Projet de recherche sur l'article **Handling algebraic effects** Résumé et mise en lien avec **erpl**

Jordan Ischard[1]

[1] *Université d'Orléans*

Sommaire

Introduction

Le pionnier dans la gestion des effets algébriques est **Eugenio Moggi**. Il proposa une représentation uniforme de calcul d'effets grâce aux **Monades** [1]. Cette représentation est utilisé dans le langage **Haskell**.

Définition 1. *Une **Monade** est une structure permettant de manipuler les langages fonctionnels purs avec des traits impératifs. On peut représenter une monade comme un triplet constitué de :*

1. *un **constructeur de type** $\mathcal{M}$ appelé type monadique, qui associe au type $\boldsymbol{t}$ le type $\mathcal{M}\boldsymbol{t}$.*
2. *une **fonction** `unit/return` qui construit à partir d'un élément de type sous-jacent $\boldsymbol{a}$ un autre élément de type monadique $\mathcal{M}\boldsymbol{a}$. Cette fonction a la signature suivante :*

   `unit/return` : $\boldsymbol{t} \to \mathcal{M}\boldsymbol{t}$.
3. *une **fonction** bind, représenté par l'opérateur infixe `>>=`, associant à une valeur de type monadique et une fonction d'association une autre valeur de type monadique. Cette opérateur à la signature suivante :*

   `>>=` : $\mathcal{M}\boldsymbol{t} \to (\boldsymbol{t} \to \mathcal{M}\boldsymbol{u}) \to \mathcal{M}\boldsymbol{u}$.

   L'idée derrière cette fonction est que l'on doit passé par un calcul pour accéder et traiter une valeur de type monadique afin de continuer le calcul.

Exemple 1. Voici un exemple de **Monade** trouvé sur le site d'Haskell france. On représente une variable optionnelle.

```
-- Ici le constructeur de type est Maybe
instance Monad Maybe where
        -- La fonction qui construit notre type monadique
        return x = Just x

        -- La fonction bind qui nous permet de travailler avec le type monadique
        Nothing >>= f = Nothing
        Just x >>= f  = f x

        -- Au cas où, une erreur se lève on dit qu'il n'y a rien
        fail _ = Nothing
```

Les effets et leurs gestions sont utiles pour représenter les exceptions, les états mémoires, le non-déterminisme, les I/O, etc. **Moggi** ne fut pas le seul à proposer une gestion des effets, en effet **Plotkin et Power** proposerons plus tard une représentation basée sur un ensemble d'opérations qui représente la sources des effets et une théorie d'équations, pour ces dites opérations, qui décrit leurs propriétés.

L'intuition derrière est que chaque calcul retourne soit une **valeur**, soit effectue une **opération** avec un retour déterminant la continuation. Les arguments de cette opération représente donc les potentiels continuations.

Exemple 2. On prend une opération binaire de choix **choose**. C'est une opération qui choisit de manière non déterministe un booléen à retourner :

choose(return true,return false)

On a deux possibilités : soit on continue avec le calcul donné par le première argument soit on continue avec le calcul donné par le second.

Les représentations respectives de **Moggi** et de **Plotkin et Power** permettent de gérer les effets et il est possible de passer d'une représentation à l'autre pour la plupart des effets. Typiquement les effets représentant les I/O et les états mémoires peuvent être exprimé dans les deux représentations. Les effets ayant cette caractéristique sont appelés **effets Algébriques**.

La vision algébrique des effets a permis de les combiner et de raisonner dessus. Cependant, le gestionnaire d'exception apporte un challenge.

Soit la monade $\_ + \mathbf{exc}$ pour l'ensemble des exceptions **exc**. Cette monade est représentée par une opération $\mathbf{raise}_e()$ pour chaque $e \in \mathbf{exc}$ et aucune équations. $\mathbf{raise}_e()$ ne prend par d'arguments et il n'y a pas de continuation directement après une levée d'exception. Maintenant la monade défini, comment créer le gestionnaire d'exception ? L'approche commune est la suivante:

$$\mathbf{handle}_e(M, N)$$

M s'effectue à moins qu'une exception e soit levée, dans ce cas on effectue N. Cette construction manque d'une certaine propriété caractérisant les opérations spécifiés dans les équations qui les décrivent.

Dans l'article, une explication algébrique des gestionnaires d'exceptions est donné. L'idée principale est que :

1. les gestionnaires correspondent à des modèles de théorie équationnelle;
2. la sémantique des gestionnaires est donné à l'aide d'homomorphismes uniques qui ciblent de tels modèles et est induit par la propriété universelle du modèle libre.

Il souhaite adopter une vision générale suggéré par Benton et Kennedy. Dans leur approche, la valeur de retour est passé à la continuation définit par l'utilisateur. Conceptuellement, les opérations algébriques et les gestionnaires d'effets sont duals, on a le *constructeur d'effets* qui crée l'effet et le *destructeur d'effets* qui produit un calcul en lien avec l'effet détruit.

Le projet recherche a pour but de choisir un article et de le résumé. Mon résumé restera fidèle à l'article en gardant la forme du papier tout en allégeant le fond. En effet, il m'est plus important de montrer le concept général via des exemples ainsi que les résultats importants sur la validité des gestionnaires que d'expliquer la logique mathématique interne.

De plus, le projet recherche étant lié à un autre module projet, je vais ajouter une plus-valu à mon résumé. Durant ma 1ère année de master, j'ai eu l'occasion d'implémenter une représentation des effets algébrique dans un noyau de langage fonctionnel créé lors de mon stage de fin de licence. Je vais donc profiter d'une section supplémentaire pour comparer les deux approches.

1 Gestionnaire des exceptions

La structure du gestionnaire est important. Le choix se pose sur les travaux de **Benton et Kennedy**. Le principe est exprimé à travers un exemple sur un effet simple : les exceptions.

On considère un ensemble fini d'exceptions **exc**, un second ensemble A et une monade d'exception TA telle que $TA \stackrel{\mathbf{def}}{=} A + \mathbf{exc}$ (avec $A + \mathbf{exc}$ l'union disjointe des deux ensembles).

Un calcul retournant une valeur $a \in A$ est modélisé par un élément $ta \in TA$. Cette monade a pour unité $\eta_A : A \to A + \mathbf{exc}$, et le calcul $return(V)$ est interprété par $\eta_A(V)$ tandis que $\mathbf{raise}_e()$ est interprété par $in_2(e)$ (une fonction injective de $A + \mathbf{exc}$ vers **exc**).

1.1 Construction des gestionnaires étendus

Benton et Kennedy ont généralisé la construction des gestionnaires, dans [2], avec la forme suivante :

$$M \textbf{ handled with}\{\mathbf{raise}_e() \mapsto N_e\}_{e \in \mathbf{exc}} \textbf{ to } x : A.N(x)$$

avec $\{...\}_{e \in \mathbf{exc}}$ représentant l'ensemble des calculs, un pour chaque $e \in \mathbf{exc}$. Dans cette construction, on effectue le calcul de $M \in A + \mathbf{exc}$. Si on lève une exception $e \in \mathbf{exc}$ alors on effectue un calcul N_e qui retourne un élément de B (où B peut différer de A). Sachant que N_e peut lui même lever une exception on en déduit que $N_e \in B + \mathbf{exc}$. Les valeurs retournées par le gestionnaire sont passées dans une continuation défini par l'utilisateur $N : A \to B + \mathbf{exc}$. La construction satisfait 2 équations :

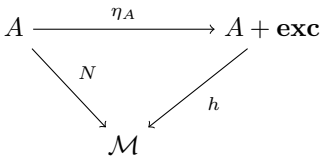
$$\textbf{return } V \textbf{ handled with}\{\mathbf{raise}_e() \mapsto N_e\}_{e \in \mathbf{exc}} \textbf{ to } x : A.N(x) = N(V)$$
$$\mathbf{raise}_{e'}() \textbf{ handled with}\{\mathbf{raise}_e() \mapsto N_e\}_{e \in \mathbf{exc}} \textbf{ to } x : A.N(x) = N_{e'}$$

Comme discuté dans , Cette construction permet d'écrire de façon simple un idiome de programmation qui était lourd à mettre en place.

