Théorie des Types Dépendants
Conception et Implémentation

Roman Delgado

Table des matières

1 Introduction

a relire

Dans ce rapport, il sera question de comprendre le typage, une étape clée de la chaîne de compilation. Résumons brièvement la composition d'un compilateur. Celui-ci traduit d'abord le texte écrit par le programmeur en un arbre de syntaxe abstraite. Cette représentation lui permet de manipuler les différents éléments du programme. Si le compilateur ne génère pas d'erreur durant cette étape, on dit que le programme est syntaxiquement correct. Cependant le programmeur à pu écrire des instructions telles que `x = 2 + true`. Même si cette instruction est syntaxiquement correcte, celle-ci n'a aucun sens. Il lui faut maintenant, à partir de l'arbre de syntaxe abstraite, générer du code assembleur, un code compréhensible par les microprocesseurs. Le compilateur peut très bien générer une suite d'instructions assembleur correspondant à l'expression précédente. C'est durant l'exécution du programme que le processeur va interrompre celui-ci. Lorsque ces erreurs surviennent, il est très compliqué de trouver leur source car le programme compilé est très différent de celui écrit par le programmeur. Pour éviter ce genre d'erreurs, qui peuvent survenir très longtemps après le lancement du programme, il nous faut donc vérifier la validité du programme. Afin d'effectuer cette vérification, nous allons insérer un vérificateur de type dans la chaîne de compilation. L'intérêt du typage est donc de vérifier que le programme s'exécutera correctement, sans avoir besoin d'exécuter celui-ci. Il permet aussi d'obtenir des informations très précises sur la source des erreurs.

2 Le λ-calcul non typé

En 1936, Alonzo Church introduit le λ-calcul non typé [Church, 1936]. Le λ-calcul est un modèle de calcul universel, tout comme les machines de Turing. Celui-ci repose sur une syntaxe minimaliste capturant exactement la notion de fonction. Ce formalisme est au cœur de nombreuses recherches dans le domaine de la programmation fonctionnelle. Le λ-calcul non typé a connu un formidable succès pratique, donnant lieu à de multiples variantes dont on trouvera une présentation moderne et synthétique dans l'ouvrage de Pierce [2002].

2.1 Syntaxe

Le λ-calcul est composé uniquement de λ-termes dont voici la syntaxe

$$\begin{array}{rcll} t & ::= & & (\lambda\text{-terme}) \\ & | & x & (\text{variable}) \\ & | & \lambda x.t & (\text{abstraction}) \\ & | & t\,t & (\text{application}) \end{array}$$

Pour faire une analogie avec les mathématiques, une abstraction peut être vue comme une définition de fonction anonyme.

Exemple 1. La fonction identité est définie par le λ-terme suivant : $\lambda x.x$

On peut constater avec cette syntaxe que les abstractions ne prennent qu'un seul argument. Cependant, l'ensemble des fonctions à plusieurs arguments peut être représenté, par curryfication, à l'aide d'une succession de fonctions à un seul argument.

Exemple 2. La fonction $\lambda(x, y).t$, qui a deux arguments x et y, correspond au λ-terme $\lambda x.\lambda y.t$

2.1.1 Variables libres et variables liées

Une variable est *liée* lorsque celle-ci est déclarée dans le corps d'une abstraction. Dans l'Exemple 2, x était ainsi une variable liée. Si la variable n'a pas été déclarée, elle est dite *libre*. Formellement, l'ensemble des variables libres d'un terme est défini récursivement de la façon suivante :

$$\begin{aligned} \text{isFree}(x) &\triangleq \{x\} \\ \text{isFree}(\lambda x.t) &\triangleq \text{isFree}(t) \backslash \{x\} \\ \text{isFree}(f\,s) &\triangleq \text{isFree}(f) \cup \text{isFree}(s) \end{aligned}$$

Comme les abstractions lient les variables, elles sont communéments appelées *lieurs*.

Exemple 3. Dans le terme $\lambda x.\lambda y.x\ y\ z$, les variables x et y sont liées tandis que la variable z est libre .

Un terme ne contenant aucune occurrence de variable *libre* est dit *clos*.

Exemple 4. Dans le λ-terme $(\lambda x.\lambda y.x\ y)\ y$, la variable x ainsi que la première occurence de la variable y sont liées. Cependant la seconde occurence de variable y est libre.

2.1.2 α-équivalence

Le problème avec cette représentation est que les termes $\lambda x.x$ et $\lambda y.y$ sont *à priori* distincts du fait des noms de variable utilisés. Or cette distinction n'est pas souhaitable : ces deux termes représentent de façon équivalente la fonction identité, indépendamment du choix de nom de variable. Nous introduisons donc la notion d'α-équivalence.

On considère que deux termes sont α-équivalents si et seulement si ils sont égaux au renommage de leurs variables liées près.

Exemple 5. Voici deux termes α-équivalents

$$\lambda x.x \equiv_\alpha \lambda y.y$$

2.1.3 Représentation concrète

Pour pallier à la remarque 2.1.2, notre implémentation utilise une représentation particulière pour les variables liées : les indices de De Bruijn. Nous allons ainsi représenter les variables liées non pas par un nom mais par un entier naturel. Cette valeur est déterminée à partir du nombre d'abstractions entre la variable et le lieur qui l'introduit.

Exemple 6. Le λ-terme représentant l'identité, que l'on écrivait $\lambda x.x$, est représenté par $\lambda.0$ et le terme $\lambda x.\lambda y.x$ est représenté par $\lambda.\lambda.1$

Tandis que les variables liées seront representées par des indices de De Bruijn, les variables libres quant à elles seront représentées par un (unique) nom de variable, comme dans la présentation formelle. Une telle représentation est traditionnellement appelée "locally nameless" [Charguéraud, 2011]

Exemple 7. Soit x une variable libre. Cela manque d'explication : je ne comprends pas cet exemple, qu'est-ce que tu veux montrer ? En théorie : $\lambda x.(\lambda y.\lambda z.y)x) \rightsquigarrow \lambda x.\lambda z.x$

En pratique avec les indices de De Bruijn : $\lambda.((\lambda.\lambda.1)\ 0) \rightsquigarrow \lambda.\lambda.0$

Ici on constate bien que le terme obtenu n'est pas correct. Il nous faut introduire une opération intermédiaire afin de remédier à ce problème. Cette opération est appelée par Pierce les "shifts"

En pratique avec du locally nameless : $\lambda.((\lambda.\lambda.1)\ x) \rightsquigarrow \lambda.\lambda.x$

Il n'est plus nécessaire d'effectuer des "shifts", cette représentation possède les avantages de la représentation avec les indices de De Bruijn et s'affranchit du problème de la valeur des variables libres.

Voici le code Ocaml traduisant cette définition inductive des termes :

```
type lambda_term =
  | FreeVar of string
  | BoundVar of int
  | Abs of string × lambda_term
  | Appl of lambda_term × lambda_term
```

Cette représentation se rapproche au plus de la spécification formelle : seule la distinction entre les variables liées (`BoundVar`) et les variables libres (`FreeVar`) diffère. On voulais mettre les `name` dans `FreeVar`

2.2 Calculer dans le λ-calcul

Dans le λ-calcul pur, il n'existe pas d'opérations primitives, comme l'addition ou la multiplication d'entiers par exemple. Le seul moyen de calculer de nouveaux termes est de les appliquer entre eux. Mais il nous faut tout d'abord définir le mécanisme de substitution par lequel se réalise la notion de calcul.

2.2.1 Substitution

Sérieusement, il reste des "lambda" ?! Il est temps de rencontrer query-search-and-replace. La substitution est un mécanisme central dans le lambda calcul. Cela consiste à remplacer l'occurrence d'une variable liée par un λ-terme. On note $t[x := u]$ la substitution de la variable x dans le terme t par le terme u.

Formellement, la substitution est définie par récursion sur le λ-terme :

$$\begin{aligned}
x[x := u] &\triangleq u \\
x[y := u] &\triangleq x && (x \neq y) \\
\lambda x.t[x := u] &\triangleq \lambda x.t \\
\lambda y.t[x := u] &\triangleq \lambda y.(t[x := u]) && (x \neq y) \\
(t\ v)[x := u] &\triangleq t[x := u]\ v[x := u]
\end{aligned}$$

Exemple 8. Considérons le terme $(\lambda z.x)[x := \lambda y.y]$. Après substitution, nous obtenons le terme $\lambda z.\lambda y.y$.

Implémentation : La substitution de la variable liée `var` par le terme `tsub` dans `term` est implémentée par la fonction suivante :

```
let rec substitution term var tsub
    = match term with
    | FreeVar v → FreeVar v
    | BoundVar v when v = var → tsub
    | BoundVar v → BoundVar v
    | Abs (va,x) → Abs (va,(substitution x (var+1) tsub))
    | Appl (x,y) → Appl (substitution x var tsub,
                         substitution y var tsub)
```

L'argument `var` compte le nombre de lieurs traversés. Cette valeur correspond donc à la valeur de la variable liée à substituer. C'est pour cette raison que dans le cas où `term` est une variable liée d'indice `x` on teste l'égalité entre `var` et `x`. Le cas échéant, on effectue la substitution.

Exemple 9. Soit le terme t = $\lambda y.x\,y$ où x est liée à l'abstraction la plus proche (indice 0).

En OCaml, ce terme est représenté par

```
Abs ("y", Appl (BoundVar 1, BoundVar 0))
```

Voici le déroulement de l'appel à la fonction de substitution sur x :

```
    substitution t 0 (FreeVar "z")
→   substitution (Abs ("y", Appl (BoundVar 1, BoundVar 0))) 0 (FreeVar "z")
→   Abs("y", substitution (Appl (BoundVar 1,BoundVar 0)) 1 (FreeVar "z"))
→   Abs("y", Appl(substitution (BoundVar 1) 1 (FreeVar "z"),
                  substitution (BoundVar 0) 1 (FreeVar "z")))
→   Abs("y", Appl(FreeVar "z",
                  substitution (BoundVar 0) 1 (FreeVar "z")))
→   Abs("y", Appl(FreeVar "z", BoundVar 0))
```

Nous obtenons donc le terme attendu, soit $\lambda y.z\,y$.

2.2.2 β-réduction

Seules les applications de la forme $(\lambda x.t)\ u$, dont le membre de gauche est une abstraction, peuvent être réduites : un tel terme s'appelle un *redex* [Krivine, 1993]. Le terme $t[x := u]$, obtenu après réduction, est appelé son *contracté*.

Exemple 10. Le terme $(\lambda x.x)\ u$ est un redex tandis que le terme $x\ u$ n'en est pas un.

Formellement, on définit la β-réduction par induction sur les termes :

$$\frac{}{(\lambda x.t)\ u \rightsquigarrow t[x := u]}$$

$$\frac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'}$$

$$\frac{t \rightsquigarrow t'}{t\ u \rightsquigarrow t'\ u}$$

$$\frac{u \rightsquigarrow u'}{t\ u \rightsquigarrow t\ u'}$$

On obtient la β-conversion en prenant la clôture réflexive et transitive $t \rightsquigarrow^* t'$ de la β-réduction : on a $t \rightsquigarrow^* t'$ s'il existe une suite de β-réductions telles que $t \rightsquigarrow \ldots \rightsquigarrow t'$.

Exemple 11. Soit le terme $(\lambda x.\lambda y.y\ x)\ z$. Ce terme contient un redex : il est composé d'une application ayant une abstraction comme membre de gauche. La réduction consiste à substituer la variable liée dans la première abstraction par le membre droit de l'application. Son contracté est $\lambda y.y\ z$

Exemple 12. Voici un autre exemple de réduction

$$(\lambda x.\lambda y.y)\ w \rightsquigarrow \lambda y.y$$

x n'étant pas utilisée dans la fonction, l'argument w disparaît complètement.

