que pour deux arguments x et y on retourne le 1er. Si on a : $fst(3+4, 5/5)$ on va évaluer $3+4$. 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 œuvre dans un langage. En effet, on a la garantit que l'élément que l'on va appliquer est réduit au maximum et que plus aucun traitement ne sera à faire sur lui ce qui réduit considérablement le nombre de règle qui va régir le langage. Cependant on va perdre en vitesse d'exécution ce qui est dérangeant mais que l'on va préférer à un recalcule de chaque élément.

Exemple 15 *Si on reprend l'expression $(\lambda x.(x\ x))\ ((\lambda y.y)\ (\lambda z.z))$. Un appel par nom va faire la chose suivante :*

$\underline{(\lambda x.(x\ x))\ ((\lambda y.y)\ (\lambda z.z))}$
$\rightarrow_n^\beta \underline{((\lambda y.y)\ (\lambda z.z))}\ ((\lambda y.y)\ (\lambda z.z))$
$\rightarrow_n^\beta \underline{(\lambda z.z)\ ((\lambda y.y)\ (\lambda z.z))}$
$\rightarrow_n^\beta \underline{((\lambda y.y)\ (\lambda z.z))}$
$\rightarrow_n^\beta (\lambda z.z)$

On ne va pas traiter tout de suite $((\lambda y.y)\ (\lambda z.z))$ ce qui va nous obliger à le traiter deux fois. Alors que si on réduit avec la stratégie de l'appel par valeur, on va avoir :

$(\lambda x.(x\ x))\ \underline{((\lambda y.y)\ (\lambda z.z))}$
$\rightarrow_n^\beta \underline{(\lambda x.(x\ x))\ (\lambda z.z)}$
$\rightarrow_n^\beta \underline{(\lambda z.z)\ (\lambda z.z)}$
$\rightarrow_n^\beta (\lambda z.z)$

On va voir un exemple de langage utilisant la stratégie de l'appel par valeur : **ISWIM**.

2.2.3 ISWIM

ISWIM est un langage impératif à noyau fonctionnel ; de fait, c'est une syntaxe proche du λ-calcul à laquelle sont ajoutées des variables mutables et des définitions. Grâce aux λ-calculs, 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 du λ-calcul.

$M, N, L, K =$

les termes des λ-calculs :

$| \ X$

$| \ (\lambda X.M)$

$| \ (M \ N)$

les nouveaux termes :

$| \ b$: une constante

$| \ (o^n \ M \ ... \ N)$ 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 le λ-calcul avec deux ajouts qui sont les suivants :

— $b[X \longleftarrow M] = b$

— $(o^n \ M_1 \ ... \ M_n)[X \longleftarrow M] = (o^n \ M_1[X \longleftarrow M] \ ... \ M_n[X \longleftarrow M])$

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

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

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

Cependant l'η et l'α réduction ne sont plus vues 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 telle que **n**-réduction $= \beta_v \cup \delta$

Exemple 16 *Voici un exemple de* ***n****-réduction. On va prendre l'expression suivante :* $(\lambda f x.(f \ x) \ \lambda y.(+ \ y \ y) \ \ulcorner 1 \urcorner)$.

$\underline{(\lambda f x.(f \ x) \ \lambda y.(+ \ y \ y)} \ \ulcorner 1 \urcorner)$

On a substitué f *par* $\lambda y.(+ \ y \ y)$

$\rightarrow_n^\beta \ \underline{(\lambda x.(\lambda y.(+ \ y \ y) \ x) \ \ulcorner 1 \urcorner)}$

On a substitué x *par* $\ulcorner 1 \urcorner$

$\rightarrow_n^\beta \ \underline{(\lambda y.(+ \ y \ y) \ \ulcorner 1 \urcorner)}$

On a substitué y *par* $\ulcorner 1 \urcorner$

$\rightarrow_n^\beta \ \underline{(+ \ \ulcorner 1 \urcorner \ \ulcorner 1 \urcorner)}$

On applique l'opérateur $+$ *sur* $\ulcorner 1 \urcorner \ \ulcorner 1 \urcorner$

$\rightarrow_n^\delta \ \ulcorner 2 \urcorner$

Le résultat est $\ulcorner 2 \urcorner$.

Vous pouvez retrouver un exemple d'implantation de ce langage en Ocaml.

2.2.4 Machines abstraites

CC Machine : CC vient des termes **Control string** et **Context** qui représentent 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 prend 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 [ ] dans notre expression.

Basiquement la machine va évaluer l'expression comme ceci :

- Quand on a une application $(M\ N)$:
  - la machine évalue M et le garde dans le contexte $([\ ]\ N)$.
  - elle remet la sous-expression M évaluée, soit $M \twoheadrightarrow_{cc} V$, dans l'application ce qui donnera $(V\ N)$
  - elle évalue N et le garde dans le contexte $(V\ [\ ])$
  - elle remet la sous-expression N évaluée, soit $N \twoheadrightarrow_{cc} U$, dans l'application ce qui donnera $(V\ U)$
  - elle évalue $(V\ U)$
- 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 $(o^n\ M...N)$: c'est le même principe que l'application
  - la machine évalue M et le garde $(o^n\ []...N)$.
  - elle remet la sous-expression M évaluée, soit $M \twoheadrightarrow_{cc} V$, dans l'opération ce qui donnera $(o^n\ V...N)$
  - etc jusqu'à avoir $(o^n\ V...U)$ pour pouvoir évaluer l'opération

Les règles définies pour cette machine sont les suivantes :

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

La machine peut s'arrêter dans trois états différents :

- Soit on a une **constante** telle que $\langle M, []\rangle \twoheadrightarrow_{cc} \langle b, []\rangle$;
- Soit on a une **fonction** telle que $\langle M, []\rangle \twoheadrightarrow_{cc} \langle \lambda X.N, []\rangle$;
- Sinon on a un **état inconnu** soit une **erreur**.

Exemple 17 *Voici un exemple de la machine CC pour l'expression :* $(((\lambda f.\lambda x.f\ x)\ \lambda y.(+\ y\ y))\ \ulcorner 1 \urcorner)$.

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

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**).

Dans la machine CC, on peut réunir :

$(5)_{cc}\ \langle V, E[([]\ N)]\rangle \quad \longmapsto_{cc} \quad \langle (V\ N), E\rangle$
$(2)_{cc}\ \langle (V\ N), E\rangle \quad \longmapsto_{cc} \quad \langle N, E[(V\ [])]\rangle$
$\rightarrow$ sont combinées dans la règle $(4)_{scc}\ \langle V, E[[]\ N)]\rangle \quad \longmapsto_{scc} \quad \langle N, E[(V\ [])]\rangle$

$(4)_{cc}\ \langle V, E[(U\ [])]\rangle \quad \longmapsto_{cc} \quad \langle (U\ V), E\rangle$
$(3)_{cc}\ \langle ((\lambda X.M)\ V), E\rangle \quad \longmapsto_{cc} \quad \langle M[X \longleftarrow V], E\rangle$
$\rightarrow$ sont combinées dans la règle $(3)_{scc}\ \langle V, E[((\lambda X.M)\ [])]\rangle \quad \longmapsto_{scc} \quad \langle M[X \leftarrow V], E\rangle$

$(8)_{cc}\ \langle V, E[(o^n\ V_1...V_i\ []\ N\ ...)]\rangle \quad \longmapsto_{cc} \quad \langle (o^n\ V_1...V_i\ V\ N\ ...), E\rangle$
$(7)_{cc}\ \langle (o^n\ b_1...b_n), E\rangle \quad \longmapsto_{cc} \quad \langle V, E\rangle$ avec $V = \delta(o^n, b_1...b_n)$
$\rightarrow$ sont combinées dans la règle $(5)_{scc}\ \langle b, E[(o^n, b_1, ...b_i, [])]\rangle \quad \longmapsto_{scc} \quad \langle V, E\rangle\ avec\ \delta(o^n, b_1, ...b_i, b) = V$

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

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

- Soit on a une **constante b** telle que $\langle M, []\rangle \twoheadrightarrow_{scc} \langle b, []\rangle$;
- Soit on a une **fonction** telle que $\langle M, []\rangle \twoheadrightarrow_{scc} \langle \lambda X.N, []\rangle$;
- Sinon on a un **état inconnu** soit une **erreur**.

Exemple 18 *Voici un exemple de la machine SCC pour l'expression :* $(((\lambda f.\lambda x.f\ x)\ \lambda y.(+\ y\ y))\ \ulcorner 1\urcorner)$.

SCC : $\langle (((\lambda f.\lambda x.f\ x)\ \lambda y.(+\ y\ y))\ \ulcorner 1\urcorner), []\rangle$
> (1)
SCC : $\langle ((\lambda f.\lambda x.f\ x)\ \lambda y.(+\ y\ y)), [([]\ \ulcorner 1\urcorner)]\rangle$
> (1)
SCC : $\langle (\lambda f.\lambda x.f\ x), [([]\ \ulcorner 1\urcorner), ([]\ (\lambda y.(+\ y\ y)))]\rangle$
> (4)
SCC : $\langle (\lambda y.(+\ y\ y)), [([]\ \ulcorner 1\urcorner), ((\lambda f.\lambda x.f\ x)\ [])]\rangle$
> (3)
SCC : $\langle (\lambda x.f\ x)[f \leftarrow (\lambda y.(+\ y\ y))], [([]\ \ulcorner 1\urcorner)]\rangle$
SCC : $\langle (\lambda x.(\lambda y.(+\ y\ y))\ x), [([]\ \ulcorner 1\urcorner)]\rangle$
> (4)
SCC : $\langle \ulcorner 1\urcorner, [((\lambda x.(\lambda y.(+\ y\ y))\ x)\ [])]\rangle$
> (3)
SCC : $\langle ((\lambda y.(+\ y\ y))\ x)[x \leftarrow \ulcorner 1\urcorner], []\rangle$
SCC : $\langle ((\lambda y.(+\ y\ y))\ \ulcorner 1\urcorner, []\rangle$
> (1)
SCC : $\langle (\lambda y.(+\ y\ y)), [([]\ \ulcorner 1\urcorner)]\rangle$
> (4)
SCC : $\langle \ulcorner 1\urcorner, [(\lambda y.(+\ y\ y))\ []]\rangle$
> (3)
SCC : $\langle (+\ y\ y)[y \leftarrow \ulcorner 1\urcorner], []\rangle$
SCC : $\langle (+\ \ulcorner 1\urcorner\ \ulcorner 1\urcorner), []\rangle$
> (2)
SCC : $\langle \ulcorner 1\urcorner, (+\ []\ \ulcorner 1\urcorner)\rangle$
> (6)
SCC : $\langle \ulcorner 1\urcorner, (+\ \ulcorner 1\urcorner\ [])\rangle$
> (5)
SCC : $\langle \ulcorner 2\urcorner, []\rangle$

CK Machine Les machines CC et SCC cherchent toujours à traiter l'élément le plus à gauche , c'est-à-dire que si on a une application on va en créer une intermédiaire dans le contexte avec un trou et traiter 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 transitions dépendent directement de la forme du premier 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 de 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 κ = 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** notés précédemment [ ].

Cette machine agit 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 l'évalue et on la stocke 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 éléments de l'opération qui sont en attente d'être traités et quand c'est fait on évalue l'opération.

Les règles qui définissent la machine CK sont les suivantes :

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

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

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

Exemple 19 *Voici une partie d'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 les Annexes.*

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

CEK Machine Pour toutes les machines vues jusqu'à présent la β-réduction était appliquée immédiatement. Cela coûte cher surtout quand l'expression est 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. Cet environnement est une fonction qui pour une variable va retourner une expression.

On a alors :

ε = une fonction $\{\langle X, c\rangle, ...\}$ 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 $\overline{\kappa}$ et défini par :

$\overline{\kappa}$ = mt
| $\langle fun, v, \overline{\kappa}\rangle$
| $\langle arg, c, \overline{\kappa}\rangle$
| $\langle opd, \langle v, ..., v, o^n\rangle, \langle c, ...\rangle, \overline{\kappa}\rangle$

Les règles qui définissent la machine CEK sont les suivantes :

1. $\langle\langle(M\ N), \varepsilon\rangle, \overline{\kappa}\rangle \longmapsto_{cek} \langle\langle M, \varepsilon\rangle, \langle arg, \langle N, \varepsilon\rangle, \overline{\kappa}\rangle\rangle$
2. $\langle\langle X, \varepsilon\rangle, \overline{\kappa}\rangle \longmapsto_{cek} \langle c, \overline{\kappa}\rangle$ avec $\varepsilon(X) = c$
3. $\langle\langle V, \varepsilon\rangle, \langle fun, \langle(\lambda X1.M), \varepsilon'\rangle, \overline{\kappa}\rangle\rangle \longmapsto_{cek} \langle\langle M, \varepsilon'[X1 \leftarrow \langle V, \varepsilon\rangle]\rangle, \overline{\kappa}\rangle$ si $V \notin X$
4. $\langle\langle V, \varepsilon\rangle, \langle arg, \langle N, \varepsilon'\rangle, \kappa\rangle\rangle \longmapsto_{cek} \langle\langle N, \varepsilon'\rangle, \langle fun, \langle V, \varepsilon\rangle, \overline{\kappa}\rangle\rangle$ si $V \notin X$
5. $\langle\langle(o^n\ M\ N...), \varepsilon\rangle, \overline{\kappa}\rangle \longmapsto_{cek} \langle\langle M, \varepsilon\rangle, \langle opd, \langle o^n\rangle, \langle\langle N, \varepsilon\rangle, ...\rangle, \overline{\kappa}\rangle\rangle$
6. $\langle\langle b, \varepsilon\rangle, \langle opd, \langle\langle b_i, \varepsilon_i\rangle, ...\langle b_1, \varepsilon_1\rangle, o^n\rangle, \langle\rangle, \overline{\kappa}\rangle\rangle \longmapsto_{cek} \langle\langle V, \emptyset\rangle, \overline{\kappa}\rangle$ avec $\delta(o^n, b_1, ...b_i, b) = V$
7. $\langle\langle V, \varepsilon\rangle, \langle opd, \langle v', ...o^n\rangle, \langle\langle N, \varepsilon'\rangle, c, ...\rangle, \overline{\kappa}\rangle\rangle \longmapsto_{cek} \langle\langle N, \varepsilon'\rangle, \langle opd, \langle\langle V, \varepsilon\rangle, v', ...o^n\rangle, \langle c, ...\rangle, \overline{\kappa}\rangle\rangle$ si $V \notin X$

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

- Soit on a une **constante b** telle que $\langle\langle M, \emptyset\rangle, mt\rangle \twoheadrightarrow_{cek} \langle\langle b, \varepsilon\rangle, mt\rangle$;
- Soit on a une **fonction** telle que $\langle\langle M, \emptyset\rangle, mt\rangle \twoheadrightarrow_{cek} \langle\langle \lambda X.N, \varepsilon\rangle, mt\rangle$;
- Sinon on a un **état inconnu** soit une **erreur**.

Exemple 20 *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 : $\langle\langle(((\lambda f.\lambda x.f\ x)\ \lambda y.(+\ y\ y))\ \ulcorner 1\urcorner), \emptyset\rangle, mt\rangle$

...

On a deux parties qui appliquent une abstraction.

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

> (3)

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

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

> (4)

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

> (3)

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

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

> (1)

...

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

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

> (2)

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

...

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

L'exemple complet peut être retrouvé dans les Annexes.

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 tels que 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 :

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

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 simples comme des constantes, des variables, des fonctions et des commandes qui permettent de savoir ce que doit faire la machine. Voici les règles de conversion :

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

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 une abstraction, elle crée ce qu'on appelle une fermeture. C'est-à-dire qu'on va lier l'abstraction avec l'environnement. Ce processus permet de mettre en attente l'évaluation de l'expression présente 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 met en attente l'expression principale pour se concentrer sur une sous-expression. Pour cela, la machine fait une sauvegarde d'elle-même avant et la stocke dans le dépôt.
- Si elle a une commande $prim_{o^n}$, on effectue l'opération sur les n premiers éléments de la pile.
- Si elle a rien :
  - elle a une sauvegarde dans le dépôt, elle reprend cette sauvegarde.
  - elle n'a pas de sauvegarde, elle a fini son travail.

Les règles qui définissent la machine SECD sont les suivantes :

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

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

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

Exemple 21 *Voyons sur un exemple les avantages de cette machine. Un exemple étant assez long, on s'intéresse à certaines parties. L'exemple complet est disponible dans les Annexes.*
On teste 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 : $\langle \epsilon, \emptyset, \langle f, \langle x, f\ x\ ap\rangle\rangle\ \langle y, y\ y\ prim_+\rangle\ ap\ \ulcorner 1 \urcorner\ ap, \epsilon\rangle$
...
On voit comment l'application est traitée
SECD : $\langle\langle\langle y, y\ y\ prim_+\rangle, \emptyset\rangle\ \langle\langle f, \langle x, f\ x\ ap\rangle\rangle, \emptyset\rangle, \emptyset, ap\ \ulcorner 1 \urcorner\ ap, \epsilon\rangle$
> (4)
SECD : $\langle \epsilon, \emptyset[f \leftarrow \langle\langle y, y\ y\ prim_+\rangle, \emptyset\rangle], \langle x, f\ x\ ap\rangle, \langle \epsilon, \emptyset, \ulcorner 1 \urcorner\ ap, \epsilon\rangle\rangle$
SECD : $\langle \epsilon, \{\langle f, \langle\langle y, y\ y\ prim_+\rangle, \emptyset\rangle\rangle\}, \langle x, f\ x\ ap\rangle, \langle \epsilon, \emptyset, \ulcorner 1 \urcorner\ ap, \epsilon\rangle\rangle$
> (3)

On voit l'intérêt de la sauvegarde.
SECD : $\langle\langle\langle x, f\ x\ ap\rangle, \{\langle f, \langle\langle y, y\ y\ prim_+\rangle, \emptyset\rangle\rangle\}\rangle, \{f, \langle\langle y, y\ y\ prim_+\rangle, \emptyset\rangle\}, \emptyset, \langle \epsilon, \emptyset, \ulcorner 1 \urcorner\ ap, \epsilon\rangle\rangle$
> (5)
SECD : $\langle\langle\langle x, f\ x\ ap\rangle, \{\langle f, \langle\langle y, y\ y\ prim_+\rangle, \emptyset\rangle\rangle\}\rangle, \emptyset, \ulcorner 1 \urcorner\ ap, \epsilon\rangle$
...
On voit comment est gérée l'opération
SECD : $\langle \ulcorner 1 \urcorner\ \ulcorner 1 \urcorner, \{\langle y, \ulcorner 1 \urcorner\rangle\}, prim_+, \langle \epsilon, \{\langle f, \langle\langle y, y\ y\ prim_+\rangle, \emptyset\rangle\rangle, \langle x, \ulcorner 1 \urcorner\rangle\}, \emptyset, \langle \epsilon, \emptyset, \epsilon, \epsilon\rangle\rangle\rangle$
> (6)
SECD : $\langle \ulcorner 2 \urcorner, \{\langle y, \ulcorner 1 \urcorner\rangle\}, \emptyset, \langle \epsilon, \{\langle f, \langle\langle y, y\ y\ prim_+\rangle, \emptyset\rangle\rangle, \langle x, \ulcorner 1 \urcorner\rangle\}, \emptyset, \langle \epsilon, \emptyset, \epsilon, \epsilon\rangle\rangle\rangle$
> (5)
SECD : $\langle \ulcorner 2 \urcorner, \{\langle f, \langle\langle y, y\ y\ prim_+\rangle, \emptyset\rangle\rangle, \langle x, \ulcorner 1 \urcorner\rangle\}, \emptyset, \langle \epsilon, \emptyset, \epsilon, \epsilon\rangle\rangle$
> (5)
SECD : $\langle \ulcorner 2 \urcorner, \emptyset, \epsilon, \epsilon\rangle$

2.3 Programmation réactive

Explication formelle En informatique, la programmation réactive est un paradigme de programmation déclarative qui concerne les flux de données et la propagation du changement. Avec ce paradigme, il est possible d'exprimer facilement des flux de données statiques (par exemple, des tableaux) ou dynamiques (par exemple, des émetteurs d'événements) et également de signaler l'existence d'une dépendance inférée au sein du modèle d'exécution associé, ce qui facilite la propagation automatique des données modifiées.