Algébriquement le calcul N_e donne un nouveau modèle $\mathcal{M}$ pour la théorie des exceptions. Le porteur de ce modèle est $B + \mathbf{exc}$. Pour chaque $e \in \mathbf{exc}$, $\mathbf{raise}_e()$ est interprété par N_e. On peut voir d'après les deux équations ci-dessus que :

$$h(M) \stackrel{def}{=} M \textbf{ handled with}\{\mathbf{raise}_e() \mapsto N_e\}_{e \in \mathbf{exc}} \textbf{ to } x : A.N(x)$$

où $h : A + \mathbf{exc} \to \mathcal{M}$ est l'homomorphisme unique qui étend N, i.e, tel que l'on a le diagramme de commutation suivant :



La construction proposé par Benton et Kennedy est le plus général possible d'un point de vue algébrique.

1.2 Gestion des effets algébriques

La construction des gestionnaires étendus permet de traiter les effets algébrique. Cela équivaut à dire que les gestionnaires vu ci-dessus correspondent à des **modèles de théorie équationnelle**. La source des effets est l'ensemble des opérations comme dans la représentation de **Plotkin et Power**.

Toutes interprétations ne donnent pas forcément un modèle de la théorie équationnelle. Cela induit que certains gestionnaires sont incorrectes. Pour contourner le problème, il y a deux écoles.

1. l'utilisateur est contraint à utiliser une famille d'effet que le créateur du langage sait gérer;
2. l'utilisateur a une totale liberté et le créateur du langage autorise un gestionnaire à être incorrect.

Dans l'article la seconde option est choisie.

2 Syntaxe

2.1 Signatures

Les types de signature α, β sont définis par:

$$\alpha, \beta := \mathbf{b} \mid \mathbf{1} \mid \alpha \times \beta \mid \sum_{l \in L} \alpha_l$$

où $\mathbf{b}$ représente l'ensemble des types de base, 1 représente le type *unit* et L représente un sous-ensemble fini de label. Cependant, Un sous-ensemble de $\mathbf{b}$ est spécifié comme un **type de base d'arité** ce qui permet d'introduire la définition suivante :

Définition 2. *Un type de signature est un type de signature* ***d'arité*** *si et seulement si tous les types de base qu'il contient sont des types de base d'arité.*

Les ensembles finis peuvent être représentés par le type de signature $\sum_{l \in L} \alpha_l$ et les ensembles infinis sont représentés par le type de base correspondant.

Exemple 3. On peut représenter les booléens par $\sum_{bool \in \{true, false\}} \mathbf{1}$ et un sous-ensemble d'entier par $\sum_{n \in \{1,\dots,m\}} \mathbf{1}$.

Le symbole de fonction typé est défini par $\mathbf{f} : \alpha \to \beta$. Il représente une fonction prédéfinie.

Le symbole d'opération typé est défini par : $\mathbf{op} : \alpha \to \beta$ tel que **op** a une théorie d'effet τ. Il est évalué de la façon suivante : **op** accepte un paramètre α et après avoir exécuté l'effet approprié, son type de retour β détermine la continuation.

Exemple 4. On présente quelques exemples prit de [3].

> **Exceptions** : On prend l'unique symbole d'opération **raise** : **exc** $\to$ **0** paramétré sur l'ensemble **exc** afin de lever une exception. Le symbole de l'opération est nul (**0**) car il n'y a pas de continuation après une levée d'exception. Il faut donc gérer les exceptions.
>
> **State** : On prend le type de base d'arité **nat** (qui représente l'ensemble des naturelles), le type de base d'arité **loc** (un ensemble fini d'emplacement) et les symboles d'opérations **get** : **loc** $\to$ **nat** et **set** : **loc** $\times$ **nat** $\to$ **1**. On part du principe que l'on stocke des naturelles dans des emplacements mémoire, **get** permet de récupérer le naturelle pour un emplacement donné tandis que **set** remplace l'élément à l'emplacement donné en paramètre par le second élément donné en paramètre et ne retourne rien.

2.2 Types

Le langage que défini suit l'approche *call-by-push-value* de Levy [4], i.e, il y a une séparation stricte entre les valeurs et les calculs. Ces types sont donnés par :

$$A, B := \mathbf{b} \mid \mathbf{1} \mid A \times B \mid \sum_{l \in L} A_l \mid U\underline{C}$$
$$\underline{C} := FA \mid \Pi_{l \in L}\, \underline{C}_l \mid A \to \underline{C}$$

Les types de valeur étendent les types de signature en ajoutant le constructeur de type $U\_$. Le type $U\underline{C}$ représente les types de calcul $\underline{C}$ qui ont été bloqué dans une valeur. Le calcul pourra être forcé plus tard.

FA représente les calculs qui retourne une valeur de type A. La fonction $A \to \underline{C}$ représente les calculs de type $\underline{C}$ nécessitant un paramètre de type A. Pour finir, le type de calcul produit $\Pi_{l \in L}\underline{C}_l$ représente un ensemble fini indexé de calcul produit de type $\underline{C}_l$, pour $l \in L$. Ces tuples ne sont pas évalués de façon séquentiel comme dans la stratégie d'évaluation *call-by-value*. À la place, un composant d'un tuple est évalué seulement une fois qu'il a été sélectionné via une projection.

2.3 Termes

Les termes se sépare en trois sous-ensemble de termes : les termes **valeur** V, W, les termes **calcul** M, N et les termes **gestionnaire** H. Ils sont défini par :

$$\begin{aligned} V, W &:= x \mid f(V) \mid \langle\rangle \mid \langle V, W\rangle \mid l(V) \mid \mathbf{thunk}\ M \\ M, N &:= \mathbf{match}\ V\ \mathbf{with}\ \langle x, y\rangle \mapsto M \mid \mathbf{match}\ V\ \mathbf{with}\ \{l(x_l) \mapsto M_l\}_{l\in L} \mid \mathbf{force}\ V \mid \\ &\quad \mathbf{return}\ V \mid M\ \mathbf{to}\ x : A.N \mid \langle M_l\rangle_{l\ \in L} \mid \mathbf{prj}_l\ M \mid \lambda x : A.M \mid M\ V \mid k(V) \mid \\ &\quad \mathbf{op}_V(x : \beta.M) \mid M\ \mathbf{handled\ with}\ H\ \mathbf{to}\ x : A.N \\ H &:= \{\mathbf{op}_{x:\alpha}(k : \beta \to \underline{C}) \mapsto M_{\mathbf{op}}\}_{\mathbf{op}:\alpha\to\beta} \end{aligned}$$

avec x, y appartenant à l'ensemble des variables de valeur et k appartenant à l'ensemble des variables de continuation. $\{...\}_{l\in L}$ représente l'ensemble des calculs, un pour chaque $l \in L$. Même principe pour $\{...\}_{\mathbf{op}:\alpha\to\beta}$. Les termes non-présents dans la syntaxe de base est expliqué ci-dessous.

L'*application d'opération* symbolisé par $\mathbf{op}_V(x : \beta.M)$ fonctionne de la manière suivante : l'opération **op** paramétré par V est géré, ensuite on affecte le résultat à x et enfin on évalue la continuation M. Si il n'y a pas de gestionnaire les opérations sont effectué de manière classique.

Exemple 5. Le calcul suivant incrémente le nombre n, le stocke dans l'emplacement l et retourne l'ancienne valeur.

$$\mathbf{get}_l(n : \mathbf{nat}.\mathbf{set}_{\langle l,n+1\rangle}(x : \mathbf{1}.\mathbf{return}\ \mathbf{n}))$$

Le terme du gestionnaire $\{\mathbf{op}_{x:\alpha}(k : \beta \to \underline{C}) \mapsto M_{\mathbf{op}}\}_{\mathbf{op}:\alpha\to\beta}$ est défini par un ensemble fini de définitions d'opérations, un ensemble pour chaque opération. $M_{\mathbf{op}}$ est dépendant de la variable x et de la continuation k.