2.2.3 Stratégie de réduction : appel par nom

Un terme est dit en *forme normale* si on ne peut plus lui appliquer de réduction : il ne contient aucun redex. La *normalisation* consiste à reduire autant que possible notre terme. En particulier, on caractérise inductivement les formes normales de la façon suivante :

- Une variable x est en forme normale ;
- Si t est en forme normale, alors $\lambda x.t$ est en forme normale ;
- Si t et u sont en forme normale et que u n'est pas une abstraction, alors $t\ u$ est en forme normale.

On en déduit qu'un terme est en forme normale si et seulement s'il est de la forme $\lambda x_0. \ldots \lambda x_k. x_i\, t_0\ \ldots\ t_l$ où les t_j sont eux-mêmes des termes normaux.

Exemple 13 (Terme en forme normale)**.** Il faut écrire un petit quelque chose ici, sinon cela tombe à plat. $\lambda x.x$

On dira qu'un terme t est *normalisable* s'il existe un terme normal t' tel que $t \rightsquigarrow^* t'$. Un terme t est *fortement normalisable* s'il n'existe aucune suite infinie de réduction à partir de t. Un terme fortement normalisable est, *à fortiori*, un terme normalisable.

Exemple 14. Le terme suivant n'admet pas de forme normale :

$$\begin{aligned}\omega &\triangleq (\lambda x.x\ x)(\lambda x.x\ x)\\ &\rightsquigarrow (xx)[x \mapsto \lambda x.x\ x]\\ &\rightsquigarrow (\lambda x.x\ x)(\lambda x.x\ x)\\ &\rightsquigarrow \ldots\end{aligned}$$

Exemple 15 (Terme normalisable mais pas fortement normalisable)**.** $(\lambda x.0)\,\omega$
Il faut écrire un petit quelque chose ici, sinon cela tombe à plat. En particulier, il faut dire que si l'on commence par réduire ω, cela diverge. Si l'on commence par le redex le plus à gauche, cela normalise.

Un terme normalisable (mais pas *fortement* normalisable) pourra donc diverger ou retourner une forme normale selon les réductions effectuées lors de la conversion : l'ordre dans lequel sont effectuées ces réductions est donc important. Nous considérons ici une *stratégie* de réduction qui, pour tout terme normalisable, est assurée d'obtenir une forme normale : il s'agit de la réduction en *appel par nom*.

On remarque tout d'abord que tout terme t du λ-calcul s'écrit sous la forme

$$\lambda x_0.\lambda x_m.k\, t_0\ \ldots\ t_n$$

où k est soit une variable soit un redex (le *redex de tête*) tandis que les t_i sont des termes quelconques (et donc eux-même de cette forme).

Un terme est en *forme normale de tête* lorsque k est nécessairement une variable : le redex le plus à gauche du terme est "bloqué" sur une variable et ne peut donc réduire.

La stratégie d'*appel par nom* consiste à contracter successivement le redex de tête (et uniquement ce redex).

$$\frac{}{(\lambda x.t)\ u \rightsquigarrow_N t[x := u]}$$

$$\frac{t \rightsquigarrow_N t'}{t\ u \rightsquigarrow_N t'\ u}$$

$$\frac{}{\lambda x.t \rightsquigarrow \lambda x.t}$$

$$\frac{}{x \rightsquigarrow x}$$

Crucialement, nous avons la propriété suivante [Krivine, 1993, th.3, p.62] :

Proposition 1. *Un terme t normalise vers une forme normale de tête si et seulement si la stratégie d'appel par nom termine (donnant un terme en forme normale de tête).*

Il existe d'autres stratégies de réduction comme par exemple *l'appel par valeur*, qu'implémente OCaml. Bien que plus efficace, l'appel par valeur diverge pour certains termes normalisant (Exemple 15). Dans le contexte du λ-calcul non typé, où les termes ne sont pas tous fortement normalisables, on privilégiera la réduction en appel par nom.

Exemple 16.

$$\begin{aligned}(((\lambda x.x)\ (\lambda y.y))\ z) &\rightsquigarrow ((\lambda y.y)\ z)\\ &\rightsquigarrow z\end{aligned}$$

Ici z n'est plus réductible, c'est donc la forme normale du terme.

Exemple 17. Voici une réduction en appel par nom :

$$\begin{aligned}&(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.(\lambda x.x)\ z))\\ \rightarrow&(\lambda y.y)\ (\lambda z.(\lambda x.x)\ z)\\ \rightarrow&\lambda z.(\lambda x.x)\ z\end{aligned}$$

Le terme obtenu est en forme normale de tête.

Dans les sections suivantes, nous verrons lorsque nous introduirons les systèmes de type que l'on peut statiquement garantir que les termes acceptés par l'analyse de type sont fortement normalisants.

Remarque 1. Le fait d'obtenir pas français : la garantie que l'ensemble de nos termes sont fortement normalisants signifie que notre langage n'est plus Turing complet. En effet, nous ne pourrons plus exprimer les programmes qui ne terminent pas. Cependant, ce n'est pas gênant car nous ne cherchons pas à écrire ce genre de programmes dans notre système.

Implémentation : La fonction de β-réduction consiste donc à contracter une application dont le membre de gauche est une abstraction. Le cas échéant, l'opération échoue. Pour ce faire, nous appelons la fonction de substitution pour substituer la première variable (d'indice 0) par l'argument dans le corps de la fonction :

```
let beta t
    = match t with
    | Appl(Abs(var,x),y) → Some (substitution x 0 y)
    | _ → None
```

Les seuls termes pouvant se réduire étant les redex, nous effectuons la substitution sur celle-ci, sinon le terme est retourné sans modification. Passons maintenant à l'évaluation :il y a un problème dans le code ci-dessous

```
let evaluation env t
    =
  (×=evaluation_sig ×)
  let rec eval t =
    match t with

    | x →
       try_reduction x

  and try_reduction t =
    match reduction env t with
    | Some t' → eval  t'
    | None → t
  in
  eval t
```

La fonction `eval` a pour but de détecter les applications et d'en évaluer le membre de gauche. Une fois le membre de gauche évalué, nous pouvons tenter de réduire l'ensemble de l'application.

2.3 Encodages à la Church

Bien que le λ-calcul soit un langage minimaliste, Church [1936] a montré que l'on peut y *encoder* de nombreux types de données familiers aux développeurs, tels que les entiers et les booléens.

2.3.1 Les booléens de Church

Nous allons voir une représentation des booléens accompagnée de la structure de contrôle `if ... then ... else ...`. Voici les termes correspondants aux

constructeurs *true*, *false* et *ifte* :

$$\begin{aligned} \mathit{true} &\triangleq \lambda x.\lambda y.y \\ \mathit{false} &\triangleq \lambda x.\lambda y.x \\ \mathit{ifte} &\triangleq \lambda l.\lambda m.\lambda n.l\ m\ n \end{aligned}$$

On assimile ces deux termes aux booléens *true* et *false* car leur calcul au sein de l'application d'un *ifte* reproduit le comportement attendu.

Exemple 18. Voici un exemple de réduction du terme *ifte* appliqué à la condition *false* :

$$\begin{aligned} \mathit{ifte}\ \mathit{false}\ v\ w &\triangleq (\lambda l.\lambda m.\lambda n.l\ m\ n\ )(\lambda x.\lambda y.y\ )v\ w \\ &\to (\lambda m.\lambda n.(\ \lambda x.\lambda y.y)m\ n\ )v\ w \\ &\to (\lambda n.(\ \lambda x.\lambda y.y)v\ n)\ w \\ &\to (\ \lambda x.\lambda y.y)v\ w) \\ &\to (\ \lambda y.y)w) \\ &\to w \end{aligned}$$

Ici nous n'avons aucun moyen de vérifier avant normalisation que le premier argument de *ifte* est effectivement un booléen : nous ne pouvons donc pas garantir la bonne évaluation de notre terme.

Remarque 2. Les langages orientés objets permettent de représenter ce type d'encodage d'une manière particulière. Voici la représentation avec le langage Java :

```
Interface Bool<A>{
  A ifte(A x,A y);
}

class True<A> implements Bool<A>{
  A ifte(A x,A y){
    return x;
  }
}
class False<A> implements Bool<A>{
  A ifte(A x,A y){
    return y;
  }
}
```

2.3.2 Les entiers de Church

Ici, nous allons voir comment créer les entiers naturels *ex nihilo* en utilisant un encodage à la Church. L'idée consiste à représenter le nombre n par une

fonction d'ordre supérieur prenant en argument une fonction f et l'appliquant n fois à un argument x. On définira donc

$$\begin{aligned} \mathit{zero} &\triangleq \lambda f.\lambda x.x \\ \mathit{un} &\triangleq \lambda f.\lambda x.f\ x \\ \mathit{deux} &\triangleq \lambda f.\lambda x.f\ (f\ x) \end{aligned}$$

Afin de construire tous les entiers naturels, on s'inspire alors de la définition des entiers de Peano et on définit le successeur d'un nombre n comme

$$\mathit{successeur} \triangleq \lambda n.\lambda f.\lambda x.n\ f\ (f\ x)$$

c'est-à-dire n applications de f précédées d'une première application de f, soit $n+1$ applications.

De la même manière, on construit l'addition de deux nombres m et n en faisant m applications répétées de f précédées par n applications :

$$\mathit{plus} \triangleq \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$$

2.4 Extensions

Nous nous attardons ici sur quelques choix d'implémentation et il sera question d'analyser certaines parties du code et non sa totalité.

2.4.1 Les booléens

Afin d'implémenter les booléens dans notre langage, une possibilité consiste à exporter l'encodage à la Church des termes *true*, *false* et *ifte* définis en Section 2.3.1. Cependant nous avons choisi d'enrichir le noyau de notre langage afin de supporter, de façon primitive, les booléens. Cela permet d'alléger l'écriture des termes mais surtout de faciliter la représentation et, par la suite, la compilation des programmes manipulant des expressions booléennes.

Nous allons donc étendre la grammaire de nos λ-termes

$$\begin{array}{rcll} t & ::= & (\ldots) & \text{(lambda terme)} \\ & | & \mathsf{true} & \text{(true)} \\ & | & \mathsf{false} & \text{(false)} \\ & | & \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t & \text{(ifte)} \end{array}$$

ainsi que notre implémentation

```
type lambda_term = (...)
  | True | False
  | IfThenElse of lambda_term × lambda_term × lambda_term
```

Pour la substitution des constructeurs `True` et `False`, les règles sont triviales. Pour le terme `IfThenElse`, il faut rappeler la fonction de substitution sur l'ensemble des arguments du constructeur. En particulier, il n'est pas nécessaire

d'incrémenter la valeur de `var` étant donné que ce constructeur ne lie aucune variable.

La réduction est étendue au-delà de la β-réduction par une relation de ι-réduction, correspondant à la réduction du test conditionnel $\mathsf{if} \ldots \mathsf{then} \ldots \mathsf{else} \ldots$ face aux booléens true ou false :

$$\frac{}{\mathsf{if\ true\ then}\ u\ \mathsf{else}\ v \rightsquigarrow u} \qquad \frac{}{\mathsf{if\ false\ then}\ u\ \mathsf{else}\ v \rightsquigarrow v}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{if}\ t\ \mathsf{then}\ u\ \mathsf{else}\ v \rightsquigarrow \mathsf{if}\ t'\ \mathsf{then}\ u\ \mathsf{else}\ v}$$

L'implémentation de cette réduction ne présente pas de difficulté particulière :

```
let iota t
  = match t with

(...)

  | IfThenElse (True, y, z) → Some y
  | IfThenElse (False, y, z) → Some z
```

2.4.2 Les entiers

Tout comme pour les booléens, il nous faut enrichir la grammaire formelle du λ-calcul

$$\begin{array}{rcll} t & ::= & (\ldots) & (\lambda\text{-terme}) \\ & | & \mathsf{zero} & (\text{zero}) \\ & | & \mathsf{succ}\ t & (\text{succ}) \\ & | & \mathsf{iter}\ t\ t\ t & (\text{iter}) \end{array}$$

ainsi que sa réalisation dans l'implémentation

```
type lambda_term = (...)
  | Zero | Succ of lambda_term
  | Iter of lambda_term × lambda_term × lambda_term
```

Ces trois constructeurs n'étant pas des lieurs, l'implémentation de la substitution est triviale.

