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 abtraites : leur fonctionnement, 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 abstraite existante. J'y explique les contraintes créées par l'ajout de la concurrence dans une machine abstraite. Pour chaque commande ajoutée, une explication détaillée des changements apportés dans la machine est écrite. Deux sémantiques de la machine abstraite y sont décrites.

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 améliorer, 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.

Éléments base de c light

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 axé 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 exposé 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, le typage 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 ? La curryfication est la transformation d'une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction sur le reste des arguments.

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

  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 termesdu 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 1 *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 2 *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 lié 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 la cas ou on utilise l'η-réduction.

La réduction générale 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 occurences 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 3 *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 g4, 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" part 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 4 *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 5 *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 6 *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 7 *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 qui existent.*

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

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

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 aucune 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 rédution.*

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

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

Le problème de la boucle infinie de réduction est possible si on applique une "mauvaise" réduction.

Exemple 9 *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 10 *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

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

Version personnelle

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

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 propriétée du diamant est la suivante :

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 intéresse :

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

   Exemple 11 *Pour une fonction* fst *tels que pour deux arguments* x *et* y *on retourne le 1er. Si on a :* $fst(3+4, 5/5)$ *on va évalué* $3+4$. *Ce qui donne* $fst(7, 5/5)$ *et on retourne 7 on n'évalue pas* $5/5$.

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

   Exemple 12 *Pour une fonction* fst *tels que pour deux arguments* x *et* y *on retourne le 1er. Si on a :* $fst(3+4, 5/5)$ *on va évalué* $3+4$. *Ce qui donne* $fst(7, 5/5)$, *on évalue* $5/5$. *On arrive à* $fst(7, 1)$ *et on retourne 7.*

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

L'appel par valeur est plus facile à mettre en oeuvre dans un langage. En effet, on a la garantit que l'élément que l'on va appliquer est réduit au maximum et que plus de 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'éxécution ce qui est dérangeant mais que l'on va préférer à un recalcul de chaque élément.

Exemple 13 *Si on reprend l'expression* $(\lambda x.(x\ x))\ \underline{((\lambda y.y)\ (\lambda z.z))}$. *Un appel par nom va faire la chose suivante :*

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

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 :*

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

On va voir ISWIM qui est un langage qui utilise la stratégie de l'appel par valeur.

2.2.3 ISWIM

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

ISWIM a une grammaire étendue de la grammaire 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 tels que **n**-réduction $= \beta_v \cup \delta$

Exemple 14 *Voici un exemple de* ***n****-réduction. On va prendre la* λ*-expression suivante :* $(\lambda fx.(f \ x) \ \lambda y.(+ \ y \ y) \ \ulcorner 1 \urcorner)$.

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

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 :

$\longrightarrow$ on a une **constante** telle que $\langle M, []\rangle \twoheadrightarrow_{cc} \langle b, []\rangle$;

$\longrightarrow$ on a une **fonction** telle que $\langle M, []\rangle \twoheadrightarrow_{cc} \langle \lambda X.N, []\rangle$;

$\longrightarrow$ on a un **état inconnu** soit une **erreur**.

Exemple 15 *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**).

Par exemple dans la machine CC, on peut réunir :

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

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

$(8)_{cc}\ \langle V, E[(o^n\ V_1...V_i\ []\ N\ ...)]\rangle \longmapsto_{cc} \langle (o^n\ V_1...V_i\ V\ N\ ...), E\rangle$
$(7)_{cc}\ \langle (o^n\ b_1...b_n), E\rangle \longmapsto_{cc} \langle V, E\rangle$ avec $V = \delta(o^n, b_1...b_n)$
$\rightarrow$ sont combinées dans la règle $(5)_{scc}\ \langle b, E[(o^n, b_1, ...b_i, [])]\rangle \longmapsto_{scc} \langle V, E\rangle$ *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 toute les informations que l'on peut extraire d'un état de la machine pour rendre le fonctionnement plus rapide.

Les règles qui définissent la machine SCC sont les suivantes :

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

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

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

Exemple 16 *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 transition 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 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**.