Exemple 6. On ne veut pas modifier la valeur en mémoire mais quand tester pour une valeur arbitraire. Pour cela, on crée un gestionnaire *état-temporaire* qui dépend d'une variable n (la dite variable temporaire) :

$$H_{temporary} = \{\mathbf{get}_{l:\mathbf{loc}}(k : \mathbf{nat} \to \underline{C}) \mapsto k(n)\}$$

Le gestionnaire symbolisé par M **handled with** H **to** $x : A.N$ est évalué comme suit : on évalue M, gérant tous les calculs d'opérations grâce à H. Ensuite on associe le résultat à x et enfin évalue N. En cas de présence d'une opération $\mathbf{op}_V(y.M')$ dans M et de $\mathbf{op}_z(k) \mapsto M_{\mathbf{op}}$ dans H, $M_{\mathbf{op}}[V/z]$ substitue **op**. Ensuite les continuations $k(W)$ présent dans $M_{\mathbf{op}}$ sont substitués par

$$M'[W/y]\ \mathbf{handled\ with}\ H\ \mathbf{to}\ x : A.N$$

La continuation k reçoit la valeur W déterminé par $M_{\mathbf{op}}$. Elle est ensuite géré de la même façon que M, ce qui n'est pas le cas de $M_{\mathbf{op}}$. En effet $M_{\mathbf{op}}$ n'est pas géré par H, donc ses opérations échappent à H mais peuvent cependant être gérées par un autre gestionnaire imbriqué.

Exemple 7. On considère le gestionnaire d'état temporaire $H_{temporary}$ donné dans Exemple 6. On prend la calcul suivant :

$$\begin{aligned} &\mathbf{let}\ n : \mathbf{nat}\ \mathbf{be}\ 20\ \mathbf{in} \\ &\mathbf{get}_l(x : \mathbf{nat}.get_l(y : \mathbf{nat}.\mathbf{return}\ x + y))\ \mathbf{handled\ with}\ H_{temporary}\ \mathbf{to}\ z : A.\mathbf{return}\ z + 2 \end{aligned}$$

Dans ce calcul on va associer n avec la valeur 20. Le premier **get** va être géré par $H_{temporary}$ et va donc associer x avec la valeur de n donc 20. Le second **get** appartient à la continuation et est donc géré aussi par $H_{temporary}$, i.e, il va associer y avec 20 (par la même logique que le premier **get**). On retourne $x + y$ soit 40 et on associe ce résultat à z. On retournera au final 42.

2.4 Jugement de types

Tous les jugements de types sont fait dans le contexte de valeurs : $\Gamma = x_1 : A_1, ..., x_m : A_m$ et dans le contexte de continuations : $K = k_1 : \alpha_1 \to \underline{C}_1, ..., k_n : \alpha_n \to \underline{C}_n$.
Les valeurs sont typés en suivant les règles ci-dessous :

$$\Gamma \mid K \vdash x : A \ (x : A \in \Gamma) \qquad \frac{\Gamma \mid K \vdash V : \alpha}{\Gamma \mid K \vdash f(V) : \beta} \ (\mathbf{f} : \alpha \to \beta) \qquad \frac{\Gamma \mid K \vdash M : \underline{C}}{\Gamma \mid K \vdash \mathbf{thunk}\ M : U\underline{C}}$$

$$\frac{\Gamma \mid K \vdash V : A \qquad \Gamma \mid K \vdash W : B}{\Gamma \mid K \vdash \langle V, W \rangle : A \times B} \qquad \frac{\Gamma \mid K \vdash V : A_l}{\Gamma \mid K \vdash l(V) : \sum_{l \in L} A_l} \ (l \in L) \qquad \Gamma \mid K \vdash \langle\rangle : \mathbf{1}$$

Les calculs sont typés en suivant les règles ci-dessous :

$$\frac{\Gamma \mid K \vdash V : A \times B \qquad \Gamma\ x : A, y : B \mid K \vdash M : \underline{C}}{\Gamma \mid K \vdash \mathbf{match}\ V\ \mathbf{with}\ \langle x, y \rangle \mapsto M : \underline{C}}$$

$$\frac{\Gamma \mid K \vdash V : \sum_{l \in L} A_l \qquad \Gamma\ , x_l : A_l \mid K \vdash M_l : \underline{C} \ \ (l \in L)}{\Gamma \mid K \vdash \mathbf{match}\ V\ \mathbf{with}\ \{l(x_l) \mapsto M_l\}_{l \in L} : \underline{C}} \qquad \frac{\Gamma \mid K \vdash V : U\underline{C}}{\Gamma \mid K \vdash \mathbf{force}\ V : \underline{C}}$$

$$\frac{\Gamma \mid K \vdash M : FA \qquad \Gamma, x : A \mid K \vdash N : \underline{C}}{\Gamma \mid K \vdash M\ \mathbf{to}\ x : A.N : \underline{C}} \qquad \frac{\Gamma \mid K \vdash V : A}{\Gamma \mid K \vdash \mathbf{return}\ V : FA}$$

$$\frac{\Gamma \mid K \vdash V : \alpha \qquad \Gamma, x : \beta \mid K \vdash M : \underline{C}}{\Gamma \mid K \vdash \mathbf{op}_V(x : \beta.M) : \underline{C}} \ (\mathbf{op} : \alpha \to \beta) \qquad \frac{\Gamma \mid K \vdash M : \Pi_{l \in L} \underline{C}_l}{\Gamma \mid K \vdash prj\ M : \underline{C}_l} \ (l \in L)$$

$$\frac{\Gamma \mid K \vdash M : A \to \underline{C} \qquad \Gamma \mid K \vdash V : A}{\Gamma \mid K \vdash M\ V : \underline{C}} \qquad \frac{\Gamma, x : A \mid K \vdash M : \underline{C}}{\Gamma \mid K \vdash \lambda x : A.M : A \to \underline{C}}$$

$$\frac{\Gamma \mid K \vdash V : \alpha}{\Gamma \mid K \vdash k(V) : \underline{C}} \ (k : \alpha \to \underline{C} \in K) \qquad \frac{\Gamma \mid K \vdash M_l : \underline{C}_l \ \ (l \in L)}{\Gamma \mid K \vdash \langle M_l \rangle_{l\ \in L} : \Pi_{l \in L} \underline{C}_l}$$

$$\frac{\Gamma \mid K \vdash M : FA \qquad \Gamma \mid K \vdash H : \underline{C}\ \mathbf{handler} \qquad \Gamma, x : A \mid K \vdash N : \underline{C}}{\Gamma \mid K \vdash M\ \mathbf{handled\ with}\ H\ \mathbf{to}\ x : A.N : \underline{C}}$$

Les gestionnaires sont typés en suivant la règle ci-dessous :

$$\frac{\Gamma, x : \alpha \mid K, k : \beta \to \underline{C} \vdash M_{\mathbf{op}} : \underline{C} \ \ (\mathbf{op} : \alpha \to \beta)}{\Gamma \mid K \vdash \{\mathbf{op}_{x:\alpha}(k : \beta \to \underline{C}) \mapsto M_{\mathbf{op}}\}_{\mathbf{op}:\alpha \to \beta} : \underline{C}\ \mathbf{handler}}$$

À partir de maintenant, on utilisera une succession d'abréviations décrite en Annexe.