La ι-réduction de l'itération se définit ainsi :

$$\frac{}{\mathsf{iter}\ (\mathsf{succ}\ n)\ f\ a \rightsquigarrow \mathsf{iter}\ n\ f\ (f\ a)} \qquad \frac{}{\mathsf{iter\ zero}\ f\ a \rightsquigarrow a}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{iter}\ t\ f\ a \rightsquigarrow \mathsf{iter}\ t'\ f\ a}$$

L'implémentation de l'évaluation suit cette spécification :

```
let iota t
  = match t with

(...)

    | Iter(Zero,f,a) → Some a
    | Iter(Succ num, f, a) → Some (Iter(num,f,(Appl(f, a))))
```

Nous rappelons donc la fonction avec cette fois-ci `(f a)` et le prédécesseur de `n`.

Ici la ι-réduction n'effectue que les étapes de calcul d'`iter` sans effectuer l'évaluation des applications. En effet cela n'est pas le rôle de la ι-réduction mais de la β-réduction de réduire les applications.

2.4.3 Les paires

La paire est une construction très répandue dans les langages fonctionnels, cela nous permet de créer un ensemble de deux éléments. Afin d'en extraire les éléments, nous utiliserons les projections π_0 et π_1. Celles-ci nous permettent d'obtenir respectivement le premier ou le second élément de la paire.

$$\begin{array}{rcll} t & ::= & (\ldots) & \text{(lambda terme)} \\ & | & (t,t) & \text{(pair)} \\ & | & t.\pi_0 & (\pi_0) \\ & | & t.\pi_1 & (\pi_1) \end{array}$$

Ce qui se traduit dans notre programme par l'ajout des termes suivants :

```
type lambda_term = (...)
  | Pair of lambda_term × lambda_term
  | Pi0 of lambda_term
  | Pi1 of lambda_term
```

La substitution s'implémente trivialement car aucun de ces constructeurs n'est un lieur. La ι-réduction est étendue aux projections de la façon suivante :

$$(u,v).\pi_0 \rightsquigarrow u \qquad (u,v).\pi_1 \rightsquigarrow v$$

$$\frac{t \rightsquigarrow t'}{t.\pi_0 \rightsquigarrow t'.\pi_0} \qquad \frac{t \rightsquigarrow t'}{t.\pi_1 \rightsquigarrow t'.\pi_1}$$

Implémentons donc ces règles :

```
let iota t
  = match t with

(...)

    |Pi0(Pair(x,y)) → Some x
    |Pi1(Pair(x,y)) → Some y
```

L'évaluation des membres de la paire s'effectue dans la fonction `eval` :

```
  let rec eval t =
    match t with
```

```
(...)
    | Pair(x,y) → Pair((try_reduction x),(try_reduction y))
```

3 Le λ-calcul simplement typé

Nous allons maintenant enrichir notre λ-calcul avec des types. Cela va nous donner des propriétés quant à l'évaluation de nos termes et leur utilisation. Nous verrons que dans le λ-calcul simplement typé il est impossible d'écrire des termes qui ne soient pas fortement normalisants, comme dans l'Exemple 14. Dans un premier temps nous introduirons les types, puis nous verrons comment les utiliser en définissant des règles de typage pour nos termes.

3.1 Les types

Il est important avant tout de définir ce qu'est un type. Les types que nous considérons dans cette partie sont décrits par la grammaire suivante :

$$\begin{array}{rcll} T & ::= & & \text{(type)} \\ & | & \mathsf{int} & \text{(entiers)} \\ & | & \mathsf{bool} & \text{(booléens)} \\ & | & T \to T & \text{(fonction)} \\ & | & T \times T & \text{(produit)} \end{array}$$

Exemple 19. Voici des exemples de type :

$$\mathsf{int} \to \mathsf{bool} \tag{1}$$
$$\mathsf{bool} \times \mathsf{int} \tag{2}$$

Le type (1) est celui d'une fonction prenant un argument de type entier et retournant un résultat de type booléen. Le second type (2) correspond à une paire donc le membre de gauche est un booléen et le membre de droite un entier.

Implémentation : La spécification des types se traduit naturellement en OCaml :

```
type typ =
  | Bool
  | Nat
  | Fleche of typ × typ
  | Croix of typ × typ
```

$$\frac{}{\Gamma, x : T \in \Gamma}\ (\textsc{Var})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t \ : \ A \to B}\ (\textsc{Abs}) \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash f\ s : B}\ (\textsc{App})$$

FIGURE 1 – λ-calcul simplement typé

3.2 Système de type

Un système de type est un système formel permettant de vérifier que l'ensemble de nos termes sont correctement typés. Il sera question de définir un ensemble de règles pour chacun des termes de notre langage. Le typage nous apportera certaines propriétés quant à la normalisation de nos termes comme nous le verrons dans la Section 3.3.1.

3.2.1 Spécification

Afin de faciliter la lecture et la compréhension, nous allons dans un premier temps donner les règles de typage pour les termes appartenant au noyau du langage. Nous donnerons les autres règles dans les sections suivantes.

La vérification de type s'effectue dans un *contexte* qui assigne à chaque variable d'un programme son type. Le contexte est donc une liste ordonnée de paires variable/type :

$$\begin{array}{lll} \Gamma & ::= & \text{(contexte)} \\ & \mid \quad . & \text{(contexte vide)} \\ & \mid \quad \Gamma, x : T & \text{(type de variable)} \end{array}$$

Les règles de typage sont définies en Figure 1. La règle (VAR) spécifie le typage des variables. Cette expression se lit en partant du numérateur pour déduire le dénominateur : supposons que la variable x de type T soit présente dans le contexte, alors on conclut que le type de la variable x est T. La règle (ABS) spécifie le typage des λ-abstractions. L'abstraction construit un type fonction de la forme $A \Rightarrow B$: il faut donc que la variable associée au lambda terme soit de type A et le résultat de type B. La règle (APP) spécifie le typage de l'application. Dans un langage de programmation, appliquer un terme n'étant pas une fonction n'a pas de sens. On s'assure donc que le membre gauche de notre application est bien de type $A \to B$, pour A et B quelconque. De plus on vérifie que l'argument de la fonction est bien de type A, correspondant au domaine définition de la fonction.

Exemple 20. Voici donc quelques exemples de dérivations de type. Ces dérivations se lisent du bas vers le haut.

$$\dfrac{\dfrac{\dfrac{x : \mathsf{int} \to \mathsf{bool} \in \Gamma}{\Gamma, x : \mathsf{int} \to \mathsf{bool}, y : \mathsf{int} \vdash x : \mathsf{int} \to \mathsf{bool}} \quad \dfrac{y : \mathsf{int} \in \Gamma}{\Gamma, x : \mathsf{int} \to \mathsf{bool}, y : \mathsf{int} \vdash y : \mathsf{int}}}{\Gamma, x : \mathsf{int} \to \mathsf{bool}, y : \mathsf{int} \vdash x\ y : \mathsf{bool}}}{\Gamma, x : \mathsf{int} \to \mathsf{bool} \vdash \lambda y.x\ y : \mathsf{int} \to \mathsf{bool}}}{\Gamma \vdash \lambda x.\lambda y.x\ y : (\mathsf{int} \to \mathsf{bool}) \to \mathsf{int} \to \mathsf{bool}}$$

Ce terme est donc correctement typé.

Le système de type tel que présenté est une *spécification* : afin de l'implémenter, il nous faut effectuer des choix laissés libres par la spécification. Par exemple, la dérivation

$$\dfrac{\dfrac{\Gamma, x :?_A \vdash \mathsf{true} : \mathsf{bool}}{\Gamma \vdash (\lambda x.\mathsf{true}) :?_A \to \mathsf{bool}} \quad \Gamma \vdash \lambda y.y :?_A}{\Gamma \vdash (\lambda x.\mathsf{true})\ (\lambda y.y) : \mathsf{bool}}$$

admet une infinité de choix possibles pour le type $?_A$: on pourra choisir $?_A = \mathsf{int}$ ou encore $?_A = \mathsf{bool} \times (\mathsf{int} \to \mathsf{int})$.

Afin d'obtenir un algorithme, la règle typant l'application introduit donc une difficulté : il faut vérifier que le membre de gauche est de type $A \to B$ alors que l'on ne connait que B. Nous allons donc introduire une nouvelle représentation des termes afin de déterminer la spécification et obtenir un algorithme.

3.2.2 Syntaxe bidirectionnelle

Les applications dont le membre de gauche est une abstraction, posent problème car il faut leur calculer un type sans nécessairement avoir cette information à disposition immédiate. Nous allons donc partitionner les λ-termes en deux catégories : d'une part, les termes dont on peut automatiquement synthétiser le type (que l'on peut donc placer à gauche d'une application) et ceux pour lesquels il faudra fournir une annotation de type (annotation que l'on saura vérifier efficacement). Nous nommerons les premiers les termes *synthétisables* et les seconds les termes *vérifiables*.

Voici donc notre représentation des termes :

$$\begin{array}{lcll}
ex & ::= & & \text{(termes synthétisables)} \\
 & | & ex\ in & \text{(application)} \\
 & | & x & \text{(variable)} \\
 & | & (in\ :\ T) & \text{(annotation)} \\
in & ::= & & \text{(termes vérifiables)} \\
 & | & \lambda x.in & \text{(abstraction)} \\
 & | & \mathsf{inv}(ex) & \text{(inversion)}
\end{array}$$

On a ainsi introduit l'inversion $\mathsf{inv}(ex)$, qui permet d'inclure les termes synthétisables dans le monde des termes vérifiables, et l'annotation $(in\ :\ T)$, qui inclut les termes vérifiables dans le monde des termes synthétisables en fournissant l'information de typage manquante.

$$\boxed{\Gamma \vdash T \ni in} \qquad\qquad \boxed{\Gamma \vdash ex \in T}$$

$$\text{(Var)}\ \frac{x : T \in \Gamma}{\Gamma \vdash x \in T}$$

$$\text{(Abs)}\ \frac{T = A \to B \qquad \Gamma, x : A \vdash B \ni t}{\Gamma \vdash T \ni \lambda x.t} \qquad \text{(App)}\ \frac{\Gamma \vdash f \in A \to B \quad \Gamma \vdash A \ni s}{\Gamma \vdash f\ s \in B}$$

$$\text{(Inv)}\ \frac{\Gamma \vdash t \in T' \ \ T = T'}{\Gamma \vdash T \ni \mathsf{inv}(t)} \qquad \text{(Ann)}\ \frac{\Gamma \vdash T \ni t}{\Gamma \vdash (t\ :\ T) \in T}$$

FIGURE 2 – Lambda calcul bidirectionnel simplement typé

Implémentation : La représentation des termes bidirectionnels est sans surprise :

```
type inTm =
  | Abs of name × inTm
  | Inv of exTm

and exTm =
  | FVar of name
  | BVar of int
  | Appl of exTm × inTm
```

Le type `name` ci-dessous nous permettra de mettre des entiers à la place d'une chaîne de caractère, nous en reparlerons dans l'implémentation de l'évaluation.

```
type name =
  | Global of string
  | Bound of int
  | Quote of int
```

3.2.3 Système de type bidirectionnel

À partir de cette nouvelle présentation de la syntaxe, on adapte aisément les règles de typage (Figure 2), qui sont désormais entièrement dirigées par la syntaxe et donc purement algorithmique.

Les règles de la forme $\Gamma \vdash T \ni t$ sont des règles de vérification, celles de la forme $\Gamma \vdash t \in T$ des règles de synthèse. Lors de la vérification, on connaît le type T et l'on souhaite déterminer si le terme t est bien de ce type. Lors de la synthèse, on est en mesure, à partir d'un terme t quelconque, de déterminer son type T sans ambiguïté. On vérifie aisément que ce système de type bidirectionnel est équivalent à celui présenté en Figure 1.