Historique La programmation réactive a été portée jusqu'à nos jours à travers différents langages, un des plus vieux est l'**ESTEREL**. ESTEREL est un langage de programmation développé dans les années 80 par une équipe de chercheurs des Mines de Paris et de l'INRIA. Il a initialement été conçu pour la programmation de robots industriels et pour la simulation de circuit électrique. La programmation réactive s'est diversifiée en s'implantant soit dans des langages déjà existant comme le C, le SCHEME ou encore le JAVA via l'extension **Fair Threads** (présentée dans [4] datant de 2001) soit en créant un tout nouveau langage de programmation comme le **SL** (présenté dans [5] datant de 1995) ou encore **Icobj** (présenté dans [6] datant de 1996).

Le framework **Fair Threads** est une implantation de la programmation réactive dans différents langages. Je me suis intéressé à l'implantation java. Ce langage se base sur la notion de *fair thread*, basiquement ce sont des threads qui fonctionnent par concurrence coopérative (expliquée plus bas). Cependant il y a une forme de préemption via un planificateur qui donne les mêmes droits à tous. Son implantation est déterministe et portable. On peut remarquer cependant que le bon fonctionnement de la machine va beaucoup dépendre du développeur car il va devoir rendre ces threads coopératifs.

Le langage synchrone **SL** est, comme dit plus haut, basé sur le langage ESTEREL. Son but est de régler le problème de causalité présent dans le ESTEREL. Le problème venait de savoir si oui ou non un signal est absent. Dans l'instant courant on ne peut pas être sûr qu'il ne va pas être émis donc ESTEREL se basait sur une hypothèse. Le SL règle le problème en attendant la fin de l'instant courant pour dire si un signal est émis ou non.

Le langage **Icobj** est un langage assez atypique. En effet, il n'a pas de syntaxe car il est entièrement graphique et il possède une sémantique simple basée sur le Reactive script. De plus, il n'est pas compilé mais interprété directement. Il fonctionne comme les autres langages de programmation réactif : il peut fonctionner de façon séquentielle, parallèle ou encore dans une boucle. Il fonctionne aussi avec un système d'événement qui permet une bonne communication.

Il y a de nombreuses implantations dont je n'ai pas parlé. Si vous souhaitez plus d'informations voici le un lien des recherches de Frédéric Bussinot : http ://www-sop.inria.fr/mimosa/Frederic.Boussinot/. Ses recherches se focalisent sur la programmation réactive, tous les langages précédemment cités dans l'historique viennent de ses articles.

Le cas étudié Chaque implantations travaillent un aspect de la programmation réactive pour une utilisation spécifique. Nous n'allons pas déroger à cette remarque en travaillant sur un aspect de la programmation réactive : la **programmation réactive synchrone concurrente**. Premièrement, on va expliquer tous ces termes.

- **Synchrone** signifie que les informations seront obtenues de manière immédiate au contraire de la programmation réactive asynchrone qui attend un "instant" avant de distribuer les valeurs.

- **Concurrent** signifie que plusieurs processus vont se dérouler durant le même instant logique. Il existe deux sous-catégories à la concurrence :

  1. **Concurrence coopérative :** les processus vont régulièrement "laisser la main aux autres"
  2. **Concurrence préemptive :** le système va "donner un temps de parole"

Dans notre machine abstraite, on utilise un peu des deux principes en laissant les processus fonctionner seuls mais le système garde la possibilité de gérer les processus bloqués. On va voir à travers un exemple comment fonctionne la programmation réactive synchrone concurrente coopérative.

Le cas de Réactive ML Le langage Reactive ML utilise le modèle de programmation de concurrence coopérative. C'est-à-dire que les différents processus vont fonctionner en même temps (de façon hypothétique) tout en laissant la main pour que tout le monde puissent fonctionner.

La spécificité du Reactive ML est son analyse qui va détecter les erreurs de concurrence avant de le tester via une sémantique spéciale. L'analyse est découpée en 2 sous-analyses :

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

Cette méthode d'analyse syntaxique permet d'avoir une implantation séquentielle efficace ainsi qu'aucune création de problème de parallélisme. En revanche, la réactivité du programme est entièrement déléguée au développeur.

Le modèle de programmation réactif synchrone définit une notion de temps logique, appelé tick. On peut voir le fonctionnement du programme comme une succession de ticks qui lui permet de faire avancer ses processus. De là, on peut définir une condition nécessaire et suffisante pour vérifier la réactivité d'un programme :
Un programme est réactif si son exécution fait progresser les instants logiques.

Exemple 22 *Voici un exemple de programme qui contient une erreur de réactivité.*

```
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****. Le nouveau programme sera donc :*