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 ont 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 élément 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 :

$\longrightarrow$ on a une **constante b** telle que $\langle M, mt\rangle \twoheadrightarrow_{ck} \langle b, mt\rangle$;

$\longrightarrow$ on a une **fonction** telle que $\langle M, mt\rangle \twoheadrightarrow_{ck} \langle \lambda X.N, mt\rangle$;

$\longrightarrow$ on a un **état inconnu** soit une **erreur**.

Exemple 17 *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 l'Annexe 6.1.1.*

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 est appliquée immédiatement. Cela coûte cher surtout quand l'expression devient grande. De plus, si notre substitution n'est pas une variable elle est traitée avant d'être appliquée.

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

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

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

$\mid \langle opd, \langle v, ..., v, o^n\rangle, \langle c, ...\rangle, \overline{\kappa}\rangle$

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 :

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

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

$\longrightarrow$ on a un **état inconnu** soit une **erreur**.

Exemple 18 *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 quant on évalue une application ou un argument indépendamment de la complexité de celui-ci.

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

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

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

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

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

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 :

$\longrightarrow$ on a une **constante b** telle que $\langle \epsilon, \emptyset, [M]_{secd}, \epsilon\rangle \twoheadrightarrow_{secd} \langle b, \widehat{\varepsilon}, \epsilon, \epsilon\rangle$;

$\longrightarrow$ on a une **fonction** telle que $\langle \epsilon, \emptyset, [M]_{secd}, \epsilon\rangle \twoheadrightarrow_{secd} \langle\langle\langle X, \widehat{C}\rangle, \widehat{\varepsilon'}\rangle, \widehat{\varepsilon}, \epsilon, \epsilon\rangle$;

$\longrightarrow$ on a un **état inconnu** soit une **erreur**.

Exemple 19 *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 langage, 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** (présentée dans [5] datant de 1995) ou encore **Icobj** (présentée dans [6] présentée en 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 émit 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 émit 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/. Ces recherches se focalisent sur la programmation réactive, tous les langages précédemments cités dans l'historique viennent de ces 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 ReactiveML est sont 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, appelée tick, que l'on peut voir comme une succession de tick qui permet au programme 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 20 *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 nous faut une machine qui fait un appel par valeur, c'est-à-dire que les paramètres seront évalués avant la fonction. La machine SECD remplit ce critère donc c'est celle que nous utiliserons comme base.

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

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

Un autre point délicat est la fin d'un instant logique. 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 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 machine SECD qui se parlent donc il faut créer une structure plus large qui peut les stocker. On part cependant d'une seule chaîne de contrôle donc il faut pouvoir créer ces processus. Ils sont ce que l'ont va appeler des **threads**.

3.1.2 Description formelle du langage

Un Thread

Qu'est-ce qu'un 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.

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

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

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

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

Un Signal

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

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

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

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

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

Le Spawn

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

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

L'initialisation d'un signal

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

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

   Exemple 21 *Si on prend* $\langle s, C' \rangle$ *la version convertie de Signal s in t on a :*
   $\langle S, E, \langle s, C' \rangle C, D, SI \rangle \longrightarrow \langle \emptyset, \emptyset\ [init \leftarrow s], C', \langle S, E, C, D \rangle, SI \rangle$

   Le problème de cette version est qu'il provoque un stockage double des signaux dans la machine à cause des commandes à venir. En effet, on va garder l'initialisation dans l'environnement et l'émission dans l'élément SI. Ce n'est pas optimal.
   (b) On utilise la sauvegarde mais différemment. En effet on va faire une sauvegarde comme pour la première version cependant on ne met pas l'initialisation dans l'environnement mais directement là où l'on voudra stocker plus tard via un booléen. On va créer une variante de la sauvegarde de la machine SECD, dans le dépôt, de la forme $\langle s, \langle S, E, C, D \rangle \rangle$. Ce qui va nous permettre de savoir quand sortir de l'expression pour laquelle on avait initialisé le signal. La structure SI ici prend une autre forme, on a $SI = \{\langle s, init, emit \rangle\}$ telle qu'un signal s, un booléen représentant l'initialisation *init* et un booléen représentant l'émission du signal *emit*.