Exemple 21. Voici maintenant la même dérivation mais avec notre système de type bidirectionnel :

$$\dfrac{\dfrac{\Gamma, x : \mathsf{int} \to \mathsf{int} \vdash \mathsf{true} : \mathsf{bool}}{\Gamma \vdash (\lambda x.\mathsf{true} \ : \ (\mathsf{int} \to \mathsf{int}) \to \mathsf{bool}) \in (\mathsf{int} \to \mathsf{int}) \to \mathsf{bool}} \quad \dfrac{\Gamma, y : \mathsf{int} \vdash \mathsf{int} \ni y}{\Gamma \vdash \mathsf{int} \to \mathsf{int} \ni \lambda y.y}}{\Gamma \vdash ((\lambda x.\mathsf{true}) \ : \ (\mathsf{int} \to \mathsf{int}) \to \mathsf{bool}) \ (\lambda y.y) \in \mathsf{bool}}$$

Implémentation : Encore des cfracs pourris ? Sérieusement ?!

- $\boxed{\boxed{\Gamma \vdash T \ni in}}$

  Nous disposons d'une fonction de vérification `check` permettant de vérifier que le terme `inT` est bien de type `ty`. Le `contexte` est une liste qui nous permet de stocker des paires associant une variable à son type. Il nous faut aussi donner le type d'entrée `ty`.

```
let rec check contexte ty inT
    = match inT with
```

- $\boxed{\text{Abs} \ \dfrac{T = A \to B \qquad \Gamma, x : A \vdash B \ni t}{\Gamma \vdash T \ni \lambda x.t}}$

  Lorsque que l'on veut vérifier le terme contenu dans une abstraction il faut libérer l'ensemble des variables qui lui sont associées. Avec notre représentation "locally nameless" nous somme obligé d'effectuer une substitution.
  C'est pour cette raison que l'on génère un nouveau nom à l'aide de la fonction `gensym` qui nous garantie que ce nom n'a jamais été utilisé. On substitue l'ensemble des variables liées à l'abstraction en cours avec une variable libre dont le nom est celui généré précédement. On insère ensuite dans le contexte l'association variable type.

```
| Abs(x, b) →
   begin
     match ty with
     | Fleche(s, t) →
        (× XXX: open the de Bruijn binder ×)
        let freshVar = gensym () in
        check ((Global(freshVar), s) :: contexte) t
          (substitution_inTm b (FVar (Global(freshVar))) 0)
     | _ → failwith "SAbstraction forced into a non-functional type"
   end
```

- $\boxed{\dfrac{\Gamma \vdash t \in T' \ T = T'}{\Gamma \vdash T \ni inv(t)}(\text{Inv})}$

  Afin de tester l'égalité de type, l'égalité structurelle de OCaml est suffisante puisque les types sont purement statiques (contrairement aux types dépendants qui, eux, calculeront).

```
    | Inv(t) → ty = synth contexte t
```

- $\boxed{\Gamma \vdash ex \in T}$

La fonction de synthèse `synth` prend un contexte, un terme synthétisable et renvoie le type de celui-ci.

```
and synth contexte exT
    = match exT with
```

- $\boxed{\dfrac{\Gamma \vdash T \ni t}{\Gamma \vdash (t : T) \in T}(\text{Ann})}$

Nous sommes ici en synthèse c'est à dire que nous devons retourner un type. Si t est effectivement de type T, alors il suffit de retourner T.

```
    | Ann(tm, ty) →
       if check contexte ty tm then ty
       else failwith "Wrong annotation"
```

- $\boxed{\dfrac{x : T \in \Gamma}{\Gamma \vdash x \in T}(\text{Var})}$

Dans la spécification formelle, on ne fait pas de distinction entre les variables libres et liées. Comme nous l'avons vu dans la règle de typage de l'abstraction, l'ensemble des variables liées à chacun des lieurs sont libérées. La spécification parle donc bien des variables libres, que l'on synthétise à partir du type qui leur a été assigné dans le contexte. le code contient un TODO

```
    | FVar(x) → List.assoc x contexte
```

- $\boxed{\dfrac{\Gamma \vdash f \in A \to B \quad \Gamma \vdash A \ni s}{\Gamma \vdash f\ s \in B}(\text{App})}$

Comme la spécification nous l'indique, il est d'abord nécessaire de synthétiser le type de f. Il faut ensuite s'assurer que le type retourné par la synthèse est bien de la forme $A \to B$ puis en extraire le type A pour pouvoir cette fois-ci vérifier s. Si cette vérification réussit, on retourne le type B.

```
    | Appl(f, s) →
       let fTy = synth contexte f in
       begin
         match fTy with
         | Fleche(a, b) →
            if check contexte a s then b
            else failwith "Argument type invalid"
         | _ → failwith "Function type invalid"
       end
```

Pour cette implémentation nous avons choisi de générer des exceptions lorsque le terme n'était pas correctement typé. Cela permet de terminer le programme directement car de toute façon le terme serait refusé. Nous reviendrons cependant sur certains points négatifs de cette méthode dans la section 4.4.1.

3.3 Normalisation

Dans cette section nous verrons certaines propriétés offertes par les systèmes de type, et nous parlerons de l'évaluation. Nous ne reviendrons pas sur la substitution ni la β-réduction, qui restent identique aux définitions introduites en Section 2.2.

3.3.1 Méta-théorie

Le typage permet d'obtenir, lors de la compilation, des garanties quant à la bonne exécution des programmes. Formellement, ces garanties se traduisent en deux propriétés complémentaires d'un système de type :

Progrès : si un terme t est correctement typé alors soit t est une valeur, soit il existe une réduction telle que $t \rightsquigarrow t'$

Préservation : si un terme t a le type T et que $t \rightsquigarrow t'$ alors t' est de type T

La conjonction de ces deux propriétés donne le slogan "un programme bien typé ne plante jamais" [Milner, 1978] : tout programme bien typé réduit vers une valeur de ce type.

Exemple 22. Un programme de type bool réduit vers une valeur de type bool et donc ce programme réduit *nécessairement* vers le constructeur true ou le constructeur false.

3.3.2 Réduction forte

Dans le λ-calcul non typé, nous avions utilisé une stratégie d'évaluation en appel par nom car nous n'avions pas la garantie que les termes réduits soient fortement normalisants. D'après les propriétés de la section précédente nous disposons de la certitude que les termes acceptés par le type-checker sont fortement normalisants.

De plus, les types ne sont plus utiles lors de l'évaluation. Nous allons donc implémenter une stratégie de réduction forte sur les λ-termes non typés de la Section 2. La réduction forte du λ-calcul simplement typé sera donc obtenue par plongement des termes bidirectionnels dans le fragment fortement normalisant du λ-calcul non typé. La traduction de nos termes bidirectionnels vers les termes non typés est transparente : il suffit d'effacer les annotations de type.

Exemple 23. Roman : j'ai l'exemple sur papier faut juste que le mette dans le rapport C'est l'exemple où cela ne réduit pas

Implémentation : La fonction de réduction forte est très similaire à la fonction d'évaluation précédente

```
let reduction_forte env t
  =
  let rec eval t =
    match t with
    | Appl(f, v) →
      let vf = eval f in
      let vv = eval v in
      try_reduction (Appl(vf, vv))
    | x →
      try_reduction x
  and try_reduction t =
    match reduction env t with
    | Some t' → eval  t'
    | None → t
  in
  eval t
```

Avec les garanties offertes par notre système de type nous pouvons à présent contracter *tous* les redexes d'un terme sans risque de divergence.

3.4 Extensions

Dans notre calcul non typé, nous avions introduit les booléens, les entiers ainsi que les paires, nous verrons ici la façon de les représenter ainsi que leurs règles de typage.

3.4.1 Les entiers

Les constructeurs sont les mêmes que dans le λ-calcul non typé cependant, maintenant que nous disposons de termes bidirectionnels, il est important de préciser leur phase.

succ et zero sont des termes vérifiables étant donné que leur type sera toujours *Nat*. À l'inverse, on exige que l'itérateur soit annoté par son type de retour : celui-ci est donc synthétisable. Les règles de typage bidirectionnel sont donc :

$$\text{(Zero)}\quad \frac{}{\Gamma \vdash Nat \ni \mathsf{zero}}$$

$$\text{(Succ)}\quad \frac{\Gamma \vdash Nat \ni t}{\Gamma \vdash Nat \ni \mathsf{succ}\, t} \qquad \text{(Iter)}\quad \frac{\Gamma \vdash Nat \ni n \;\; \Gamma \vdash A \ni a \;\; \Gamma \vdash A \to A \ni f}{\Gamma \vdash \mathsf{iter}\; A\; n\; f\; a \in A}$$

Les règles d'évaluations restent inchangées.

3.4.2 Les booléens

Afin de pouvoir typer nos termes, il nous faut introduire le type bool. En suivant la même démarche que pour les entiers, les constructeurs *True* et *False* sont désormais des *termes vérifiables* et le constructeur *Ifte* un *terme synthétisable*

Nous pouvons désormais typer nos booléens de la manière suivante :

$$\frac{}{\mathsf{bool} \ni \Gamma \vdash \mathsf{true}}\ (\textsc{true}) \qquad \frac{}{\mathsf{bool} \ni \Gamma \vdash \mathsf{false}}\ (\textsc{false})$$

$$\frac{\Gamma \vdash \mathsf{bool} \ni t1 \qquad \Gamma \vdash t2 \in A \qquad \Gamma \vdash t3 \in A}{\Gamma \vdash \mathsf{if}_A\ t1\ \mathsf{then}\ t2\ \mathsf{else}\ t3 \in A}\ (\textsc{If-else})$$

3.4.3 Le produit cartésien

Les projections sont des *termes synthétisables* étant donné que leur type dépend de la paire sur laquelle ils sont appliqués. La paire est un *terme vérifiable* car chaque paire à un type statique.

Règle de typages des paires :

$$\frac{\Gamma \vdash A \ni a \ \ \Gamma \vdash B \ni b}{\Gamma \vdash A \times B \ni (a, b)}\ (\textsc{Paire}) \qquad \frac{\Gamma \vdash p \in A \times B}{\Gamma \vdash p.\pi_0 \in A}\ (\pi_0)$$

$$\frac{\Gamma \vdash p \in A \times B}{\Gamma \vdash p.\pi_1 \in A}\ (\pi_1)$$

Pour les projections, il nous faut vérifier le fait que l'argument est bien de type produit afin de retourner le type du membre de gauche ou droit.

Concernant l'évaluation, les règles sont les mêmes qu'en Section 2.4.3.

Implémentation : Nous allons maintenant implémenter le produit cartésien afin d'enrichir notre langage.

L'algorithme de typage suit litéralement sa spécification bidirectionnelle :

```
let rec check contexte ty inT
    = match inT with

(...)

    | Pair(x,y) →
       begin
         match ty with
         | Croix(a,b) → if check contexte a x
                           && check contexte b y then true
                        else failwith "In Pair(x,y) x is not of type a"
```

```
          | _ → failwith "Type of a pair must be a Croix"
        end

and synth contexte exT
    = match exT with

(...)

    | P0(x) →
       begin
         match synth contexte x with
         | Croix(a,b) → a
         | _ → failwith "Po must be applied to a pair"
       end
    | P1(x) →
       begin
         match synth contexte x with
         | Croix(a,b) → b
         | _ → failwith "P1 must be applied to a pair"
       end
```

3.5 Réduction forte à grands pas

Dans cette section, nous décrivons une implémentation alternative de la réduction forte qui exploite le fait que nous implémentons notre évaluateur dans un langage fonctionnel. Il est intéressant d'exploiter les mécanismes d'Ocaml. Nous allons créer un interpréteur pour nos termes en utilisant les termes Ocaml ce qui nous permettra de ne pas avoir à gérer les mécanismes de définition de fonction et d'application.

Comme nous l'avons vu dans la Section 3.3.2 la réduction forte aboutit forcément à une forme normale, les termes que nous obtenons par la réduction à grands pas, seront donc les mêmes que ceux obtenus avec la réduction à petits pas (Section 3.3.2).