Définition 3. *Quand un gestionnaire contient des termes gérant uniquement un sous-ensemble Θ de symbole d'opérations, on assume que les symboles d'opérations restants sont gérés par "eux-même". De façon plus formel, on va définir le gestionnaire tel que :*

$$\{\boldsymbol{op}_x(k) \mapsto M_{\boldsymbol{op}}\}_{\boldsymbol{op} \in \Theta} \stackrel{def}{=} \left\{ \boldsymbol{op}_x(k) \mapsto \begin{cases} M_{\boldsymbol{op}} & (\boldsymbol{op} \in \Theta) \\ \boldsymbol{op}_x(y : \beta.k(y)) & (\boldsymbol{op} \notin \Theta) \end{cases} \right\}_{\boldsymbol{op}}$$

3 Exemples

3.1 Non-déterminisme explicite

L'évaluation d'un calcul non-déterminisme prend habituellement un seul chemin possible. Une alternative est de prendre tous les chemins possibles et d'autoriser qu'un chemin échoue. Le second choix va être représenté dans la syntaxe du langage. Pour cela, Le symbole d'opération **fail** est ajouté afin de représenter les chemins qui échouent.

On considère un gestionnaire qui introduit le résultat d'un calcul dans une liste. Les listes de types basiques **b** sont définit par le symbole $\mathbf{list_b}$. Ainsi que les symboles de fonctions: $\mathbf{nil} : \mathbf{1} \to \mathbf{list_b}$, $\mathbf{cons} : \mathbf{b} \times \mathbf{list_b} \to \mathbf{list_b}$ et $\mathbf{append} : \mathbf{list_b} \times \mathbf{list_b} \to \mathbf{list_b}$.

On définit un calcul qui pour tout élément de type **b** retourne un élément de type $\mathbf{list_b}$ tel que :

$$\Gamma \mid K \vdash M \textbf{ handled with } H_{list} \textbf{ to } x : \mathbf{b}.\textbf{return cons}(x, \textbf{nil}) : F\mathbf{list_b}$$

avec H_{list} définit par :

$$\begin{aligned}\Gamma \mid K \vdash \{ & \\ & \textbf{fail}() \mapsto \textbf{return nil}, \\ & \textbf{choose}(k_1, k_2) \mapsto k_1 \textbf{ to } l_1 : \mathbf{list_b}.k_2 \textbf{ to } l_2 : \mathbf{list_b}.\textbf{return append}(l_1, l_2) \\ & \} : F\mathbf{list_b} \textbf{ handler}\end{aligned}$$

Voyons comment le gestionnaire fonctionne.

(1) On considère que M contient un symbole d'opération **choose**. Lorsque l'on va évaluer l'opération $\textbf{choose}(x, y)$, on va regarder si M est géré. Ici, on a H_{list}. Ensuite on va voir dans H_{list} si **choose** est géré. C'est le cas, on va donc calculer

$$M_{\textbf{choose}}[x/k1, y/k2] = (k_1 \textbf{ to } l_1 : \mathbf{list_b}.k_2 \textbf{ to } l_2 : \mathbf{list_b}.\textbf{return append}(l_1, l_2))[x/k1, y/k2]$$

Deux choses à noter : la continuation n'est pas utilisé dans $M_{\textbf{choose}}$; on part du principe que les continuations k_1 et k_2 retourne un élément de type $\mathbf{list_b}$.

(2) On considère que M contient un symbole d'opération **fail**. En suivant la même logique que ci-dessus, On va calculer

$$M_{\textbf{fail}} = \textbf{return nil}$$

Une fois encore la continuation n'est pas gardé. Il est important de noter qu'un chemin invalide retourne aussi un élément de type $\mathbf{list_b}$.

3.2 Timeout

Le gestionnaire du *timeout* donne un exemple de paramètre passé dans la continuation. On souhaite effectuer un calcul et attendre pour un certains temps donné. Si le calcul n'est pas compléter avant le temps donné alors on retourne une valeur par défaut. On représente le temps via une seule opération $\mathbf{delay} : \mathbf{nat} \to \mathbf{1}$, telle que $\mathbf{delay}_t(M)$ est le calcul qui attend un temps t et ensuite effectue le calcul M. Pour tout type A, on a le gestionnaire $H_{timeout}$ suivant:

$$\begin{aligned}\Gamma, x_0 : A, t_{wait} : \mathbf{nat} \mid K \vdash \{ & \\ & \mathbf{delay}_t(k : \mathbf{nat} \to FA) \mapsto \lambda t_{spent} : \mathbf{nat}. \\ & \textbf{if } t + t_{spent} \leq t_{wait} \\ & \quad \textbf{then } \mathbf{delay}_t(k(t + t_{spent})\ t_{spent}) \\ & \quad \textbf{else } \mathbf{delay}_{t_{wait} - t_{spent}}(\textbf{return } x_0) \\ & \} : \mathbf{nat} \to FA \textbf{ handler}\end{aligned}$$

Le terme qui gère **delay** (que l'on va nommé $M_{\mathbf{delay}}$) dépend de la variable t_{spent}, qui représente le temps que l'on a déjà attendu. Si l'on souhaite utiliser ce gestionnaire, on va devoir lui spécifier un paramètre initial. On veut gérer $M : FA$ via $H_{timeout}$ or on a le calcul suivant :

$$\Gamma, x_0 : A, t_{wait} : \mathbf{nat} \mid K \ \vdash \ M \ \mathbf{handled\ with}\ H_{timeout}\ \mathbf{to}\ x : A.\lambda t : \mathbf{nat}.\mathbf{return}\ x : \mathbf{nat} \to FA$$

Le calcul géré est de type $\mathbf{nat} \to FA$, on voit la nécessité du paramètre initial pour passer à un calcul géré de type FA.

$$\Gamma, x_0 : A, t_{wait} : \mathbf{nat} \mid K \ \vdash \ (M \ \mathbf{handled\ with}\ H_{timeout}\ \mathbf{to}\ x : A.\lambda t : \mathbf{nat}.\mathbf{return}\ x)\ 0 : FA$$

il est nécessaire d'ajouter une abstraction autour de **return** x car si on suppose que M ne contient pas de symbole d'opération **delay** alors on aura le calcul suivant : $(\lambda t : \mathbf{nat}.\mathbf{return}\ x)\ 0 : FA$.

Exemple 8. On effectue le calcul ci-dessous avec $M = \mathbf{delay}_5(t_2.(\mathbf{return}\ t_2 + 45))$ (on symbolisera le passage d'une étape de calcul via le symbole $\Rightarrow$):

$\lambda t_{wait}.(\lambda x_0.((M \ \mathbf{handled\ with}\ H_{timeout}\ \mathbf{to}\ x.\lambda t.\mathbf{return}\ x)\ 0)\ 4)\ 20$

$\Rightarrow\ \lambda x_0.((M \ \mathbf{handled\ with}\ H_{timeout}\ \mathbf{to}\ x.\lambda t.\mathbf{return}\ x)\ 0)\ 4\ [20/t_{wait}]$

$\Rightarrow\ (M \ \mathbf{handled\ with}\ H_{timeout}\ \mathbf{to}\ x.\lambda t.\mathbf{return}\ x)\ 0\ [20/t_{wait}, 4/x_0]$

Rappel : Pour évaluer un calcul géré, on évalue en premier M; ensuite on associe la valeur de retour de M à x et enfin on évalue $\lambda t.\mathbf{return}\ x$.

