Langages fonctionnels: Modélisation formelle des différentes formes d'évaluation

Guy Tremblay

Octobre 1996

1 Introduction

Dans ce rapport, nous présentons une modélisation formelle des différentes formes d'évaluation rencontrées dans les langages fonctionnels — évaluation stricte, indulgente et paresseuse. Notre modélisation suppose un contexte d'évaluation entièrement *parallèle* — aucune contrainte séquentielle d'exécution ne sera donc imposée, à moins que cela soit absolument nécessaire pour respecter la stratégie d'évaluation modélisée — et est présentée à l'aide de la notation LOTOS [1, 10]. L'avantage de cette notation est qu'elle nous permettra de bien identifier comment les différentes formes d'évaluation se distinguent en termes de contraintes séquentielles d'exécution et en termes de parallélisme.

Avant de présenter les modèles formels d'évaluation, il serait bon de comprendre, intuitivement, les différentes formes d'évaluation possibles dans un langage fonctionnel. [1] Les trois principales approches sont les suivantes:

- Évaluation stricte: Dans cette approche, le mécanisme utilisé pour réaliser les appels de fonctions est *l'appel-par-valeur*. Ce mécanisme assure que les arguments effectifs utilisés pour appeler une fonction sont évalués *avant* l'appel de la fonction. Si un des arguments n'est pas défini (e.g., boucle infinie), alors le résultat de la fonction ne sera pas défini non plus.

- Évaluation paresseuse: Ici, le mécanisme utilisé pour les appels de fonctions est *l'appel-par-nécessité*. Dans cette approche, une expression utilisée comme argument est évaluée *seulement si* sa valeur est requise pour produire le résultat de la fonction. Ceci signifie qu'une expression utilisée comme argument effectif *n'est pas* évaluée avant l'appel; une telle expression est plutôt encapsulée dans une *suspension* (appelée aussi *glaçon*). Une fois l'encapsulation effectuée, l'évaluation du corps de la fonction

[1] Pour une présentation plus détaillée des concepts de base de l'évaluation paresseuse, voir [4]; dans le cas de l'évaluation indulgente, particulièrement en ce qui touche le parallélisme, voir [6]. Finalement, pour une présentation plus détaillée des différences entre évaluation stricte vs. indulgente vs. paresseuse, voir [8].

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 correspondant à l'argument requis est évaluée (le glaçon est dégelé), après quoi l'évaluation de la fonction peut se poursuivre.

Dans le contexte d'un langage fonctionnel, où l'absence d'effets de bord permet d'obtenir les propriétés de *transparence référentielle* et de "*definiteness*" [7], chaque évaluation d'une expression encapsulée dans une suspension est certaine de toujours produire la même valeur. Une valeur calculée peut donc être *mémorisée*, de façon à pouvoir être réutilisée par la suite: [2] dans l'appel-par-nécessité, une expression n'est donc évaluée *qu'au plus* une seule fois.

- Évaluation indulgente: Un avantage de l'évaluation paresseuse est qu'elle permet de définir des fonctions non-strictes. En d'autres mots, il est possible pour une fonction de produire un résultat même si un (ou plusieurs) de ses arguments effectifs n'est pas défini.

  L'évaluation indulgente permet aussi d'obtenir une évaluation non-stricte. Toutefois, contrairement à l'approche paresseuse, ceci n'est pas obtenu par le biais de la création de suspensions. Comme dans l'évaluation stricte, une expression utilisée comme argument effectif est *toujours* évaluée. Toutefois, cette évaluation ne se fait pas nécessairement avant l'appel de la fonction: dans l'évaluation indulgente, il n'existe pas *a priori* de contraintes d'ordonnancement entre l'évaluation des arguments et l'évaluation de la fonction; les seules contraintes à respecter sont celles liées au respect des dépendances de données.

  Dans un modèle d'évaluation *parallèle*, l'idée sous-jacente à l'évaluation indulgente s'exprime facilement: l'indulgence implique que le corps de la fonction et les arguments effectifs peuvent être évalués *en parallèle* et le respect des dépendances signifie que lorsqu'un argument devient requis dans le corps de la fonction, une synchronisation appropriée — entre l'argument et l'expression utisant cet argument — doit être effectuée. C'est, entres autres, cette caractéristique de l'évaluation indulgente que nous allons tenter de modéliser dans les sections qui suivent.

2 Types de données utilisés par la spécification

Dans cette section, nous présentons les types de données, spécifiés en `ACT ONE` [2], utilisés pour modéliser divers concepts et aspects du langage source ainsi que les types utilisés pour représenter les concepts servant à modéliser

[2] C'est ce dernier point qui distingue le mécanisme d'appel-par-nécessité du mécanisme d'*appel-par-nom* initialement introduit en `ALGOL-60`.

certains aspects sémantiques de ce langage. Ces types de données sont spécifiés de façon algébrique, en utilisant le langage `ACT ONE`, langage de spécification algébrique de types de données utilisé dans le cadre de LOTOS [3, 10].

2.1 La structure de `Sequence`

La figure 1 présente la spécification d'un type qui définit une structure de `Sequence`. Une particularité de ce type, qui sera utilisée lors de la modélisation de l'évaluation paresseuse, est la définition d'opérations permettant de `demander` un élément et permettant de déterminer si un élément `estDemande`. À l'exception de cet aspect, il s'agit d'un type `Sequence` classique.