```
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.         pause
8.     end
```

L'analyse étant spéciale, l'écriture du programme a des règles strict par exemple : 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 ce qui permet de vérifier la réactivité du programme très rapidement. Cette partie nous intéresse moins car on ne va pas travailler sur l'analyse mais sur la méthode d'exécution mais si cela vous intéresse vous pouvez retrouver cet article dans la bibliographie.

Chapitre 3

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 faut pouvoir incorporer les ajouts facilement. La machine SECD remplit ce critère via la composition de la chaîne de contrôle : des éléments simples et des commandes. Nous utiliserons donc la machine SECD comme base.

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

3.1 Machine réactive pure

3.1.1 Description informelle du langage

Avant tout développement sur l'implantation il faut pouvoir bien cerner le principe de la réactivité synchrone avec concurrence. Le but est de créer un "dialogue" entre plusieurs processus gérés par 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. Il représente notre processus principal. Son but est d'initier la discussion puis d'y participer.

Monsieur X va commencer a parler en introduisant des personnes, ce sont nos processus secondaires. À noter que tous les processus peuvent introduire un nombre non limité de processus. Lorsque Monsieur X va finir de parler ou lorsqu'il attendra une information pour continuer à parler, ce point est important, il va passer son tour. Il devient un processus commun. Un autre processus prend le relais 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 du traitement de la machine.

Un point qu'il faut bien comprendre est que l'absence d'une information 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. Si on reprend l'exemple précédent, si on attend une réponse d'une personne qui dort, on peut attendre longtemps. 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 , et au moins une personne attend une information, 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'aient d'arguments.

Le principe étant expliqué, nous pouvons rentrer dans les détails. L'idée est d'avoir plusieurs processus 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 processus. Ils sont ce que l'on va appeler des **threads**.

Récapitulatif : Notre machine va devoir pouvoir :

- Créer un thread
- Traiter un thread
- Passer d'un thread à un autre
- Detecter une fin d'instant logique
- Créer un signal
- Emettre un signal
- Verifier l'emission d'un signal
- Detecter une fin de fonctionnement

3.1.2 Description formelle du langage

Le Thread Un thread est l'équivalent d'un processus, il a sa propre pile d'éxécution 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.

La forme 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, en effet on n'a pas la nécessité de différencier les threads.

Le stockage 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 suite unique de transitions possibles.

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

Le fonctionnement Un petit point dessus histoire de ne pas laisser de zone d'ombre dans le fonctionnement de la machine. Les threads sont des machines SECD. Du coup lorsqu'un thread sera exécuté, il fonctionnera comme une machine SECD classique avec les ajouts que l'on est entrain d'expliquer et il se finira comme une machine SECD aussi avec un autre état expliqué et donné plus tard dans la sémantique.

Le roulement Quand un thread fini son exécution, on prend un nouveau thread. On a définit une file de threads. Lorsque le thread courant a fini, on va aller chercher le thread en tête de la file etc. Lorsqu'on prend un nouveau thread, le thread courant est effacé. On aura une règle ressemblant à la suivante :
$\langle\langle S, E, \epsilon, \emptyset\rangle, \langle S', E', C, D\rangle\ TL\rangle \longrightarrow \langle\langle S', E', C, D\rangle, TL\rangle$

La création La création d'un thread va se faire dans la chaîne de contrôle du thread en cours d'exécution. Or un thread est , comme définit plus haut, de la forme $\langle S, E, C, D\rangle$. Il faut donc savoir quoi mettre dans chaque élément. On va debrieffer rapidement :

- la pile d'exécution S, elle indépendante pour chaque thread et de plus il y a dans la pile d'exécution seulement ce que le thread est entrain de traiter or il est en création donc pas encore actif donc S sera vide.
- l'environnement E, il contient toutes les substitutions faisables par le thread courant, il y a potentiellement des substitutions que notre thread en cours de création va avoir besoin.

  Exemple 23 *L'exemple le plus concret vient de l'initialisation d'un signal. On va le voir plus tard mais l'initialisation va donner une substitution pour le thread courant. Or un signal est ,dans notre version final, initialisé pour tout le monde donc tout le monde doit pouvoir y accèder. On a donc besoin de l'environnement du thread courant ou un environnement général et un environnement privé. On va se contenter de prendre l'environnement du thread courant dans un premier temps.*

- la chaîne de contrôle C, on va devoir prendre une portion de la chaîne de contrôle du thread courant pour l'insérer dans le thread en cours de création.
  (a) La première façon de faire est de délimiter une portion via deux mots-clés que j'ai nommé *bspawn* et *espawn*. Quand la machine va tomber sur le *bspawn* elle sait qu'elle va devoir prendre dans la chaîne de contrôle jusqu'à trouver le mot-clé *espawn*. Cette façon de faire a été utilisée dans les premières versions de la machine cependant elle a un défaut majeur : ce ne sont pas des commandes et on ne travaille pas dans la pile d'exécution mais dans la chaîne de contrôle. C'est décalée par rapport au fonctionnement global de la machine.

  (b) Le seconde façon de faire m'a été proposée par l'un de mes encadrants. On va utiliser les abstractions à notre avantage. La règle de l'abstraction est la suivante :

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

  On voit que la machine ne tente pas de traiter la chaîne de contrôle C' car elle est "protégée" par l'abstraction. On va utiliser ce principe et l'additionner à une commande que l'on nomme *spawn*. Elle va permettre à la machine de comprendre que la fermeture contenue dans la pile d'exécution va servir à créer un nouveau thread.
- le dépôt D, Il n'y a pas à se poser de quetions pour lui. Il est vide au départ de chaque thread car il n'a aucune sauvegarde à garder d'une potentielle application.

On arrive à un prototype de règle assez clair : $\langle\langle\langle\langle X, C'\rangle, E'\rangle\ S, E, spawn\ C, D\rangle, TL\rangle \longrightarrow \langle\langle S, E, C, D\rangle, TL\ \langle\epsilon, E', C', \emptyset\rangle\rangle$

Le 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.

La forme 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**.

Le stockage 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, à la partie **L'initialisation**. 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.

L'initialisation La base est de pouvoir créer un signal ou plutôt de l'initialiser. Comment le fonctionnement de l'initialisation diffère par rapport au champ d'action proposé ?

On se retrouve avec trois possibilités de règle pour initialiser un signal :

- On initialise un signal pour une partie de la chaîne de contrôle : Cela revient à devoir se focaliser sur une partie de la chaîne de contrôle qui obtient le droit d'émettre. Cela ressemble fortement à une application qui, je le rappelle, à la règle suivante :

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

  On va donc devoir créer une forme reconnaissable par la machine. On va s'inspirer des abstractions et créer un couple $\langle s, C\rangle$ avec le signal s et la chaîne de contrôle où le signal s est initialisé pour C. Quand on tombera sur ce couple on saura qu'il faut initialiser un signal. On aura donc un règle de la forme :

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

  On remarque que l'identifiant du signal est choisie par l'utilisateur. Cette règle fonctionne cependant plusieurs points sont problématiques.

  1. L'identifiant ne devrait pas choisie par l'utilisateur.
  2. La forme $\langle s, C'\rangle$ ressemble trop à une abstraction et peut être confondu si s à la même forme q'une variable X.
  3. L'information de l'initialisation est stockée dans l'environnement mais l'émission est général et est donc stockée dans la liste des signaux qui sera de la forme $SI = \{..., s, ...()\}$ avec s le signal émis. Stocker à deux endroits des informations complémentaires sur un élément n'est pas optimal.

  Pour palier au premier problème on devrait pouvoir produire nos identifiants, ce qui n'est pas facile car les signaux ne sont pas tous réunis dans une liste. Sinon il nous faudrait un producteur d'identifiant. Un identifiant qu'il faudrait récupérer après via une application. Avec ces contraintes ce n'est pas la plus facile des tâches.

  Pour palier au second problème, il faut modifier l'approche en utilisant l'abstraction une nouvelle fois et en créant une commande que l'on va nommer *init*, on dira que la variable X de l'abstraction $\langle X, C\rangle$ sera la variable représentant le signal. On aura un règle de la forme :

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

  Pour palier au troisième problème on peut réunir les informations de l'initialisation et de l'émission dans la liste des signaux. Cependant on va devoir ajouter une façon d'identifier les threads. Voyons cela avec un exemple.

  Exemple 24 *On crée deux threads que l'on va nommer, avec beaucoup d'imagination,* $t1$ *et* $t2$*.*

  $t1$ *initialise un signal* s *et va être bloquer par l'attente d'une émission.* $t2$ *lui va initialiser le signal* s *aussi. Un problème survient si on ne vérifie qui a initialisé* s*, on peut voir la seconde intialisation comme un problème car le signal est déjà théoriquement initialisé.*

  Avec cette forme et la contrainte du champs d'action la résolution du problème est compliqué. On ne va pas prendre cette possibilité dans la machine mais si cela vous intéresse je mets en Annexe les explications pour pouvoir créer une règle qui fonctionne avec ces contraintes.

- On initialise un signal pour le thread courant : Je ne vais pas m'étendre sur cette version car elle rejoint de peu la version précédente et comme la version précédente une explication est donnée en Annexe.

- On initialise un signal pour la machine : La contrainte retire pas mal de problèmes, l'initialisation étant général on peut dire que : si un signal est dans la liste SI alors il est initialisé. On aura donc $\forall\ si \in SI\ :\ si = \langle s, emit\rangle$ avec l'identifiant s du signal et le booléen représentant l'émission $emit$. Il suffit maintenant que la machine supporte un nouvelle élément : un signal s. Ainsi que la commande $init$ On va peut créer la règle suivante :

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

  On peut aller plus loin en changeant la liste des signaux en file des signaux ce qui nous permet de créer un identifiant de signal facilement et donc éviter que l'utilisateur choisisse l'identifiant. De plus, ce ne sera plus une variable mais une constante que l'on pourra appliquer après. Cela convient plus à la machine. Ce qui nous donne la règle suivante :

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

On va utiliser la dernière version dans la machine finale. En effet, les deux premières possibilités ont le grand défaut de donner la liberté d'avoir plusieurs variables (dans différents threads) pour un même identifiant. Je m'explique. Lorsque l'on va initialiser un signal pour un thread, on va lui retourner un identifiant que l'on va ensuite appliquer à une variable. Si autre thread initialise ce même signal pour lui il peut lui donner un nom de variable qui diffère du thread précédent ce qui peut créer des imcompréhensions. Des versions sont cependant trouvable avec leurs implantations, dans l'optique de leurs trouver une utilitée que je n'ai moi-même pas vu .

Vérifier la présence *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 $emit$ pour savoir si un signal est émis. Le cas émis est simple. Cependant comment sait-on qu'il ne l'est pas ?

Un signal est absent s'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'éxécuter à la chaîne, la fin d'un instant sera lorsqu'aucun thread ne s'éxécute. Cependant certains vont tomber dans ce cas où ils devront attendre l'absence d'un signal. Cette contrainte prise en compte, on doit redéfinir la fin d'un instant logique.

Fin d'un instant logique : *la fin ou le blocage de tous les threads de la machine dans l'instant courant.*

Dans notre machine on va devoir différencier les threads en attente 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 $TL = \langle W, ST\rangle$ tel 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 necéssité de stocker quel signal attend un thread pour chaque thread 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 ne 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. $s\ \langle\langle X, C'\rangle\ \langle\langle X, C''\rangle\ present$ avec un signal s. Cette version parait 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 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 reprend le tout, pour faire notre test de présence nous avons défini :

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

Émettre 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 bougé, seulement deux versions existent :

1. $emit_s$ une forme inspirée de $prim_{o^n}$. 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 indépendante du signal qu'il émet.
2. s *emit* avec un signal s. Celle-ci est privée de toute ambiguïté et s'intègre bien à la machine.

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

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

3.1.3 Sémantique de la machine abstraite

Maintenant que l'on a défini les nouvelles commandes et ce dont 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 :**

$b, s, n =$ une constante ou un identifiant de signal (un entier)

$V = b$
| $\langle\langle X, C'\rangle E\rangle$

$S = \emptyset$
| $V\ S$

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

$C = \epsilon$
| $b\ C$ (une constante ou un identifiant de signal)
| $X\ C$ (une variable)
| $\langle X, C'\rangle\ C$ (une abstraction)
| $ap\ C$ (une application)
| $prim_{o^n}\ C$ (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 :** $\forall\ si \in SI \mid si = \langle s, \langle emit, ST\rangle\rangle$ avec :

- **un identifiant de signal :** s
- **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 futures :

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

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'incrémente pour en créer un nouveau et retourne l'identifiant du signal créé avec la liste mise à jour.

Exemple 25 *2 cas sont possibles :*

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

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

$\varepsilon(s, SI)$ une fonction qui prend un signal s et met à vrai son booléen représentant l'émission et retourne la liste de threads bloqués.

Exemple 26 $\varepsilon(s, \{..., \langle s, \langle emit, ST\rangle\rangle, ...\}) = (ST, \{..., \langle s, \langle vraie, \{\}\rangle\rangle, ...\})$

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

Exemple 27 $SI(s) = \langle emit, ST\rangle$

$\tau(SI)$ 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 28 $\tau(SI) = \forall\ si \in SI$:

- $\langle s, \langle true, \{\}\rangle\rangle \ \rightarrow\ \langle s, \langle false, \{\}\rangle\rangle$
- $\langle s, \langle true, ST\rangle\rangle \ \rightarrow\ \langle s, \langle false, \{\}\rangle\rangle$ *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 ajouté 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 ou Signal : On a une constante, on la déplace dans la pile.

$\langle S, E, n\ C, D\rangle \longrightarrow_{TTS} \langle n\ S, E, C, D\rangle$ où n est une constante b ou un identifiant de signal s

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

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

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

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

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.

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

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

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

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

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

Partie pour la concurrence : Voici les règles ajoutées 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.

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

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

$\langle\langle S, E, init\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTS} \langle\langle s\ S, E, C, D\rangle, TL, SI'\rangle$ 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 prend le premier choix.

$\langle\langle\langle\langle X', C''\rangle, E\rangle\ \langle\langle X, C'\rangle, E\rangle\ s\ S, E, present\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTS} \langle\langle S, E, C'\ C, D\rangle, TL, SI\rangle$
avec $SI(s) = \langle vraie, ST\rangle$

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.

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

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.

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

Émettre : On émet un signal via la fonction ε.

$\langle\langle s\ S, E, emit\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTS} \langle\langle S, E, C, D\rangle, TL\ ST, SI'\rangle$ avec $\varepsilon(s, SI) = (ST, SI)'$

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

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

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

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

Partie commune : Quand on ajoute 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 ne rien faire.

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

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

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

la machine TTS peut s'arrêter dans 4 états différents :

- Soit on a une **constante** telle que $\langle\langle\emptyset, \emptyset, [M]_{TTS}, \emptyset\rangle, \emptyset, \emptyset\rangle \twoheadrightarrow_{TTS} \langle\langle b\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a une **fonction** telle que $\langle\langle\emptyset, \emptyset, [M]_{TTS}, \emptyset\rangle, \emptyset, \emptyset\rangle \twoheadrightarrow_{TTS} \langle\langle\langle X, C\rangle, E\rangle\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a **rien** telle que $\langle\langle\emptyset, \emptyset, [M]_{TTS}, \emptyset\rangle, \emptyset, \emptyset\rangle \twoheadrightarrow_{TTS} \langle\langle\epsilon, E, \epsilon, \emptyset\rangle, \emptyset, SI\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Sinon on a un **état inconnu** : une **erreur**

Cette version des règles est la plus optimale car on a pris le meilleur de chaque possibilité. Cependant il faut savoir que des règles intermédiaires ont été créées, implantées et testées en OCaml. En effet je vais vous présenter 2 versions 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. Un exemple de fonctionnement de la machine TTS se trouve dans les Annexes.

3.2 Le partage des valeurs dans la machine

3.2.1 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 soient liées à un signal précis. On va ajouter 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 problème se pose par rapport à cela : l'attribution des identifiants.

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 éviter des problèmes de déterminisme, on va devoir séparer les valeurs partagées en deux parties :

1. Une liste de valeurs courantes : c'est là que l'on va insérer les valeurs, on ne peut pas prendre dans cette liste, on peut la voir comme une liste tampon. Cela permet 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 met 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. Si le signal est 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 à été faite durant le stage mais n'est pas si éloignée 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'identifiants 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'identifiants qui représente une position possible de l'itérateur IL.

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

Accèder 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 pourrait 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 retourne 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 on veut une valeur de ces valeurs partagées et le neutre n.

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 réunit les commandes *emit* et *put*.

3.2.2 Sémantique de la machine abstraite

Ces deux commandes ajoutées engrangent énormément de changements. On va donc devoir redéfinir nos règles.

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 :**

$b, s, n =$ une constante ou un identifiant de signal (un entier)

$V = b$
| $\langle\langle X, C'\rangle E\rangle$

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

$S = \emptyset$
| $V\ S$

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

$C = \epsilon$
| $b\ C$ (une constante ou un signal)
| $X\ C$ (une variable)
| $\langle X, C'\rangle\ C$ (une abstraction)
| $ap\ C$ (une application)
| $prim_{o^n}\ C$ (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 :** $\forall\ si \in SI\ :\ si = \langle s, \langle emit, CS, SSI, TL\rangle\rangle$ avec :

- **un identifiant de signal :** s
- **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 :** $\forall\ ssi \in SSI\ :\ ssi = \langle I, \langle CI, IL\rangle\rangle$ avec
  - **une liste d'identifiant de threads telle que :** $\forall\ il \in IL\ :\ il = I$
  - **une liste de constante avec itérateur telle que :** $\forall\ ci \in CI\ :\ ci = \langle b, IL\rangle$

$IP =$ 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'incrémente pour en créer un nouveau et retourne l'identifiant du signal créé avec la liste mise à jour.

Exemple 29

Si on initialise pour la première fois alors $\iota(\{\}) = (0, \{\langle 0, \langle false, \{\}, \{\}, \{\}\rangle\rangle\})$
sinon $\iota(\{..., \langle s, data\rangle\}) = (s+1, \{..., \langle s, data\rangle, \langle s+1, \langle false, \{\}, \{\}, \{\}\rangle\rangle\})$ *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 30 $SI(s) = \langle emit, CS, SSI, ST\rangle$

- $\tau(SI)$ 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 31 $\tau(SI) = \forall\ si \in SI$:

- $\langle true, CS, SSI, \{\}\rangle \rightarrow \langle false, \{\}, CS', \{\}\rangle$ *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.

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

Exemple 32 *Trois cas sont possibles :*

1. *Première fois que l'on prend :* $\langle id', \langle\{\langle b, \emptyset\rangle, \langle n, IL\rangle, ...\}, IL\rangle\rangle \rightarrow \langle id', \langle\{\langle b, \emptyset\rangle, \langle n, IL\ id\rangle, ...\}, IL\rangle\rangle$ *et on retourne b*
2. *On a déjà pris :* $\langle id', \langle\{..., \langle b, \{..., id, ...\}\rangle, \langle n, IL\rangle, ...\}, IL\rangle\rangle \rightarrow \langle id', \langle\{..., \langle b, \{...\}\rangle, \langle n, IL\ id\rangle, ...\}, IL\rangle\rangle$ *et on retourne b*
3. *On prend le dernier :* $\langle id', \langle\{..., \langle b, \{..., id, ...\}\rangle\}, IL\rangle\rangle \rightarrow \langle id', \langle\{..., \langle b, \{...\}\rangle\}, IL\ id\rangle\rangle$ *et on retourne b*

- $SI[(s, i) \leftarrow b]$ 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 $emit$.

Exemple 33 *Pour* $SI(s) = \langle emit, CS, SSI, ST\rangle$ *on change SI telle que* $SI(s) = \langle true, CS, SSI', ST\rangle$
avec $SSI' = \gamma(id, i, SSI)$ *avec id l'identifiant du thread courant.*

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

Exemple 34 $SSI(i) = \langle CI, IL\rangle$

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

Dans tous les cas :

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

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

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

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