$\Rightarrow\ \mathbf{delay}_5(t_2.(\mathbf{return}\ t_2 + 45)\ \mathbf{handled\ with}\ H_{timeout}\ \mathbf{to}\ x.\lambda t.\mathbf{return}\ x)\ 0\ [20/t_{wait}, 4/x_0]$

delay est un symbole d'opération géré par $H_{timeout}$. On va donc prendre le terme gérant $M_{\mathbf{delay}}$. Pour alléger l'écriture des substitutions ci-après on définit *conti* telle que

$conti = (\mathbf{return}\ t_2 + 45)[t + t_{spent}/t2]\ \mathbf{handled\ with}\ H_{timeout}\ \mathbf{to}\ x.\lambda t.\mathbf{return}\ x\ [20/t_{wait}, 4/x_0]$

$\Rightarrow\ (\lambda t_{spent}.\mathbf{if}\ t + t_{spent} \leq t_{wait}\ \mathbf{then}\ \mathbf{delay}_t(k(t + t_{spent})\ t_{spent})\ \mathbf{else}\ \mathbf{delay}_{t_{wait}-t_{spent}}(\mathbf{return}\ x_0)$
$[conti/k(t + t_{spent}), 5/t])\ 0\ [20/t_{wait}, 4/x_0]$

$\Rightarrow\ (\lambda t_{spent}.\mathbf{if}\ 5 + t_{spent} \leq 20\ \mathbf{then}\ \mathbf{delay}_5(k(5 + t_{spent})\ t_{spent})\ \mathbf{else}\ \mathbf{delay}_{20-t_{spent}}(\mathbf{return}\ 4)$
$[conti/k(5 + t_{spent}), 5/t])\ 0\ [20/t_{wait}, 4/x_0]$

$\Rightarrow\ \mathbf{if}\ 5 \leq 20\ \mathbf{then}\ \mathbf{delay}_5(k(5)\ 0)\ \mathbf{else}\ \mathbf{delay}_{20}(\mathbf{return}\ 4)\ [0/t_{spent}, conti/k(5), 5/t, 20/t_{wait}, 4/x_0]$

$\Rightarrow\ \mathbf{delay}_5(k(5)\ 0)[conti/k(5), 20/t_{wait}, 4/x_0]$

delay n'est pas géré donc on effectue juste l'opération.

$\Rightarrow\ k(5)\ 0\ [conti/k(5), 20/t_{wait}, 4/x_0]$

$\Rightarrow\ (\mathbf{return}\ 5 + 45\ \mathbf{handled\ with}\ H_{timeout}\ \mathbf{to}\ x.\lambda t.\mathbf{return}\ x)\ 0\ [20/t_{wait}, 4/x_0]$

$\Rightarrow\ (\lambda t.\mathbf{return}\ x)\ 0\ [50/x, 20/t_{wait}, 4/x_0]$

$\Rightarrow\ \mathbf{return}\ 50\ [0/t, 50/x, 20/t_{wait}, 4/x_0]$

3.3 Rollback

Quand un calcul provoque une levée d'exception, il est primordial d'annuler toute modification faite dans la mémoire. Ce comportement est nommé *rollback*. On va supposer que l'on a qu'un seul emplacement (pour plus de simplicité) l_0. Un gestionnaire avec passage de paramètre, qui ne modifie pas la mémoire, mais garde la trace de tous les changements effectués dans M permet d'effectuer un *rollback* efficace. Si aucune exception est levée alors on met à jour l'emplacement. Ce gestionnaire $H_{param-rollback}$ est défini par :

$$\begin{aligned}\Gamma \mid K \ \vdash \ \{ & \\ & \mathbf{get}_{l_0}(k : \mathbf{nat} \to (\mathbf{nat} \to \underline{C})) \mapsto \lambda n : \mathbf{nat}.k(n)\ n \\ & \mathbf{set}_{l_0,n'}(k : \mathbf{1} \to (\mathbf{nat} \to \underline{C})) \mapsto \lambda n : \mathbf{nat}.k()\ n' \\ & \mathbf{raise}_e() \mapsto \lambda n : \mathbf{nat}.M'\ e \\ & \} : \mathbf{nat} \to \underline{C}\ \mathbf{handler}\end{aligned}$$

Il est utilisable de la manière suivante :

$$\Gamma \mid K \ \vdash \ \mathbf{get}_{l_0}(n_{init} : \mathbf{nat}.(M\ \mathbf{handled\ with}\ H_{rollback}\ \mathbf{to}\ x : A.\lambda n : \mathbf{nat}.\mathbf{set}_{l_0,n}(\mathbf{return}\ x))\ n_{init})$$

On récupère la valeur initial de l'emplacement l_0 que l'on associe à la variable n_{init}.

4 Sémantique

La partie interprétation sera omit dans ce résumé car il fait appel à des notions de catégorie et des notions poussées sur les modèles. Je n'ai pas la prétention d'être en capacité de le comprendre entièrement et de l'expliquer.

4.1 La théorie des effets

Les propriétés des effets sont décrites avec des équation entre patrons T. Ils décrivent la forme général de tous les calculs sans présumer des types. T est définit par :

$$T := z(V) \mid \mathbf{match}\ V\ \mathbf{with}\ \langle x, y\rangle \mapsto T \mid \mathbf{match}\ V\ \mathbf{with}\ \{l(x_l) \mapsto T_l\}_{l\in L} \mid \mathbf{op}_V(x : \beta.T)$$

avec z appartenant à un ensemble de variable de patron. Dans les patrons, on se limite à la signature des valeurs. Ceux sont des valeurs qui peuvent être typées par $\Gamma \vdash V : \alpha$, avec $\Gamma = x_1 : \alpha_1, ..., x_m : \alpha_m$.

Les patrons sont construit dans le contexte de variables de valeur Γ et un contexte de patron : $Z = z_1 : \alpha_1, ..., z_n : \alpha_n$. Attention, $z_j : \alpha_j$ ne représente pas une valeur de type α_j mais un calcul qui dépend d'une telle valeur. Le jugement de typage pour un patron bien-formé est $\Gamma \mid Z \vdash T$ et est donné par les règles suivantes :

$$\frac{\Gamma \vdash V : \alpha}{\Gamma \mid Z \vdash z(V)}\ (z : \alpha \in Z) \qquad \frac{\Gamma \vdash V : A \times B \qquad \Gamma\ x : \alpha, y : \beta \mid Z \vdash T}{\Gamma \mid Z \vdash \mathbf{match}\ V\ \mathbf{with}\ \langle x, y\rangle \mapsto T}$$

$$\frac{\Gamma \vdash V : \sum_{l\in L} \alpha_l \qquad \Gamma\ , x_l : \alpha_l \mid Z \vdash T_l\ \ (l \in L)}{\Gamma \mid Z \vdash \mathbf{match}\ V\ \mathbf{with}\ \{l(x_l) \mapsto T_l\}_{l\in L}} \qquad \frac{\Gamma \vdash V : \alpha \qquad \Gamma, x : \beta \mid Z \vdash T}{\Gamma \mid Z \vdash \mathbf{op}_V(x : \beta.T)}\ (\mathbf{op} : \alpha \to \beta)$$

Une théorie d'effet τ est un ensemble fini d'équation $\Gamma \mid Z \vdash T_1 = T_2$, avec T_1 et T_2 bien formé par rapport à Γ et Z.

Exemple 9. On reprend les symboles d'opérations de l'Exemple 4.

> **Exceptions:** La théorie d'effet est un ensemble vide, car les exceptions ne satisfont aucune équations non-trivial.

États: La théorie d'effet est définie par les équations suivantes (on n'écrit pas les contextes pour une meilleur lecture) :

$$\begin{aligned}
\mathbf{get}_l(x.z) &= z \\
\mathbf{get}_l(x.\mathbf{set}_{l,x}(z)) &= z \\
\mathbf{set}_{l,x}(\mathbf{set}_{l,x'}(z)) &= \mathbf{set}_{l,x'}(z) \\
\mathbf{set}_{l,x}(\mathbf{get}_l(x'.z(x'))) &= \mathbf{set}_{l,x}(z(x)) \\
\mathbf{get}_x(x.\mathbf{get}_l(x'.z(x,x'))) &= \mathbf{get}_l(x.z(x,x)) \\
\mathbf{set}_{l,x}(\mathbf{set}_{l',x'}(z)) &= \mathbf{set}_{l',x'}(\mathbf{set}_{l,x}(z)) && (l \neq l') \\
\mathbf{set}_{l,x}(\mathbf{get}_{l'}(x'.z(x'))) &= \mathbf{get}_{l'}(x'.\mathbf{set}_{l,x}(z(x'))) && (l \neq l') \\
\mathbf{get}_l(x.\mathbf{get}_{l'}(x'.z(x,x'))) &= \mathbf{get}_{l'}(x'.\mathbf{get}_l(x.z(x,x'))) && (l \neq l')
\end{aligned}$$

