Langages fonctionnels non-stricts: De la paresse et de l'indulgence

Guy Tremblay

Dépt. d'informatique — Univ. du Québec À Montréal
Montréal, Qué.
`tremblay.guy@uqam.ca`

Résumé

Qu'est-ce qu'un langage fonctionnel non-strict? Qu'est-ce qu'un langage paresseux? Un langage non-strict est-il nécessairement paresseux? Dans cet article, nous tentons d'apporter des éléments de réponse à ces questions. Dans un premier temps, une présentation informelle des divers modes d'évaluation (strict, indulgent, paresseux) est faite. Ensuite nous examinons, par le biais de divers petits exemples, certaines des différences entre évaluation non-stricte mais non-paresseuse et évaluation paresseuse. Finalement, nous présentons un modèle opérationnel (en LOTOS) permettant de mieux comprendre les différences entre les divers modes d'évaluation. Ce modèle, qui utilise une approche d'évaluation entièrement parallèle, permet de bien saisir les caractéristiques particulières de l'évaluation indulgente lorsque mise en oeuvre de façon parallèle.

1. Introduction

Les langages fonctionnels sont généralement divisés en deux grandes classes: les stricts et les non-stricts. Lorsqu'on dit qu'un langage fonctionnel est non-strict, on sous-entend généralement qu'il est paresseux. Ainsi, Goldberg écrivait récemment:

> In a class of functional languages called *non-strict* functional languages, of which `Haskell` is the most popular, no function evaluates its arguments unless they are needed.
>
> Another feature of non-strictness is that it allows the construction of infinite data structures. [...] [I]n a non-strict language, the :: [le constructeur de liste] does not evaluate its operands until they are actually needed. [10, p. 251]

Les deux propriétés ainsi décrites — appel par-nécessité et structures de données infinies — sont en fait des propriétés des langages *paresseux*. Or, il existe des langages qui sont non-stricts, dans un sens que nous préciserons plus loin, mais sans posséder aucune des propriétés décrites par Goldberg: ces langages sont dits *indulgents* (*"lenient"*).

Dans cet article, nous tentons d'identifier certaines des caractéristiques des langages non-stricts, tant paresseux qu'indulgents. Nous verrons que des différences existent entre ces deux types de langage, et ce même si les deux sont non-stricts. Finalement, nous présenterons un aperçu d'un modèle opérationnel formel permettant de mieux comprendre, dans un contexte parallèle, les différences entre les divers modes d'évaluation (strict vs. indulgent vs. paresseux).

2. Langages stricts vs. langages non-stricts

Dans cette section nous présentons, de façon informelle, trois modes d'évaluation: strict, indulgent et paresseux. Nous verrons que ces deux derniers sont des modes d'évaluation non-stricts, car un résultat peut être produit par une fonction même lorsqu'un argument n'est pas défini.

2.1. Langages fonctionnels stricts

Soit f une fonction de n arguments définie par: $f\ x_1 \ldots\ x_n = e$. La fonction f est dite stricte en son $i^{ème}$ argument si la condition suivante est satisfaite ($\perp$ dénote une valeur indéfinie appelée "*bottom*"):

$$x_i = \perp \Rightarrow f\ x_1 \ldots\ x_n = \perp$$

Informellement, on a donc que f est stricte en un argument si lorsque l'argument effectif correspondant est indéfini ($x_i = \perp$) alors le résultat de la fonction *n'est pas* défini ($= \perp$). En d'autres mots, le résultat ne peut pas être produit tant que l'argument n'est pas disponible.

Pour les programmeurs familiers avec les langages impératifs ou les langages fonctionnels style `SCHEME` ou `ML`, il est naturel qu'une fonction soit stricte en *tous* ses arguments: ces langages utilisent un mécanisme de passage des paramètres *par valeur* ayant pour effet de rendre strictes (en tous leurs arguments) toutes les fonctions. Dans le contexte d'une mise en oeuvre séquentielle, l'utilisation de l'appel-par-valeur signifie que les arguments peuvent être évalués *avant* que l'appel de fonction soit effectué. Si l'évaluation d'un argument ne se termine pas ($= \perp$), alors la fonction ne sera jamais appelée et, donc, aucun résultat ne sera produit.

On peut donc dire qu'un langage fonctionnel est *strict* lorsque sa *mise en oeuvre* assure que toutes les fonctions sont strictes en tous leurs arguments, ce qui peut être assuré par l'utilisation de l'appel-par-valeur.

2.2. Langages fonctionnels non-stricts

Par opposition, on peut dire qu'un langage fonctionnel est *non-strict* lorsqu'il est possible pour une fonction de produire un résultat même lorsqu'un de ses arguments est indéfini. Il existe deux grandes stratégies pour mettre en oeuvre les langages non-stricts: la paresse et l'indulgence.

Langages paresseux. L'idée sous-jacente à la mise en oeuvre des langages paresseux peut aussi être expliquée par le mécanisme utilisé pour le passage des paramètres. Dans ce cas, le mécanisme utilisé est l'*appel-par-nécessité* ("*call-by-need*"). Dans cette approche, une expression utilisée comme argument est évaluée *seulement si* sa valeur est *requise* pour produire le résultat de la fonction. Une telle expression n'a donc *pas* à être évaluée avant l'appel; elle est plutôt encapsulée dans une *suspension* (un *glaçon*). Une fois cette encapsulation effectuée, l'évaluation du corps de la fonction peut s'amorcer. Si/lorsque, durant l'évaluation de ce corps, la valeur de l'argument est requise, alors l'évaluation du corps est suspendue et l'expression encapsulée dans la suspension appropriée est évaluée (le glaçon est dégelé), après quoi l'évaluation de la fonction peut se poursuivre. [1]

Dans un contexte fonctionnel, où l'absence d'effets de bord permet d'obtenir les propriétés de *transparence référentielle* et de "*definiteness*" [20], chaque évaluation d'une expression encapsulée dans une suspension produit toujours la même valeur. Cette valeur peut donc être mémorisée pour ensuite être réutilisée. C'est ce dernier point qui distingue le mécanisme d'appel-par-nécessité de l'*appel-par-nom*: dans l'appel-par-nécessité, une expression n'est évaluée *qu'au plus* une seule fois.