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

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

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

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

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

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 présents dans la fermeture et on ajoute une substitution dans le nouvel environnement.

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

Récupération de sauvegarde : On n'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 ajoute la concurrence dans notre machine.

Création thread : On crée un nouveau thread.

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

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

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

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

$\langle\langle I, s\ b\ n\ \langle\langle X, C'\rangle, E'\rangle\ S, E, get\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSI} \langle\langle I, \emptyset, E'[X \leftarrow V], C', \langle S, E, C, D\rangle\rangle, TL, SI\rangle$
si pour $SI(s) = \langle emit, CS, SSI\rangle$ et $SSI(b) = \langle CI, IL\rangle$ on a $I \notin IL$ alors $\gamma(I, b, SSI(b)) = V$ sinon $n = V$

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

$\langle\langle I, S, E, init\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSI} \langle\langle I, s\ S, E, C, D\rangle, TL, SI'\rangle$ 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 premier choix.

$\langle\langle I, \langle\langle X', C''\rangle, E\rangle\ \langle\langle X, C'\rangle, E\rangle\ s\ S, E, present\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSI} \langle\langle I, S, E, C'\ C, D\rangle, TL, SI\rangle$
avec $SI(s) = \langle vraie, CS, SSI, TL\rangle$

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.

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

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.

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

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

$\langle\langle I, S, E, \epsilon, \emptyset\rangle, \emptyset, SI\rangle \longrightarrow_{TTSI} \langle\langle I, S, E, \epsilon, \emptyset\rangle, TL, SI'\rangle$ 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 ne rien faire.

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

Récupération de sauvegarde avec pile vide : On n'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 4 états différents :

- Soit on a une **constante** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSI} \langle\langle I, b\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a une **fonction** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSI} \langle\langle I, \langle\langle X, C\rangle, E\rangle\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a **rien** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSI} \langle\langle I, \epsilon, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Sinon on a un **état inconnu** : on a une **erreur**

Un exemple de fonctionnement de la machine TTS se trouve dans les Annexes.

3.3 Preuve du déterminisme

Notre machine a besoin d'être déterministe. En effet, il faut pouvoir savoir exactement ce qu'il se passe dans la machine. Si on a plusieurs sortie possible pour une même entrée, la machine abstraite ne serait pas utilisable. Avec les différentes explications sur la machine TTSI on peut avoir l'intution quelle est déterministe. Cependant il faut le prouver.

Pour cela on va faire une preuve par induction. Avant tout, je vais reprendre les différents point de la machine et montrer qu'il ne peuvent pas avoir d'incidence sur le déterminisime de la machine. Tout cela afin de réduire notre preuve aux règles et plus sur la structure.

Reprenons la machine dans son entièreté :

1. T : le thread courant est de la forme $\langle I, S, E, C, D\rangle$. En enlevant I on retrouve notre machine SECD. La machine SECD étant déterministe, sa structure ne peut pas provoquer de non déterminisme de notre machine. T est vérifié.
2. TL : la file d'attente est ,comme son nom l'indique, une file (une FIFO). Ce qui veut dire que l'on va prendre toujours au "début" de notre file et ajouter à la "fin". Ce qui revient à avoir un seul ordre possible dans le traitement des threads. Avoir un non déterminisme ici reviendrait à prendre un thread au hasard à chaque fois dans notre file or par la définition de file c'est impossible. L'ordre de traitement est créé lors de la création des threads et peut être uniquement modifier par la commande *present*. En effet si un signal n'est pas émis on va mettre le thread dans la file de threads bloqués du signal qu'il attend. On va donc l'enlever de TL et le remettre plus tard à l'intérieur. La file d'attente est vérifiée.
3. SI : On va d'abord s'intéresser à SI puis à ce qui la compose. C'est une liste que l'on ne vide pas. En effet =, quand on initialise un signal, il est initialisé pour toute la durée du fonctionnement de la machine. On a vu que l'on avait besoin de les mettre dans un ordre précis pour pouvoir initialiser (pour créer leurs identifiants on les mets dans l'ordre croissant). On a une sorte de file où l'on ne prend jamais. SI est composé $\langle s, \langle emit, CS, SSI, TL\rangle\rangle$. TL

Prenons maintenant toute les formes que peut prendre la chaîne de contrôle qui sont traités par notre machine :

- $\underline{b\ C}$
- ... $\langle X, C'\rangle\ C''\ ap$ ... $X\ C$ avec $C'' \twoheadrightarrow_{TTSI} V$
- $\underline{\langle X, C'\rangle\ C}$
- $\underline{ap\ C}$
- $\langle X, C'\rangle\ C''\ ap\ C$ avec $C'' \twoheadrightarrow_{TTSI} V$
- $C_n...C_0\ prim_{o^n}\ C$ avec $C_n...C_0 \twoheadrightarrow_{TTSI} b_n...b_0$
- $\langle X, C'\rangle\ spawn\ C$
- $\langle X', C''\rangle\ \langle X, C'\rangle\ s\ present\ C$
- $\underline{init\ C}$
- $s\ b\ n\ get\ C$ avec un signal s et un entier n
- $b\ put\ C$

Les cas soulignés sont les seuls cas que l'on peut faire durant la première transition car ils ne nécéssitent pas plusieurs éléments.

On va commencer la preuve par induction et plus précisement une preuve par récurrence :

Hypothèse : La machine TTSI est déterministe.

Initialisation : On teste le déterminisme de la machine pour la première transition de la machine. À l'état 0 on a la machine TTSI de la forme $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle$. On a 4 possibilités pour la chaîne de contrôle :

1. $b\ C$: On va utiliser la règle **Constante ou Signal**
2. $\langle X, C'\rangle\ C$: On va utiliser la règle **Abstraction**
3. $ap\ C$: On va utiliser la règle **Application neutre**
4. $init\ C$: On va utiliser la règle **Initialisation signal**

Tous les autres éléments et commandes ne sont pas possible et provoque une erreur. La machine TTSI est déterministe à l'état 1.

Hérédité : On suppose que l'hypothèse est vrai au rang n, c'est-à-dire que l'on a n transition déterministe. Montrons que l'hypothèse est vrai au rang n+1. Le but est de montrer que pour chaque cas que notre machine peut supporter il y a q'une seule règle qui puisse s'appliquer. En priorité, ce qui va impacter le choix d'une transition se sera la chaîne de contrôle, on va donc voir tous les éléments que la chaîne peut avoir en tête pour la transition n+1.

Au total 21 cas sont traités par la machine. Voici les cas :

1 - b : on a une constante, une règle est applicable **Constante**.

- X : on a une variable, deux cas sont possibles :
  - **2** - Il y a une substitution possible dans l'environnement E, une règle est applicable **Substitution**.
  - **x** - Il n'y a pas de substitution possible dans l'environnement E, aucune règle est applicable : c'est une **erreur**.
- $prim_{o^n}$: on a la commande qui gère les opérations, deux cas sont possibles :
  - **3** - il y a assez de constantes consécutives dans la pile S, une règle est applicable **Opération**.
  - **x** - il n'y a pas assez de constantes consécutives dans la pile S, aucune règle est applicable : c'est une **erreur**.

4 - $\langle X, C' \rangle$: on a une abstraction, une règle est applicable **Abstraction**.

- ap : on a la commande qui gère l'application, deux cas sont possibles :
  - **5** - soit il y a V $\langle \langle X, C' \rangle, E' \rangle$ dans la pile S, une règle est applicable **Application**.
  - **6** - soit il y a rien de spécifique dans la pile S, une règle est applicable **Application neutre**.
- *spawn* : on a la commande qui gère la création de thread, deux cas sont possibles :
  - **7** - si on a $\langle \langle X, C' \rangle, E' \rangle$ dans la pile S, une règle est applicable **Création thread**.
  - **x** - sinon on a aucune règle est applicable : c'est une **erreur**.
- *put* : on a la commande qui gère l'ajout dans un signal, deux cas sont possibles :
  - **8** - si on a s b en tête dans la pile S, une règle est applicable **Ajouter dans un signal**.
  - **x** - sinon on a aucune règle est applicable : c'est une **erreur**.
- *get* : on a la commande qui gère la prise de valeur, deux cas sont possibles :
  - **9** - si on a s b n en tête dans la pile S, une règle est applicable **Prendre une valeur partagée**.
  - **x** - sinon on a aucune règle est applicable : c'est une **erreur**.

10 - *init* : on a la commande qui gère l'initialisation d'un signal, une règle est applicable **Initialisation signal**.

- *present* : on a la commande qui gère la présence d'un signal, deux cas sont possibles :
  - si on a $\langle \langle X', C'' \rangle, E'' \rangle$ $\langle \langle X, C' \rangle, E' \rangle$ s en tête dans la pile S, deux cas sont possibles :
    - **11** - soit on teste un signal émis tel que $SI = \{..., \langle s, \langle true, CS, SSI, TL \rangle \rangle, ...\}$, une règle est applicable **Présence du signal**
    - soit on teste un signal non émis tel que $SI = \{..., \langle s, \langle false, CS, SSI, TL \rangle \rangle, ...\}$, le cas se divise alors encore en deux :
      - **12** - soit on met en attente ce thread et on prend un nouveau thread dans TL, une règle est applicable **Thread bloqué remplacé**.
      - **13** - soit on met en attente ce thread et TL est vide, une règle est applicable **Thread bloqué non remplacé**.
  - **x** - sinon on a aucune règle est applicable : c'est une **erreur**.
- ϵ : la chaîne de contrôle est vide, deux cas sont possibles :
  - le dépôt D contient une sauvegarde, deux cas sont possibles :
    - **14** - on a un élément dans la pile S, une règle est applicable **Récupération de sauvegarde**.
    - **15** - on a rien dans la pile S, une règle est applicable **Récupération de sauvegarde neutre**.
  - le dépôt D est vide, deux cas sont possibles :
    - **16** - on a un élément dans la file d'attente TL, une règle est applicable **Récupération dans la file d'attente**.
    - on a la file d'attente TL vide, deux cas sont possibles :
      - **17** - au moins un signal contient un ou plusieurs threads bloqués, une règle est applicable **Fin d'instant logique**.
      - aucun signal ne contient de threads bloqués, c'est la **fin de fonctionnement de la machine**. 4 fin sont possibles :
        - **18** - la tête de pile S est une **constante**.
        - **19** - la tête de pile S est une **fonction**.
        - **20** - la tête de pile S est **vide**.
        - **21** - on a un état inconnu qui est une **erreur**.

3.4 Les types inductifs et la récursion

3.4.1 Description informelle du langage

Récursion La récursion existe déjà dans les λ-calculs cependant on la perd avec le typage. En effet, on va voir ça à travers un exemple.

Exemple 35 *Si on prend* $fun\ x \rightarrow fun\ y \rightarrow x\ y$ *et que l'on tente de le typer. On va avoir :*

- $fun\ x \rightarrow fun\ y \rightarrow\ x\ y$
  $'a\ \ \rightarrow\ \ 'b\ \ \rightarrow 'c$

Or x est une fonction du coup on aurait : $'a\ =\ 'a\ \ \rightarrow\ \ 'b\ \ \rightarrow 'c$*. Ce n'est pas possible. D'où l'impossibilité d'avoir une récurrence innée.*

Pour ajouter la récursivité il va donc falloir indiquer à la machine que la fonction sur laquelle on va travailler est récursive. On peut voir cet indicateur en Ocaml, par exemple, via le mot-clé *rec*. Pour notre machine, on crée la commande fix. Cette commande va se situer juste après une abstraction pour indiquer que celle-ci est récursive. L'abstraction devra être de la forme $\langle f, \langle x, t\rangle\rangle$ avec f le nom de la fonction récursive, le premier paramètre x et le reste de la fonction (avec potentiellement d'autre paramètres) t. Notre but est de garder l'information de la récursion, pour ça on sait que lorsqu'une abstraction passe de la chaîne de contrôle à la pile d'exécution, elle devient une fermeture qui est un couple contenant une abstraction et un environnement. C'est dans cette environnement que l'on va stocker l'information.

Exemple 36 *Si on se focalise sur la machine SECD de base, on peut voir un prototype de règle sur l'exemple suivant :*
$\langle\langle f, \langle x, t\rangle\rangle, E'\rangle\ S, E, fix\ C, D\rangle \longrightarrow \langle\langle\langle x, t\rangle, E'[f \leftarrow A]\rangle\ S, E, fix\ C, D\rangle$

Le but, maintenant, est de savoir quelle forme va prendre A.

A est un élément qui fonctionne récursivement, en effet il va se contenir lui même.
On va avoir $A\ =\ \langle\langle\langle x, t\rangle, E'[f \leftarrow B]\rangle$ avec $B\ =\ \langle\langle\langle x, t\rangle, E'[f \leftarrow A]\rangle$

Personnellement pour gérer son implantation il m'a fallu mettre un booléen représentant la récursion pour contourner la définition récursive de l'élément A. Cependant une seconde difficulté vient se greffer à la récursion.

En effet notre machine a pour stratégie d'évaluation l'appel par valeur, c'est-à-dire que l'on va d'abord évaluer les paramètres avant la fonction. Or la conditionnelle en λ-calcul s'écrit comme une fonction qui prend trois paramètres $(\lambda vtf.(v\ t\ f))$.

Exemple 37 *Soit la fonction factorielle telle que :*
$let\ fact\ n\ =\ if\ (n = 0)\ then\ 1\ else\ n * fact\ (n - 1)\ ;;$
Réecrit en pseudo λ-calcul cela donne :
$\lambda fact.n.(\lambda vtf.(v\ t\ f)\ \underline{(n = 0)}\ \underline{1}\ \underline{(n * fact\ (n - 1))})$

Les parties soulignées sont les paramètres de la conditionnelle. Fonctionnant par appel par valeur on doit les évaluer en premier. Dedans il y a une récursion qui va être évaluée etc. On va toujours évaluer la récursion avant la conditionnelle.

Pour palier à cela on va retarder artificiellement l'évaluation des paramètres en les encapsulants dans des abstractions. On va aussi modifier les booléens qui sont de base de la forme suivante en λ-calcul :

$true\ =\ \lambda xy.x$
$false\ =\ \lambda xy.y$

On aura donc $\lambda vtf.(v\ t\ f)\ p^v\ \lambda x.(p^t)\ \lambda x.(p^f)$ avec x une variable quelconque et p^v , p^t, p^f les paramètres de la conditionnelleS. La représentation des booléens changera un peu et donnera cela : $true\ =\ \lambda xy.(x\ a)$ et $false\ =\ \lambda xy.(y\ a)$ avec a une variable quelconque.

Type Les types sont extrêmement importants dans la programmation fonctionnelle. Ils nous permettent de définir des structures complexes et nous donnent la possibilités de travailler avec. Tout d'abord il faut savoir comment on va représenter un type dans notre machine ensuite comment le créer et enfin comment l'utiliser.

Structure d'un type Les types sont des structures composées de plusieurs paramètres qui sont liées via un constructeur.

Exemple 38 *type tree = Node of tree * int * tree | Leaf of int ;;*
Ici Node et Leaf sont deux constructeurs pour le type tree.

Ce qui est intéressant à savoir c'est que lorsque l'on arrive au niveau de la machine abstraite dans la procédure, le langage est déjà vérifié syntaxiquement et sémantiquement. Cela peut paraître anodin mais cela fait toute la différence. Quand notre chaîne de contrôle est donnée à la machine on n'a plus à se préoccuper des types en eux-même car on sait déjà que si la chaîne de contrôle est donnée à la machine, les types sont déjà vérifiés et corrects. Les types deviennent alors seulement deux informations : le constructeur qu'il les a créé et les paramètres qui les composent. Pour stocker cela on va utiliser un tableau qui aura comme première élément l'identifiant du constructeur et le reste, les éléments du type. On aura donc un type de la forme : $[c, V_1, ..., V_n]$ avec n le nombre de paramètre pour le constructeur c.

Création d'un type Prenons un exemple pour créer un type.

Exemple 39 *On reprend le type tree. On va créer un élément de type tree.*
let arbre = Node(Leaf(1), 2, Leaf(3)) ;;

En prenant en compte que la machine garde en mémoire uniquement l'identifiant de constructeur (ici on va garder leurs noms pour que l'exemple soit plus clair) et le nombre de paramètre, on aurait :
let arbre = Node 3 (Leaf 1 1) 2 (Leaf 1 3)

Remettons cela en pseudo λ-calcul :
λarbre.(t) Node 3 (Leaf 1 1) 2 (Leaf 1 3) avec t le reste du programme à exécuter
On a l'intuition de ce qu'il faut rajouter pour que cela fonctionne.

On va devoir ajouter une commande que l'on va nommer *build*.Elle va indiquer la création d'un type. Cette commande *build* va avoir besoin de trois informations : l'identifiant de constructeur, le nombre de paramètres et les paramètres. La règle pour une machine SECD serait la suivante :
$\langle n\ b\ V^n\ ...\ V^1\ S, E, build\ C, D, \rangle \longrightarrow \langle [b, V^1, ..., V^n]\ S, E, C, D, \rangle$

Filtrage Les types sont utilisés via le filtrage. C'est la base pour pouvoir faire des opérations sur un type. La forme du filtrage est la suivante :

1. *match elem with*
2. $|\ pattern^1 \rightarrow processus^1$
3. ...
4. $|\ pattern^n \rightarrow processus^n$

Le but est d'écrire le filtrage avec ce que l'on a déjà pour ajouter uniquement le nécessaire. On peut voir le filtrage comme une succession de conditionnelle, ce qui donne la forme suivante :

1. *if* $(elem = pattern^1)$
2. *then decomposer elem* $pattern^1$; $processus^1$
3. *else if* $(elem = pattern^2)$
4. *then decomposer elem* $pattern^2$; $processus^2$
5. *else ...*
6. *if* $(elem = pattern^n)$
7. *then decomposer elem* $pattern^n$; $processus^n$
8. *else erreur*

La conditionnelle existe déjà en λ-calcul du coup on peut encore descendre d'un niveau d'écriture, ce qui donne la forme qui suit :

1. $\lambda vtf.(v\ t\ f)\ (elem\ =\ pattern^1)$
2. $\lambda x.(decomposer\ elem\ pattern^1\ ;\ processus^1)$
3. $\lambda x.(\lambda vtf.(v\ t\ f)\ (elem\ =\ pattern^2)$
4. $\lambda x.(decomposer\ elem\ pattern^2\ ;\ processus^2)$
5. $\lambda x.(...\lambda x.(\lambda vtf.(v\ t\ f)\ (elem\ =\ pattern^n))$
6. $\lambda x.(decomposer\ elem\ pattern^n\ ;\ processus^n)$
7. $\lambda x.(error))...))$

On ne peut pas découper plus que cela, nos objectifs sont définis. On doit définir ce qu'est un pattern (un motif), ensuite il faut pouvoir comparer un pattern et un type et enfin il faut pouvoir décomposer un type via un pattern pour pouvoir utiliser les éléments du type.

Pattern Le filtre fonctionne avec un pattern. Ce pattern est soit une variable, soit un type contenant des variables. Du coup on aura un pattern tel que : $pattern\ =\ [c, X_1, ..., X_n]$ avec le nombre de paramètre n et l'identifiant de constructeur c ou $pattern\ =\ X$.

Comparaison de deux types Pouvoir comparer deux types est la base pour le filtrage. Ici la comparaison ne sera pas trop difficile car on va comparer leurs identifiants de constructeurs pour savoir si c'est le même type. Pour cela on va ajouter une commande *compare* qui va indiquer la volonté de comparer le type et le pattern en tête de la pile d'exécution. Le résultat sera retourné dans la pile sous la forme d'un booléen écrit en λ-calcul. La règle sera donc de la forme :

$\langle [c', V^1...V^n]\ [c, X^1...X^n]\ S, E, compare\ C, D\rangle \longrightarrow_{TTSI} \langle\langle bool, E\rangle\ S, E, C, D\rangle$
si $c' = c$ alors $bool = true$ sinon $bool = false$

ou

$\langle [c', V^1...V^n]\ X\ S, E, compare\ C, D\rangle \longrightarrow_{TTSI} \langle\langle true, E\rangle\ S, E, C, D\rangle$

Décomposition d'un type Lorsque le filtrage a été fait, il faut que l'on décompose le type via un pattern pour pouvoir utiliser les éléments du type. Pour cela, on peut voir la décomposition de deux façons :

- On fait une décomposition récursive car on prend en compte les patterns imbriqués ;
- On fait une décomposition en prenant en compte qu'un pattern contient uniquement des variables.

La première version est plus lourde à implanter et plus complexe à définir mais si cela vous intéresse, une version implantée en Ocaml comporte la règle suivante :

$\langle S, E, destruct\ C, D\rangle \longrightarrow_{TTSI}$
si $S = \epsilon$ alors si $D = \langle S', E', C', D\rangle$
alors $\langle S', E \cup E', C', D\rangle$
sinon si $C \neq \epsilon$ alors $\langle S, E, C, D\rangle$
sinon si $S = [c, V^1...V^n]\ [c, X^1...X^n]\ S'$
alors $\langle \epsilon, \emptyset, \epsilon, \langle V^1\ X^1, E, destruct, \langle ...\langle V^n\ X^n, E, destruct, \langle S', E, destruct\ C, D\rangle\rangle...\rangle\rangle\rangle$
sinon si $S = V\ X\ S'$
alors $\langle S', E\ [X \leftarrow V], destruct\ C, D\rangle$
sinon si $S = V\ \_\ S$
alors $\langle S', E, destruct\ C, D\rangle$
sinon $\langle S, E, C, D\rangle$

On peut remarquer que la machine comprend qu'elle est dans une sous décomposition en voyant qu'il n'y que la commande *destruct* dans la chaîne de contrôle. En effet si on prend l'exemple de l'Ocaml, je prends en compte que le *destruct* fait référence à la flèche $\rightarrow$ or cette flèche est toujours suivie sinon elle provoque une erreur. Du coup la commande *destruct* ne peut fatalement pas être le dernière élément de la chaîne de contrôle normalement.

La seconde version est, pour ça part, beaucoup plus légère que la première. En effet, on part du principe que la machine ne traite que des patterns contenant uniquement des variables. Cela ne veut pas dire que le langage ne supporte pas les patterns imbriqués mais seulement que le pattern sera convertit pour ne pas inclure de patterns imbriqués. Prenons un exemple :

Exemple 40 *Si on prend le type tree et* $arbre = Node(Leaf(1), 2, Leaf(3))$. *On va créer le filtrage suivant :*

1. *match arbre with*
2. $Node(Leaf(a), b, c) \rightarrow processus$

Par rapport à ce que l'on a vu plus haut on peut écrire le match sous la forme suivante :

1. $\lambda vtf.(v\ t\ f)\ (arbre = Node(Leaf(a), b, c))$
2. $\lambda x.(decomposer\ arbre\ Node(Leaf(a), b, c)\ ;\ processus)$
3. $\lambda x.(erreur)$

Maintenant il faut que l'on décompose le pattern pour avoir seulement des variables. Ce qui donne :

1. $\lambda vtf.(v\ t\ f)\ (arbre = Node(Leaf(a), b, c))$
2. $\lambda x.(decomposer\ Leaf(1)\ Leaf(a)\ ;\ decomposer\ Node(Leaf(1), 2, Leaf(3))\ Node(\_, b, c)\ ;\ processus)$
3. $\lambda x.(erreur)$

Deux décompositions vont se faire et permettre de ne pas faire une règle aussi complexe que la première version. On ne connaît normalement pas la forme de l'élément quand on fait un match du coup on peut aussi l'écrire sous la forme suivante :

1. $\lambda vtf.(v\ t\ f)\ (arbre = Node(Leaf(a), b, c))$
2. $\lambda x.(decomposer\ arbre\ Node(x, b, c)\ decomposer\ x\ Leaf(a)\ ;\ processus)$
3. $\lambda x.(erreur)$

Cette version a l'avantage de ne pas avoir besoin de connaître la forme du type à décomposer.

Dans la sémantique qui suit, on prendra la décomposition sans patterns imbriqués ce qui nous donnera la règle suivante :

$\langle [c, V_1...V_n]\ [c, X_1...X_n]\ S, E, destruct\ C, D\rangle \longrightarrow_{TTSI} \langle S, E\ [X_1 \leftarrow V_1]...[X_n \leftarrow V_n], C, D\rangle$

Un point non abordé est la décomposition neutre, je m'explique. Si on prend l'exemple d'un filtrage en Ocaml : _ considère que l'on ne souhaite pas garder l'information. On va avoir la même chose ce qui va nous donner la règle suivante :

$\langle [c, V_1...V_{k-1}, V_k, V_{k+1}...V_n]\ [c, X_1...X_{k-1}, \_, X_{k+1}...X_n]\ S, E, destruct\ C, D\rangle$
$\longrightarrow_{TTSI} \langle S, E\ [X_1 \leftarrow V_1]...[X_{k-1} \leftarrow V_{k-1}][X_{k+1} \leftarrow V_{k+1}]...[X_n \leftarrow V_n], C, D\rangle$

3.4.2 Sémantique de la machine abstraite

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 :**

$b, s, n =$ une constante ou un identifiant de signal (un entier)

$V = b$
- $\mid\ \langle \langle X, C' \rangle E \rangle$
- $\mid\ [c, V_1, ..., V_n]$ avec n le nombre de paramètre pour le constructeur c