   Exemple 22 *Si on prend* $\langle s, C' \rangle$ *la version convertie de Signal s in t on a :*
   $\langle S, E, \langle s, C' \rangle C, D, SI \rangle \longrightarrow \langle \emptyset, \emptyset, C', \langle s, \langle S, E, C, D \rangle \rangle, SI\ \langle s, true, false \rangle \rangle$

   Cette version est assez efficace mais reste lourde dans la machine au niveau du stockage et la façon dont cette commande est convertie crée le même problème que pour la première version du *Spawn* : on n'est pas censé avoir de structure à traiter dans la machine.
2. *Init s* : l'identifiant du signal est donné et va être initialisé pour tous les éléments de la machine. Cette version va drastiquement simplifier le fonctionnement de la machine car on n'a plus besoin de le limiter l'action de l'initialisation à une partie de la chaîne de contrôle ni à un thread en particulier. Simplement, on met le signal dans le stockage des signaux. Ce fait sera nécessaire et suffisant pour savoir qu'il est initialisé. La structure de SI va donc être de la forme $\langle s, emit \rangle$ avec un signal s et un booléen représentant l'émission du signal *emit*.

   Exemple 23 *Si on prend s Init la version convertie de Signal s in t on a :*
   $\langle s\ S, E, Init\ C, D, SI \rangle \longrightarrow \langle S, E, C, D, SI\ \langle s, false \rangle \rangle$
3. *Init* : l'identifiant du signal sera retourné par la machine, l'utilisateur ne lui donne plus l'identifiant. On créera l'identifiant en incrémentant le dernier identifiant de signal de un. Le reste est identique au fonctionnement du *Init s*. Cette possibilité aura la même structure pour SI que la précédente à l'exception que l'on va ajouter un ordre car il faudra créer des identifiants donc on va toujours mettre en fin de la liste afin de pouvoir créer facilement un nouvel identifiant.

   Exemple 24 *Si on prend s Init la version convertie de Signal s in t on a :*
   $\langle s\ S, E, Init\ C, D, SI \rangle \longrightarrow \langle S, E, C, D, SI\ \langle s, false \rangle \rangle$

La présence d'un signal

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

C'est ici que la notion de temps logique va nous être utile. Via l'initialisation d'un signal on sait que l'on a un booléen pour savoir si un signal est émis donc le cas émis est simple mais s'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 donc 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 redéfinit la fin d'un instant logique comme la fin ou le blocage de tous les threads de la machine.

Dans notre machine on va devoir différencier les threads en 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 un signal Cette commande est sûrement la plus simple à mettre en place car tout a été fait en amont via les trois commandes précédentes. La forme de la commande *emit* n'a pas beaucoup 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 W

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 :**

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

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

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

$C = \epsilon$
$\mid b\ C$ (une constante)
$\mid X\ C$ (une variable)
$\mid signal\ C$ (un signal)
$\mid \langle X, C'\rangle\ C$ (une abstraction)
$\mid ap\ C$ (une application)
$\mid prim_{o^n}\ C$ (un opérateur)
$\mid spawn\ C$ (créateur d'un nouveau thread)
$\mid present\ C$ (teste la présence d'un signal)
$\mid init\ C$ (initialise un signal)
$\mid emit\ C$ (émet un signal)

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

$SI =$ **une liste de signaux telle que :** $\forall\ si \in SI \mid si = \langle signal, \langle emit, ST\rangle\rangle$ avec :
- **un booléen représentant l'émission de ce signal :** $emit$
- **la liste des threads bloqués par ce signal :** ST avec $\forall st \in ST\ :\ st = T$

On va définir une règle pour simplifier les règles 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.

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

$SI(s)$ 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 : 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 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 prends le premer 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.

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

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.

$\langle\langle V\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI\rangle \longrightarrow_{TTS} \langle\langle V\ S, E, \epsilon, \emptyset\rangle, TL, SI'\rangle$
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 $\langle\langle\emptyset, \emptyset, [M]_{TTS}, \emptyset\rangle, \emptyset, \emptyset\rangle \twoheadrightarrow_{TTS} \langle\langle b\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI\rangle$;

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

- Soit on a **rien** telle que $\langle\langle\emptyset, \emptyset, [M]_{TTS}, \emptyset\rangle, \emptyset, \emptyset\rangle \twoheadrightarrow_{TTS} \langle\langle\epsilon, E, \epsilon, \emptyset\rangle, \emptyset, SI\rangle$;

- 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 autre problème se pose par rapport à cela : l'attribution d'un identifiant.