Langages indulgents. L'indulgence (*leniency*) comme forme d'évaluation non-stricte fut introduite par Traub [21]. Contrairement à l'évaluation paresseuse, donc comme l'évaluation stricte, l'approche indulgente ne conduit à la création d'aucune suspension. Une expression utilisée comme argument est *toujours* évaluée. Toutefois, cette évaluation n'a pas à se faire avant l'appel de la fonction: dans le cas indulgent, il n'y a pas *a priori* de contraintes d'ordonnancement entre l'évaluation des

[1] Il est parfois possible — par le biais d'analyses de stricticité [8, 13, 22] — d'éviter d'utiliser le mécanisme de gel/dégel et d'utiliser plutôt l'appel-par-valeur. Malheureusement, ces analyses ne peuvent pas être complètes et certaines expressions doivent donc inévitablement utiliser le mécanisme de gel/dégel.

arguments et l'évaluation de la fonction; les seules contraintes à respecter sont celles liées au respect des dépendances de données.

Bien que l'approche indulgente s'applique aussi bien dans un contexte d'évaluation séquentielle que parallèle, l'idée sous-jacente s'exprime plus facilement dans le cas parallèle: l'indulgence implique que le corps de la fonction *et* les arguments peuvent être évalués *en parallèle*, alors que le respect des dépendances signifie que lorsqu'un argument devient requis dans la fonction, une synchronisation appropriée (entre l'argument et l'expression utilisant cet argument) doit être effectuée.

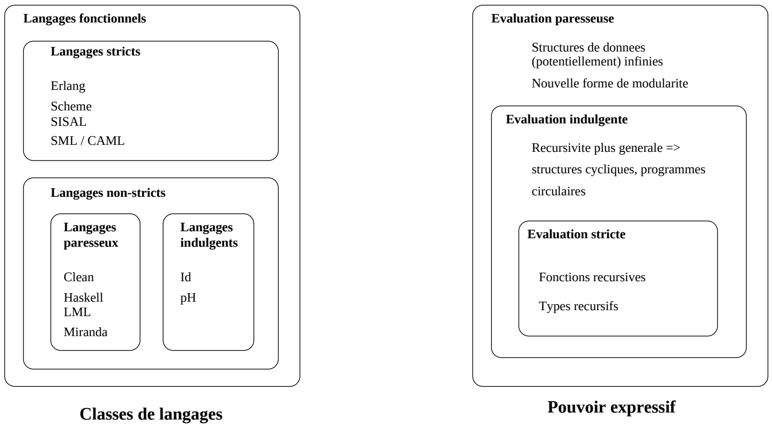

Figure 1: Les différentes classes de langages fonctionnels

Lorsqu'on dit qu'un langage est non-strict, il est donc important de préciser s'il est paresseux ou indulgent. La partie gauche de la figure 1 présente quelques langages fonctionnels connus et indique dans quelle catégorie se situe chacun des langages: comme on le voit, la plupart des langages non-stricts sont effectivement paresseux, ce qui explique peut-être la confusion existant entre évaluation non-stricte et paresse.

Quant à la partie droite de la figure 1, elle indique la relation d'inclusion existant, au niveau du pouvoir expressif, entre ces différentes classes; ainsi, tout ce qui peut être traité correctement et efficacement dans un langage indulgent peut aussi l'être dans un langage paresseux, *mais pas* l'inverse. Dans les sections qui suivent (3 et 4) nous examinons, à l'aide de petits exemples, les différences existant entre évaluation non-stricte non-paresseuse et évaluation paresseuse.

3. Évaluation non-stricte *sans* paresse

3.1. Évaluation non-stricte et récursivité

Les programmes fonctionnels contiennent fréquemment des fonctions ou des types de données définis récursivement. D'un point de vue pragmatique, la récursivité permet de référer, à l'intérieur même du corps de sa définition, à l'entité en cours de définition. Par exemple, on pourrait définir en `Haskell` un type représentant des listes d'entiers comme suit:

```
> data ListeInt  =  NilInt  |  ConsInt Int ListeInt
```

Supposons maintenant que l'on veuille utiliser le type `ListeInt` de façon à définir récursivement une *valeur*:

```
> desUns  =  ConsInt 1 desUns
```

Il s'agit ici de l'application du même principe d'utilisation d'une entité dans sa définition. Formellement, comme pour un type ou une fonction, l'entité ainsi définie peut être obtenue comme point fixe d'une équation appropriée. Ici, on cherche donc une liste qui, lorsqu'on lui ajoute un "`1`" en tête, produit une liste identique à elle-même; intuitivement, on peut voir que la solution appropriée est une liste *infinie* de 1.