Voici donc le code correspondant à la nouvelle représentation de nos termes en Ocaml :

```
type value =
  | VLam of (value → value)
  | VNeutral of neutral

and neutral =
  | NFree of name
  | NApp of neutral × value
```

Le terme `VLam` correspond à l'abstraction. Cette représentation nous permet de reconsidérer la façon de voir la réduction d'une abstraction. En effet lorsque l'on applique une abstraction à un terme, nous avions introduit la substitution, qui consiste à remplacer les occurences des variables liées à ce terme. La transformation d'une abstraction consiste donc à créer une fonction anonyme puis à

évaluer le terme contenu dans celle-ci en ajoutant dans le contexte la variable liée à cette fonction. Lors de l'évaluation d'une variable liée, on va retrouver dans le contexte l'argument correspondant. On aura donc bien le comportement attendu lors de l'application de cette fonction anonyme car cela va remplacer les occurences de la variable.

Exemple 24. Soit le terme $\lambda x.x\ u$, avec la méthode d'évaluation à petit pas, on évalue en effectuant $x[x := u]$ ce qui nous donne comme résultat u. Avec l'évaluation à grands pas, on évalue le terme $\lambda x.x$ en la fonction OCaml `fun arg ->arg`. Cette fonction une fois appliquée à la traduction v_u de u, nous retourne v_u comme souhaité.

Comme nous l'avons vu précédement le type contenu dans une annotation n'est pas utile lors de l'évaluation. Il nous faut simplement évaluer le terme `x` contenu dans celle ci.

```
let rec eval_exTm t envi =
  match t with
    | Ann(x,_) → eval_inTm x envi
```

Il en est de même pour l'inversion :

```
and eval_inTm t envi =
  match t with
  | Inv(i) → eval_exTm i envi
```

Pour les variables liées il faut retrouver dans le contexte la variable à laquelle elle est associée. Cela nous permettra de définir cette variable comme un argument de la fonction anonyme que nous avons créée avec l'évaluation de l'abstraction.

```
    | FVar v → vfree v
    | BVar v → List.nth envi v
```

Nous avons besoin de créer des `neutral` pour tous les termes dont l'évaluation est terminée. C'est le rôle de la fonction `vfree` dont voici le code :

```
let vfree name = VNeutral(NFree name)
```

Pour évaluer l'application d'un coup, définissons une fonction `vapp` permettant d'effectuer la réduction. Nous appelons donc cette fonction avec le couple formé par l'évaluation de la fonction et son argument. Si la fonction est une abstraction, il nous suffit d'appliquer celle-ci *dans Ocaml*, étant donné qu'elle est encodée par une fonction anonyme OCaml. Sinon l'évaluation est bloquée sur un terme neutre, il faut donc créer un nouveau terme neutre contenant la fonction suspendue. Voici le cas de l'application dans notre fonction d'évaluation :

```
    | Appl(x,y) → vapp((eval_exTm x envi),(eval_inTm y envi))
```

Ainsi que le code de `vapp` :

```
and vapp v =
  match v with
```

```
  | ((VLam f),v) → f v
  | ((VNeutral n),v) → VNeutral(NApp(n,v))
  | _ → failwith "Impossible (vapp)"
```

Comme nous l'avons vu précédement, nous avons créé une `value` correspondant à l'abstraction, `VLam` prenant en argument une fonction de type `value → value`. L'évaluation de l'abstraction consiste donc à transformer celle-ci en une fonction.

```
  | Abs(x,y) → VLam(function arg → (eval_inTm y (arg :: envi)))
```

Une fois nos termes évalués dans OCaml, il s'agit de les *réifier* sous forme de termes de notre λ-calcul. Cette fonction doit nécessairement prendre un argument de typevalue afin de nous retourner un *terme synthétisable*.

```
let rec value_to_inTm i v =
  match v with
  | VLam(f) → let var = gensym2 () in
              begin
                Abs(Global(var),(value_to_inTm (i+1) (f(vfree(Quote i)))))
              end
  | VNeutral(x) → Inv(neutral_to_exTm i x)

and neutral_to_exTm i v =
  match v with
  | NFree x → boundfree i x
  | NApp(n,x) → Appl((neutral_to_exTm i n),(value_to_inTm i x))
```

Le type `name` nous permet de faire la distinction entre les variables réellement libres et celles que nous souhaitons lier pendant la réification. Afin de transformer une valeur `VLam f` en *Abs* il nous faut appliquer la fonction f à une variable de réification, que l'on distinguera des autres variables libres grâce au type `name`. On s'assure que les variables ainsi introduites sont fraîches en utilisant la fonction `gensym`. La fonction `boundfree` va nous permettre de relier les `NFree` dont la variable est un `Quote`.

```
let boundfree i n =
  match n with
  | Quote k → BVar (i - k - 1)
  | x → FVar x
```

4 Les types dépendants

Maintenant que les bases du λ-calcul et du typage on été présentés, nous allons découvrir les types dépendants.

4.1 Automatisation de démonstration de preuve

Durant les deux premières parties du rapport nous avons présenté le λ-calcul qui nous a permis de nous initier à la conception et à l'implémentation d'un

système de type. Cependant nous avions une vision calculatoire des types. En effet notre seul but était de nous assurer que nos λ-termes étaient fortement normalisants. Un des aspects primordial des systèmes de types est la correspondance de Curry-Howard. Celle-ci permet de d'établir des relations entre la logique propositionnelle et les systèmes de type. Nous verrons que "t est une preuve de P" est équivalent à $t : P$ c'est à dire t est de type P. Tout d'abord intéressons nous aux bases de la logique propositionnelle.

4.1.1 La logique propositionnelle

Les règles de la logique propositionnelle associent à chaque connecteur logique une *règle d'introduction* qui permet de formuler une conclusion à partir de certaines hypothèses. Par exemple, la règle d'introduction de l'implication est

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

Les connecteurs logiques sont éliminés à travers une *règle d'élimination*, qui se traduit calculatoirement par des règles de réduction. Par exemple, la règle d'élimination de l'implication est

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

On constate que la règle d'introduction de l'implication est presque identique à celle du typage de l'abstraction. De même pour l'élimination vis à vis de la règle de l'application. C'est cette intuition qu'ont eu Haskell Curry et Alvin Howard ce qui à permis d'établir la correspondance entre les systèmes de type et la logique propositionnelle.

Regardons maintenant un second exemple, le constructeur $\wedge$ signifiant le *et* logique. La règle d'introduction énonce que pour prouver $A \wedge B$, il nous faut prouver d'une part A et d'autre part B :

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma A \wedge B}$$

Si l'on désire prouver une proposition A et que l'on suppose $A \wedge B$ alors celle-ci est vraie. Ce raisonnement est aussi valable pour une propriété B.

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

On reconnaît ici le produit cartésien que l'on avait introduit dans notre λ-calcul simplement typé : la règle d'introduction correspond au constructeur de paires tandis que les éliminateurs correspondent aux projections $.\pi_0$ et $.\pi_1$.

Pour l'instant, nous n'avons pas établi de lien réel entre ces connecteurs et nos termes, c'est l'enjeu de la section suivante.

4.1.2 Correspondance de Curry Howard

Dans la section précédente nous avons regardé quelques intuitions permettant de voir le lien entre la logique et l'informatique au travers des systèmes de type. Nous avons étudié le λ-calcul qui permet de faire le lien avec la déduction naturelle. Voici un tableau montrant les différents liens entre la logique et le λ-calcul.

logique	$\leftrightarrow$	λ-calcul
axiome	$\leftrightarrow$	variable
règles d'introduction	$\leftrightarrow$	constructeurs
règles d'élimination	$\leftrightarrow$	destructeurs
normalisation des preuves	$\leftrightarrow$	réduction du calcul

Le fait d'avoir séparé nos termes permet de rendre ces correspondances plus aisées. En effet les constructeurs correspondent aux *termes vérifiables* tandis que les *termes synthétisables* sont les destructeurs. Introduire de nouveaux termes dans notre λ-calcul nous a ainsi permis d'introduire de nouveaux connecteurs logiques dans notre système. Nous allons suivre la même motivation en introduisant les types dépendants.

Si nous souhaitons prouver une proposition, c'est à dire un type, il nous faudra trouver un terme accepté par notre vérificateur de type. Ceci permet d'aider grandement la recherche de démonstration étant donné que le vérificateur de type nous assiste dans la création des preuves, notamment grâce à une amélioration que nous verrons en Section 4.4.

4.2 Système de type dépendant

Comme nous allons le voir dans cette section, les types dépendants vont nous permettre d'écrire des programmes/termes qu'il était impossible d'écrire dans le cadre simplement typé. Faisant écho à la section précédente, nous allons introduire un nouveau lien entre logique et typage : le quantificateur $\forall$.

4.2.1 Equivalence entre les types et les termes

Jusqu'à présent, nous avions d'une part l'ensemble des types et l'ensemble des termes. Cette séparation avait un sens car les termes et les types ne se comportaient pas de la même façon et n'avaient pas les mêmes propriétés. La plus grande différence se situait au niveau de l'évaluation, en effet nos types étant statiques il n'était pas nécessaire de les évaluer. Cependant certains termes ne peuvent pas s'exprimer dans un système de type statique, prenons par exemple la fonction identité.

Exemple 25. Dans notre système précédent, il existait une fonction identité

pour chaque type :

$$\begin{aligned}
&\lambda x.x \ : Bool \to Bool \\
&\lambda x.x \ : Int \to Int \\
&...
\end{aligned}$$

On souhaiterait écrire une fonction polymorphe dont le type de retour est influencé par le type en entrée.

Il existe différentes façon de permettre l'écriture de fonctions comme celles-ci. Inspirons-nous de la logique, le quantificateur $\forall$ nous permet d'écrire ce type de termes.

Exemple 26. On souhaiterait pouvoir écrire une fonction identité de la manière suivante :

$$\lambda A.\lambda x.x \ : (\forall A, A \to A)$$

Avec des variables au sein de nos types nous pouvons désormais introduire des *familles de type*, comme par exemple les vecteurs de taille n.

Exemple 27. Les vecteurs représentent des listes dont la taille est connue (et vérifiée) lors de la compilation. Voici des vecteurs d'éléments de type α, pour certaines tailles n fixées :

$$\begin{aligned}
&\mathsf{nil} && : \mathsf{vec}\ 0\ \alpha \\
&\mathsf{cons\ true\ nil} && : \mathsf{vec}\ 1\ \mathsf{bool} \\
&\mathsf{cons\ false\ (cons\ true\ nil)} && : \mathsf{vec}\ 2\ \mathsf{bool}
\end{aligned}$$

En introduisant une dépendance des types sur les types, ceux-ci se comportent de la même façon que les termes. En particulier, la notion de réduction s'élève au niveau des types : on s'attend à ce que le type $\mathsf{vec}\ (2+2)\ \mathsf{bool}$ soit "équivalent" au type $\mathsf{vec}\,4\,\mathsf{bool}$. Poussant cette logique à l'extrême, on est amené à considérer les types et les termes comme appartenant à une unique famille syntaxique régie par les même propriétés.

Afin de distinguer statiquement types et termes, on introduit le type $\star$, le type des types.

Exemple 28. Pour être tout à fait explicite, notre fonction identité est de type

$$\forall A : \star, A \to A$$

Quand le quantificateur porte sur une variable de type, on aura tendance à garder son type (nécessairement $\star$) implicite.