Non-déterminisme: La théorie d'effet est définie par les équations suivants :

$$\begin{aligned}
\mathbf{choose}(z,z) &= z \\
\mathbf{choose}(z_1,z_2) &= \mathbf{choose}(z_2,z_1) \\
\mathbf{choose}(z_1,\mathbf{choose}(z_2,z_3)) &= \mathbf{choose}(\mathbf{choose}(z_1,z_2),z_3)
\end{aligned}$$

Temps: La théorie d'effet est définie par les équations suivants :

$$\begin{aligned}
\mathbf{delay}_0(z) &= z \\
\mathbf{delay}_{t_1}(\mathbf{delay}_{t_2}(z)) &= \mathbf{delay}_{t_1+t_2}(z)
\end{aligned}$$

Exception destructrice: La signature des exceptions destructives est l'union de la signature de l'état et de l'exception. La théorie d'effet comprend toute les équations de l'état plus les équations suivantes :

$$\begin{aligned}
\mathbf{get}_l(x.\mathbf{raise}_e()) &= \mathbf{raise}_e() \\
\mathbf{set}_{l,x}(\mathbf{raise}_e()) &= \mathbf{raise}_e()
\end{aligned}$$

Les équations ci-dessus impliquent que les opérations sur la mémoire suivit par une levée d'exception est la même chose que juste lever une exception. Cela implique que toutes les opérations sur la mémoire sont discutables si une exception apparaît. La théorie des exceptions destructives est discutée comme celle du "rollback" dans [3].

5 Raisonnement à propos des gestionnaires

Deux questions se posent : Quels calculs sont égaux ? Quels gestionnaires sont valides ?

On fixe une signature et une théorie d'effet τ avec ses interprétations.

Remarque. Dans la suite on utilisera l'égalité, l'inégalité de Kleene ainsi que les assertions d'existence. On introduit donc les notations ainsi que leurs signification dans cette remarque.

assertion d'existence: on note $e\downarrow$ l'assertion et elle est vraie si et seulement si l'expression e est définie.

égalité de kleene: on la note $e \simeq e'$ et elle vrai si les expressions sont définies et équivalente ou les expressions sont indéfinies.

inégalité de kleene: on la note $e \lesssim e'$ et elle abrège la notation $e\downarrow \;\Rightarrow\; e \simeq e'$

On interprétera $H\downarrow$ comme la validité du gestionnaire.

Exemple 10. On peut définir une β-inéquation de Kleene avec la construction suivante :

$$\textbf{return } x \textbf{ to } x : A.M \lesssim M$$

De la même façon, l'η-équation peut se définir comme $\lambda x : A.M\ x \simeq M$

On définit deux inéquations qui font écho au équation de la section 1 sur les gestionnaire d'exception. Pour tout $H = \{\mathbf{op}_y(k) \mapsto M_{\mathbf{op}}\}_{\mathbf{op}:\alpha\to\beta}$, les inéquations sont :

$$\textbf{return } x \textbf{ handled with } H \textbf{ to } x : A.N \lesssim N$$
$$\mathbf{op}_y(x' : \beta.M) \textbf{ handled with } H \textbf{ to } x : A.N \lesssim M_{\mathbf{op}}[x' : \beta.M \textbf{ handled with } H \textbf{ to } x : A.N/k]$$

où la substitution de la forme $x' : \beta.M'/k$ remplace chaque occurrence du calcul $k(W)$ par $M'[V/x']$. Voici l'équivalence qui permet d'admettre l'existence d'un gestionnaire et sa validité.

$$H \downarrow \ \Leftrightarrow\ \bigwedge\{\forall x : \alpha.M_{\mathbf{op}} \downarrow \mid \mathbf{op} : \alpha \to \beta\} \ \wedge\ \bigwedge\{T_1^H \simeq T_2^H \ \mid\ \Gamma \mid Z \vdash T_1 = T_2 \in \tau\}$$

elle affirme que le gestionnaire est valide si les opérations sont définit et respect les équations de la théorie des effets. T^H représente un patron ayant un gestionnaire. La construction du gestionnaire en elle-même amène à l'équivalence suivante :

$$M \textbf{ handled with } H \textbf{ to } x : A.N \downarrow \ \Leftrightarrow\ M \downarrow \ \wedge\ H \downarrow \ \wedge\ \forall x : A.N \downarrow$$

6 Validité des gestionnaires

L'équivalence ci-dessus exprime la validité du gestionnaire. Savoir si cette validité est décidable est décrit dans la section qui suit.

Définition 4. *Un problème de décision est dit* ***décidable*** *s'il existe un algorithme qui se termine en un nombre fini d'étapes, qui le décide, i.e, qui répond par oui ou par non à la question posée par le problème.*

Définition 5. *Un gestionnaire* $\Gamma \mid K \vdash H : \underline{C}$ ***handler*** *est* ***simple*** *si*

- *en réorganisant,* Γ *a la forme*

$$x_1 : \alpha_1, ..., x_m : \alpha_m, f_1 : U(\beta_1 \to \underline{C}), ..., f_n : U(\beta_n \to \underline{C})$$

- *en réorganisant,* K *a la forme*

$$k'_1 : \beta'_1 \to \underline{C}, ..., k'_p : \beta'_p \to \underline{C}$$

- *et pour tout* $\boldsymbol{op} : \alpha \to \beta$*, il y a un patron*

$$x_1 : \alpha_1, ..., x_m : \alpha_m, x : \alpha \mid z_1 : \beta_1, ..., z_n : \beta_n, z'_1 : \beta'_1, ..., z'_p : \beta'_p, z : \beta \vdash T_{\boldsymbol{op}}$$

tel que le terme gérant

$$\Gamma, x : \alpha \mid K, k : \beta \to \underline{C} \vdash M_{\boldsymbol{op}} : \underline{C}$$

est obtenu par la substitution de $T_{\boldsymbol{op}}$ *qui remplace tous les* x_i *par eux-même,* x *par lui-même, tous les* z_j *par* $(y_j : \beta_j.(\boldsymbol{force}\ f_j)\ y_j)$*, tous les* z'_l *par* $(y'_l : \beta'_l.k'_l(y'_l))$ *et* z *par* $(y : \beta.k(y))$*.*

Le gestionnaire $H_{Temporary}$ (Exemple 6) est **simple**, aucun des gestionnaires avec des paramètres passés sont simple car ils contiennent tous une lambda abstraction dans leur terme gérant. Le gestionnaire d'exception

$$\{\mathbf{raise}_{y:exc}(k : \mathbf{0} \to \underline{C}) \mapsto \textbf{match } y \textbf{ with } \{e(z) \mapsto N_e\}_{e\in\mathbf{exc}}\}$$

n'est pas simple. Toutefois, on peut utiliser le gestionnaire simple

$$f : U(\mathbf{exc} \to \underline{C}) \vdash \{\mathbf{raise}_{e:\mathbf{exc}}(k : \mathbf{0} \to \underline{C}) \mapsto (\textbf{force } f)\ e\}$$

avec f la fonction bloquée tel que

$$f = \lambda y : \mathbf{exc}.\textbf{match } y \textbf{ with } \{e(z) \mapsto N_e\}_{e\in\mathbf{exc}}$$

Ce gestionnaire a le même comportement que celui définit plus haut.

Remarque. Une signature est **simple** si elle n'a pas de types de base et pas de symboles de fonction. Les signatures données pour les exceptions sont simple mais pas pour les états.

Remarque. Dans la suite, on va parler de la hiérarchie arithmétique. Un petit point s'impose.

Dans la théorie de la calculabilité, la hiérarchie arithmétique est une hiérarchie des sous-ensembles de $\mathbb{N}$ définissables dans le langage du premier ordre de l'arithmétique de Peano. Un ensemble d'entiers est classé suivant les alternances de quantificateurs d'une formule sous forme prénexe qui permet de le définir.