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

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

1. Une liste de valeurs courante : c'est là que l'on va insérer les valeurs, on ne peut pas prendre dans cette liste, on peut 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 et à la condition que le signal soit émis, on transfère ces valeurs.

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

Comment stocker ces valeurs ? La liste des signaux est pour l'instant de la forme $SI \mid \forall\ si \in SI\ :\ \langle s, \langle emit, ST\rangle\rangle$. On va se contenter d'expliquer pour cette forme car une autre forme 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 retournera le neutre.

On a donc une commande *get* de la forme s id n *get* avec un signal s, l'identifiant du thread i dont 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 redéfinir nos règles pour voir les changements provoqués.

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

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

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

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

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

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

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

$C = \epsilon$

$\mid\ b\ C$	(une constante)
$\mid\ X\ C$	(une variable)
$\mid\ signal\ C$	(un signal)
$\mid\ \langle X, C'\rangle\ C$	(une abstraction)
$\mid\ ap\ C$	(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)
$\mid\ get\ C$	(prends une valeurs dans un signal)

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

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

- **un booléen représentant l'émission du signal :** $emit$
- **un identifiant de thread :** I
- **une liste des signaux courants telle que :** $\forall\ cs \in CS\ :\ cs = \langle I, CL\rangle$ avec
  - **une liste de constantes telle que :** $\forall\ cl \in CL\ :\ cl = b$
- **la liste des signaux partagés telle que :** $\forall\ ssi \in SSI\ :\ ssi = \langle I, \langle CI, IL\rangle\rangle$ 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 : 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 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.

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

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 $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSI} \langle\langle I, b\ S, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$;

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

- Soit on a **rien** telle que $\langle\langle 0, \emptyset, \emptyset, [M]_{TTSI}, \emptyset\rangle, \emptyset, \emptyset, 1\rangle \twoheadrightarrow_{TTSI} \langle\langle I, \epsilon, E, \epsilon, \emptyset\rangle, \emptyset, SI, IP\rangle$;

- 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 La gestion des erreurs

Cette partie est facultative à votre lecture. Une version de la machine à été créée avec pour but de gérer les exceptions. Pour cela deux versions ont été créées avec et sans la propagation des erreurs. Je vais donc présenter brièvement le langage.

3.3.1 Description informelle du langage

Gérer des erreurs : Quand on parle d'erreur on a deux choses qui viennent en tête :

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

La présence d'une erreur est-elle toujours signe d'un problème ? Non, on peut créer une erreur pour pouvoir gérer un cas, par exemple en java on peut créer une erreur et l'afficher sans que le programme soit erroné en lui-même. Il faut donc faire une différence entre une erreur et une erreur levé. On va créer deux éléments en plus pour notre chaîne de contrôle :

- *throw* : indique qu'un erreur est levée
- e : une erreur

La structure de contrôle a toujours la même forme : la partie protégé et la partie de remplacement si une erreur est levée. On va devoir trouver une structure qui est adaptée pour la machine. On va utilisé une astuce déjà présent pour le *spawn*, c'est-à-dire utilisé une abstraction pour éviter la structure inappropriée. Il faut une commande pour savoir que l'on a une structure de gestion d'erreur. On arrive donc à la forme suivant $\langle X, C\rangle$ $\langle X, C\rangle$ *catch*.

On peut se demander ce qu'on doit sauvegarder avec la structure de gestion d'erreur. En effet, il faudrait pouvoir revenir au moment où le try...catch est traité. On crée H le gestionnaire d'erreur. On ne peut pas limiter le gestionnaire d'erreur H à un thread précis car les threads communiquent entre eux. Donc le gestionnaire va être un élément à part entière de la machine. On aura donc $\langle T, TL, SI, H, IP\rangle$.

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. On sait aussi intéressé à la programmation réactive. De là, nous sommes partie d'une machine existante pour créer notre machine fonctionnelle réactive.