4.2.2 Réduction

Etant donné que nos types sont désormais des termes et que l'on peut les normaliser il nous faut d'abord parler de l'évaluation. L'évaluation du terme $\star$ est triviale car ce terme est déja en forme normale. Pour $\forall$, il nous faut évaluer ses deux arguments.

$$\frac{}{* \Downarrow *}$$

$$\frac{p \Downarrow t \quad p' \Downarrow t'}{\forall x : p.p' \Downarrow \forall x : t.t'}$$

En général le connecteur $\forall$ se note Π, nous garderons cette convention.
Les deux notations suivantes sont équivalentes :

$$\forall x : S.T \Leftrightarrow \Pi x : S.T$$

Afin d'alléger la syntaxe, un Π anonyme sera noté comme une simple implication :

$$\forall x : Nat.Nat \Leftrightarrow \Pi x : Nat.Nat \Leftrightarrow (Nat \rightarrow Nat)$$

L'implémentation suit la stratégie de normalisation par évaluation présentée en Section 3.5. Les deux *value* nécessaires à l'implémentation de cette évaluation sont :

```
type value =

(...)

  | VStar
  | VPi of value × (value → value)
```

En suivant le même raisonnement que pour l'évaluation d'une abstraction avec la fonction `big_step_eval`, nous allons transformer le second membre de $\forall$ en fonction anonyme.

```
let rec big_step_eval_inTm t envi =

(...)

  | Star → VStar
  | Pi (v,x,y) →
     VPi ((big_step_eval_inTm x envi),
          (function arg → (big_step_eval_inTm y (arg :: envi))))
```

Le premier membre est évalué simplement tandis que le second est evalué vers une fonction anonyme puis, comme pour l'abstraction, cela nous permet de pouvoir directement substituer les valeurs normalisées dans notre type.

La fonction de réification est étendue sans surprise :

$$\boxed{\Gamma \vdash T \ni t} \qquad\qquad \boxed{\Gamma \vdash t \in T}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x \in T}(\text{Var})$$

$$\frac{\Gamma \vdash t \in T' \;\; T = T'}{\Gamma \vdash T \ni inv(t)}(\text{Inv}) \qquad \frac{\Gamma \vdash * \ni T \;\; \Gamma \vdash T \ni t}{\Gamma \vdash (t : T) \in T}(\text{Ann})$$

$$\frac{}{\Gamma \vdash * \ni *}(\text{Star})$$

$$\frac{\Gamma \vdash * \ni S \;\; \Gamma, x : S \vdash * \ni T}{\Gamma \vdash * \ni \forall x : S.T}(\text{Pi})$$

$$\frac{\Gamma, x : S \vdash T \ni t}{\Gamma \vdash \forall x : S.T \ni \lambda x.t}(\text{Abs}) \qquad \frac{\Gamma \vdash f \in \forall x : S.T \quad \Gamma \vdash S \ni s}{\Gamma \vdash f \; s \in T[s := x]}(\text{App})$$

FIGURE 3 – Typage dépendant

```
let rec value_to_inTm i v =
  match v with

(...)

  | VPi(x,f) → let var = gensym () in
               begin
                 Pi(Global(var),
                    (value_to_inTm i x),
                    (value_to_inTm (i+1) (f(vfree(Quote i)))))
               end
```

4.2.3 Formalisation

Maintenant que nous avons l'intuition des nouveaux constructeurs, nous pouvons enrichir notre langage. Les règles de typage sont présentées en style bidirectionnel en Figure 3.

Les deux nouvelles constructions Pi et $\star$ sont des termes vérifiables étant donné que l'on possède toute l'information nécessaire pour les vérifier efficacement. De manière générale les types seront des termes vérifiables. Le caractère bidirectionnel du langage est utilisé ici de façon cruciale pour obtenir une présentation algorithmique du test de conversion : il est tout simplement traité par le changement de phase (règle INV).

Pour le moment, mis à part la dépendance introduite par l'abstraction, notre langage ne comporte pas de termes exploitant l'expressivité des types dépendants. Nous découvrirons leur richesse au fur et à mesure des extensions.

Avant de poursuivre, il est important de se convaincre que l'ensemble des termes exprimables avec un système de type simple est inclus dans l'ensemble des termes exprimables dans un système de type dépendant.

Exemple 29. On peut toujours exprimer une fonction identité applicable uniquement aux entiers de la manière suivante :

$$\lambda x.x : \forall x : Int.\, Int$$

Implémentation : Représentons nos termes vérifiables et synthétisables dans le programme :

```
type inTm =

  | Abs of name × inTm
  | Inv of exTm
  | Pi of name × inTm × inTm
  | Star

and exTm =

  | Ann of inTm × inTm
  | BVar of int
  | FVar of name
  | Appl of exTm × inTm
```

Nous n'avons rajouté que deux nouveaux termes dans le noyau de notre langage mais le fait de ne plus faire la distinction entre les termes et les types change beaucoup de choses dans l'architecture du programme.

- $\boxed{\Gamma \vdash T \ni t}$
  Nous avons désormais une forme normale pour les types, il est donc intéressant de fournir cette forme normale directement au vérificateur de type. La fonction `check` prend donc maintenant un contexte, un terme vérifiable ainsi qu'un type en forme normale. Nous somme obligé d'effectuer cette opération car nos termes ne sont plus statiques.

  ```
  let rec lcheck contexte ty inT =
    match inT with
  ```

- $\boxed{\dfrac{\Gamma, x : S \vdash T \ni t}{\Gamma \vdash \forall x : S.T \ni \lambda x.t}\ (\textsc{Abs})}$
  Pour l'abstraction il nous faut libérer les occurences de x dans T. Etant donné que nous avons désormais des variables liées dans le type Π il nous faut effectuer la même opération que pour l'abstraction à savoir une substitution avec une variable fraîche. Notre type étant sous forme normale son second membre est une fonction anonyme comme vu en Section 4.2.2. La substitution s'effectue donc au travers de l'application de fonction en Ocaml.

```
| Abs(x,y) →
   begin
     match ty with
     | VPi(s,t) → let freshVar = gensym() in
                  lcheck (((Global freshVar),s)::contexte)
                    (t (vfree (Global freshVar)))
                    (substitution_inTm y (FVar(Global(freshVar))) 0)
     | _ → false
   end
```

- $\boxed{\dfrac{\Gamma \vdash t \in T' \;\; T = T'}{\Gamma \vdash T \ni inv(t)}\,(\textsc{Inv})}$

Le principal changement de cette implémentation est le changement d'égalité. En effet, ici les types sont des valeurs OCaml d'ordre supérieur (pouvant contenir des fonctions anonymes) dont on ne peut tester l'égalité. Il nous faut donc d'abord transformer ces valeurs en termes. Le second problème est que le nom associé aux différents lieurs tels que le λ ou le $\forall$ n'auront pas nécessairement le même nom. Comme nous l'avons vu dans la Section 2.1.2, deux termes sont égaux même si le nom de leur variable est différents. La fonction `equal_inTm` parcourt simplement les deux termes passés en arguments sans tenir compte du nom de variable associé au lieur. Par contre les variables libres doivent nécessairement être les mêmes.

```
| Inv(x) →
   let ret = lsynth contexte x in
       if equal_inTm (value_to_inTm 0 (ty)) (value_to_inTm 0 ret)
       then true
       else false
```

- $\boxed{\dfrac{}{\Gamma \vdash * \ni *}\,(\text{Star})}$

L'implémentation de la règle *Star* est triviale :

```
| Star →
   begin
    match ty with
      | VStar → true
      | _ → false
   end
```

- $\boxed{\dfrac{\Gamma \vdash * \ni S \;\; \Gamma, x : S \vdash * \ni T}{\Gamma \vdash * \ni \forall x : S.T}\,(\text{Pi})}$

Ici, Π n'est pas sous forme normale étant donné qu'il est le terme à vérifier. Nous sommes donc obligés d'effectuer la substitution de x dans T avec notre fonction `substitution` :

```
| Pi (v,s,t) →
   begin
     match ty with
     | VStar → let freshVar = gensym () in
               lcheck contexte VStar s
               && lcheck (((Global freshVar),
                           (big_step_eval_inTm s []))::contexte)
                 VStar
                 (substitution_inTm t (FVar(Global(freshVar))) 0)
     | _ → false
   end
```

- $\boxed{\boxed{\Gamma \vdash t \in T}}$

Notre fonction de synthèse nécessite un contexte `ctxt` qui est de même type que celui de la fonction `check` ainsi qu'un terme synthétisable `exT`

```
and lsynth ctxt exT =
  match exT with
```

- $\boxed{\dfrac{x : T \in \Gamma}{\Gamma \vdash x \in T}(\text{Var})}$

Tout comme dans l'implémentation précédente, il n'est pas possible de rencontrer de variable liée lors de la vérification de type.

```
  | BVar x → failwith "Impossible to check a BoundVar"
  | FVar x → List.assoc x ctxt
```

- $\boxed{\dfrac{\Gamma \vdash f \in \forall x : S.T \quad \Gamma \vdash S \ni s}{\Gamma \vdash f\ s \in T[s := x]}(\text{App})}$

Si la synthèse de f nous donne bien un Π et que l'on vérifie s de type S alors il nous faut retourner T en substituant les occurences de variables liées. Etant donné que Π est sous forme normale cette substitution s'effectue comme dans la règle de l'abstraction, par application de fonction dans Ocaml.

```
| Appl(f,s) →
   let synth_f = lsynth ctxt f in
   begin
     match synth_f with
     | VPi(s_pi,fu) → if lcheck ctxt s_pi s
                      then (fu (big_step_eval_inTm s []))
                      else failwith "fail synth Appl"
     | _ → failwith "fail synth Appl"
   end
```

- $\boxed{\dfrac{\Gamma \vdash * \ni T \ \Gamma \vdash T \ni t}{\Gamma \vdash (t : T) \in T}(\text{Ann})}$

Etant donné que termes et types sont équivalents, il faut vérifier que le type T proposé par l'annotation est bien un type (donc de type $\star$), sans quoi l'utilisateur pourrait introduire un type mal formé et donc, potentiellement, une contradiction logique. Il faut aussi évaluer celui-ci afin de le mettre en forme normale.

```
  | Ann(x,t) → let eval_t = big_step_eval_inTm t [] in
               if lcheck ctxt VStar t
                 && lcheck ctxt (big_step_eval_inTm t []) x
               then eval_t
               else failwith "fail synth Ann"
```

4.3 Extensions

Comme nous l'avons vu dans l'Exemple 27 nous pouvons typer de nouveaux termes inexistants auparavant. Nous commencerons par mettre à jour les types de données courants. Nous présenterons ensuite les vecteurs ainsi que l'égalité.

4.3.1 Les entiers

La représentation de nos termes n'a pas changé :

```
type inTm =

(...)

  | Zero
  | Succ of inTm
  | Nat

and exTm =

(...)

  | Iter of inTm × inTm × inTm × inTm
```

Passons maintenant aux règles de typages :

$$\text{(Nat)}\ \frac{}{\Gamma \vdash * \ni Nat} \qquad \text{(Zero)}\ \frac{}{\Gamma \vdash Nat \ni \mathsf{zero}} \qquad \text{(Succ)}\ \frac{\Gamma \vdash Nat \ni n}{\Gamma \vdash Nat \ni \mathsf{succ}\ n}$$

$$\text{(Iter)}\ \frac{\Gamma \vdash n \ni Nat \qquad \Gamma \vdash Nat \to * \ni P \qquad \Gamma \vdash \forall n : Nat.(P\ n \to P\ (\mathsf{succ}\ n)) \ni f \qquad \Gamma \vdash P\ \mathsf{zero} \ni a}{\Gamma \vdash \mathsf{iter}\ P\ n\ f\ a \in P\ n}$$

Comme nous l'énoncions précédement les types sont maintenant des termes, il nous faut donc ajouter une règle *Nat*.

Pour continuer l'analogie avec la logique propositionnelle, on constate que la règle de typage de iter correspond à un raisonnement par *induction*. En effet

on vérifie d'abord que l'argument a est bien du type du prédicat dans le cas de zero. Puis l'on vérifie que la fonction f est définie pour tout n vers $n + 1$.