- $\mid\ [c, X_1, ..., X_n]$ avec n le nombre de paramètre pour le constructeur c

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

$S = \emptyset$
- $\mid\ V\ S$

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

$C = \epsilon$
- $\mid\ b\ C$ (une constante ou un signal)
- $\mid\ X\ C$ (une variable)
- $\mid\ \langle X, C' \rangle\ C$ (une abstraction)
- $\mid\ [c, X_1, ..., X_n]\ C$ (un pattern)
- $\mid\ ap\ C$ (une application)
- $\mid\ prim_{o^n}\ C$ (un opérateur)
- $\mid\ spawn\ C$ (créer d'un nouveau thread)
- $\mid\ present\ C$ (le test de présence d'un signal)
- $\mid\ init\ C$ (initialise un signal)
- $\mid\ put\ C$ (insère une valeur dans un signal)
- $\mid\ get\ C$ (prend une valeurs dans un signal)
- $\mid\ build\ C$ (construit un type)
- $\mid\ compare\ C$ (compare deux types)
- $\mid\ destruct\ C$ (décompose un type par rapport à un pattern)

$D = \emptyset$
- $\mid\ \langle S, E, C, D \rangle$ (une sauvegarde liée à une abstraction)

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

- **un identifiant de signal :** s
- **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 :** $\forall\ ssi \in SSI\ :\ ssi = \langle I, \langle CI, IL \rangle \rangle$ avec
  - **une liste d'identifiant de threads telle que :** $\forall\ il \in IL\ :\ il = I$
  - **une liste de constante avec itérateur telle que :** $\forall\ ci \in CI\ :\ ci = \langle b, IL \rangle$

$IP =$ 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'incrémente pour en créer un nouveau et retourne l'identifiant du signal créé avec la liste mise à jour.

Exemple 41

Si on initialise pour la première fois alors $\iota(\{\}) = (0, \{\langle 0, \langle false, \{\}, \{\}, \{\}\rangle\rangle\})$
sinon $\iota(\{..., \langle s, data\rangle\}) = (s+1, \{..., \langle s, data\rangle, \langle s+1, \langle false, \{\}, \{\}, \{\}\rangle\rangle\})$ *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 42 $SI(s) = \langle emit, CS, SSI, ST\rangle$

- $\tau(SI)$ 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 43 $\tau(SI) = \forall\ si \in SI$:

- $\langle true, CS, SSI, \{\}\rangle \rightarrow \langle false, \{\}, CS', \{\}\rangle$ *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.

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

Exemple 44 *Trois cas sont possibles :*

1. *Première fois que l'on prend :* $\langle id', \langle\{\langle b, \emptyset\rangle, \langle n, IL\rangle, ...\}, IL\rangle\rangle \rightarrow \langle id', \langle\{\langle b, \emptyset\rangle, \langle n, IL\ id\rangle, ...\}, IL\rangle\rangle$ *et on retourne b*
2. *On a déjà pris :* $\langle id', \langle\{..., \langle b, \{..., id, ...\}\rangle, \langle n, IL\rangle, ...\}, IL\rangle\rangle \rightarrow \langle id', \langle\{..., \langle b, \{...\}\rangle, \langle n, IL\ id\rangle, ...\}, IL\rangle\rangle$ *et on retourne b*
3. *On prend le dernier :* $\langle id', \langle\{..., \langle b, \{..., id, ...\}\rangle\}, IL\rangle\rangle \rightarrow \langle id', \langle\{..., \langle b, \{...\}\rangle\}, IL\ id\rangle\rangle$ *et on retourne b*

- $SI[(s, i) \leftarrow b]$ 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 $emit$.

Exemple 45 *Pour* $SI(s) = \langle emit, CS, SSI, ST\rangle$ *on change* SI *telle que* $SI(s) = \langle true, CS, SSI', ST\rangle$
avec $SSI' = \gamma(id, i, SSI)$ *avec id l'identifiant du thread courant.*

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

Exemple 46 $SSI(i) = \langle CI, IL\rangle$

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

Dans tous les cas :

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

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

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

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

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

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

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

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

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

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 présents dans la fermeture et on ajoute une substitution dans le nouvel environnement.

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

Récupération de sauvegarde : On n'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 ajoute la concurrence dans notre machine.

Création thread : On crée un nouveau thread.

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

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

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

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

$\langle\langle I, s\ b\ n\ \langle\langle X, C'\rangle, E'\rangle\ S, E, get\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSI} \langle\langle I, \emptyset, E'[X \leftarrow V], C', \langle S, E, C, D\rangle\rangle, TL, SI\rangle$
si pour $SI(s) = \langle emit, CS, SSI\rangle$ et $SSI(b) = \langle CI, IL\rangle$ on a $I \notin IL$ alors $\gamma(I, b, SSI(b)) = V$ sinon $n = V$

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

$\langle\langle I, S, E, init\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSI} \langle\langle I, s\ S, E, C, D\rangle, TL, SI'\rangle$ 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 premier choix.

$\langle\langle I, \langle\langle X', C''\rangle, E\rangle\ \langle\langle X, C'\rangle, E\rangle\ s\ S, E, present\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSI} \langle\langle I, S, E, C'\ C, D\rangle, TL, SI\rangle$
avec $SI(s) = \langle vraie, CS, SSI, TL\rangle$

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.

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

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.

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

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

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

Partie pour les types : Cette partie ajoute les types dans la machine.

Pattern : On a un pattern, on le met dans la pile.

$\langle S, E, [c, X_1 ... X_n]\ C, D\rangle \longrightarrow_{TTSI} \langle [c, X_1 ... X_n]\ S, E, C, D\rangle$

Construction de type : On a la commande qui nous dit que l'on veut construire un type, on a le nombre de paramètre indiqué ainsi que le numéro de constructeur.

$\langle n\ b\ V_n\ ...\ V_1\ S, E, build\ C, D, \rangle \longrightarrow_{TTSI} \langle [b, V_1, ..., V_n]\ S, E, C, D, \rangle$

Comparer deux types : On a deux types en tête de la pile et on a la commande compare, on compare leurs constructeurs et on retourne un booléen.

$\langle [c', V_1 ... V_n]\ [c, X_1 ... X_n]\ S, E, compare\ C, D\rangle \longrightarrow_{TTSI} \langle\langle bool, E\rangle\ S, E, C, D\rangle$
si $c' = c$ alors $bool = true$ sinon $bool = false$

Comparaison neutre : On a un type et une variable en tête de la pile et on a la commande compare, on retourne vraie.

$\langle [c', V_1 ... V_n]\ X\ S, E, compare\ C, D\rangle \longrightarrow_{TTSI} \langle\langle true, E\rangle\ S, E, C, D\rangle$

Décomposition pattern : On a un type et un pattern et on a la commande destruct, on décompose le type via le pattern et on met dans l'environnement chaque couple (X, V).

$\langle [c, V_1 ... V_n]\ [c, X_1 ... X_n]\ S, E, destruct\ C, D\rangle \longrightarrow_{TTSI} \langle S, E\ [X_1 \leftarrow V_1] ... [X_n \leftarrow V_n], C, D\rangle$

Décomposition neutre : On a un type et une variable "_" et on a la commande destruct, on ne stocke pas la valeur liée à "_".

$\langle [c, V_1 ... V_{k-1}, V_k, V_{k+1} ... V_n]\ [c, X_1 ... X_{k-1}, \_, X_{k+1} ... X_n]\ S, E, destruct\ C, D\rangle$
$\longrightarrow_{TTSI} \langle S, E\ [X_1 \leftarrow V_1] ... [X_{k-1} \leftarrow V_{k-1}][X_{k+1} \leftarrow V_{k+1}] ... [X_n \leftarrow V_n], C, D\rangle$

Partie pour la récursion : Cette partie rajoute la récursion qui n'était pas innée dans la machine.

Récursion : On ajoute dans l'environnement une substitution qui est elle-même récursive pour pouvoir rappeler notre élément récursive.

$\langle\langle\langle X, \langle X', C'\rangle\rangle, E'\rangle\ S, E, fix\ C, D\rangle \longrightarrow_{TTSI} \langle\langle\langle X', C'\rangle, E'[X \leftarrow \langle\langle X', C'\rangle, E'[X \leftarrow ...]\rangle]\rangle\ S, E, C, D\rangle$

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 ne rien faire.

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

Récupération de sauvegarde avec pile vide : On n'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 4 états différents :

- Soit on a une **constante** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSI} \langle\langle I, b\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a une **fonction** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSI} \langle\langle I, \langle\langle X, C\rangle, E\rangle\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a **rien** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSI} \langle\langle I, \epsilon, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a un **type** tel que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSI} \langle\langle I, [c, V^1 ... V^n]\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Sinon on a un **état inconnu** : on a une **erreur**

3.5 Les exceptions

La machine devient de plus en plus complète cependant une partie n'a pas encore été traitée : *les exceptions*. Cette partie traite donc de l'ajout des exceptions dans la machine TTSI. Trois versions ont été créé pour cette partie, seul la dernière est présente dans la deuxième partie de cette section cependant les deux autres peuvent être retrouvées en Annexe.

3.5.1 Description informelle du langage

Quand on parle d'exceptions on a trois choses qui viennent en tête :

- créer une exception
- lever une exception
- avoir une structure de contrôle (try...catch)

Créer une exception Ce n'est pas la première fois que ma réflexion se pose sur les exceptions. En effet avant même la première version de la machine TTSI finie un travail a été fait sur les exceptions. Dans cette version elles été vu comme des erreurs que l'on pouvait lever ou non. Un élément *erreur* a été créé pour représenter les erreurs dans la machine cependant le gros défaut c'est que l'on pouvait pas créer des erreurs, seules celle reconnaissable par la machine pouvaient être utilisées. Cela nous avançait pas vraiment et de plus on perdait complètement l'idée de création d'exception. Heureusement la dernière version de la machine TTSI ajoute les types. L'idée est de voir les exceptions comme des types. La création est déjà toute faite car on va créer une exception via *build*.

Lever un exception La présence d'une exception est-elle toujours signe d'un problème ? Nan par exemple en Ocaml une exception peut être utilisé comme filtre dans un *match* ou encore en java on peut utiliser une exception sans que la machine soit compromise. Il faut donc faire la différence entre une exception et une exception levée. Pour cela on va utiliser un mot-clé que l'on va reprendre de l'Ocaml : *raise*. On part du principe que l'on n'utilise pas la propagation des erreurs lorsqu'une exception est levée mais que l'on vérifiera directement si une structure de gestion est présente.

Gérer les exceptions La structure de contrôle a toujours la même forme : la partie protégée et la partie de remplacement si une exception est levée. On va devoir trouver une structure qui est adaptée pour la machine. On va utiliser une astuce déjà présente pour le *spawn* et le *present*, c'est-à-dire utiliser une abstraction pour éviter la structure inappropriée. Il faut une commande pour savoir que l'on a une structure de gestion d'exception. On arrive donc à la forme suivant $\langle X, C\rangle\ \langle X', C'\rangle\ catch$. On ne spécifie pas l'exception qui devra être gérée, en effet on est au niveau le plus bas du traitement du langage ce n'est donc pas nécessaire. Voyons cela sur un exemple.

Exemple 47 *Soit f qui a x y associe x/y. On peut avoir l'erreur de la division par zero.*

1. *try (f x y)*
2. *with DivByZero → printf "Division par zéro"*

On peut découper ce try with en :

1. *try (f x y)*
2. *catch*
3. *match exception with*
4. *DivByZero → printf "Division par zéro"*
5. *_ → raise exception*

Le try...catch n'a donc pas la nécessité de connaître l'erreur qu'il traite.

Comment garder le catch dans la machine ? Lorsque l'on tombe sur la forme $\langle X, C\rangle\ \langle X', C'\rangle\ catch$ dans la chaîne de contrôle on sait que l'on va devoir garder la partie de remplacement de côté le temps de voir si on en a la nécessité. Pour cela on va ajouter un emplacement dans notre thread T qui est pour l'instant de la forme $T = \langle I, S, E, C, D\rangle$. On va ajouter l'élément H qui va garder les informations nécessaires. On aura donc $T = \langle I, S, E, C, D, H\rangle$. L'utilité de mettre le gestionnaire d'exception ici est de le garder privé à chaque thread. Ce qui évite les comportements étranges tels que le thread 2 utilise le gestionnaire d'exception du thread 5.

Que garder dans le gestionnaire ? Deux versions ont été faite pour la forme de H.

La première part du postulat que si on est dans le contenu protégée que l'on fait une action inérante à la concurrence de la machine comme par exemple ajouter une valeur dans un signal et qu'ensuite nous avons une exception levée alors la valeur ajouter ne doit plus exister. Pour faire cela, H doit contenir la totalité de la machine avant l'utilisation du contenu protégé. H est donc de la forme $H = \langle\langle I, \langle\langle X, C'\rangle E'\rangle\ S, E, C, D, H'\rangle, TL, SI, IP\rangle$. Le problème majeur de cette version est la mémoire prise pour stocker tout. En effet la machine ici peut potentiellement contenir une autre machine.

La seconde part du postulat que les effets de bords existent et que donc on peut garder seulement les informations du thread courants.

Effets de bords : *Une fonction est dite à effet de bord si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que retourner une valeur.*

Exemple 48 *Prenons un exemple en java.*

1. $int\ x = 3;$
2. $try\ \{$
3. $\quad System.out.println(\text{"}coucou\text{"});$
4. $\quad int\ x = 5/0;$
5. $\}catch(Exception\ e)\{$
6. $\quad System.out.println(\text{"}exception\ traitée\text{"});$
7. $\}$
8. $System.out.println(x);$

Ici le résultat sera :

coucou

erreur

3

Ici l'effet de bord remarquable est l'affichage de coucou alors le contenu protégé est erroné. C'est le cas aussi en Ocaml.

Le gestionnaire des exceptions sera donc de la forme $H = \langle\langle\langle X, C'\rangle E'\rangle\ S, E, C, D, H'\rangle$. La aussi le champs d'action du gestionnaire d'exceptions se limitera au thread qui le contient.

3.5.2 Sémantique de la machine abstraite

Cette extension est une vision des exceptions comme d'un type et non comme une entitée à part entière. Cependant il existe une version si cela vous intéresse en Annexe.

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

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

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

$b, s, n =$ une constante ou un identifiant de signal (un entier)

$V = b$
- $\mid\ \langle\langle X, C'\rangle E\rangle$
- $\mid\ [c, V_1, ..., V_n]$ avec n le nombre de paramètre pour le constructeur c
- $\mid\ [c, X_1, ..., X_n]$ avec n le nombre de paramètre pour le constructeur c

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

$S = \emptyset$
- $\mid\ V\ S$

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

$C = \epsilon$
- $\mid\ b\ C$ (une constante ou un signal)
- $\mid\ X\ C$ (une variable)
- $\mid\ \langle X, C'\rangle\ C$ (une abstraction)
- $\mid\ [c, X_1, ..., X_n]\ C$ (un pattern)
- $\mid\ ap\ C$ (une application)
- $\mid\ prim_{o^n}\ C$ (un opérateur)
- $\mid\ spawn\ C$ (créer d'un nouveau thread)
- $\mid\ present\ C$ (le test de présence d'un signal)
- $\mid\ init\ C$ (initialise un signal)
- $\mid\ put\ C$ (insère une valeur dans un signal)
- $\mid\ get\ C$ (prend une valeurs dans un signal)
- $\mid\ build\ C$ (construit un type)
- $\mid\ compare\ C$ (compare deux types)
- $\mid\ destruct\ C$ (décompose un type par rapport à un pattern)
- $\mid\ raise\ C$ (lève une erreur)
- $\mid\ catch\ C$ (try...catch)

$D = \emptyset$
- $\mid\ \langle S, E, C, D, H\rangle$ (une sauvegarde liée à une abstraction)

$H =$ une liste de thread servant à stocker le thread avant tentative de calcul du contenu protégé tels que $\forall\ h \in H\ :\ h = \langle S, E, C, D, H\rangle$.

$SI =$ **une liste de signaux telle que :** $\forall\ si \in SI\ :\ si = \langle s, \langle emit, CS, SSI, TL\rangle\rangle$ avec :
- **un identifiant de signal :** s
- **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 :** $\forall\ ssi \in SSI\ :\ ssi = \langle I, \langle CI, IL\rangle\rangle$ avec
  - **une liste d'identifiant de threads telle que :** $\forall\ il \in IL\ :\ il = I$
  - **une liste de constante avec itérateur telle que :** $\forall\ ci \in CI\ :\ ci = \langle b, IL\rangle$

$IP =$ 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'incrémente pour en créer un nouveau et retourne l'identifiant du signal créé avec la liste mise à jour.

Exemple 49

Si on initialise pour la première fois alors $\iota(\{\}) = (0, \{\langle 0, \langle false, \{\}, \{\}, \{\}\rangle\rangle\})$
sinon $\iota(\{..., \langle s, data\rangle\}) = (s+1, \{..., \langle s, data\rangle, \langle s+1, \langle false, \{\}, \{\}, \{\}\rangle\rangle\})$ *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 50 $SI(s) = \langle emit, CS, SSI, ST\rangle$

- $\tau(SI)$ 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 51 $\tau(SI) = \forall\ si \in SI$:

- $\langle true, CS, SSI, \{\}\rangle \rightarrow \langle false, \{\}, CS', \{\}\rangle$ *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.

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

Exemple 52 *Trois cas sont possibles :*

1. *Première fois que l'on prend :* $\langle id', \langle\{\langle b, \emptyset\rangle, \langle n, IL\rangle, ...\}, IL\rangle\rangle \rightarrow \langle id', \langle\{\langle b, \emptyset\rangle, \langle n, IL\ id\rangle, ...\}, IL\rangle\rangle$ *et on retourne b*
2. *On a déjà pris :* $\langle id', \langle\{..., \langle b, \{..., id, ...\}\rangle, \langle n, IL\rangle, ...\}, IL\rangle\rangle \rightarrow \langle id', \langle\{..., \langle b, \{...\}\rangle, \langle n, IL\ id\rangle, ...\}, IL\rangle\rangle$ *et on retourne b*
3. *On prend le dernier :* $\langle id', \langle\{..., \langle b, \{..., id, ...\}\rangle\}, IL\rangle\rangle \rightarrow \langle id', \langle\{..., \langle b, \{...\}\rangle\}, IL\ id\rangle\rangle$ *et on retourne b*

- $SI[(s, i) \leftarrow b]$ 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 $emit$.

Exemple 53 *Pour* $SI(s) = \langle emit, CS, SSI, ST\rangle$ *on change SI telle que* $SI(s) = \langle true, CS, SSI', ST\rangle$
avec $SSI' = \gamma(id, i, SSI)$ *avec id l'identifiant du thread courant.*

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

Exemple 54 $SSI(i) = \langle CI, IL\rangle$

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

Si la règle utilisée n'est ni **Erreur levée gérée** ni **Erreur levée non gérée** ni **Création gestionnaire** :

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

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

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

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

$\langle S, E, n\ C, D\rangle \longrightarrow_{TTSIH} \langle n\ S, E, C, D\rangle$ avec $n =$ une constante b ou un identifiant de signal s

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

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

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

$\langle b_n, ..., b_1\ S, E, prim_{o^n}\ C, D\rangle \longrightarrow_{TTSIH} \langle V\ S, E, C, D\rangle$ avec $\delta(o^n\ b_1 ... b_n) = V$

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

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

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

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

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

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

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

Création thread : On crée un nouveau thread.

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

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

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

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

$\langle\langle I, s\ b\ n\ \langle\langle X, C'\rangle, E'\rangle\ S, E, get\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSIH} \langle\langle I, \emptyset, E'[X \leftarrow V], C', \langle S, E, C, D\rangle\rangle, TL, SI\rangle$
si pour $SI(s) = \langle emit, CS, SSI\rangle$ et $SSI(b) = \langle CI, IL\rangle$ on a $I \notin IL$ alors $\gamma(I, b, SSI(b)) = V$ sinon $n = V$

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

$\langle\langle I, S, E, init\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSIH} \langle\langle I, s\ S, E, C, D\rangle, TL, SI'\rangle$ 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 premier choix.

$\langle\langle I, \langle\langle X', C''\rangle, E\rangle\ \langle\langle X, C'\rangle, E\rangle\ s\ S, E, present\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSIH} \langle\langle I, S, E, C'\ C, D\rangle, TL, SI\rangle$
avec $SI(s) = \langle vraie, CS, SSI, TL\rangle$

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.

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

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.

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

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

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

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

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

Partie pour les types : Cette partie ajoute les types dans la machine.

Pattern : On a un pattern, on le met dans la pile.

$\langle S, E, [c, X_1 ... X_n]\ C, D\rangle \longrightarrow_{TTSIH} \langle [c, X_1 ... X_n]\ S, E, C, D\rangle$

Construction de type : On a la commande qui nous dit que l'on veut construire un type, on a le nombre de paramètre indiqué ainsi que le numéro de constructeur.

$\langle n\ b\ V_n\ ...\ V_1\ S, E, build\ C, D\rangle \longrightarrow_{TTSIH} \langle [b, V_1, ..., V_n]\ S, E, C, D\rangle$

Comparer deux types : On a deux types en tête de la pile et on a la commande compare, on compare leurs constructeurs et on retourne un booléen.

$\langle [c', V_1 ... V_n]\ [c, X_1 ... X_n]\ S, E, compare\ C, D\rangle \longrightarrow_{TTSIH} \langle\langle bool, E\rangle\ S, E, C, D\rangle$
si $c' = c$ alors $bool = true$ sinon $bool = false$

Comparaison neutre : On a un type et une variable en tête de la pile et on a la commande compare, on retourne vraie.

$\langle [c', V_1 ... V_n]\ X\ S, E, compare\ C, D\rangle \longrightarrow_{TTSIH} \langle\langle true, E\rangle\ S, E, C, D\rangle$

Décomposition pattern : On a un type et un pattern et on a la commande destruct, on décompose le type via le pattern et on met dans l'environnement chaque couple (X, V).

$\langle [c, V_1 ... V_n]\ [c, X_1 ... X_n]\ S, E, destruct\ C, D\rangle \longrightarrow_{TTSIH} \langle S, E\ [X_1 \leftarrow V_1] ... [X_n \leftarrow V_n], C, D\rangle$

Décomposition neutre : On a un type et la variable "_" et on a la commande destruct, on fait rien.

$\langle [c, V_1 ... V_n]\ [\_]\ S, E, destruct\ C, D\rangle \longrightarrow_{TTSIH} \langle S, E, C, D\rangle$

Partie pour la récursion : Cette partie rajoute la récursion qui n'était pas innée dans la machine.

Récursion : On ajoute dans l'environnement une substitution qui est elle-même récursive pour pouvoir rappeler notre élément récursive.

$\langle\langle\langle X, \langle X', C'\rangle\rangle, E'\rangle\ S, E, fix\ C, D\rangle \longrightarrow_{TTSIH} \langle\langle\langle X', C'\rangle, E'[X \leftarrow \langle\langle X', C'\rangle, E'[X \leftarrow ...]\rangle]\rangle\ S, E, C, D\rangle$

Partie pour les exceptions : Cette partie ajoute les exceptions et leurs gestions.

Erreur levée gérée : Une erreur est levée mais le gestionnaire d'exception contient un élément.

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

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

Erreur levée non gérée : Une erreur est levée et le gestionnaire d'exception est vide. Rien ne peut gérer cette exception on a donc une erreur de levée et un arrêt de la machine.