Dans un langage strict, une telle définition récursive d'une valeur *ne serait pas* acceptable: les arguments d'une fonction (ou d'un constructeur) doivent être entièrement évalués *avant* que la fonction puisse être appelée. Par contre, dans un langage non-strict, une telle définition serait acceptable et ce même si le langage n'est pas paresseux. Par exemple, dans le cas du langage `pH` [16][2], la stratégie utilisée pour traiter de telles définitions consiste à *découpler* l'allocation d'une structure de données de son remplissage. En termes de mise en oeuvre, une telle définition récursive pourrait donc être interprétée comme suit:

```
> let   desUns = IAlloc ConsInt  -- Allouer une cellule ConsInt
>       desUns_0 = 1             -- Remplir la tete avec l'entier 1
>       desUns_1 = desUns        -- Remplir la queue avec l'adresse
> in                             --         de la cellule
>       desUns                   -- Retourner l'adresse de la cellule
```

Dans ce qui suit, nous présentons d'autres exemples illustrant l'expressivité permise par la récursivité plus générale associée à l'évaluation non-stricte (définition récursive de valeurs *non*-fonctionnelles).

3.2. Structures récursives non-strictes

[2]Le langage `pH` est une version non-stricte indulgente de `Haskell`; les deux ont donc la même syntaxe, `pH` possédant toutefois certaines constructions additionnelles.

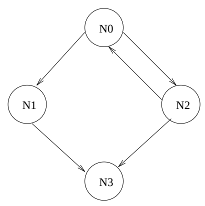

Figure 2: Graphe cyclique

Graphes cycliques. L'exemple suivant, adapté de [9], montre que des structures de données cycliques relativement complexes peuvent être construites grâce à l'évaluation non-stricte: [3]

```
> data Graphe a = Noeud a [Graph a]
> noeuds = [
>   Noeud "N0" [noeuds!!1, noeuds!!2],
>   Noeud "N1" [noeuds!!3],
>   Noeud "N2" [noeuds!!0, noeuds!!3],
>   Noeud "N3" []
> ]
```

Ici, un graphe cyclique (cf. fig. 2) est construit de façon statique et est représenté par une liste de noeuds: à chaque noeud est associé une chaîne et une liste des noeuds adjacents, lesquels sont référés directement via l'indexation de la liste `noeuds`. Il est aussi possible (cf. [9]) de construire de tels graphes de façon dynamique: pour autant que ces graphes soient finis, ceci ne requiert pas non plus la paresse.

Structures non-cycliques. Une structure définie récursivement et de façon non-stricte n'est pas nécessairement cyclique, par exemple:

```
> deux2 = (2, fst deux2)
```

L'évaluation non-stricte permet ici, comme dans le cas précédent, une dépendance entre l'une des composantes et le résultat produit.

Programmation dynamique. Des structures de données non-strictes peuvent aussi être utilisées pour résoudre des problèmes grâce à la *programmation dynamique*, stratégie qui assure que "the answer [to a subproblem] is stored and never recalculated" [1]. Un exemple simple est le suivant, qui crée la liste des `n` premiers nombres de Fibonacci et

[3] Les auteurs affirment que cet exemple n'est valide que dans un langage paresseux; ce n'est pas le cas: il suffit que le langage soit non-strict, i.e., indulgent ou paresseux.

où le calcul du $\texttt{i}^{ième}$ nombre utilise (via l'indexation de `l` avec `!!`) les valeurs déjà calculées pour les $\texttt{i-1}^{ième}$ et $\texttt{i-2}^{ième}$ nombres:

```
> fib_liste n =
>   let
>     l = 1 : 1 : [l!!(i-1) + l!!(i-2) | i <- [2..n]]
>   in
>     l
```

3.3. Programmes *circulaires*

Selon Allison [4, p. 14], un programme circulaire "[...] contains a data structure whose definition is self-referential or recursive". De tels programmes, initialement introduits par Bird [6], peuvent être utilisés, entre autres, pour éviter les traversées multiples d'une structure de données. Par exemple, supposons que l'on désire trouver les éléments d'une liste qui sont strictement supérieurs à la moyenne. La stratégie naïve consisterait à parcourir une première fois la liste pour calculer la moyenne, puis parcourir à nouveau la liste pour trouver les éléments appropriés. Dans un langage non-strict, cette tâche peut être effectuée en *une seule* passe, comme l'illustre l'exemple qui suit:

```
> trouverSupMoyenne [] =
>   error "La liste ne doit pas etre vide."
> trouverSupMoyenne xs =
>     let
>       (laMoyenne, listeSupM) = trvSupMoyenne laMoyenne xs 0 0 []
>     in
>       listeSupM
>
> trvSupMoyenne laMoyenne [] total nb listeSupM =
>     (total `div` nb, listeSupM)
> trvSupMoyenne laMoyenne (x : xs) total nb listeSupM =
>     let
>       listeSupM' = if x > laMoyenne then (x : listeSupM)
>                                     else listeSupM
>     in
>       trvSupMoyenne laMoyenne xs (total + x) (nb + 1) listeSupM'
```

La fonction `trouverSupMoyenne` prend comme argument une liste de nombres et retourne les éléments qui sont supérieurs à la moyenne. L'évaluation non-stricte est requise du fait que la valeur `laMoyenne`, utilisée dans la fonction auxiliaire `trvSupMoyenne` pour faire les comparaisons, est elle-même calculée par cette fonction, i.e., un des arguments à `trvSupMoyenne` est le résultat produit par cette fonction.