Nous allons maintenant définir un λ-terme permettant d'effectuer l'addition de deux nombres avec l'aide d'une syntaxe Lispienne (parsée par notre prototype) :

```
plus = (lambda n (lambda a
          (iter (lambda x N) n
                (lambda ni (lambda x (succ x))) a)))
```

Pour effectuer l'addition $n + m$ la stratégie consiste à itérer n fois la fonction successeur sur m. Afin d'alléger la syntaxe nous utiliserons dorénavant `(+ n m)` correspondant au terme *plus* appliqué aux entiers n et m.

Implémentation : Commençons par les règles de vérification :

```
let rec lcheck contexte ty inT =
  match inT with

(...)

  | Nat → ty = VStar
  | Zero → ty = VNat
  | Succ(x) →
     begin
       match ty with
         | VNat → lcheck contexte VNat x
         | _ → false
     end
```

Pour *iter* si l'ensemble des hypothèses sont vérifiées il nous faut retourner l'application du prédicat P sur n.

```
  | Iter(p,n,f,a) →
     let big_p = big_step_eval_inTm p [] in
     let big_n = big_step_eval_inTm n [] in
     if lcheck ctxt (big_step_eval_inTm (read "(→ N ×)") []) p &&
       lcheck ctxt (big_step_eval_inTm (read "N") []) n &&
       lcheck ctxt (big_step_eval_inTm
                          (Pi(Global("n"),Nat,
                              Pi(Global("NO"),(Inv(Appl(Ann(p,Pi(Global"NO",Nat,Star)
                                 (Inv(Appl(Ann(p,Pi(Global"NO",Nat,Star)),Succ(n)))))
       f &&
       lcheck ctxt (vapp(big_p,VZero)) a
     then (vapp(big_p,big_n))
     else failwith "Iter synth fail"
```

Dans l'implémentation 3.4.1 nous n'avons pas présenté l'évaluation à grands pas des entiers naturels, voici donc le code correspondant :

```
  | Nat → VNat
  | Zero → VZero
  | Succ(n) → VSucc(big_step_eval_inTm n envi)
```

Pour l'évaluation de `iter` nous utilisons la même technique que pour l'application. On évalue l'ensemble des arguments puis on appelle la fonction `vitter`

```
| Iter(p,n,f,a) → vitter ((big_step_eval_inTm p envi),
                          (big_step_eval_inTm n envi),
                          (big_step_eval_inTm f envi),
                          (big_step_eval_inTm a envi))
```

Le prédicat ne sert pas dans l'évaluation mais étant donné que l'on doit retourner un terme neutre si le terme contient des variables libres nous le gardons dans les arguments de la fonction.

```
and vitter (p,n,f,a) =
  match n,f with
  | (VZero,VLam fu) → a
  | (VSucc(x),VLam fu) → vapp(fu n,(vitter (p,x,f,a)))
  | _ → VNeutral(NIter(p,n,f,a))
```

4.3.2 Les vecteurs

Les vecteurs permettent d'introduire réellement de la dépendance de type. Nous avons vu une présentation assez informelle des vecteurs afin de présenter les types dépendants, il sera donc question ici de formaliser l'intuition de la Section 4.2.1. Commençons par enrichir notre grammaire :

$$
\begin{array}{rcll}
t & ::= & (\ldots) & \text{(termes vérifiables)} \\
 & | & \mathit{vec}\ t\ t & \text{(type des vecteurs)} \\
 & | & \mathit{dnil}\ t & \text{(vecteur vide)} \\
 & | & \mathit{dcons}\ t\ t & \text{(concaténation)} \\
s & ::= & (\ldots) & \text{(termes synthétisables)} \\
 & | & \mathit{dfold}\ t\ t\ t\ t\ t\ t & \text{(itérateur)}
\end{array}
$$

Ce qui se traduit dans notre programme par le code suivant :

```
| Vec of inTm × inTm
| DNil of inTm
| DCons of inTm × inTm

| DFold of inTm × inTm × inTm × inTm × inTm × inTm
```

Les arguments des différents constructeurs seront expliqués une fois que nous aurons vu les règles de typage présentes en figure 4.3.2.

Dans la règle *vec*, α sera le type des éléments contenus dans le vecteur et n sa taille. Cette représentation des vecteurs n'est pas familière aux programmeurs, en général si l'on veut appliquer une fonction sur des vecteurs de taille précise il faut effectuer un test à l'entrée de la fonction. C'est un des aspects très intéressant des types dépendants qui permettent d'assister les programmeurs dans la vérification de leurs programmes.

Le terme *dnil* correspond au vecteur vide, et *dcons* qui nous permet d'ajouter un élément dans un vecteur.

(Vec)
$$\frac{\Gamma \vdash * \ni \alpha \;\; \Gamma \vdash Nat \ni n}{\Gamma \vdash * \ni Vec\,\alpha\,n}$$

(dnil)
$$\frac{}{\Gamma \vdash Vec\,\alpha\,\mathsf{zero} \ni dnil\,\alpha}$$

(dcons)
$$\frac{\Gamma \vdash Vec\,\alpha\,n \ni v \;\; \Gamma \vdash \alpha \ni a}{\Gamma \vdash Vec\,\alpha\,(\mathsf{succ}\,n) \ni dcons\,a\,v}$$

Exemple 30. Voici un vecteur de taille trois dont les éléments sont des entiers :

$$dcons\ zero\ (dcons\ (succ\ zero)\ (dcons\ zero\ (dnil\ Nat)))$$

Nous pouvons maintenant créer des vecteurs mais nous ne possédons pas encore d'éliminateur pour les utiliser. Nous allons implémenter une fonction bien connue des programmeurs fonctionnels, dfold dont voici une définition intuitive : `dfold f vec b` $\rightarrow$ `f a1 (f a2(...(f an b)...))`. Définissons maintenant l'éliminateur dfold comme un terme synthétisable :

(Dfold)
$$\frac{\begin{array}{c}\Gamma \vdash * \ni \alpha\;\Gamma \vdash Nat \ni n \qquad \Gamma \vdash \forall n : Nat.\forall v : Vec\,\alpha\,n.* \ni P \\ \Gamma \vdash Vec\,\alpha\,n \ni v \qquad \Gamma \vdash P\,\mathsf{zero}\,(dnil\,\alpha) \ni a \\ \Gamma \vdash \forall n : N.\forall v : Vec\,\alpha\,n.\forall a : \alpha.(P\,n\,v \rightarrow P\,(succ\,n)\,(dcons\,a\,v)) \ni f\end{array}}{\Gamma \vdash dfold\,\alpha\,P\,n\,v\,f\,a \in P\,n\,v}$$

De la même manière que pour l'éliminateur des entiers naturels *iter*, le prédicat p nous permet de rendre le type de retour dépendant des différents arguments.

Il existe de nombreuses façons d'utiliser *dfold*, une application assez simple consiste à faire la somme des éléments d'une liste. Voici le terme correspondant à cette utilisation :

```
somme_liste =
  (lambda v (lambda n
    (dfold N (lambda n (lambda xs N)) n v
      (lambda n (lambda xs (lambda a (lambda x (+ a x)))))
      zero)))
```

Voici donc le code du type-checker correpondant :

- $\boxed{\dfrac{\Gamma \vdash * \ni \alpha \;\; \Gamma \vdash Nat \ni n}{\Gamma \vdash * \ni Vec\,\alpha\,n}(\text{Vec})}$

  Il nous faut vérifier les deux arguments du terme `Vec`.

```
| Vec(alpha,n) → ty = VStar &&
                    lcheck contexte VStar alpha  &&
                      lcheck contexte VNat n
```

- $\boxed{\dfrac{}{\Gamma \vdash Vec\ \alpha\ \mathsf{zero} \ni dnil\ \alpha}(\text{Dnill})}$

Afin de vérifier le terme *Dnill* il nous faut simplement vérifier que les termes α sont égaux.

```
| DNil(alpha) →
   begin
     match ty with
     | VVec(alpha_vec,VZero) →
        equal_inTm (value_to_inTm 0
                      (big_step_eval_inTm alpha []))
          (value_to_inTm 0 alpha_vec)
     | _ → false
   end
```

- $\boxed{\dfrac{\Gamma \vdash Vec\ \alpha\ n \ni v \quad \Gamma \vdash \alpha \ni a}{\Gamma \vdash Vec\ \alpha\ (succ\ n) \ni dcons\ a\ v}(\text{Dcons})}$

```
| DCons(a,xs) →
   begin
     match ty with
     | VVec(alpha,VSucc(n)) →
        lcheck contexte (VVec(alpha,n)) xs  &&
          lcheck contexte alpha a
     | _ → false
   end
```

- $\boxed{\dfrac{\begin{array}{c}\Gamma \vdash * \ni \alpha \quad \Gamma \vdash Nat \ni n \qquad \Gamma \vdash \forall n : Nat. \forall v : Vec\ \alpha\ n. * \ni P \\ \Gamma \vdash Vec\ \alpha\ n \ni v \qquad \Gamma \vdash P\ \mathsf{zero}\ (dnil\ \alpha) \ni a \\ \Gamma \vdash \forall n : N. \forall v : Vec\ \alpha\ n. \forall a : \alpha. (P\ n\ v \to P\ (succ\ n)\ (dcons\ a\ v)) \ni f\end{array}}{\Gamma \vdash dfold\ \alpha\ P\ n\ v\ f\ a \in P\ n\ v}(\text{Dfold})}$

Il nous faut tout d'abord vérifier l'ensemble des hypothèses puis retourner l'application du terme P aux termes n et v

```
| DFold(alpha,p,n,xs,f,a) →
   let type_p = (Pi(Global"n",Nat,(Pi(Global"xs",Vec(alpha,Inv(BVar 0)),Star))
   if lcheck ctxt VStar alpha &&
     lcheck ctxt (big_step_eval_inTm type_p []) p &&
     lcheck ctxt VNat n &&
     lcheck ctxt (big_step_eval_inTm (Vec(alpha,n)) []) xs &&
     lcheck ctxt
     (big_step_eval_inTm
        (Pi(Global"n",Nat,
            Pi(Global"xs",Vec(alpha,Inv(BVar 0)),
               Pi(Global"a",alpha,
```

```
                    Pi(Global"NO",Inv(Appl(Appl(Ann(p,type_p),n),xs)),
                       Inv(Appl(Appl(Ann(p,type_p),Succ(n)),DCons(a,xs))))))))
  lcheck ctxt (big_step_eval_inTm (Inv(Appl(Appl(Ann(p,type_p),Zero),DNil(a
then (big_step_eval_inTm (Inv(Appl(Appl(Ann(p,type_p),n),xs))) [])
else failwith "DFOld synth something goes wrong"
```

Evaluation : Avant d'implémenter l'évaluation dans notre programme il nous faut tout d'abord rajouter les *valeurs* suivantes à notre langage.

```
| VVec of value × value
| VDNil of value
| VDCons of value × value
```

Passons maintenant à l'évaluation des vecteurs, nos trois constructeurs n'ont pas d'évaluation particulière :

```
| Vec(alpha,n) → VVec((big_step_eval_inTm alpha envi),(big_step_eval_inTm n envi))
| DNil(alpha) → VDNil(big_step_eval_inTm alpha envi)
| DCons(a,xs) → VDCons((big_step_eval_inTm a envi),(big_step_eval_inTm xs envi))
```

Si *dfold* contient des variables libres il nous faut un élément neutre à retourner :

```
| NDFold of value × value × value × value × value × value
```

Pour l'implémentation, comme pour iter, nous avons crée une fonction `vfold` qui nous permet d'effectuer l'évaluation :

```
and vfold(alpha,p,n,xs,f,a) =
  match xs,f,n with
  | (VDNil(alphi),VLam fu,VZero) → a
  | (VDCons(elem,y),VLam fu,VSucc(ni)) → vapp(vapp(vapp(fu n,xs),elem),vfold(alpha,p
  | _ → VNeutral(NDFold(alpha,p,n,xs,f,a))
```

4.3.3 L'égalite

Pour implémenter l'égalité au sein de notre langage, il faut nous tourner vers la logique. La théorie des types implémente une définition de l'égalité proche de la conception de Leibniz : deux éléments sont identiques s'ils ont les mêmes propriétés. Pour que deux éléments soient égaux, il faut nécessairement qu'ils soient de même type. Ce n'est pas une condition suffisante mais elle est nécessaire. On notera $a =_A b$ l'égalité de deux élements de type A.