$\langle\langle I, [e]\ S, E, raise\ C, D, \emptyset\rangle, TL, SI, IP\rangle \longrightarrow_{TTSIH} \langle\langle I, [e], E, \epsilon, \emptyset, \emptyset\rangle, \emptyset, SI', IP\rangle$
avec $SI' = \forall\ si \in SI\ :\ si = \langle s, \langle emit, CS, SSI, TL\rangle\rangle$ devient $si = \langle s, \langle emit, CS, SSI, \emptyset\rangle\rangle$

Création gestionnaire : On a deux fermetures dans la pile d'exécution et on a la commande *catch*. On crée un nouveau gestionnaire que l'on met dans H, on sauvegarde la machine dans le dépôt et on commence à utiliser le contenu protègé.

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

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

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 ne rien faire.

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

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

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

la machine TTSIH peut s'arrêter dans 4 états différents :

- Soit on a une **constante** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSIH}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSIH} \langle\langle I, b\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a une **fonction** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSIH}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSIH} \langle\langle I, \langle\langle X, C\rangle, E\rangle\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a **rien** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSIH}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSIH} \langle\langle I, \epsilon, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a un **type** tel que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSIH}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSIH} \langle\langle I, [c, V^1 ... V^n]\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Sinon on a un **état inconnu** : on a une **erreur**

Chapitre 4

Conclusion

Le sujet traitait *la programmation réactive synchrone* à travers un travail d'implantation d'une machine virtuelle. Pour cela, nous sommes repartis de la base en apprenant le langage de programmation λ-calcul. On a continué notre apprentissage avec le langage de programmation ISWIM. On s'est intéressé aux machines abstraites qui découle de ces langages, c'est-à-dire les machines CC, SCC, CK, CEK et SECD. On sait aussi intéressé à la programmation réactive. De là, nous sommes partis d'une machine existante, la machine SECD, pour créer notre machine fonctionnelle réactive.

Dans un premier temps on a créé une machine abstraite fonctionnelle réactive pure. On a mis les bases de la programmation réactive via l'ajout des threads et des signaux en prenant le principe du langage SL pour connaître l'absence d'un signal.

Dans un deuxième temps on a créé une machine abstraite fonctionnelle réactive avec partage de valeurs. Les signaux ne sont plus seulement présents ou absents mais permettent de transférer des données entre chaque thread. On a créé une mémoire partagée découpée grâce au signal qui le contient et au thread qui la mis dans la mémoire.

Avant d'aller plus loin dans les implantations on a prouvé le déterminisme de la machine TTSI. C'est une preuve simple mais qui reste indispensable pour que notre machine abstraite fonctionnelle réactive puisse être utilisable.

Dans un troisième temps on a ajouté la récursivité à la machine. En effet, on l'avait perdu avec le typage. On a implanté une commande fix. C'est un ajout que l'on aurait pu faire bien plus tôt mais il ne nous était pas encore indispensable. Le prochain point le nécessite.

Dans un quatrième temps on a ajouté les types à notre machine. Les types sont des éléments quasi vitaux pour une machine abstraite fonctionnelle. On gère la construction et le filtrage sans ce soucier de ce que représente le type.

Enfin on a ajouté les exceptions, qui sont vu comme des types dans notre machine. Cela permet d'éviter de créer un élément différent pour les exceptions. Cependant les exceptions ne se propagent pas.

J'aimerai souligner tout le travail effectué durant le stage qui n'est pas présent dans le rapport ou seulement en Annexe. Beaucoup de versions intermédiaires de notre travail final existent ainsi que leurs implantations en OCaml.

Notre machine commence à être assez complète, il nous manque cependant la preuve du déterminisme de la dernière version de la machine. Les ajouts restent simples à implanter dans notre machine donc encore beaucoup d'ajouts peuvent être faits dans le temps tant que nous restons dans la logique de fonctionnement prise depuis le début. Personnellement je ne vois pas ce que l'on pourrait ajouter à notre machine à part peut être une gestion des variables globales et locales plus explicite car cela reste assez flou.

Ce stage m'a permi d'entrevoir le monde de la recherche, ses avantages et ses inconvénients. Je pense avoir gagné en rigueur, grâce à l'écriture de la sémantique stricte ainsi que la preuve de déterminisme qui est en cours d'écriture, ainsi qu'en rapidité d'exécution grâce aux dates butoirs données pour faire chaque parties de la machine. Je suis content d'avoir postulé pour ce stage, le monde de la recherche est fascinant.

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] *Java Fair Threads* de Frédéric Bussinot

[5] *The SL Synchronous Language* de Frédéric Bussinot et Robert de Simone

[6] *Icobj Programming* de Frédéric Bussinot

Annexes

Les Exemples des machines

Cette section des annexes regroupent les exemples des machines étudiées et créées.

Exemple 55 *Voici un exemple de fonctionnement de la machine CK :*

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

Exemple 56 *Voici un exemple de fonctionnement de la machine CEK :*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$> (5)\ \langle\langle(o^n\ M\ N...), \varepsilon\rangle, \overline{\kappa}\rangle \longmapsto_{cek} \langle\langle M, \varepsilon\rangle, \langle opd, \langle o^n\rangle, \langle\langle N, \varepsilon\rangle, ...\rangle, \overline{\kappa}\rangle\rangle$

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

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

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

$> (7)\ \langle\langle V, \varepsilon\rangle, \langle opd, \langle v', ...o^n\rangle, \langle\langle N, \varepsilon'\rangle, c, ...\rangle, \overline{\kappa}\rangle\rangle \longmapsto_{cek} \langle\langle N, \varepsilon'\rangle, \langle opd, \langle\langle V, \varepsilon\rangle, v', ...o^n\rangle, \langle c, ...\rangle, \overline{\kappa}\rangle\rangle\ si\ V \notin X$

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

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

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

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

$\langle\langle\ulcorner 2\urcorner, \emptyset\rangle, mt\rangle$

Exemple 57 *Voici un exemple de fonctionnement de la machine SECD :*

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

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

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

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

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

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) $\langle \widehat{S}, \widehat{\varepsilon}, \langle X, C'\rangle\ \widehat{C}, \widehat{D}\rangle \longmapsto_{secd} \langle\langle\langle X, C'\rangle, \varepsilon\rangle\ \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D}\rangle$

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) $\langle \widehat{S}, \widehat{\varepsilon}, \langle X, C'\rangle\ \widehat{C}, \widehat{D}\rangle \longmapsto_{secd} \langle\langle\langle X, C'\rangle, \varepsilon\rangle\ \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D}\rangle$

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

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

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

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

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

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

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

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

> (1) $\langle \widehat{S}, \widehat{\varepsilon}, b\ \widehat{C}, \widehat{D}\rangle \longmapsto_{secd} \langle b\ \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D}\rangle$

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

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

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

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

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

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

> (2) $\langle \widehat{S}, \widehat{\varepsilon}, X\ \widehat{C}, \widehat{D}\rangle \longmapsto_{secd} \langle \widehat{V}\ \widehat{S}, \widehat{\varepsilon}, v\ \widehat{C}, \widehat{D}\rangle$ *où* $\widehat{V} = \varepsilon(X)$

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) $\langle \widehat{V}\ \langle\langle X, C'\rangle, \varepsilon'\rangle\ \widehat{S}, \widehat{\varepsilon}, ap\ \widehat{C}, \widehat{D}\rangle \longmapsto_{secd} \langle \epsilon, \varepsilon'[X \leftarrow \widehat{V}], C', \langle \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D}\rangle\rangle$

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

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

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

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

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

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

> (6) $\langle b_1\ ...\ b_n \widehat{S}, \widehat{\varepsilon}, prim_{o^n}\ \widehat{C}, \widehat{D}\rangle \longmapsto_{secd} \langle \widehat{V}\ \widehat{S}, \widehat{\varepsilon}, \widehat{C}, \widehat{D}\rangle$ *où* $\widehat{V} = \delta(o^n, b_1, ...b_n)$

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

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

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

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

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

Exemple 58 *Voici un exemple de fonctionnement de la machine SECD :*

$TTS\ Machine : \langle\langle\epsilon, \emptyset, \langle s, \langle, s\ \langle, 6\rangle\ \langle, 9\rangle\ present\rangle\ spawn\ \langle, s\ emit\rangle\ spawn\ ap\rangle\ init\ ap, \emptyset\rangle, \emptyset, \emptyset\rangle$

$>$ ***Abstraction*** $\langle S, E, \langle X, C'\rangle\ C, D\rangle \longrightarrow_{TTS} \langle\langle\langle X, C'\rangle, E\rangle\ S, E, C, D\rangle$

$TTS\ Machine : \langle\langle\langle\langle s, \langle, s\ \langle, 6\rangle\ \langle, 9\rangle\ present\rangle\ spawn\ \langle, s\ emit\rangle\ spawn\ ap\rangle, \emptyset\rangle, \emptyset, init\ ap, \emptyset\rangle, \emptyset, \emptyset\rangle$

$>$ ***Initialisation signal*** $\langle\langle S, E, init\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTS} \langle\langle s\ S, E, C, D\rangle, TL, SI'\rangle$ *avec* $\iota(SI) = (s, SI')$

$TTS\ Machine : \langle\langle 0\ \langle\langle s, \langle, s\ \langle, 6\rangle\ \langle, 9\rangle\ present\rangle\ spawn\ \langle, s\ emit\rangle\ spawn\ ap\rangle, \emptyset\rangle, \emptyset, ap, \emptyset\rangle, \emptyset, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

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

$TTS\ Machine : \langle\langle\epsilon, \emptyset[s \leftarrow 0], \langle, s\ \langle, 6\rangle\ \langle, 9\rangle\ present\rangle\ spawn\ \langle, s\ emit\rangle\ spawn\ ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

$TTS\ Machine : \langle\langle\epsilon, \{\langle s, 0\rangle\}, \langle, s\ \langle, 6\rangle\ \langle, 9\rangle\ present\rangle\ spawn\ \langle, s\ emit\rangle\ spawn\ ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

$>$ ***Abstraction*** $\langle S, E, \langle X, C'\rangle\ C, D\rangle \longrightarrow_{TTS} \langle\langle\langle X, C'\rangle, E\rangle\ S, E, C, D\rangle$

$TTS\ Machine : \langle\langle\langle, s\ \langle, 6\rangle\ \langle, 9\rangle\ present\rangle, \{\langle s, 0\rangle\}\rangle, \{\langle s, 0\rangle\}, spawn\ \langle, s\ emit\rangle\ spawn\ ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

$>$ ***Création thread*** $\langle\langle\langle\langle X, C'\rangle, E\rangle\ S, E, spawn\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTS} \langle\langle S, E, C, D\rangle, TL\ \langle S, E, C', D\rangle, SI\rangle$

$TTS\ Machine : \langle\langle\epsilon, \{\langle s, 0\rangle\}, \langle, s\ emit\rangle\ spawn\ ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle\epsilon, \{\langle s, 0\rangle\}, s\ \langle, 6\rangle\ \langle, 9\rangle\ present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

$>$ ***Abstraction*** $\langle S, E, \langle X, C'\rangle\ C, D\rangle \longrightarrow_{TTS} \langle\langle\langle X, C'\rangle, E\rangle\ S, E, C, D\rangle$

$TTS\ Machine : \langle\langle\langle, s\ emit\rangle, \{\langle s, 0\rangle\}\rangle, \{\langle s, 0\rangle\}, spawn\ ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle\epsilon, \{\langle s, 0\rangle\}, s\ \langle, 6\rangle\ \langle, 9\rangle\ present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}$
$, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

$>$ ***Création thread*** $\langle\langle\langle\langle X, C'\rangle, E\rangle\ S, E, spawn\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTS} \langle\langle S, E, C, D\rangle, TL\ \langle S, E, C', D\rangle, SI\rangle$

$TTS\ Machine : \langle\langle\epsilon, \{\langle s, 0\rangle\}, ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle\epsilon, \{\langle s, 0\rangle\}, s\ \langle, 6\rangle\ \langle, 9\rangle\ present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \langle\epsilon, \{\langle s, 0\rangle\}, s\ emit, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}$
$, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

$>$ ***Application neutre*** $\langle S, E, ap\ C, D\rangle \longrightarrow_{TTS} \langle S, E, C, D\rangle$

$TTS\ Machine : \langle\langle\epsilon, \{\langle s, 0\rangle\}, \epsilon, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle\epsilon, \{\langle s, 0\rangle\}, s\ \langle, 6\rangle\ \langle, 9\rangle\ present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \langle\epsilon, \{\langle s, 0\rangle\}, s\ emit, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}$
$, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

$>$ ***Récupération de sauvegarde avec pile vide*** $\langle S, E, \epsilon, \langle S', E', C, D\rangle\rangle \longrightarrow_{TTS} \langle S', E', C, D\rangle$

$TTS\ Machine : \langle\langle\epsilon, \emptyset, \epsilon, \emptyset\rangle, \{\langle\epsilon, \{\langle s, 0\rangle\}, s\ \langle, 6\rangle\ \langle, 9\rangle\ present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \langle\epsilon, \{\langle s, 0\rangle\}, s\ emit, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

$>$ ***Récupération dans la file d'attente*** $\langle\langle S, E, \epsilon, \emptyset\rangle, \langle S', E', C, D\rangle\ TL, SI\rangle \longrightarrow_{TTS} \langle\langle S', E', C, D\rangle, TL, SI\rangle$

$TTS\ Machine : \langle\langle\epsilon, \{\langle s, 0\rangle\}, s\ \langle, 6\rangle\ \langle, 9\rangle\ present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle\epsilon, \{\langle s, 0\rangle\}, s\ emit, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

$>$ ***Substitution*** $\langle S, E, X\ C, D\rangle \longrightarrow_{TTS} \langle V\ S, E, C, D\rangle$ *avec* $E(X) = V$

$TTS\ Machine : \langle\langle 0, \{\langle s, 0\rangle\}, \langle, 6\rangle\ \langle, 9\rangle\ present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle\epsilon, \{\langle s, 0\rangle\}, s\ emit, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

$>$ ***Abstraction*** $\langle S, E, \langle X, C'\rangle\ C, D\rangle \longrightarrow_{TTS} \langle\langle\langle X, C'\rangle, E\rangle\ S, E, C, D\rangle$

$TTS\ Machine : \langle\langle\langle\langle, 6\rangle, \{\langle s, 0\rangle\}\rangle\ 0, \{\langle s, 0\rangle\}, \langle, 9\rangle\ present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle\epsilon, \{\langle s, 0\rangle\}, s\ emit, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle false, \emptyset\rangle\rangle\}\rangle$

$>$ ***Abstraction*** $\langle S, E, \langle X, C'\rangle\ C, D\rangle \longrightarrow_{TTS} \langle\langle\langle X, C'\rangle, E\rangle\ S, E, C, D\rangle$

$TTS\ Machine : \langle\langle\langle\langle, 9\rangle, \{\langle s, 0\rangle\}\rangle\ \langle\langle, 6\rangle, \{\langle s, 0\rangle\}\rangle\ 0, \{\langle s, 0\rangle\}, present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle\epsilon, \{\langle s, 0\rangle\}, s\ emit, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle false,$
$\emptyset\rangle\rangle\}\rangle$

$>$ ***Thread bloqué remplacé*** $\langle\langle\langle\langle X', C''\rangle, E\rangle\ \langle\langle X, C'\rangle, E\rangle\ s\ S, E, present\ C, D\rangle, \langle S', E', C''', D'\rangle\ TL, SI\rangle$
$\longrightarrow_{TTS} \langle\langle S', E', C''', D'\rangle, TL, SI'\rangle$
avec $SI(s) = \langle faux, ST\rangle$ *et* $SI'(s) = \langle faux, ST\ \langle\langle\langle X', C''\rangle, E\rangle\ \langle\langle X, C'\rangle, E\rangle\ s\ S, E, present\ C, D\rangle\rangle$

$TTS\ Machine : \langle\langle\epsilon, \{\langle s, 0\rangle\}, s\ emit, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false,$
$\{\langle\langle\langle, 9\rangle, \{\langle s, 0\rangle\}\rangle\ \langle\langle, 6\rangle, \{\langle s, 0\rangle\}\rangle\ 0, \{\langle s, 0\rangle\}, present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}\rangle\rangle\}\rangle$

$>$ ***Substitution*** $\langle S, E, X\ C, D\rangle \longrightarrow_{TTS} \langle V\ S, E, C, D\rangle$ *avec* $E(X) = V$

$TTS\ Machine : \langle\langle 0, \{\langle s, 0\rangle\}, emit, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false,$
$\{\langle\langle\langle, 9\rangle, \{\langle s, 0\rangle\}\rangle\ \langle\langle, 6\rangle, \{\langle s, 0\rangle\}\rangle\ 0, \{\langle s, 0\rangle\}, present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}\rangle\rangle\}\rangle$

$>$ ***Émettre :*** $\langle\langle s\ S, E, emit\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTS} \langle\langle S, E, C, D\rangle, TL, SI'\rangle$ *avec* $\varepsilon(s, SI) = SI'$

$TTS\ Machine : \langle\langle\epsilon, \{\langle s, 0\rangle\}, \epsilon, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle\langle\langle, 9\rangle, \{\langle s, 0\rangle\}\rangle\ \langle\langle, 6\rangle, \{\langle s, 0\rangle\}\rangle\ 0, \{\langle s, 0\rangle\}, present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle true,$
$\emptyset\rangle\rangle\}\rangle$

$>$ ***Récupération de sauvegarde avec pile vide*** $\langle S, E, \epsilon, \langle S', E', C, D\rangle\rangle \longrightarrow_{TTS} \langle S', E', C, D\rangle$

$TTS\ Machine : \langle\langle\epsilon, \emptyset, \epsilon, \emptyset\rangle, \{\langle\langle\langle, 9\rangle, \{\langle s, 0\rangle\}\rangle\ \langle\langle, 6\rangle, \{\langle s, 0\rangle\}\rangle\ 0, \{\langle s, 0\rangle\}, present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle true, \emptyset\rangle\rangle\}\rangle$

$>$ ***Récupération dans la file d'attente*** $\langle\langle S, E, \epsilon, \emptyset\rangle, \langle S', E', C, D\rangle\ TL, SI\rangle \longrightarrow_{TTS} \langle\langle S', E', C, D\rangle, TL, SI\rangle$

$TTS\ Machine : \langle\langle\langle\langle, 9\rangle, \{\langle s, 0\rangle\}\rangle\ \langle\langle, 6\rangle, \{\langle s, 0\rangle\}\rangle\ 0, \{\langle s, 0\rangle\}, present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle true, \emptyset\rangle\rangle\}\rangle$

$>$ ***Présence d'un signal*** $\langle\langle\langle\langle X', C''\rangle, E\rangle\ \langle\langle X, C'\rangle, E\rangle\ s\ S, E, present\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTS} \langle\langle S, E, C'\ C, D\rangle, TL, SI\rangle$
avec $SI(s) = \langle vraie, ST\rangle$

$TTS\ Machine : \langle\langle\epsilon, \{\langle s, 0\rangle\}, 6, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle true, \emptyset\rangle\rangle\}\rangle$

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

$TTS\ Machine : \langle\langle 6, \{\langle s, 0\rangle\}, \epsilon, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle true, \emptyset\rangle\rangle\}\rangle$

$>$ ***Récupération de sauvegarde*** $\langle V\ S, E, \epsilon, \langle S', E', C, D\rangle\rangle \longrightarrow_{TTS} \langle V\ S', E', C, D\rangle$

$TTS\ Machine : \langle\langle 6, \emptyset, \epsilon, \emptyset\rangle, \emptyset, \{\langle 0, \langle true, \emptyset\rangle\rangle\}\rangle$

Exemple 59 *Voici un exemple de fonctionnement de la machine TTSI :*

Machine TTSI $\langle\langle 0, \epsilon, \emptyset, \langle s, \langle s', \langle, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present\rangle\ spawn\ 10\ s\ put\ ap\rangle\rangle\ init\ ap\ init\ ap, \emptyset\rangle, \emptyset, \emptyset, 1\rangle$

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

Machine TTSI $\langle\langle 0, \langle\langle s, \langle s', \langle, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present\rangle\ spawn\ 10\ s\ put\ ap\rangle\rangle, \emptyset\rangle, \emptyset, init\ ap\ init\ ap, \emptyset\rangle,$
$\emptyset, \emptyset, 1\rangle$

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

Machine TTSI $\langle\langle 0, \langle 0\ \langle s, \langle s', \langle, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present\rangle\ spawn\ 10\ s\ put\ ap\rangle\rangle, \emptyset\rangle, \emptyset, ap\ init\ ap, \emptyset\rangle,$
$\emptyset, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 1\rangle$

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

Machine TTSI $\langle\langle 0, \epsilon, \emptyset[s \leftarrow 0], \langle s', \langle, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present\rangle\ spawn\ 10\ s\ put\ ap\rangle, \langle\epsilon, \emptyset, init\ ap, \emptyset\rangle\rangle,$
$\emptyset, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 1\rangle$

Machine TTSI $\langle\langle 0, \epsilon, \{\langle s, 0\rangle\}, \langle s', \langle, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present\rangle\ spawn\ 10\ s\ put\ ap\rangle, \langle\epsilon, \emptyset, init\ ap, \emptyset\rangle\rangle,$
$\emptyset, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 1\rangle$

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

Machine TTSI $\langle\langle 0, \langle\langle s', \langle, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present\rangle\ spawn\ 10\ s\ put\ ap\rangle, \{\langle s, 0\rangle\}\rangle, \{\langle s, 0\rangle\}, \epsilon,$
$\langle\epsilon, \emptyset, init\ ap, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 1\rangle$

$>$ ***Récupération de sauvegarde*** $\langle V\ S, E, \epsilon, \langle S', E', C, D\rangle\rangle \longrightarrow_{TTSI} \langle V\ S', E', C, D\rangle$

Machine TTSI $\langle\langle 0, \langle\langle s', \langle, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present\rangle\ spawn\ 10\ s\ put\ ap\rangle, \{\langle s, 0\rangle\}\rangle, \emptyset, init\ ap, \emptyset\rangle$
$, \emptyset, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 1\rangle$

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

Machine TTSI $\langle\langle 0, 1\ \langle\langle s', \langle, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present\rangle\ spawn\ 10\ s\ put\ ap\rangle, \{\langle s, 0\rangle\}\rangle, \emptyset, ap, \emptyset\rangle$
$, \emptyset, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 1\rangle$

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

Machine TTSI $\langle\langle 0, \epsilon, \{\langle s, 0\rangle\}[s' \leftarrow 1], \langle, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present\rangle\ spawn\ 10\ s\ put\ ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle$
$, \emptyset, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 1\rangle$

Machine TTSI $\langle\langle 0, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, \langle, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present\rangle\ spawn\ 10\ s\ put\ ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle$