4. Évaluation paresseuse

La principale caractéristique de l'évaluation paresseuse est de n'évaluer une expression que si elle est absolument nécessaire à la production du résultat. Dans ce qui suit nous présentons diverses utilisations de cette propriété. Auparavant, il faut souligner que l'exécution, dans un langage indulgent, des programmes qui suivent pourrait, si la mise en oeuvre est *équitable*[4], produire les résultats désirés. Par contre, des calculs non-nécessaires, possiblement sans fins dans le cas de structures infinies, pourraient être générés. En d'autres mots, les deux approches produiraient le même résultat mais l'approche indulgente utiliserait, à cause des structures infinies, une quantité non-bornée de ressources.

4.1. Structures de données (potentiellement) infinies

Un avantage intéressant de l'évaluation paresseuse est qu'elle permet de manipuler des structures de données *potentiellement* infinies. Par exemple, il est possible de définir la liste des nombres naturels ou encore la liste des nombres premiers. Évidemment, pour que le programme se termine, seule une partie finie de ces listes devra être utilisée:

```
> aPartirDe n = n : (aPartirDe (n+1))
>
> naturels = aPartirDe 0
> pairs = [2 * n | n <- naturels ]
>
> estPremier n = ...   -- Determine si un nombre est premier ou non
> premiers = [ n | n <- naturels, estPremier n ]
>
> take 1000 premiers   -- Les 1000 premiers nombres premiers
```

De telles structures de données sont particulièrement intéressantes pour des problèmes de fouille [12], la paresse permettant une approche modulaire, tel qu'expliqué dans la prochaine section.

4.2. Paresse et modularité

Un avantage de la paresse est son impact positif sur la modularité des programmes [12]. Dans de nombreux cas, l'évaluation paresseuse permet d'ignorer les questions liées au contrôle et à l'ordre d'évaluation des expressions. Ceci permet par exemple, dans le programme qui suit

[4] Informellement, une mise en oeuvre est équitable ("*fair*") si elle assure le progrès constant de toute expression prête à s'évaluer.

(adapté de [12]), de découpler la production des approximations de leur examen: ce programme calcule la racine carrée d'un nombre en produisant une séquence (potentiellement) infinie d'approximations de plus en plus précise de cette racine et en fouillant cette séquence pour trouver la première approximation acceptable en terme d'erreur:

```
> prochaineApprox x a = (a + (x / a)) / 2
>
> genererApproximations x0 =
>    let
>       premiereApprox = (x0 + 1) / 2
>       approxSuivantes = map (prochaineApprox x0) approximations
>       approximations = premiereApprox : approxSuivantes
>    in
>       approximations
>
> approxAcceptable eps (a0 : a1 : rest) =
>       if abs (a0 - a1) <= eps
>         then a1
>         else approxAcceptable eps (a1 : rest)
>
> racineCarre x eps =
>       approxAcceptable eps (genererApproximations x)
```

L'évaluation paresseuse permet ici de découpler la tâche de production des approximations de la tâche de sélection du résultat. Le critère de sélection pourrait donc être changé (e.g., erreur relative plutôt qu'absolue) sans avoir à modifier la production des approximations.

4.3. Structures de données à coûts amortis

Une autre application intéressante de l'évaluation paresseuse a été présentée par Okasaki [17], où des structures de données purement fonctionnelles possédant des coûts amortis intéressants sont définies. Une analyse amortie des coûts vise à assurer que le coût moyen d'une opération est borné asymptotiquement, ce qui permet de borner le coût d'une *séquence* arbitraire d'opérations. Dans les travaux d'Okasaki, c'est la propriété de la paresse de retarder le plus possible l'évaluation d'une expression qui permet d'obtenir des coûts amortis intéressants.

4.4. Mécanisme d'exception

La paresse peut aussi être utilisée comme une forme de mécanisme de traitement des exceptions [5]. Prenons un exemple simplifié:

```
> diviser x y err = if y == 0 then err else x / y
>
> r = diviser a b (error "Division par 0 dans calcul de r.\n")
```

Dans le cas où le diviseur est 0, une valeur d'erreur peut être spécifiée par l'utilisateur de la fonction. Dans notre exemple, cette valeur est un appel à la fonction prédéfinie `error`, appel qui ne sera exécuté que si l'erreur survient effectivement.

4.5. Utilisation effective de la paresse dans divers programmes *benchmarks*

Dans le but de mieux comprendre comment la paresse est utilisée en pratique, nous avons examiné un certain nombre de programmes visant à évaluer des compilateurs paresseux ou nécessitant supposément la paresse et avons traduit ces programmes, soit en `Id` [15] (indulgent) soit en `SML` (strict).

Plusieurs exemples de programmes circulaires sont présentés dans [6, 2, 3, 4]. Les auteurs mentionnent que ces programmes ne peuvent fonctionner qu'avec l'évaluation paresseuse. Or, sauf pour un programme, tous fonctionnent correctement lorsque traduits directement en `Id` (indulgent). La seule exception est un programme, présenté dans [4], visant à construire l'ensemble des séquences irréductibles sur un alphabet: cet ensemble étant infini, la paresse est requise.

De la même façon, *aucun* des programmes utilisés dans [19] pour évaluer une nouvelle méthode d'analyse de stricticité ne nécessite l'évaluation paresseuse (sic) et tous ont pu être traduits en `Id`. [5]

Nous avons aussi traduit en SML la plupart des programmes présentés par Hartel et al. [11]. Dans certains cas, la paresse était effectivement utilisée mais, règle générale, de façon très limitée. Voici les exemples d'utilisation de la paresse qui ont pu être trouvés:

- Utilisation pour générer une liste potentiellement infinie de nombres aléatoires ou une liste potentiellement infinie d'une même constante ou d'une série répétitive de constantes. Par exemple:

  $$[0, 1, 2, 0, 1, 2, \ldots]$$

  Dans tous les cas identifiés, seul un nombre *fini* d'éléments, *fixé a priori*, était utilisé: la traduction a donc pu se faire en émulant la production d'un "*stream*" de nombres.