- Dans un premier temps 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 une machine abstraite fonctionnelle réactive avec partage de valeurs. Les signaux ne sont plus seulement présent ou absent 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 mit dans la mémoire.
- Dans un troisième temps un petit mot a été glissé sur la gestion des erreurs.

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

Encore beaucoup de travail reste à faire. En effet, il manque une preuve du déterminisme de la machine qui a été effectuée en partie au brouillon. Une implantation avec un système de gestion d'erreur est une prochaine étape à franchir pour que notre machine abstraite fonctionnelle réactive soit entièrement opérationnelle.

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 strict ainsi que la preuve de déterminisme qui est en cours d'écriture, et 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 35 *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 36 *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 37 *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 38 *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*** $\langle\langle\langle X', C''\rangle, E\rangle\ \langle\langle X, C'\rangle, E\rangle\ s\ S, E, present\ C, D\rangle, \langle S', E', C''', D'\rangle\ TL, SI\rangle$
$\longrightarrow_{TTS} \langle\langle S', E', C''', D'\rangle, TL, SI'\rangle$
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 39 *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 $\langle\langle 1, 13, \emptyset, \epsilon, \emptyset\rangle, \emptyset, \{\langle 0, \langle false, \emptyset, \{\langle 0, \{\langle 10, \emptyset\rangle\}, \{1\}\rangle\}, \emptyset\rangle\rangle, \langle 1, \langle false, \emptyset, \emptyset, \emptyset\rangle\rangle\}, 3\rangle$

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

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
$\langle S, E, \epsilon, \emptyset, \emptyset, ST, SI\rangle \longmapsto_{secdv1-2} \langle S, E, \emptyset, \emptyset, W, \emptyset, \emptyset, \emptyset\rangle$
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 $\langle \epsilon, \emptyset, [M]_{secdv1-2}, \emptyset, \emptyset, \emptyset, \epsilon\rangle \twoheadrightarrow_{secdv1-2} \langle b, E, \epsilon, \emptyset, \emptyset, SI, \epsilon\rangle$;
$\longrightarrow$ Soit on a une **abstraction function** tels que $\langle \epsilon, \emptyset, [M]_{secdv1-2}, \emptyset, \emptyset, \emptyset, \epsilon\rangle \twoheadrightarrow_{secdv1-2} \langle\langle\langle X, C\rangle, E'\rangle, E, \epsilon, \emptyset, \emptyset, SI, \epsilon\rangle$;
$\longrightarrow$ Soit on a un **remplacement** tels que $\langle \epsilon, \emptyset, [M]_{secdv1-2}, \emptyset, \emptyset, \emptyset, \epsilon\rangle \twoheadrightarrow_{secdv1-2} \langle Remp, E, \epsilon, \emptyset, \emptyset, SI, \epsilon\rangle$;
$\longrightarrow$ 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 40 $\rho(\{..., \langle s, \{..., \langle id, valeur\rangle, ...\}, emit\rangle, ...\}, v, s, id) = \{..., \langle s, \{..., \langle id, valeur\ v\rangle, ...\}, emit\rangle, ...\}$

$\gamma(l, s, i, i')$ = 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 41 *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 42 $\iota(\{...\}, s) = \{..., \langle s, \{\}, false\rangle\}$

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

Exemple 43 $\beta(\{..., \langle s, \{...\}, vraie\rangle, ...\}, s) = vraie$ *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 44 $\varepsilon(\{..., \langle s, \{...\}, faux\rangle, ...\}, s) = \{..., \langle s, \{...\}, vraie\rangle, ...\}$

$\alpha(\langle CS, SSI\rangle)$ = la fonction qui ,pour la liste des signaux courant 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 45 $\alpha(\langle CS, SSI\rangle) = SSI$ *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.

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

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 $\langle 0, \emptyset, \emptyset, [M]_{secdv4}, \emptyset, \langle\emptyset, \emptyset\rangle, \langle\emptyset, \emptyset\rangle, \emptyset, \emptyset, 1\rangle$
$\twoheadrightarrow_{secdv4} \langle I, throw_e\ S, E, \epsilon, \emptyset, \langle\emptyset, \emptyset\rangle, SI, H, IP\rangle$;