Exemple 31. L'égalité sur les entiers s'écrit donc $2 + 2 =_N 4$

Si l'on souhaite exprimer la même idée en logique nous écrirons : $Id(A, a, b)$. Nous allons donc introduire les règles de typage du terme *id* ainsi que celle de son introducteur *refl* et de son éliminateur *Trans*.

- (Id)
$$\frac{\Gamma \vdash * \ni A \quad \Gamma \vdash A \ni a \quad \Gamma \vdash A \ni b}{\Gamma \vdash * \ni Id\, A\, a\, b}$$

Pour le terme $Id\, A\, a\, b$ il nous faut tout d'abord évaluer A afin de pouvoir vérifier les termes a et b

```
let rec lcheck contexte ty inT =
  match inT with

  | Id(gA,a,b) →
     let eval_gA = big_step_eval_inTm gA [] in
     ty = VStar &&
     lcheck contexte VStar gA  &&
       lcheck contexte eval_gA a  &&
       lcheck contexte eval_gA b
```

- (Refl)
$$\frac{\Gamma \vdash A \ni a}{\Gamma \vdash Id\, A\, a\, a \ni refl\, a}$$

La spécification nous indique implicitement qu'il faut tester l'égalité entre les deux termes a. C'est le point central de cette règle. Deux termes sont donc éqaux si leur évaluation est la même et que l'on peut les vérifier avec le type A.

```
  | Refl(a) →
     begin
       match ty with
       | VId(gA,ta,ba) →
          if equal_inTm a (value_to_inTm 0 ta) &&
            equal_inTm a (value_to_inTm 0 ba)
          then lcheck contexte gA a
          else false
       | _ → false
     end
```

- (Trans)
$$\frac{\begin{array}{c}\Gamma \vdash * \ni A \\ \Gamma \vdash A \ni a \qquad \Gamma \vdash A \ni b \qquad \Gamma \vdash \forall a : A. \forall b : A. (Id\, A\, a\, b \to *) \\ \Gamma \vdash Id\, A\, a\, b \ni q \qquad \Gamma \vdash P\, a\, a\, (refl\, a) \ni x\end{array}}{\Gamma \vdash trans\, A\, P\, a\, b\, q\, x \in P\, a\, b\, q}$$

La relation de transitivité nous indique que si les éléments a et b sont égaux et que l'on vérifie x alors celui-ci est égal à a et b.

```
and lsynth ctxt exT =
  match exT with

  | Trans(gA,p,a,b,q,x) →
```

```
let type_p = Pi(Global"a",gA,Pi(Global"b",gA,Pi(Global"NO",Id(gA,Inv(BVar
if lcheck ctxt VStar gA &&
  lcheck ctxt (big_step_eval_inTm gA []) a &&
  lcheck ctxt (big_step_eval_inTm gA []) b &&
  lcheck ctxt  (big_step_eval_inTm (Id(gA,a,b)) []) q &&
  lcheck ctxt (big_step_eval_inTm type_p []) p &&
  lcheck ctxt (big_step_eval_inTm (Inv(Appl(Appl(Appl(Ann(p,type_p),a),b),
then (big_step_eval_inTm (Inv(Appl(Appl(Appl(Ann(p,type_p),a),b),q))) [])
else failwith "Trans synth fail"
```

4.4 Exemples de preuves

Dans la section 4.1.2 nous avons parlé de l'utilisation du vérificateur de types pour assister la démonstration de propositions logiques. Nous allons étendre notre implémentation afin de faciliter cette tâche. Nous verrons ensuite un exemple de démonstration de la proposition "2 + 2 = 4", qui était l'objectif initial de ce projet.

4.4.1 Amélioration du vérificateur

Dans les sections précédentes, afin que le code soit présentable, nous avons présenté une version "allégée" de la fonction de vérification. Cependant, nous avons également implémenté une version plus complète comportant des messages d'erreurs précis.

Dans l'implémentation du vérificateur de type du λ-calcul simplement typé lorsqu'un terme n'était pas correctement typé la fonction stoppait son éxecution par le moyen d'une exception. Nos programmes devenant de plus en plus complexes, il peut être intéressant de renvoyer un rapport détaillé de l'échec. Voici la structure que nous allons utiliser :

```
type debug =
  | Report of debug × debug × debug × debug
  | Success of bool
  | Contexte of string
  | Steps of string
  | Error of string
and debug_synth =
  | RetSynth of debug × value
```

Nous aurons aussi besoin des fonctions suivantes :

- `res_debug` : retourne le booléen présent dans le rapport.
- `create_report` : créé un rapport en prenant un argument pour chaque champs du `report`

Les équivalents de ces fonctions pour le type `debug synth` sont suivis du suffixe `_synth`. Cette assistance est nécessaire pour aider les programmeurs à connaître la source de l'erreur, tout comme le font les compilateurs modernes. Cela nous aide aussi à tester la validité de celui-ci car on peut maintenant tester si la vérification d'un terme est fausse.

Afin que notre vérificateur de type permette d'assister dans la recherche de preuves nous allons implémenter un nouveau constructeur : ?¿?.

Le but de ce constructeur est d'interrompre la vérification et de demander au vérificateur de type de retourner le type que l'on souhaite vérifier.

Exemple 32. Soit le terme `(lambda x ?)` que l'on veut vérifier avec le type `(pi x nat bool)`, le type checker énoncera le fait que à la place du terme ?¿? nous souhaitons un terme de type booléen.

Nous allons donc rajouter ce terme parmi les termes vérifiables :

```
type inTm =

(...)

  | What
```

Etant donné que ce terme est simplement là pour améliorer l'utilisation de notre programme, il ne possède pas de règle d'évaluation ou tout autre règle de calcul. Dans notre vérificateur de type nous allons implémenter le code suivant :

```
| What → create_report true (contexte_to_string contexte) steps
   ("What : we try to push this terme " ^
        (pretty_print_inTm (value_to_inTm 0 ty)  []))
```

Nous acceptons quand même le terme pour laisser la possibilité au programmeur de placer différents ?¿? dans son programme.

4.4.2 2 + 2 = 4

Attelons-nous maintenant à notre première preuve. Nous avons désormais tout le bagage nécessaire, les entiers et l'égalité. La construction de cette preuve est assez simple. Comme nous l'avons vu dans la section sur l'égalité 4.3.3 la contruction $2+2 =_N 4$ est équivalente au type $(id\ Nat\ (2\ +\ 2)\ 4)$. Il nous faut maintenant déterminer un terme correspondant à ce type. La règle d'introduction du type id est le terme $refl$. Le seul terme permettant de vérifier ce type est le terme $refl4$

Afin de vérifier cette preuve dans notre type checker il nous faut exécuter la fonction suivante :

```
lcheck [] (read ‘‘(succ (succ (succ (succ zero))))’’)
(read ‘‘(id (+ (succ (succ zero)) (succ (succ zero))) (succ (succ (succ (succ zero)))))’’)
```

La fonction `read` permet de parser une chaine de caractères pour obtenir le terme correspondant. On appelle la fonction `lcheck` avec comme premier argument `[]` qui correspond à la liste vide en Ocaml car nous débutons avec un contexte vide.

Pour vérifier que l'implémentation de notre vérificateur de type se comporte correctement tentons d'exécuter $1+2 =_N 4$ de la même façon. Cela nous retourne bien le booléen `false` comme attendu.

4.4.3 Preuve sur les vecteurs

Nous allons effectuer la preuve que la concaténation d'un vecteur de taille n et d'un vecteur de taille m nous donne un vecteur de taille $n+m$. Commençons tout d'abord par essayer d'énoncer cette preuve de façon intuitive. Nous souhaitons introduire de la dépendance sur le type des vecteurs cependant celui-ci doit être le même. Il nous faut aussi de la dépendance sur la taille de nos vecteurs. Voici donc le type dont nous souhaitons trouver un terme :

$\forall A : * \forall\, m : Nat \forall\, xs : Vec\, A\, m \forall\, n : Nat \forall\, ys : Vec\, A\, n \;\rightarrow\; Vec\, A\, (n\; +\; m)$

Cette spécification nous demande donc de créer un terme prenant quatre arguments :
-A qui correspond au type des deux vecteurs
-m la taille du premier vecteur
-xs un vecteur de type A et de taille m
-n la taille du second vecteur
-ys un vecteur de type A et de taille n
On doit ensuite retourner un élément de type $Vec\, A\, (n\; +\; m)$.

Le seul terme dont nous disposons pour construire un vecteur avec ce genre de dépendance est *dfold*. Construisons ce terme dans notre langage :
Tout d'abord nous avons quatre types $\forall$ ce qui nous force à commencer par un terme avec quatre abstractions `(lambda A (lambda m (lambda xs (lambda n (lambda ys (?))))))`

A l'aide du constructeur ?¿? le vérificateur de type nous indique qu'il attend un terme de type $Vec\, A\, (n\; +\; m)$ nous allons donc construire ce terme à l'aide de *dfold* car c'est le seul constructeur qui nous permet d'introduire de la dépendance dans les vecteurs.

Rappelons la règle de typage de Dfold :

(Dfold)

$$\frac{\Gamma \vdash * \ni \alpha \quad \Gamma \vdash Nat \ni n \qquad \Gamma \vdash \forall n : Nat. \forall v : Vec\, \alpha\, n. * \ni P \qquad \Gamma \vdash Vec\, \alpha\, n \ni v \qquad \Gamma \vdash P\ \mathsf{zero}\ (dnil\ \alpha) \ni a \qquad \Gamma \vdash \forall n : N. \forall v : Vec\, \alpha\, n. \forall a : \alpha. (P\, n\, v \rightarrow P\ (succ\, n)\ (dcons\, a\, v)) \ni f}{\Gamma \vdash dfold\, \alpha\, P\, n\, v\, f\, a \in P\, n\, v}$$

Il nous faut donc définir un prédicat permettant P permettant de générer le type $Vec\, A\, (n\; +\; m)$ une fois appliqué. Voici notre prédicat P :
`(lambda m (lambda v (Vec A (+ m n))))`

Trouvons maintenant la fonction f nous permettant de concatener nos vecteurs :
`(lambda f (lambda vec1 (lambda a (lambda vec2 (dcons a vec2)))))` Ce qui nous donne comme terme final :

```
(lambda A
 (lambda m
  (lambda xs
   (lambda n
    (lambda ys
     (dfold A (lambda mp (lambda vp (Vec A (+ mp n))))
```

```
        m xs (lambda nf (lambda vecun (lambda a (lambda xsf (dcons a xsf))))) ys))))
```

Voici la preuve rédigée à l'aide de AGDA un langage implémentant les types dépendants :

```
  concat : (A : Set)(m : N)(xs : Vec A m) → (n : N)(ys : Vec A n) →
CONCAT A m xs n ys
  concat A m xs n ys = dfold A (lambda m xs → CONCAT A m xs n ys)
                               m xs (lambda m a xs xs++ys → cons a xs++ys) ys
```


Remerciements

Je remercie les professeurs qui m'ont aidé dans la réalisation de ce mémoire. Pierre-Evariste Dagand pour m'avoir éclairé et soutenu tout au long de ce projet. Frédéric Peschanski pour sa pédagogie et Jérémie Salvucci pour avoir répondu à certaines de mes interrogations.

Références

A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3) :363–408, 2011. doi : 10.1007/s10817-011-9225-2.

A. Church. An Unsolvable Problem of Elementary Number Theory. *Amer. J. Math.*, 58 :345–363, 1936.

J. Krivine. *Lambda-calculus, types and models.* Ellis Horwood series in computers and their applications. Masson, 1993. ISBN 978-0-13-062407-9.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, 12 1978. doi : 10.1016/0022-0000(78)90014-4.

B. C. Pierce. *Types and programming languages.* MIT Press, Feb. 2002. ISBN 0262162091.