[5] En fait, ni la paresse ni l'évaluation non-stricte ne sont utilisés dans ces programmes; tous auraient donc pu être traduits en SML.

- Utilisation pour retarder l'évaluation d'une expression requise uniquement dans une branche d'une instruction conditionnelle, expression qui, sous certaines conditions, génère une erreur d'exécution liée à l'utilisation explicite de sélecteurs. Par exemple:

  ```
  > let
  >    x = head l
  > in
  >    if null l
  >       then ...
  >       else ... x ...
  ```

  Ici, la traduction consistait à déplacer les sélecteurs dans la branche appropriée:

  ```
  > if null l
  >    then ...
  >    else
  >       let
  >          x = head l
  >       in ...
  ```

- Utilisation de l'évaluation paresseuse pour définir des analyseurs lexicaux et syntaxiques tels que décrit dans [24, 14]. Dans cette approche de "*combinator parsing*", des fonctions d'ordre supérieur — correspondant aux différents mécanismes de définition de la syntaxe, e.g., choix, répétition, etc. — sont utilisées pour combiner les fonctions associées à chacune des règles. Dans ce cas-ci, l'évaluation paresseuse est fondamentale car elle assure que, dans le cas d'un choix, seule la bonne alternative est ultimement traitée.

Sauf pour deux programmes du dernier type, tous les programmes de Hartel et al. ont pu être traduits en SML en apportant de légères modifications à la version LML de ces programmes.

5. Un modèle LOTOS des diverses formes d'évaluation

Nous présentons maintenant un aperçu[6] d'un modèle formel et opérationnel permettant de mieux comprendre les différents modes d'évaluation dans leur incarnation pour un contexte d'évaluation *parallèle*. Dans notre modèle, aucune contrainte séquentielle d'exécution n'est imposée à moins que cela ne soit absolument nécessaire pour

[6]Pour plus de détails, cf. [23].

respecter la stratégie d'évaluation modélisée. Notre modélisation est présentée à l'aide de la notation LOTOS [7], notation de spécification algébrique de processus, donc bien adapté à la représentation des aspects parallèles vs. séquentiels.

5.1. Types auxiliaires

La première composante d'une spécification LOTOS est la spécification (algébrique) des types de données manipulées et échangées par les processus. Faute de place, nous omettrons les définitions détaillées et nous contenterons de les présenter informellement.

Séquences. Un type `Sequence` assez usuel est utilisé dans notre modèle, mais avec la particularité que ce type inclut des opérations permettant de `demander` un élément et de déterminer si un élément `estDemande`. En ce qui concerne l'accès aux éléments, l'élément à une position donnée ne peut être obtenu (via l'opération d'indexation `_[_]`) que si l'index est valide (à l'intérieur des bornes) et a été défini.

Environnement de fonctions. Un type `FEnv` est utilisé pour modéliser un ensemble de définition de fonctions. Essentiellement, il s'agit d'un dictionnaire (*mapping*) associant à un nom de fonction (`FIdent`) son arité (`nbArguments`) et sa définition, i.e., son corps (`corps`).

Langage source. Le langage pour lequel nous allons modéliser différentes formes d'évaluation est un langage purement fonctionnel de premier ordre ne manipulant que des valeurs scalaires entières. On suppose l'existence de divers opérateurs primitifs binaires mais, pour simplifier, aucun opérateur unaire.

Les expressions de notre langage sont définies par le type `Expr`. Une caractéristique de nos fonction est que leur corps sont des expressions fermées par rapport à leurs formels (des combinateurs). On suppose donc qu'une variable (`VIdent`) est dénotée par un entier compris entre 0 et $n-1$, où n est le nombre d'arguments de la fonction. Quant à un appel de fonction, il est modélisé par l'application d'une fonction (`FIdent`) à un objet de type `ArgsEffectifs`, lequel représente une séquence d'expressions (longueur = nombre d'arguments effectifs).

L'environnement d'une fonction. Comme nos fonctions sont des combinateurs, l'environnement d'une fonction (son bloc d'activation) doit simplement conserver la valeur effective des arguments formels pour l'appel de fonction considéré. Nos variables étant dénotées par des

```
process GererEnv[definir, demander, obtenir](env : Env) : exit :=
  -- Une demande ne peut etre acceptee que lorsque l'element est defini.
  demander ?i : Nat [estDefini(env, i)];
  obtenir !i !env[i];
  GererEnv[definir, demander, obtenir](env)
  []
  -- Une seule definition d'un element donne peut etre acceptee.
  definir ?i : Nat ?v : Valeur [not estDefini(env, i)];
  GererEnv[definir, demander, obtenir](definir(env, i, v))
  []
  exit
endprocess
```

Figure 3: Processus de gestion de l'environnement pour l'évaluation non-paresseuse

index, l'environnement d'une fonction sera donc une séquence de `Valeurs` (séquence d'entiers indexée par des naturels).

5.2. La modélisation de l'évaluation non-paresseuse

5.2.1. Gestion de l'environnement pour l'évaluation non-paresseuse

La figure 3 présente le processus servant à gérer l'environnement de l'activation d'une fonction dans le cas de l'évaluation non-paresseuse. Un tel processus accepte trois types de requêtes et peut être représenté graphiquement comme à la figure 4:

- `demander`: signale une demande pour obtenir la valeur d'un argument; lorsqu'acceptée, elle ne peut qu'être immédiatement suivie de l'action `obtenir`. À souligner: dans le cas non-paresseux, une demande n'est acceptée que lorsque l'élément est *déjà* défini.
- `obtenir`: permet à la fonction appelée d'obtenir la valeur d'un argument *après* l'avoir demandé.
- `definir`: permet à l'appelant de définir la valeur d'un argument.