L'alternance des quantificateurs est séparé en 2 classes : $\sum$ et Π. Au niveau 0, ces deux classes sont identique. On peut définir, le reste des niveaux inductivement.

Pour un entier naturel n non nul:

- Si A est une formule $\sum_0$ (ou Π_0), $\exists x\ A$ est une formule $\sum_1$ et $\forall x\ A$ une formule Π_1 ;
- Si S est une formule $\sum_n$, si P est une formule Π_n, et si x est une variable quelconque alors :
  - $\exists x\ S$ reste $\sum_n$;
  - $\exists x\ P$ est une formule $\sum_{n+1}$;
  - $\forall x\ S$ est une formule Π_{n+1} ;
  - $\forall x\ P$ reste Π_n.

J'ai donné la définition de base, pour en savoir plus n'hésitez pas à aller voir le cours de l'irif.

Théorème 1. *Le problème de décider (avec une signature simple, une théorie d'effet et un simple gestionnaire clos $\vdash H : F\mathbf{0}$ **handler**) si un gestionnaire est valide, est Π_2-**complet**.*

La nature polymorphique des variables de patron signifie que les patrons peuvent définir une famille entière de gestionnaire, une pour chaque type de calcul. Cela nous amène à la définition d'une famille de gestionnaire uniformément simple.

Définition 6. *Une famille de gestionnaire $\{\Gamma_{\underline{C}} \mid K_{\underline{C}} \vdash H_{\underline{C}} : \underline{C}$ **handler**$\}_{\underline{C}}$, avec $\underline{C}$ l'ensemble des types de calcul, est **uniformément simple** si*

- *en réorganisant, $\Gamma_{\underline{C}}$ a la forme*

$$x_1 : \alpha_1, ..., x_m : \alpha_m, f_1 : U(\beta_1 \to \underline{C}), ..., f_n : U(\beta_n \to \underline{C})$$

- *en réorganisant, $K_{\underline{C}}$ a la forme*

$$k'_1 : \beta'_1 \to \underline{C}, ..., k'_p : \beta'_p \to \underline{C}$$

- *et pour tout $\mathbf{op} : \alpha \to \beta$, il y a un patron*

$$x_1 : \alpha_1, ..., x_m : \alpha_m, x : \alpha \mid z_1 : \beta_1, ..., z_n : \beta_n, z'_1 : \beta'_1, ..., z'_p : \beta'_p, z : \beta \vdash T_{\mathbf{op}}$$

  tel que le terme gérant

$$\Gamma_{\underline{C}}, x : \alpha \mid K_{\underline{C}'}, k : \beta \to \underline{C} \vdash M_{\mathbf{op}} : \underline{C}$$

  est obtenu par la substitution de $T_{\mathbf{op}}$ qui remplace tous les x_i par eux-même, x par lui-même, tous les z_j par $(y_j : \beta_j.(\mathbf{force}\ f_j)\ y_j)$, tous les z'_l par $(y'_l : \beta'_l.k'_l(y'_l))$ et z par $(y : \beta.k(y))$.

Tous les gestionnaires du famille uniformément simple sont simple. Corollairement, tous les gestionnaires simple, via l'aspect polymorphique des variables de patron, peuvent définir une famille uniformément simple. Dans ceux vu en amont, le gestionnaire $H_{Temporary}$ est uniformément simple. Tous ceux qui sont définit pour un type de calcul précis (ex **F0**) ne sont pas uniformément simple.

Définition 7. *Une famille de gestionnaires $\{\Gamma_{\underline{C}} \mid K_{\underline{C}} \vdash H_{\underline{C}} : \underline{C}$ **handler**$\}_{\underline{C}}$ pour une signature donnée, avec $\underline{C}$ l'ensemble des calculs, est valide si chaque gestionnaire de la famille est valide.*

Sachant qu'une famille de gestionnaires uniformément simple ne peut utiliser les propriétés d'un type de calcul spécifique, la validité peut devenir **semi-décidable**.

Théorème 2. *Le problème de décider (avec une signature donnée, une théorie d'effet et une famille de gestionnaires* $\{\vdash H_{\underline{C}} : \underline{C}$ ***handler***$\}_{\underline{C}}$ *close uniformément simple) si une famille est valide, est* $\sum_1$***-complet.***

Les théories d'effet, avec une signature simple, correspondent à des théories d'équations finies ordinaire. Ceci étant, on peut transférer la notion de décidabilité sur eux. Avec ça on peut dire :

Théorème 3. *Le problème de décider (avec une signature donnée, une théorie d'effet décidable et une famille de gestionnaires* $\{\vdash H_{\underline{C}} : \underline{C}$ ***handler***$\}_{\underline{C}}$ *close uniformément simple) si une famille est valide, est* ***décidable.***

7 Confrontation

La section qui va suivre va confronter le langage proposé dans l'article avec le langage créé avec M. Dabrowski et Mme. Bousdira : **erpl**. C'est un langage fonctionnel avec des traits impératifs reprenant la syntaxe d'Ocaml. Les effets algébrique ont été ajouté dans ce langage mais avec une autre implémentation.

Toute la partie qui va suivre sera informelle et les principes mathématiques seront écartés. Le but étant de confronter les idées et l'implémentation. Ainsi que les potentiels problèmes liés à une implémentation du langage proposé dans l'article.

7.1 Point sur la représentation des effets dans la syntaxe

L'article propose une extension du langage décrit dans [4] en ajoutant trois termes afin de gérer les effets algébrique :

$$\begin{aligned} \textit{une structure de gestion} : &\quad M \textbf{ handled with } H \textbf{ to } x : A.N \\ \textit{une operation instigatrice des } effets : &\quad \mathbf{op}_V(x : \beta.M) \\ \textit{un gestionnaire} : &\quad \{\mathbf{op}_{x:\alpha}(k : \beta \to \underline{C}) \mapsto M_{\mathbf{op}}\}_{\mathbf{op}:\alpha\to\beta} \end{aligned}$$

L'**erpl** propose, quant à lui, part du langage Ocaml en allègé et ajoute trois termes similaire :

$$\begin{aligned} \textit{une structure de gestion} : &\quad \textbf{handle } M \textbf{ with } H \\ \textit{une operation instigatrice des } effets : &\quad \textbf{perform } V \\ \textit{un gestionnaire} : &\quad \{l(x_l), (k : \beta \to \underline{C}) \mapsto M_{\mathbf{l}}\}_{l\in L} \end{aligned}$$

Remarque. Les deux langages différents ont a le même nombre de termes ajouté. On peut interpréter cela comme l'ajout minimum nécessaire pour travailler avec des effets algébrique.

7.2 Représentation divergente

Il faut mettre au clair plusieurs points.

Premièrement, les deux langages n'ont pas été créé de la même façon. Le langage de l'article prend à coeur le côté mathématique de la gestion des effets, ce qui a pour conséquence un langage plus lourd mais aussi plus général. De son côté, le langage **erpl** a été conçu surtout pour avoir un langage dérivant de l'Ocaml plus léger. Il n'avait pas pour but de généraliser un concept.

Deuxièmement, le temps de travail sur les deux langages est tout à fait inégal. Le langage **erpl** est le fruit d'un an de travail avec une pause entre deux.

Dernièrement, les concept du langage **erpl** a été donnée par un chercheur mais l'implémentation a été laissé à un stagiaire, or je vais m'appuyer sur son travail pour confronter les langages. Il est possible que l'état actuel du langage diverge un peu de ce qui est voulu au final.

Malgré cela, il est intéressant de voir la différence entre deux langages qui utilisent le même principe mais ayant des buts différents.

La structure de gestion Il n'y a pas grande différence entre les deux langages sur ce point. On peut noter qu'aucun n'est tombé dans le travers de la structure de gestion binaire qui ne respecte pas les propriétés implicites (voir l'Exemple **??**).