$, \emptyset, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 1\rangle$

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

Machine TTSI $\langle\langle 0, \langle, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present\rangle, \{\langle s, 0\rangle, \langle s', 1\rangle\}\rangle, \{\langle s, 0\rangle, \langle s', 1\rangle\}, spawn\ 10\ s\ put\ ap,$
$\langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 1\rangle$

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

Machine TTSI $\langle\langle 0, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, 10\ s\ put\ ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle 1, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present$
$\langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 2\rangle$

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

Machine TTSI $\langle\langle 0, 10, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s\ put\ ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle 1, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present,$
$\langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 2\rangle$

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

Machine TTSI $\langle\langle 0, 0\ 10, \{\langle s, 0\rangle, \langle s', 1\rangle\}, put\ ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle 1, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present,$
$\langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 2\rangle$

$>$ ***Ajouter dans un signal*** $\langle\langle I, s\ b\ S, E, put\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSI} \langle\langle I, S, E, C, D\rangle, TL, SI\ [(s, I) \leftarrow b]\rangle$

Machine TTSI $\langle\langle 0, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle 1, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present,$
$\langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}[(0, 0) \leftarrow 10], 2\rangle$

Machine TTSI $\langle\langle 0, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, ap, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle 1, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present,$
$\langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle true, \{\langle 0, \{10\}\rangle\}, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 2\rangle$

$>$ ***Application neutre*** $\langle S, E, ap\ C, D\rangle \longrightarrow_{TTSI} \langle S, E, C, D\rangle$

Machine TTSI $\langle\langle 0, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, \epsilon, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \{\langle 1, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present,$
$\langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle true, \{\langle 0, \{10\}\rangle\}, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 2\rangle$

$>$ ***Récupération de sauvegarde neutre*** $\langle S, E, \epsilon, \langle S', E', C, D\rangle\rangle \longrightarrow_{TTSI} \langle S', E', C, D\rangle$

Machine TTSI $\langle\langle 0, \epsilon, \emptyset, \epsilon, \emptyset\rangle, \{\langle 1, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present,$
$\langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}, \{\langle 0, \langle true, \{\langle 0, \{10\}\rangle\}, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 2\rangle$

$>$ ***Récupération dans la file d'attente*** $\langle\langle I, S, E, \epsilon, \emptyset\rangle, \langle I', S', E', C, D\rangle\ TL, SI\rangle \longrightarrow_{TTSI} \langle\langle I', S', E', C, D\rangle, TL, SI\rangle$

Machine TTSI $\langle\langle 1, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s'\ \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle true, \{\langle 0, \{10\}\rangle\}, \emptyset, \emptyset\rangle\rangle,$
$\langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 2\rangle$

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

Machine TTSI $\langle\langle 1, 1, \{\langle s, 0\rangle, \langle s', 1\rangle\}, \langle, 1\rangle\ \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle true, \{\langle 0, \{10\}\rangle\}, \emptyset, \emptyset\rangle\rangle,$
$\langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 2\rangle$

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

Machine TTSI $\langle\langle 1, \langle\langle, 1\rangle, \{\langle s, 0\rangle, \langle s', 1\rangle\}\rangle\ 1, \{\langle s, 0\rangle, \langle s', 1\rangle\}, \langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle\ present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset,$
$\{\langle 0, \langle true, \{\langle 0, \{10\}\rangle\}, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 2\rangle$

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

Machine TTSI $\langle\langle 1, \langle\langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle, \{\langle s, 0\rangle, \langle s', 1\rangle\}\rangle\ \langle\langle, 1\rangle, \{\langle s, 0\rangle, \langle s', 1\rangle\}\rangle\ 1, \{\langle s, 0\rangle, \langle s', 1\rangle\}, present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle,$
$\emptyset, \{\langle 0, \langle true, \{\langle 0, \{10\}\rangle\}, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 2\rangle$

Thread bloqué non remplacé $\langle\langle I, \langle\langle X', C''\rangle, E\rangle\ \langle\langle X, C'\rangle, E\rangle\ s\ S, E, C, D\rangle, \emptyset, SI, IP\rangle \longrightarrow_{TTSI}$
$\langle\langle IP, \emptyset, \epsilon, \emptyset, \emptyset\rangle, \emptyset, SI', IP + 1\rangle$ *avec* $SI(s) = \langle faux, CS, SSI, ST\rangle$
et $SI'(s) = \langle faux, CS, SSI, ST\ \langle I, \langle\langle X', C''\rangle, E\rangle\ \langle\langle X, C'\rangle, E\rangle\ s\ S, E, present\ C, D\rangle\rangle$

Machine TTSI $\langle\langle 2, \epsilon, \emptyset, \epsilon, \emptyset\rangle, \emptyset, \{\langle 0, \langle true, \{\langle 0, \{10\}\rangle\}, \emptyset, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \{\langle 1, \langle\langle, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+\rangle,$
$\{\langle s, 0\rangle, \langle s', 1\rangle\}\rangle\ \langle\langle, 1\rangle, \{\langle s, 0\rangle, \langle s', 1\rangle\}\rangle\ 1, \{\langle s, 0\rangle, \langle s', 1\rangle\}, present, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\}\rangle\rangle\}, 3\rangle$

$>$ ***Fin d'instant logique*** $\langle\langle I, S, E, \epsilon, \emptyset\rangle, \emptyset, SI\rangle \longrightarrow_{TTSI} \langle\langle I, S, E, \epsilon, \emptyset\rangle, TL, SI'\rangle$ *avec* $\tau(SI) = (SI', TL)$

Machine TTSI $\langle\langle 2, \epsilon, \emptyset, \epsilon, \emptyset\rangle, \{\langle 1, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle\},$
$\{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \emptyset\rangle\}, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

$>$ ***Récupération dans la file d'attente*** $\langle\langle I, S, E, \epsilon, \emptyset\rangle, \langle I', S', E', C, D\rangle\ TL, SI\rangle \longrightarrow_{TTSI} \langle\langle I', S', E', C, D\rangle, TL, SI\rangle$

Machine TTSI $\langle\langle 1, \epsilon, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s\ 0\ 3\ get\ s\ 0\ 3\ get\ prim_+, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \emptyset\rangle\}, \emptyset\rangle\rangle,$
$\langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

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

Machine TTSI $\langle\langle 1, 0, \{\langle s, 0\rangle, \langle s', 1\rangle\}, 0\ 3\ get\ s\ 0\ 3\ get\ prim_+, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \emptyset\rangle\}, \emptyset\rangle\rangle,$
$\langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

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

Machine TTSI $\langle\langle 1, 0\ 0, \{\langle s, 0\rangle, \langle s', 1\rangle\}, 3\ get\ s\ 0\ 3\ get\ prim_+, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \emptyset\rangle\}, \emptyset\rangle\rangle,$
$\langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

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

Machine TTSI $\langle\langle 1, 3\ 0\ 0, \{\langle s, 0\rangle, \langle s', 1\rangle\}, get\ s\ 0\ 3\ get\ prim_+, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \emptyset\rangle\}, \emptyset\rangle\rangle,$
$\langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

$>$ ***Prendre une valeur partagée*** $\langle\langle I, s\ b\ n\ \langle\langle X, C'\rangle, E'\rangle\ S, E, get\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSI}$
$\langle\langle I, \emptyset, E'[X \leftarrow V], C', \langle S, E, C, D\rangle\rangle, TL, SI\rangle$
si pour $SI(s) = \langle emit, CS, SSI\rangle$ *et* $SSI(b) = \langle CI, IL\rangle$ *on a* $I \notin IL$ *alors* $\gamma(SSI(b)) = V$ *sinon* $n = V$

Machine TTSI $\langle\langle 1, 10, \{\langle s, 0\rangle, \langle s', 1\rangle\}, s\ 0\ 3\ get\ prim_+, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \{1\}\rangle\}, \emptyset\rangle\rangle,$
$\langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

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

Machine TTSI $\langle\langle 1, 0\ 10, \{\langle s, 0\rangle, \langle s', 1\rangle\}, 0\ 3\ get\ prim_+, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \{1\}\rangle\}, \emptyset\rangle\rangle,$
$\langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

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

Machine TTSI $\langle\langle 1, 0\ 0\ 10, \{\langle s, 0\rangle, \langle s', 1\rangle\}, 3\ get\ prim_+, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \{1\}\rangle\}, \emptyset\rangle\rangle,$
$\langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

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

Machine TTSI $\langle\langle 1, 3\ 0\ 0\ 10, \{\langle s, 0\rangle, \langle s', 1\rangle\}, get\ prim_+, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \{1\}\rangle\}, \emptyset\rangle\rangle,$
$\langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

$>$ ***Prendre une valeur partagée*** $\langle\langle I, s\ b\ n\ \langle\langle X, C'\rangle, E'\rangle\ S, E, get\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSI}$
$\langle\langle I, \emptyset, E'[X \leftarrow V], C', \langle S, E, C, D\rangle\rangle, TL, SI\rangle$
si pour $SI(s) = \langle emit, CS, SSI\rangle$ *et* $SSI(b) = \langle CI, IL\rangle$ *on a* $I \notin IL$ *alors* $\gamma(SSI(b)) = V$ *sinon* $n = V$

Machine TTSI $\langle\langle 1, 3\ 10, \{\langle s, 0\rangle, \langle s', 1\rangle\}, prim_+, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \{1\}\rangle\}, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

Opération $\langle b_n, ..., b_1\ S, E, prim_{o^n}\ C, D\rangle \longrightarrow_{TTSI} \langle V\ S, E, C, D\rangle$ *avec* $\delta(o^n\ b_1...b_n) = V$

Machine TTSI $\langle\langle 1, 13, \{\langle s, 0\rangle, \langle s', 1\rangle\}, \epsilon, \langle\epsilon, \emptyset, \epsilon, \emptyset\rangle\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \{1\}\rangle\}, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

$>$ ***Récupération de sauvegarde*** $\langle V\ S, E, \epsilon, \langle S', E', C, D\rangle\rangle \longrightarrow_{TTSI} \langle V\ S', E', C, D\rangle$

Machine TTSI $\langle\langle 1, 13, \emptyset, \epsilon, \emptyset\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \{1\}\rangle\}, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

Format de la machine et règle en fonction du champs d'action de l'initialisation

Un signal peut être initialisé pour 3 champs d'actions possibles :

1. la machine
2. le thread courant
3. la chaîne de contrôle

Le premier cas a été traité et présenté dans la version officiel. Il reste intéressant de voir les versions alternatives que peut engranger un simple changement de champs d'action. On va d'abord prendre au plus large pour aller au plus restreint, on passe donc à la restriction du thread courant.

Initialisation pour le thread courant Lorsque l'on va initialiser il va falloir garder en mémoire quel thread l'a fait. Ce point nous contraint à créer des identifiants pour les threads. Pour cela on va ajouter un entier dans notre machine que l'on va nommer *producteur d'identifiant (IP)* et qui , comme son nom l'indique, produira un identifiant pour chaque thread créé. Cela va nous obliger à modifier un peu les threads tels que $T = \langle I, S, E, C, D\rangle$ avec l'identifiant I ainsi que la règle du *spaw*n qui va donner la règle suivant :

$\langle\langle I, \langle\langle X, C'\rangle, E\rangle\ S, E, spawn\ C, D\rangle, TL, SI, IP\rangle \longrightarrow \langle\langle I, IP\ S, E, C, D\rangle, TL\ \langle IP, \epsilon, E', C', \emptyset\rangle, SI, IP + 1\rangle$

Maintenant on peut différencier nos threads. On va s'en servir pour stocker l'information de l'initialisation. Notre liste SI va contenir des éléments composés de trois informations :

1. L'identifiant du signal *id* : on va éviter que l'utilisateur contrôle les identifiants, on lui retournera simplement l'identifiant dans la pile d'exécution pour qu'il puisse le lié à une variable
2. une liste d'identifiant IL : on va stocker les identifiants des threads pour lesquels ce signal est initialisé
3. un booléen *emit* : il va nous permettre de savoir si un signal est émis

Les signaux vont devoir être différenciable eux aussi. On va utiliser des identifiants pour cela. Comme dit plus haut on ne souhaite pas que l'utilisateur est la main sur les identifiants on va donc devoir les créer. On pourrait simplement reprendre ce qui à été fait pour les threads, c'est-à-dire, utiliser un élément ajouter à la machine qui va nous servir de compteur. Cependant il faudrait un compteur par thread du coup ce ne serait pas très efficace.

Nos signaux ne sont pas détruit lors du fonctionnement de la machine, ils restent jusqu'à la fin. Avec cela en tête, on peut modifier la liste en file et donc garder un ordre dans les signaux et finalement pouvoir donner un nouvelle identifiant sans utiliser de compteur exterieur.

Une difficulté s'ajoute à cela. On va devoir faire une recherche dans notre file, on va prendre les signaux deux par deux et si le premier a été initialisé par notre thread mais pas le second alors le second le devient.

Exemple 60 *On cherche à initialise un signal pour le thread qui a pour identifiant I.*

1. *Pour chaque pair $(\langle s, il, emit\rangle, \langle s', il', emit'\rangle)$ de SI :*
2. *Si $I \in il$ et $I \notin il'$ alors : on ajoute I à il' et on retourne s'*

Quand on initialise un signal on doit savoir si le thread a déjà initialisé un signal auparavant. Car sinon on ne saura pas qu'il faut prendre le premier signal. Du coup il va falloir vérifier préalablement si le thread a déjà initialisé un signal.

Exemple 61 *On cherche à vérifier si le thread I a déjà initialisé un signal.*

1. *Pour chaque $\langle s, il, emit\rangle$ de SI :*
2. *Si $I \in il$ alors : on retourne $false$*
3. *on retourne $true$*

Si il n'y a pas de signal dans SI on va créer un premier signal avec l'identifant 0. Si la file est impair alors on peut avoir le cas où le dernière élément de SI contient aussi notre identifiant de thread il va donc falloir créer un nouveau signal en prenant l'identifiant du dernier signal et en l'incrémentant de un.

Exemple 62 *On cherche à initialiser un signal pour le thread qui a pour identifiant I.*

1. *Pour chaque $\langle s, il, emit\rangle$ de SI :*
2. *Si $I \in il$ et $\langle s, il, emit\rangle$ est le dernier élément*
3. *alors : on ajoute $\langle s + 1, \{I\}, false\rangle$ dans la file SI et on retourne $s + 1$*

On va donc se retrouver avec la règle suivante : $\langle\langle I, S, E, init\ C, D\rangle, TL, SI, IP\rangle \longrightarrow \langle\langle I, id\ S, E, C, D\rangle, TL, SI', IP\rangle$

avec $initialise(SI, I)$, une fonction qui modifie SI en initialisant un signal pour un thread donné I et qui retourne le couple identifiant, nouvelle file de signaux (id, SI').

Il est possible que ce ne soit pas encore très clair pour vous du coup je vous renvoie à l'adresse du git comprenant toutes mes implantations ainsi vous pourrez voir comment je l'ai implanté.

Initialisation pour une chaîne de contrôle Limité à une chaîne de contrôle va nous obliger à créer une nouvelle forme de sauvegarde. Je m'explique, on va utiliser une abstraction pour stocker la partie de la chaîne de contrôle pour laquelle le signal va être initialisé. On va prendre le principe de la règle d'application qui est la suivante :

$\langle\langle\langle X, C'\rangle, E'\rangle\ V\ S, E, ap\ C, D\rangle \longrightarrow_{SECD} \langle\epsilon, E'[X \leftarrow V], C', \langle S, E, C, D\rangle\rangle$

L'idée est que lorsque l'on va initialiser un signal on va sauvegarder la machine dans le dépôt pour se concentrer sur la chaîne de contrôle pour laquelle notre signal est initialisé. Le problème est de différencier les sauvegardes normales et les sauvegardes pour l'initialisation. Pour cela, on va créer une nouvelle forme de la sauvegarde qui contient l'identifiant de signal telle que : $D = \langle id, \langle S, E, C, D\rangle\rangle$ avec l'identifiant du signal initialisé id.

Il va aussi falloir enlever l'initialisation en sortie de la chaîne de contrôle et donc la reprise de la sauvegarde. Pour cela on va ajouter un booléen pour chaque élément de la liste IL. Du coup on aura une file de couple $\langle i, init\rangle$ avec l'identifiant du thread i et le booléen représentant l'initialisation $init$.

La règle d'initialisation va donc être un peu et va donner :

$\langle\langle I, \langle\langle X, C'\rangle, E'\rangle\ S, E, init\ C, D\rangle, TL, SI, IP\rangle \longrightarrow \langle\langle I, id, E', C', \langle id, \langle S, E, C, D\rangle\rangle\rangle, TL, SI', IP\rangle$

avec $initialise(SI, I)$, une fonction qui modifie SI en initialisant un signal pour un thread donné I et qui retourne le couple identifiant, nouvelle file de signaux (id, SI').

Les différentes versions faite pour rendre la machine SECD concurrente

Cette partie énumére 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.

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 et les émettre où encore de tester la présence d'un signal. Cette version est un condensée de 2 versions.

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

$V = b$
$\quad | \ \langle\langle X, C\rangle, E\rangle$
$S = \epsilon$
$\quad | \ V \ S$
$\quad | \ Remp \ S$
$E =$ une fonction $\{\langle X, V\rangle, ...\}$
$C = \epsilon$
$\quad | \ b \ C$
$\quad | \ X \ C$
$\quad | \ ap \ C$
$\quad | \ prim_{o^n} \ C$
$\quad | \ \langle X, C\rangle \ C$
$\quad | \ bspawn \ C$
$\quad | \ espawn \ C$
$\quad | \ \langle s, C', C''\rangle \ C$
$\quad | \ \langle s, C'\rangle \ C$
$\quad | \ emit_s \ C$
$D = \emptyset$
$\quad | \ \langle S, E, C, D\rangle$
$W = \{D, ...\}$
$ST = \{..., \langle s, D\rangle, ...\}$
$SI = \{s, ...\}$

Les nouvelles règles sont les suivantes :

Partie de base de la machine SECD

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

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

Substitution : On a une variable, on prend la substitution dans l'environnement lié à la variable via la fonction E et on la met dans la pile.

$\langle S, E, X \ C, D, W, ST, SI\rangle \longmapsto_{secdv1-2} \langle V \ S, E, C, D, W, ST, SI\rangle$ 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.

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

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

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

Application : On a une application, donc on 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.
$\langle V\ \langle\langle X, C'\rangle, E'\rangle\ S, E, ap\ C, D, W, ST, SI\rangle \longmapsto_{secdv1-2} \langle \epsilon, E'[X \leftarrow V], C', \langle S, E, C, D\rangle, W, ST, SI\rangle$

Récupération de sauvegarde : On a rien mais le dépôt comporte une sauvegarde donc on prend celle-ci.
$\langle V\ S, E, \epsilon, \langle S, E, C, D\rangle, W, ST, SI\rangle \longmapsto_{secdv1-2} \langle V\ S', E, C', D, W, ST, SI\rangle$

Partie pour la concurrence

Création thread : On crée un nouveau thread
$\langle S, E, bspawn\ C'\ espawn\ C, D, W, ST, SI\rangle \longmapsto_{secdv1-2} \langle Remp\ S, E, C, D, W\ \langle S, E, C', D\rangle, ST, SI\rangle$

Initialisation signal : On initialise le signal
$\langle S, E, \langle s, C'\rangle\ C, D, W, ST, SI\rangle \longmapsto_{secdv1-2} \langle \epsilon, E\ [init \leftarrow s], C', \langle S, E, C, D\rangle, W, ST, SI\rangle$

Présence d'un signal : On teste la présence d'un signal et il l'est donc on prend la 1ère option.
$\langle S, E, \langle s, C', C''\rangle\ C, D, W, ST, SI\rangle \longmapsto_{secdv1-2} \langle S, E, C'\ C, D, W, ST, SI\rangle$
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.
$\langle S, E, \langle s, C', C''\rangle\ C, D, \langle S', E', C''', D'\rangle W, ST, SI\rangle \longmapsto_{secdv1-2} \langle S', E', C''', D', W, ST\ \langle S, E, \langle s, C', C''\rangle\ C, D\rangle, SI\rangle$
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, on met juste le thread courant dans la liste de threads bloqués.
$\langle S, E, \langle s, C', C''\rangle\ C, D, \emptyset, ST, SI\rangle \longmapsto_{secdv1-2} \langle \emptyset, \emptyset, \epsilon, \emptyset, \emptyset, ST\ \langle S, E, \langle s, C', C''\rangle\ C, D\rangle, SI\rangle$
si $s \notin SI$ et $s \in E$

Émettre : on émet un signal
$\langle S, E, emit_s\ C, D, W, ST, SI\rangle \longmapsto_{secdv1-2} \langle S, E, C, D, W', ST', SI\rangle$
avec $W' = W \cup$ tous les éléments de ST qui attendent l'émission de s et
avec $ST' = ST \setminus$ tous 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.
$\langle S, E, \epsilon, \emptyset, \langle S', E', C, D\rangle\ W, ST, SI\rangle \longmapsto_{secdv1-2} \langle S', E', C, D, W, ST, SI\rangle$

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
$\langle S, E, \epsilon, \emptyset, \emptyset, ST, SI\rangle \longmapsto_{secdv1-2} \langle S, E, \emptyset, \emptyset, W, \emptyset, \emptyset\rangle$
avec $W =$ tous les éléments de ST prennent leurs 2nd choix

Partie commune

Application neutre droite : on a une application avec un neutre dans la pile donc on l'enlève
$\langle V\ Remp\ S, E, ap\ C, D, W, ST, SI\rangle \longmapsto_{secdv1-2} \langle V\ S, E, C, D, W, ST, SI\rangle$

Application neutre gauche : on a une application avec un neutre dans la pile donc on l'enlève
$\langle Remp\ V\ S, E, ap\ C, D, W, ST, SI\rangle \longmapsto_{secdv1-2} \langle V\ S, E, C, D, W, ST, SI\rangle$

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

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

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 a y avoir beaucoup d'éléments dans notre machine donc on va en rassembler certains. 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 :***

$V = b$
| $\langle \langle X, C \rangle, E \rangle$
| $erreur_e$

$S = \epsilon$
| $V\ S$
| $Remp\ S$
| $throw_e\ S$

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

$C = \epsilon$
| $b\ C$
| $X\ C$
| $ap\ C$
| $prim_{o^n}\ C$
| $\langle X, C \rangle\ C$
| $bspawn\ C$
| $espawn\ C$
| $\langle s, C', C'' \rangle\ C$
| $\langle s, C' \rangle\ C$
| $emit_s\ C$
| $throw_e\ C$
| $\langle e, \langle C', \langle X, C'' \rangle \rangle \rangle\ C$

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

$TL = \langle W, ST \rangle$ avec
- $W = \{D, ...\}$
- $ST = \{..., \langle s, D \rangle, ...\}$

$SI = \{s, ...\}$

$H = \epsilon$
| $\langle e, \langle S, E, C, D, TL, SI, H \rangle \rangle$

Les nouvelles règles sont les suivantes :

Partie de base de la machine SECD

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

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

Substitution : On a une variable, on prend la substitution dans l'environnement lié à la variable via la fonction E et on la met dans la pile.

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

$\langle b_1\ ...\ b_n\ S, E, prim_{o^n}\ C, D, TL, SI, H \rangle \longmapsto_{secdv3} \langle V\ S, E, C, D, TL, SI, H \rangle$
où $V = \delta(o^n, b_1, ...b_n)$

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

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

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

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

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

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

Partie pour les erreurs

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

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