5.2.2. Modélisation de l'évaluation stricte

Le processus de haut niveau d'évaluation est modélisé à la figure 5. Le processus `Eval` reçoit en entrée un programme, i.e., une expression (`Expr`) et un ensemble de définitions de fonctions (`FEnv`), et produit en sortie la valeur appropriée (via l'appel à `EvalExpr`).

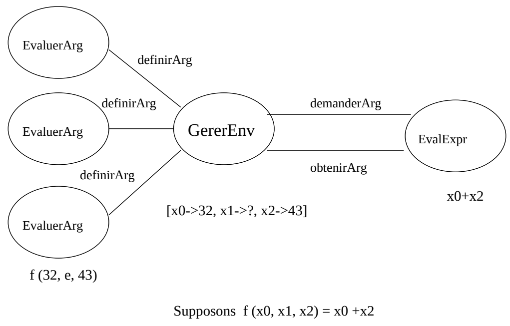

Figure 4: Fonction appelante vs. fonction appelée vs. processus de gestion de l'environnement

```
process Eval[entree, sortie] : noexit :=
  entree ?e : Expr ?fenv : FEnv;
  hide definir, demander, obtenir in
     GererEnv[definir, demander, obtenir](creer(0))
     |[demander, obtenir]|
     EvalExpr[demander, obtenir, sortie](e, fenv)
  >>
  Eval[entree, sortie]
endproc
```

Figure 5: Processus de haut niveau pour l'évaluation d'une expression

Le coeur du modèle est le processus d'évaluation d'une expression présenté à la figure 6 (`EvalExpr`). Ce processus est paramétrisé par les ports d'accès à l'environnement approprié (`demander` et `obtenir`), par le port de sortie, et par l'expression à évaluer combinée avec un ensemble de fonctions (`fenv`).

Les éléments à souligner sont les suivants:

- Lorsqu'une variable est rencontrée, une demande appropriée est envoyée (`demander`), suivie immédiatement d'une action pour `obtenir` sa valeur, laquelle est ensuite transmise en sortie. Si la valeur associée n'est pas disponible (possible dans le cas indulgent) `demander` sera suspendu jusqu'à ce qu'une valeur soit transmise par l'appelant (via `definir` de l'environnement approprié).
- Dans le cas d'une expression binaire ou conditionnelle, des processus auxiliaires sont amorcés pour évaluer les sous-expressions

```
process EvalExpr[demander, obtenir, sortie](e : Expr, fenv : FEnv) : exit :=
     [sorte(e) == Const] ->
     sortie !constante(e); exit
  [] [sorte(e) == Variable] ->
     demander !variable(e); obtenir !variable(e) ?v : Valeur; sortie !v; exit
  [] [sorte(e) == BinOp] ->
     hide sortie1, sortie2 in
        (EvalExpr[demander, obtenir, sortie1](operande1(e), fenv)
         |||
         EvalExpr[demander, obtenir, sortie2](operande2(2), fenv)
         )
        |[sortie1, sortie2]|
         sortie1 ?v1 : Valeur; sortie2 ?v2 : Valeur;
         ([operateur(e) == Plus] -> sortie !(v1 + v2); exit
          []
          [operateur(e) == Moins] -> sortie !(v1 - v2); exit
          ...
          )
  [] [sorte(e) == Si] ->
     hide sortie0 in
       EvalExpr[demander, obtenir, sortie0](condition(e), fenv)
       |[sortie0]|
       sortie0 ?cond : Valeur;
       ([cond != 0] ->
          EvalExpr[demander, obtenir, sortie](partie_si(e), fenv)
        [cond == 0] ->
          EvalExpr[demander, obtenir, sortie](partie_sinon(e), fenv)
        )
  [] [sorte(e) == Appel] ->
     let args = arguments(e) in
     let n = longueur(args) in
     hide definirArg, demanderArg, obtenirArg in
       GererEnv[definirArg, demanderArg, obtenirArg](creer(n))
       |[definirArg, demanderArg, obtenirArg]|
       (par i in [0, ..., n-1] |||
           EvaluerArg[demander, obtenir, definirArg](args[i], i, fenv)
        >>
        EvalExpr[demanderArg, obtenirArg, sortie]
                (corps(fenv, fonction(e)), fenv)
        )
endprocess
```

Figure 6: Évaluation stricte d'une expression

appropriées et les résultats requis sont reçus puis utilisés. [7]

- Dans le cas d'un appel de fonction, un nouvel environnement est créé pour gérer l'association entre les arguments effectifs et les formels (`GererEnv[...](creer(n))`). Les arguments effectifs sont

[7]Pour simplifier, on suppose une représentation des conditions "à la `C`".

```
process EvaluerArg[demander, obtenir, definirArg]
                  (e : Expr, i : Nat, fenv : FEnv) : exit :=
   hide sortie in
    EvalExpr[demander, obtenir, sortie](e, fenv)
    |[sortie]|
    sortie ?v : Valeur;  definirArg !i !v;  exit
endprocess
```