La seule différence notable est l'affectation du résultat de l'évaluation de la strucutre de gestion à une variable x utilisé pour un autre terme N.

$$M \textbf{ handled with } H \textbf{ to } x : A.N$$

Dans le langage **erpl**, on part du principe que la structure va se réduire à une valeur ou une fonction que l'on pourra associer si on veut. Si on veut avoir un terme équivalent à celui ci-dessus, on écrira :

$$\textbf{let } x = (\textbf{handle } M \textbf{ with } H) \textbf{ in } N$$

L'opération instigatrice La source des effets est différents dans les deux langages.

D'un côté, le langage de l'article a choisi les opérations $\mathbf{op}_V(x : \beta.M)$ comme source des effets. Ce choix est intéressant et limitant à la fois. Il permet de limité les effets à un ensemble fini d'opérations décidé par le créateur du langage. On a un contrôle plus fort sur ce qui provoque les effets et de plus leurs utilisation est totalement transparente. En effet, une opération à un double effet, soit il va effectuer ce que l'opération est censée faire soit elle va provoquer un effet (ce point sera rediscuté plus bas).

De l'autre, le langage **erpl** s'inspire plus des **Monades** en prenant comme source les types. Cela ouvre plus de possible (une infinité même) car l'utilisateur peut créer un type et le voir comme un effet. Cependant, un type ne va pas "activé" un gestionnaire, pour palier à cela on a le mot-clé **perform**. En plus d'avoir plus de possibilités on peut décider quand un effet va être "activé" ou non via le mot-clé alors que dans le langage de l'article chaque, opération va être vérifié par un gestionnaire. l'application d'un effet est cependant explicite dans le langage **erpl**.

La gestion de la continuation est elle aussi différente, nous la développerons plus bas.

Le gestionnaire La différence du gestionnaire vient de la source de l'effet. À part ça, leurs fonctionnement est le même.

7.3 Fonctionnement et exemple

On rappelle le fonctionnement de la structure de gestion pour le langage de l'article. On suppose le terme suivant :

$$M \textbf{ handled with } H \textbf{ to } x : A.N$$

avec $\mathbf{op}_V(y.M') \in M$ et $\{\mathbf{op}_z(k) \mapsto M_{\mathbf{op}}\} \in H$. Lorsque l'effet est "activé", on évalue $M_{\mathbf{op}}$ avec des subsitutions comme suit :

$$M_{\mathbf{op}}[V/z, M'[W/y] \textbf{ handled with } H \textbf{ to } x : A.N/k(W)]$$

Le langage **erpl** suit la même logique de subsitution. On définit le terme suivant :

$$\textbf{handle } M \textbf{ with } H$$

avec $\textbf{perform } l(x) \in M$ et $\{l(y), k \mapsto M_l\} \in H$. On évalue l'"activation" de l'effet de la manière suivante :

$$M_l[l(x)/l(y), \textbf{handle } M[W/\textbf{perform } l(x)] \textbf{ with } H/k(W)]$$

Les deux langages fonctionnent de manière similaire et il est possible de représenter le même effet avec les deux langages. Une restriction existe, à la différence du langage de l'article, en **erpl** un seul appel à la continuation est possible dans le terme gérant.

Exemple 11. On va reproduire l'effet de l'Exemple 6 et recréer l'expression de l'Exemple 7. Le langage **erpl** ayant ces termes proches de l'Ocaml, je vais simplifier l'exemple en écrivant la suite en **erpl** pour le confort de tous (les points spécifiques au langage seront expliqués).

```
(**
    On commence par créer notre type qui va être source de l'effet.
    Pour cela on créer un type
*)
type memory = Get of 'a ref -> 'a | Set of 'a ref * 'a -> unit;;

(**
    Ensuite on définit l'expression. En erpl, un gestionnaire prend
    une succession de filtre comme pour les match.
*)
let get = Get(fun x -> !x) and n = 20 in
handle ((fun x -> fun y -> x + y + 2) (perform (get,1))) (perform (get,1))
with (Get f,x),k -> k n ;;
```

7.4 Difficultés inhérentes au langage proposé dans l'article

Le langage utilise une propriété qui peut être compliqué à mettre en place. Lorsqu'une opération n'est pas géré, elle se gère elle-même. Intuitivement, lors de l'implémentation d'une telle propriété on peut se dire que l'on va créé un gestionnaire global qui va récupérer tous les effets non capté auparavant et lui appliquer son "effet de base". Hors si on ajoute ce gestionnaire global, c'est lors de la compilation et ce gestionnaire sera dans le langage de base. Mais pour une opération comme **get** ou encore **delay** cela implique un appel système. Cela pose problème. Il faudrait donc modifier la machine abstraite pour pouvoir implémenter cette option se qui rajoute un travail conséquent.

References

[1] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 42–122. Springer, 2000.

[2] Nick Benton and Andrew Kennedy. Exceptional syntax journal of functional programming. *J. Funct. Program.*, 11(4):395–410, 2001.

[3] Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3):70–99, 2006.

[4] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *High. Order Symb. Comput.*, 19(4):377–414, 2006.

[5] Paul Blain Levy. Monads and adjunctions for global exceptions. In Stephen D. Brookes and Michael W. Mislove, editors, *Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2006, Genova, Italy, May 23-27, 2006*, volume 158 of *Electronic Notes in Theoretical Computer Science*, pages 261–287. Elsevier, 2006.

[6] Gordon D. Plotkin and Matija Pretnar. A logic for algebraic effects. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 118–129. IEEE Computer Society, 2008.

Annexes

Abréviations

Afin de rendre le tout plus lisible, on va appliquer une succession d'abréviations que l'on décrit dans cette section.

Premièrement, on va simplifier les résultats binaires en résultat fini:

$$A_1 \times ... \times A_n \stackrel{\textbf{def}}{=} (A_1 \times ... \times A_{n-1}) \times A_n \qquad (n \geq 3)$$
$$\langle V_1, ..., V_n \rangle \stackrel{\textbf{def}}{=} \langle \langle V_1, ..., V_{n-1} \rangle, V_n \rangle \qquad (n \geq 3)$$

On comprend les résultats binaires, quand $n = 2$, les résultats unaires, quand $n = 1$ (simplement A_1), et le résultat vide, quand $n = 0$ ($\mathbf{1}$).

On détermine les abréviations suivantes :

$$\mathbf{f}(V_1, ..., V_n) \stackrel{\textbf{def}}{=} \mathbf{f}(\langle V_1, ..., V_n \rangle)$$
$$l(V_1, ..., V_n) \stackrel{\textbf{def}}{=} l(\langle V_1, ..., V_n \rangle)$$
$$\mathbf{op}_{V_1,...,V_n}(x : \beta.M) \stackrel{\textbf{def}}{=} \mathbf{op}_{\langle V_1,...,V_n \rangle}(x : \beta.M)$$
$$k(V_1, ..., V_n) \stackrel{\textbf{def}}{=} k(\langle V_1, ..., V_n \rangle)$$

On va omettre les parenthèses vides :

$$\mathbf{f} \stackrel{\textbf{def}}{=} \mathbf{f}() \qquad l \stackrel{\textbf{def}}{=} l() \qquad k \stackrel{\textbf{def}}{=} k()$$

On adapte aussi les destructeurs de tuples de taille finis. À la place d'un destructeur on va s'autoriser une écriture avec une association de multiple variables.

Exemple 12. Par exemple :

$$\mathbf{op}_{x_1,...,x_n}(k) \mapsto M \stackrel{\textbf{def}}{=} (\textbf{match } x \textbf{ with } \langle x_1, ..., x_n \rangle \mapsto M)$$

$$\textbf{if } V \textbf{ then } M \textbf{ else } N \stackrel{\textbf{def}}{=} \textbf{match } x \textbf{ with } \{\textbf{true} \mapsto M,\ \textbf{false} \mapsto N\}$$

Lorsqu'une opération ne retourne rien on va abrégé de la façon suivante :

$$\mathbf{op}_V() \stackrel{\textbf{def}}{=} \mathbf{op}_V(x : \mathbf{0}.\textbf{match } x \textbf{ with } \{\})$$