Traiter erreur via gestionnaire d'erreur : On a plus rien, cependant il y a une erreur levée dans la pile du coup on regarde si le gestionnaire d'erreur gère celle-ci ; c'est la cas du coup prend la sauvegarde.

$\langle throw_e\ S, E, C, D, TL, SI, \langle e, \langle S', E', \langle X, C'' \rangle\ C', D', TL', SI', H \rangle \rangle \rangle$
$\longmapsto_{secdv3} \langle \epsilon, E'[X \leftarrow erreur_e], C'', \langle S', E', C', D' \rangle, TL', SI', H \rangle$

Traitement erreur récursif : On a plus rien, cependant on a une erreur levée dans la pile du coup on regarde si le gestionnaire d'erreur gère celle-ci ; ce n'est pas le cas, du coup on regarde pour le gestionnaire sauvegardé.

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

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

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

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

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

Partie pour la concurrence

Création thread : On crée un nouveau thread.

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

Initialisation signal : On initialise le signal.

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

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

$\langle S, E, \langle s, C', C'' \rangle\ C, D, TL, SI, H \rangle \longmapsto_{secdv3} \langle S, E, C'\ C, D, TL, SI, H \rangle$
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 émis et il y a un thread dans la file d'attente donc on mets le thread courant dans la liste des threads bloqués et on prend le thread en tête de la file.

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

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

$\langle S, E, \langle s, C', C''\rangle\ C, D, \langle \emptyset, ST\rangle, SI, H\rangle \longmapsto_{secdv3} \langle \epsilon, \emptyset, \epsilon, \emptyset, \langle \emptyset, ST\ \langle s, \langle S, E, \langle s, C', C''\rangle\ C, D\rangle\rangle\rangle, SI, H\rangle$
si $s \notin SI$ et $s \in E$

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

$\langle S, E, emit_s\ C, D, \langle W, ST\rangle, SI, H\rangle \longmapsto_{secdv3} \langle S, E, C, D, \langle W', ST'\rangle, SI, H\rangle$
avec $W' = W \cup$ tous les éléments de ST qui attendent l'émission de s et
avec $ST' = ST \setminus$ tous 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 le thread courant par le thread en tête de la file d'attente.

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

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.

$\langle S, E, \epsilon, \emptyset, \langle \emptyset, ST\rangle, SI, H\rangle \longmapsto_{secdv3} \langle S, E, \epsilon, \emptyset, \langle W, \emptyset\rangle, \emptyset, H\rangle$
avec $W =$ tous 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 *Remp*.

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

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

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

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

$\longrightarrow$ Soit on a une **constante** telle que $\langle \emptyset, \emptyset, [M]_{secdv3}, \emptyset, \langle \emptyset, \emptyset\rangle, \emptyset, \emptyset\rangle \twoheadrightarrow_{secdv3} \langle b, E, \epsilon, \emptyset, \langle \emptyset, \emptyset\rangle, SI, H\rangle$;

$\longrightarrow$ Soit on a une **fonction** telle que $\langle \epsilon, \emptyset, [M]_{secdv3}, \emptyset, \langle \emptyset, \emptyset\rangle, \emptyset, \emptyset\rangle \twoheadrightarrow_{secdv3} \langle\langle\langle X, C\rangle, E'\rangle, E, \epsilon, \emptyset, \langle \emptyset, \emptyset\rangle, SI, H\rangle$;

$\longrightarrow$ Soit on a un **remplacement** telle que $\langle \epsilon, \emptyset, [M]_{secdv3}, \emptyset, \langle \emptyset, \emptyset\rangle, \emptyset, \emptyset\rangle \twoheadrightarrow_{secdv3} \langle Remp, E, \epsilon, \emptyset, \langle \emptyset, \emptyset\rangle, SI, H\rangle$;

$\longrightarrow$ Sinon on a une **erreur** telle que $\langle \epsilon, \emptyset, [M]_{secdv3}, \emptyset, \langle \emptyset, \emptyset\rangle, \emptyset, \emptyset\rangle \twoheadrightarrow_{secdv3} \langle throw_e, E, \epsilon, \emptyset, \langle \emptyset, \emptyset\rangle, SI, H\rangle$.

4ème version des règles de la machine SECD Concurrente

Cette version ajoute la propagation des erreurs ainsi que la gestion des listes de valeurs partagées. Cette version se rapproche beaucoup de la dernière version de la partie 2. De base la gestion des erreurs étaient aussi présente dans la dernière version. Cependant la forme du gestionnaire d'erreur n'arrivant pas à être choisie, il a été décidés d'enlever la gestion des erreurs.

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 i donnés, renvoie la liste l' avec v ajoutée à la liste des valeurs du signal s pour le thread i.

Exemple 63 $\rho(\{..., \langle s, \{..., \langle id, valeur\rangle, ...\}, emit\rangle, ...\}, v, s, id) = \{..., \langle s, \{..., \langle id, valeur\ v\rangle, ...\}, emit\rangle, ...\}$

$\gamma(l, s, i, i')$ = la fonction qui ,pour une liste de valeurs partagées l classés par signal s et par thread i, un signal s, l'identifiant du thread courant i' et l'identifiant du thread i auquel 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.

Exemple 64 *Trois cas sont possibles :*

1. *On prend pourla première fois :*
   $\gamma(\{..., \langle s, \{..., \langle id, \{\langle b, \emptyset\rangle, \langle n, \{...\}\rangle, ...\}, \{...\}\rangle, ...\}\rangle, ...\}, s, id, id') =$
   $\langle b, \{..., \langle s, \{..., \langle id, \{\langle b, \emptyset\rangle, \langle n, \{..., id'\}\rangle, ...\}, \{...\}\rangle, ...\}\rangle, ...\}\rangle$
2. *On a déjà pris :*
   $\gamma(\{..., \langle s, \{..., \langle id, \{..., \langle b, \{..., id', ...\}\rangle, \langle n, \{...\}\rangle, ...\}, \{...\}\rangle, ...\}\rangle, ...\}, s, id, id') =$
   $\langle b, \{..., \langle s, \{..., \langle id, \{..., \langle b, \{...\}\rangle, \langle n, \{..., id'\}\rangle, ...\}, \{...\}\rangle, ...\}\rangle, ...\}\rangle$
3. *On prend le dernier :*
   $\gamma(\{..., \langle s, \{..., \langle id, \{..., \langle b, \{..., id', ...\}\rangle\}, \{...\}\rangle, ...\}\rangle, ...\}, s, id, id') =$
   $\langle b, \{..., \langle s, \{..., \langle id, \{..., \langle b, \{...\}\rangle\}, \{..., id'\}\rangle, ...\}\rangle, ...\}\rangle$
4. *On a déjà tout pris :*
   $\gamma(\{..., \langle s, \{..., \langle id, valeurs, \{..., id', ...\}\rangle, ...\}\rangle, ...\}, s, id, id') = throw\ erreur_e$

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

Exemple 65 $\iota(\{...\}, s) = \{..., \langle s, \{\}, false\rangle\}$

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

Exemple 66 $\beta(\{..., \langle s, \{...\}, vraie\rangle, ...\}, s) = vraie$ *ou* $\beta(\{..., \langle s, \{...\}, faux\rangle, ...\}, s) = faux$

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

Exemple 67 $\varepsilon(\{..., \langle s, \{...\}, faux\rangle, ...\}, s) = \{..., \langle s, \{...\}, vraie\rangle, ...\}$

$\alpha(\langle CS, SSI\rangle)$ = la fonction qui, pour la liste des signaux courants CS et la liste des signaux partagées SSI données, renvoie la liste des signaux courants vidée de ses listes de valeurs et avec le booléen représentant l'émission *emit* mis à nul. La liste des signaux partagées est remplacée par les listes de valeurs de la liste des signaux courants qui sont émis.

Exemple 68 $\alpha(\langle CS, SSI\rangle) = SSI$ *vidé et*
$\forall\ x \in CS$ *telle que* $\langle s, \{..., \langle id, \{..., b, ...\}\rangle, ...\}, true\rangle$, *on ajoute* x *dans* SSI
$\forall\ x \in CS$ *telle 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)
| $s\ C$ (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)
| $put\ C$ (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)

$SI = \langle CS, SSI\rangle$

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

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

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

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.

$SSI = \{..., \langle s, \{..., \langle id, \{..., \langle b, \{..., id', ...\}\rangle, ...\}, \{..., id'', ...\}\rangle, ...\}\rangle, ...\}$ (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

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

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 variable, on prend la substitution dans l'environnement lié à la variable via la fonction E et on la met 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.

$\langle I, b_n, ..., b_1\ S, E, prim_{o^n}\ C, D, TL, SI, H, IP\rangle \longrightarrow_{secdv4} \langle I, V\ S, E, C, D, TL, SI, H, IP\rangle$
avec $\delta(o^n\ b_1 ... b_n) = V$

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

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

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

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

Récupération de sauvegarde : On a rien mais le dépôt comporte une sauvegarde donc on prend 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 met le résultat de la fonction δ dans la pile.

$\langle I, throw\ erreur_e\ S, E, prim_{o^{1_e}}\ C, D, TL, SI, H, IP\rangle \longrightarrow_{secdv4} \langle I, V\ S, E, C, D, TL, SI, H, IP\rangle$
avec $\delta(o^{1_e}\ erreur_e) = V$

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 $M =$ un élément de $C \backslash prim_{o^{1_e}}$

Traiter erreur via gestionnaire d'erreur : On a plus rien mais on a une erreur levée 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

$\langle I, throw\ erreur_e\ S, E, \epsilon, D, TL, SI, \emptyset, IP\rangle \longmapsto_{secdv4} \langle I, throw\ erreur_e, E, \epsilon, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, IP\rangle$

Traitement erreur récursif : On a plus rien mais on a une erreur levée 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 teste avec la chaîne de contrôle du try et on sauvegarde catch dans le gestionnaire d'erreur.

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

Partie pour la concurrence

Création thread : On crée un nouveau thread.

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

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

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

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

$\langle I, s\ b\ S, E, put\ C, D, TL, \langle CS, SSI\rangle, H, IP\rangle \longrightarrow_{secdv4} \langle I, S, E, C, D, TL, \langle CS', SSI\rangle, H, IP\rangle$
avec $CS' = \rho(CS, b, s, I)$ et s initialisé

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

$\langle I, s\ b\ \langle\langle X, C'\rangle, E'\rangle\ S, E, get\ C, D, TL, \langle CS, SSI\rangle, H, IP\rangle$
$\longrightarrow_{secdv4} \langle I, \emptyset, E'[X \leftarrow V], C', \langle S, E, C, D\rangle, TL, \langle CS, SSI'\rangle, H, IP\rangle$
avec $\gamma(SSI, s, I, b) = \langle V, SSI'\rangle$ si il reste une valeur à prendre et s un signal partagé

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

$\langle I, s\ b\ \langle\langle X, C'\rangle, E'\rangle\ S, E, get\ C, D, TL, \langle CS, SSI\rangle, H, IP\rangle$
$\longrightarrow_{secdv4} \langle I, throw\ erreur_e\ S, E, C, D, TL, \langle CS, SSI'\rangle, H, IP\rangle$
avec $\gamma(SSI, s, I, b) = throw\ erreur_e$ si il reste aucune valeur à prendre et s un signal partagé

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

$\langle I, s\ S, E, init\ C, D, TL, \langle CS, SSI\rangle, H, IP\rangle \longrightarrow_{secdv4} \langle I, S, E, C, D, TL, \langle CS', SSI\rangle, H, IP\rangle$
avec $\iota(CS, s) = CS'$

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

$\langle I, s\ S, E, \langle C', C''\rangle\ C, D, TL, \langle CS, SSI\rangle, H, IP\rangle \longrightarrow_{secdv4} \langle I, S, E, C'\ C, D, TL, \langle CS, SSI\rangle, H, IP\rangle$
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 émis et il y a un thread dans la file d'attente donc on met ce thread dans la liste de threads bloqués et on prend le thread en tête de la file.

$\langle I, s\ S, E, \langle C', C''\rangle\ C, D, \langle\langle I', S', E', C''', D'\rangle W, ST\rangle, \langle CS, SSI\rangle, H, IP\rangle$
$\longrightarrow_{secdv4} \langle I', S', E', C''', D', \langle W, ST\langle s, \langle I, s\ S, E, \langle C', C''\rangle\ C, D\rangle\rangle\rangle, \langle CS, SSI\rangle, H, IP\rangle$
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 émis donc on met ce thread dans la liste de threads bloqués.

$\langle I, s\ S, E, \langle C', C''\rangle\ C, D, \langle\emptyset, ST\rangle, \langle CS, SSI\rangle, H, IP\rangle$
$\longrightarrow_{secdv4} \langle IP, \emptyset, \emptyset, \epsilon, \emptyset, \langle W, ST\langle s, \langle I, s\ S, E, \langle C', C''\rangle\ C, D\rangle\rangle\rangle, \langle CS, SSI\rangle, H, IP + 1\rangle$
avec $\beta(CS, s) = faux$

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

$\langle I, s\ S, E, emit\ C, D, TL, \langle CS, SSI\rangle, H, IP\rangle \longrightarrow_{secdv4} \langle I, Unit\ S, E, C, D, TL', \langle CS', SSI\rangle, H, IP\rangle$
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 traiter et on a aucune sauvegarde, du coup on change le thread courant par le thread en tête de la file d'attente.

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

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.

$\langle I, V\ S, E, \epsilon, \emptyset, \langle\emptyset, ST\rangle, SI, H, IP\rangle \longrightarrow_{secdv4} \langle I, V\ S, E, \epsilon, \emptyset, \langle W, \emptyset\rangle, SI', H, IP\rangle$
avec $W = ST$ avec tous ses 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 à rien faire.

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

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

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

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

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

- Soit on a une **fonction** telle que $\langle 0, \emptyset, \emptyset, [M]_{secdv4}, \emptyset, \langle\emptyset, \emptyset\rangle, \langle\emptyset, \emptyset\rangle, \emptyset, \emptyset, 1\rangle$
$\twoheadrightarrow_{secdv4} \langle I, \langle\langle X, C\rangle, E'\rangle\ S, E, \epsilon, \emptyset, \langle\emptyset, \emptyset\rangle, SI, H, IP\rangle$;

- Soit on a **rien** telle que $\langle 0, \emptyset, \emptyset, \emptyset, [M]_{secdv4}, \langle\emptyset, \emptyset\rangle, \langle\emptyset, \emptyset\rangle, \emptyset, \emptyset, 1\rangle$
$\twoheadrightarrow_{secdv4} \langle I, \epsilon, E, \epsilon, \emptyset, \langle\emptyset, \emptyset\rangle, SI, H, IP\rangle$;

- Sinon on a une **erreur** telle que $\langle 0, \emptyset, \emptyset, [M]_{secdv4}, \emptyset, \langle\emptyset, \emptyset\rangle, \langle\emptyset, \emptyset\rangle, \emptyset, \emptyset, 1\rangle$
$\twoheadrightarrow_{secdv4} \langle I, throw_e\ S, E, \epsilon, \emptyset, \langle\emptyset, \emptyset\rangle, SI, H, IP\rangle$;

1ère version des règles de la machine TTSIH

Cette version de la machine TTSI montre comment on peut ajouter les erreurs et les gérer. Ici les erreurs sont vu comme une entitée à part entière. Ces erreurs peuvent être levées ou non grâce à une commande *throw*. Les erreurs ne se propage pas et on va vérifier directement si la machine contient un gestionnaire d'erreur. Le problème que l'on peut soulevé sur cette version c'est le gestionnaire d'erreur H qui contient bien trop d'information, il en a été décidé au début pour éviter les effets de bords via les commandes inérants à la concurrence.

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

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

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

$b, s, n =$ une constante ou un identifiant de signal (un entier)

$V = b$
| e
| $\langle\langle X, C'\rangle E\rangle$

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

$S = \emptyset$
| $V\ S$
| $throw\ S$

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

$C = \epsilon$
| $b\ C$ (une constante ou un signal)
| $X\ C$ (une variable)
| $e\ C$ (une erreur)
| $\langle X, C'\rangle\ C$ (une abstraction)
| $ap\ C$ (une application)
| $prim_{o^n}\ C$ (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)
| $get\ C$ (prend une valeurs dans un signal)
| $throw\ C$ (lève une erreur)
| $catch\ C$ (créer un gestionnaire d'erreur)

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

$H = \emptyset$
| $\langle T, TL, SI, IP, H\rangle$

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

- **un identifiant de signal :** s
- **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 :** $\forall\ ssi \in SSI\ :\ ssi = \langle I, \langle CI, IL\rangle\rangle$ avec
  - **une liste d'identifiant de threads telle que :** $\forall\ il \in IL\ :\ il = I$
  - **une liste de constante avec itérateur telle que :** $\forall\ ci \in CI\ :\ ci = \langle b, IL\rangle$

$IP =$ 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'incrémente pour en créer un nouveau et retourne l'identifiant du signal créé avec la liste mise à jour.

Exemple 69

Si on initialise pour la première fois alors $\iota(\{\}) = (0, \{\langle 0, \langle false, \{\}, \{\}, \{\}\rangle\rangle\})$
sinon $\iota(\{..., \langle s, data\rangle\}) = (s+1, \{..., \langle s, data\rangle, \langle s+1, \langle false, \{\}, \{\}, \{\}\rangle\rangle\})$ *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 70 $SI(s) = \langle emit, CS, SSI, ST\rangle$

- $\tau(SI)$ 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 71 $\tau(SI) = \forall\ si \in SI$:

- $\langle true, CS, SSI, \{\}\rangle \rightarrow \langle false, \{\}, CS', \{\}\rangle$ *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.

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

Exemple 72 *Trois cas sont possibles :*

1. *Première fois que l'on prend :* $\langle id', \langle\{\langle b, \emptyset\rangle, \langle n, IL\rangle, ...\}, IL\rangle\rangle \rightarrow \langle id', \langle\{\langle b, \emptyset\rangle, \langle n, IL\ id\rangle, ...\}, IL\rangle\rangle$ *et on retourne b*
2. *On a déjà pris :* $\langle id', \langle\{..., \langle b, \{..., id, ...\}\rangle, \langle n, IL\rangle, ...\}, IL\rangle\rangle \rightarrow \langle id', \langle\{..., \langle b, \{...\}\rangle, \langle n, IL\ id\rangle, ...\}, IL\rangle\rangle$ *et on retourne b*
3. *On prend le dernier :* $\langle id', \langle\{..., \langle b, \{..., id, ...\}\rangle\}, IL\rangle\rangle \rightarrow \langle id', \langle\{..., \langle b, \{...\}\rangle\}, IL\ id\rangle\rangle$ *et on retourne b*

- $SI[(s, i) \leftarrow b]$ 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 $emit$.

Exemple 73 *Pour* $SI(s) = \langle emit, CS, SSI, ST\rangle$ *on change SI telle que* $SI(s) = \langle true, CS, SSI', ST\rangle$
avec $SSI' = \gamma(id, i, SSI)$ *avec id l'identifiant du thread courant.*

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

Exemple 74 $SSI(i) = \langle CI, IL\rangle$

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

Si la règle utilisée est ni **Lever une erreur et la gérer** ni **Lever une erreur et arrêter la machine** ni **Créer un gestionnaire d'erreur** :

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

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

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

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

$\langle S, E, n\ C, D\rangle \longrightarrow_{TTSIH} \langle n\ S, E, C, D\rangle$ avec $n =$ une constante b ou un identifiant de signal s

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

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

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

$\langle b_n, ..., b_1\ S, E, prim_{o^n}\ C, D\rangle \longrightarrow_{TTSIH} \langle V\ S, E, C, D\rangle$ avec $\delta(o^n\ b_1 ... b_n) = V$

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

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

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

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

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

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

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

Création thread : On crée un nouveau thread.

$\langle\langle I, \langle\langle X, C'\rangle, E\rangle\ S, E, spawn\ C, D, H\rangle, TL, SI, IP\rangle \longrightarrow_{TTSIH} \langle\langle I, IP\ S, E, C, D, H\rangle, TL\ \langle IP, \epsilon, E, C', \emptyset, \emptyset\rangle, SI, IP+1\rangle$

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

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

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

$\langle\langle I, s\ b\ n\ \langle\langle X, C'\rangle, E'\rangle\ S, E, get\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSIH} \langle\langle I, \emptyset, E'[X \leftarrow V], C', \langle S, E, C, D\rangle\rangle, TL, SI\rangle$
si pour $SI(s) = \langle emit, CS, SSI\rangle$ et $SSI(b) = \langle CI, IL\rangle$ on a $I \notin IL$ alors $\gamma(I, b, SSI(b)) = V$ sinon $n = V$

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

$\langle\langle I, S, E, init\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSIH} \langle\langle I, s\ S, E, C, D\rangle, TL, SI'\rangle$ 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 premier choix.

$\langle\langle I, \langle\langle X', C''\rangle, E\rangle\ \langle\langle X, C'\rangle, E\rangle\ s\ S, E, present\ C, D\rangle, TL, SI\rangle \longrightarrow_{TTSIH} \langle\langle I, S, E, C'\ C, D\rangle, TL, SI\rangle$
avec $SI(s) = \langle vraie, CS, SSI, TL\rangle$

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.

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

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.

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

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

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

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

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

Partie pour la gestion des erreurs : Cette partie ajoute la gestion des erreurs dans notre machine.

Erreur : On a une erreur, on la déplace dans la pile

$\langle S, E, e\ C, D\rangle \longrightarrow_{TTSIH} \langle e\ S, E, C, D\rangle$

Lever une erreur et la gérer : On a la commande throw, on regarde si on peut gérer l'erreur. On le peut donc on prend la sauvegarde dans le gestionnaire d'erreur

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

Lever une erreur et arrêter la machine : On a la commande throw, on regarde si on peut gérer l'erreur. On ne le peut pas donc on arrête la machine

$\langle\langle I, e\ S, E, throw\ C, D, \emptyset\rangle, TL, SI, IP\rangle \longrightarrow_{TTSIH} \langle\langle I, e, E, throw, \emptyset, \emptyset\rangle, \emptyset, \emptyset, IP\rangle$

Créer un gestionnaire d'erreur : On a la commande catch,

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

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 ne rien faire.

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

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

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

la machine TTSIH peut s'arrêter dans 5 états différents :

- Soit on a une **constante** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSIH}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSIH} \langle\langle I, b\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a une **fonction** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSIH}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSIH} \langle\langle I, \langle\langle X, C\rangle, E\rangle\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a **rien** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSIH}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSIH} \langle\langle I, \epsilon, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a **une erreur traitée** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSIH}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSIH} \langle\langle I, e, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;

- Soit on a **une erreur non traitée** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSIH}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSIH} \langle\langle I, e, E, throw, \emptyset\rangle, \emptyset, SI, IP\rangle$
avec $\forall\ si \in SI\ :\ si = \langle s, \langle emit, \emptyset\rangle\rangle$;