Figure 7: Évaluation d'un argument effectif

```
  [] -- Evaluation indulgente d'un appel de fonction
  [sorte(e) == Appel] ->
     let args = arguments(e) in
     let n = longueur(args) in
     hide definirArg, demanderArg, obtenirArg in
       GererEnv[definirArg, demanderArg, obtenirArg](creer(n))
       |[definirArg, demanderArg, obtenirArg]|
       (par i in [0, ..., n-1] |||
           EvaluerArg[demander, obtenir, definirArg](args[i], i, fenv)
        |||
        EvalExpr[demanderArg, obtenirArg, sortie]
                (corps(fenv, fonction(e)), fenv)
        )
```

Figure 8: Traitement d'un appel de fonction pour l'évaluation indulgente

alors évalués et définis, et ce en parallèle via les divers processus `EvaluerArg`. Comme on veut modéliser l'évaluation stricte, le corps de la fonction ne sera évalué *qu'après* que les arguments auront tous été évalués. Cette séquentialisation est modélisée par l'utilisation de "`>>`", opérateur qui assure que le processus qui suit ne pourra débuter que lorsque ce qui précède sera terminé.

La figure 7 présente le processus d'évaluation d'un argument effectif. Ici aussi un processus auxiliaire est créé; lorsque la valeur de l'expression effective est reçue (via `sortie`), elle est utilisée pour définir le paramètre formel associé (via `definirArg`).

5.2.3. Modélisation de l'évaluation indulgente

Dans le cadre présent, la modélisation de l'évaluation indulgente consiste simplement à modifier la façon dont l'appel de fonction est traité. Cette modification est présentée à la figure 8. En fait, le seul changement consiste à remplacer l'opérateur "`>>`" par l'opérateur "`|||`". En d'autres mots, dans un contexte d'évaluation parallèle tel que celui modélisé

```
[] -- Evaluation paresseuse d'un appel de fonction
[sorte(e) == Appel] ->
   let args = arguments(e) in
   let n = longueur(args) in
   hide definirArg, demanderArg, obtenirArg in
     GererEnv[demander, obtenir, definirArg, demanderArg, obtenirArg]
             (args, creer(n), fenv)
     |[demanderArg, obtenirArg]|
     EvalExpr[demanderArg, obtenirArg, sortie]
             (corps(fenv, fonction(e)), fenv)
```

Figure 9: Traitement d'un appel de fonction pour l'évaluation paresseuse

ici, l'évaluation indulgente est caractérisée par le fait que l'évaluation du corps d'une fonction s'amorce *en parallèle* avec l'évaluation de ses arguments. Si un argument n'est pas disponible au moment où il est requis dans le corps de la fonction, l'action `demander` sera bloquée jusqu'à ce que l'argument soit finalement défini par l'appelant.

5.3. Modélisation de l'évaluation paresseuse

La modélisation de l'évaluation paresseuse requiert, outre des modifications au niveau des appels, des changements à la gestion des environnements: un argument doit être associé à une expression *non*-évaluée, ainsi qu'à une référence à l'environnement approprié. Ces références se font (cf. figure 9) via les ports `demander` et `obtenir`, ports qui sont hérités par `GererEnv` du contexte de l'appelant. Ici, aucun processus n'est amorcé pour évaluer les arguments, alors que l'évaluation du corps de la fonction est amorcée aussitôt que l'appel est fait: ce sont les demandes pour les arguments qui activeront les processus appropriés d'évaluation des arguments effectifs.

La gestion de l'environnement d'une fonction se fait tel que présenté à la figure 10. Les ports supplémentaires (`demander/obtenir-Nonlocal`) sont utilisés pour référer à l'environnement statique approprié. Ici, contrairement aux approches non-paresseuses, `demander` peut *toujours* être effectué; l'effet dépend du statut de l'argument demandé:

- Si l'argument a déjà été demandé, alors aucun effet.
- Si l'argument n'a jamais été demandé, alors un processus parallèle d'évaluation est amorcé (`EvaluerArg`), tout en indiquant qu'une demande a maintenant été faite (`demander(env, i)`).

Aucune modification au traitement d'un identificateur dans `EvalExpr`

```
process GererEnv[demanderNonlocal, obtenirNonlocal, definir, demander, obtenir]
                (args : ArgsEffectifs, env : Env, fenv : FEnv) : exit :=
  demander ?i : Nat;
  ([estDemande(env, i)] ->
      GererEnv[demanderNonlocal, obtenirNonlocal, definir, demander, obtenir]
              (args, n, fenv)
   []
   [! estDemande(env, i)] ->
     EvaluerArg[demanderNonlocal, obtenirNonlocal, definir](args[i], i, fenv)
     |[definir]|
     GererEnv[demanderNonlocal, obtenirNonlocal, definir, demander, obtenir]
             (args, demander(env, i), fenv)
   )
  []
  obtenir ?i : Nat !env[i] [estDefini(env, i)];
  GererEnv[demanderNonlocal, obtenirNonlocal, definir, demander, obtenir]
          (args, env, fenv)
  []
  definir ?i : Nat ?v : Valeur [not estDefini(env, i)];
  GererEnv[demanderNonlocal, obtenirNonlocal, definir, demander, obtenir]
          (args, definir(env, i, v), fenv)
endprocess
```

Figure 10: Gestion de l'environnement (bloc d'activation) dans un contexte d'évaluation paresseuse

n'est requis. Comme précédemment, on doit `demander` puis `obtenir`. Par contre, le comportement de ces actions est légèrement différent:

- L'action `demander` peut toujours s'effectuer, i.e., ne peut jamais bloquer, contrairement aux approches non-paresseuses.
- L'action `obtenir` peut bloquer si l'élément n'est pas encore défini (si l'évaluation amorcée par la $1^{ère}$ demande n'est pas terminée).

6. Conclusion

Dans cet article, nous avons examiné les différences entre évaluation stricte, non-stricte et paresseuse, montrant, contrairement à ce qui est souvent cru, qu'un langage peut être non-strict sans être paresseux.

Le modèle LOTOS que nous avons présenté permet de mieux voir pourquoi l'évaluation indulgente semble adaptée à la programmation parallèle [15]: les arguments effectifs et le corps de la fonction sont évalués en parallèle, pouvant générer ainsi une quantité intéressante de parallélisme, en contraste avec l'approche stricte (où une séquentialisation est imposée) ou paresseuse (où les arguments ne sont évalués qu'au dernier moment[8] lorsqu'absolument requis). De plus, bien que l'approche indulgente soit non-stricte, dans un contexte parallèle, comme l'a montré le modèle, elle est beaucoup plus près de l'évaluation stricte que paresseuse, donc sans la complexité associée au gel/dégel des arguments.

Clairement, le modèle présenté est un modèle opérationnel. Ceci s'explique dans la mesure où, tel que mentionné au début de la section 4, la distinction entre évaluation indulgente et paresseuse se situe effectivement à ce niveau (ressources); au niveau dénotationnel, les deux ont la même interprétation (non-stricte). Un travail futur tentera de formaliser plus à fond ces idées.

D'un coté plus pratique, nous développons présentement un compilateur pour `pH` ciblé vers une machine parallèle. Un de nos objectifs, entre autres, est de pouvoir analyser et exécuter les programmes présentés dans [18] et tenter de mieux comprendre comment la paresse est utilisée ... ou ne l'est pas. Bien que la paresse soit parfois utile, son *omni-présence* ne nous apparaît pas nécessaire. Il serait donc intéressant d'identifier divers mécanismes d'utilisation de la paresse et, possiblement, de les inclure dans un langage non-strict ... mais non paresseux.

7. Remerciements

Cette recherche a bénéficié du support financier du PAFACC (Support aux nouveaux chercheurs, UQAM) ainsi que du CRSNG. Je tiens aussi à remercier Bruno Malenfant qui a effectué la traduction en SML des programmes de [11] à partir de leur version LML. Merci aussi aux rapporteurs anonymes pour leurs commentaires utiles et pertinents.

Bibliographie

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Co., 1974.

[2] L. Allison. Circular programs and self-referential structures. *Software — Practice and Experience*, 19(2):99–109, Feb. 1989.

[3] L. Allison. Lazy dynamic-programming can be eager. *Info. Proc. Letters*, 43(4):207–212, 1992.

[4] L. Allison. Application of recursively defined data structures. *Australian Comp. Journal*, 25(1):14–20, 1993.

[5] L. Augustsson. Communication privée. Avril 1995.

[6] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(4):239–250, 1984.

[8] Évidemment, tel que mentionné précédemment, le gel/dégel peut parfois être évité grâce à une analyse de stricticité, donc augmentant le parallélisme.

[7] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[8] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *Programs as Data Objects*, pages 42–62. Springer-Verlag, LNCS-217, 1985.

[9] C. Clack, S. Clayman, and D. Parrott. Dynamic cyclic data structures in lazy functional languages. http://www.cs.ucl.ac.uk/staff/C.Clack/papers/NotSubmitted/graph.ps, Oct. 1995.

[10] B. Goldberg. Functional programming languages. *Computing Surveys*, 28(1), 1996.

[11] P.H. Hartel and K.G. Langendoen. Benchmarking implementations of lazy functional languages. In *FPCA '93*, pages 341–349. ACM Press, 1993.

[12] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1988.

[13] R.J.M. Hughes and P. Wadler. Projections for strictness analysis. In *Functional Prog. Lang. and Comp. Arch.*, pages 385–407. Springer-Verlag, LNCS-274, 1987.

[14] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[15] R.S. Nikhil. The parallel programming language Id and its compilation for parallel machines. Computation Structures Group Memo 313, Laboratory for Computer Science, MIT, July 1990.

[16] R.S. Nikhil, Arvind, J. Hicks, S. Aditya, L. Augustsson, J.-W. Meassen, and Y. Zhou. pH language reference manual, version 1.0 — preliminary. Technical Report 369, Laboratory for Computer Science, MIT, Jan. 1995.

[17] C. Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *Symposium on Foundations of Computer Science*, pages 646–654. IEEE, Oct. 1995.

[18] W. Partain. The `nofib` benchmark suite of Haskell programs. In *Proceedings of the 1993 Glasgow Workshop on Functional Programming*. Springer-Verlag, 1993.

[19] R.C. Sekar, S. Pawagi, and I.V. Ramakrishnan. Small domains spell fast strictness analysis. In *Proc. of the ACM Symp. on Principles of Prog. Lang.*, pages 169–183, 1990.

[20] H. Sondergaard and P. Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(4):505–517, 1990.

[21] K.R. Traub. Sequential implementation of lenient programming languages. Technical Report MIT/LCS/TR-417, Laboratory for Computer Science, MIT, 1988.

[22] G. Tremblay. *Parallel Implementation of Lazy Functional Languages using Abstract Demand Propagation*. PhD thesis, School of Computer Science, McGill University, November 1994.

[23] G. Tremblay. Modélisation LOTOS des différentes formes d'évaluation fonctionnelle. Rapport interne de recherche, Dépt. d'Informatique, UQAM, Octobre 1996.

[24] P.L. Wadler. How to replace failure by a list of successes: Exception handling, backtracking and pattern matching in lazy functional languages. In *Functional Prog. Lang. and Comp. Arch.*, pages 113–128. Springer-Verlag, LNCS-201, 1985.