2.2 Définition des types modélisant le langage

Le langage pour lequel nous allons modéliser différentes formes d'évaluation est un langage purement fonctionnel du premier ordre sans structures de données, i.e., ne manipulant que des valeurs scalaires entières. On suppose l'existence de divers opérateurs primitifs binaires mais, pour simplifier la présentation, aucun opérateur unaire. De plus, on suppose que les fonctions sont toutes des expressions fermées par rapport à leur arguments formels (des combinateurs); ceci permet que les variables (`VIdent` = `V`ariable `Ident`ificateur) soient dénotées par un entier, lequel entier est compris entre 0 et $n-1$ où n = nombre d'arguments formels de la fonction.

Un programme peut être vu comme un ensemble de définitions de fonction. C'est cet aspect qui est modélisé par le type présenté dans la figure 2, type représentant un *environnement* de fonctions. Un tel environnement est simplement un *dictionnaire* (*finite map*, *finite function*) associant à un identificateur de fonction (`FIdent`) son arité (`nbArguments`) ainsi que sa définition (`corps`).

Les types utilisés pour représenter divers concepts auxiliaires à la définition de notre langage sont présentés à la figure 3. La notion d'`ArgumentsEffectifs` est utilisée pour modéliser les appels de fonctions à plusieurs arguments: au moment de l'appel, une *séquence* formée des expressions effectives est créé et transmise à la fonction.

La figure 4 présente la syntaxe abstraite des expressions du langage. On suppose, tel que mentionné plus haut, l'existence d'opérateurs binaires ainsi que d'une primitive conditionnelle (`Si`). Quant à un appel de fonction, il est représenté par une expression d'`Appel`.

2.3 La notion d'environnement (bloc d'activation)

La figure 5 présente un type qui définit l'environnement d'une fonction comme étant une séquence de `Valeur`, ces dernières étant des nombres naturels. Un tel environnement sert à modéliser le bloc d'activation d'une fonction, donc contient la valeur des variables formelles utilisées dans le corps de la fonction (les *bindings*

```
type Sequence is Natural, Boolean
  formalsorts Element
  sort Sequence
  opns
    creer: Nat -> Sequence
    definir: Sequence, Nat, Element -> Sequence
    demander: Sequence, Nat -> Sequence
    longueur: Sequence -> Nat
    indexValide, estDefini, estDemande: Sequence, Nat -> Bool
    _[_]: Sequence, Nat -> Element
  eqns
  forall i, j, nb: Nat, e: Element, s: Sequence
   ofsort Nat
    longueur( creer(nb) ) = nb
    longueur( definir(s, i, e) ) = longueur(s)
    longueur( demander(s, i) ) = longueur(s)
   ofsort Bool
    -- Determine si j represente un index valide
    indexValide( creer(nb), j ) = (0 <= j < nb)
    indexValide( definir(s, i, e), j ) = indexValide(s, j)
    indexValide( demander(s, i), j ) = indexValide(s, j)
    -- Determine si l'element a la position j a bien ete defini
    estDefini( creer(nb), j ) = false
    indexValide(s, j) && i = j
       => estDefini( definir(s, i, e), j ) = true
    indexValide(s, j) && i ~= j
       => estDefini( definir(s, i, e), j ) = estDefini(s, j)
    indexValide(s, j)
       => estDefini( demander(s, i), j ) = estDefini(s, j)
    -- Determine si l'element a la position j a ete demande
    estDemande( creer(nb), j ) = false
    indexValide(s, j)  && i = j
       => estDemande( demander(s, i), j ) = true
    indexValide(s, j) && i ~= j
       => estDemande( demander(s, i), j ) = estDemande(s, j)
    indexValide(s, j)
       => estDemande( definir(s, i, e), j ) = estDemande(s, j)
   ofsort Element
    -- Retourne l'element a la position j
    indexValide(s, j) && i = j
       => definir(s, i, e)[j] = e
    indexValide(s, j) && i ~= j
       => definir(s, i, e)[j] = s[j]
    indexValide(s, j)
       => demander(s, i)[j] = s[j]
endtype
```

Figure 1: La notion de `Sequence`

```
type FEnvironnement is Natural, FIdent, Valeur, Expression
  sorts FEnv
  opns
    creer: FEnv
    definir: FEnv, FIdent, Nat, Expr -> FEnv
    nbArguments: FEnv, FIdent -> Nat
    corps: FEnv, FIdent -> Expr
  eqns
    forall n: Nat, f1, f2: FIdent, e: Expr, fenv: FEnv
    ofsort Nat
     f1 = f2 => nbArguments( definir(fenv, f1, n, e), f2 )
                 = n
     f1 ~= f2 => nbArguments( definir(fenv, f1, n, e), f2 )
                 = nbArguments(fenv, f2)
    ofsort Expr
     f1 = f2 => corps( definir(fenv, f1, n, e), f2 )
                 = e
     f1 ~= f2 => corps( definir(fenv, f1, n, e), f2 )
                 = corps(fenv, f2)
endtype
```

Figure 2: Environnement de fonctions

courants). Comme nos fonctions sont des combinateurs, un tel environnement contient les valeurs auxquelles sont associées les paramètres formels pour un appel donné de la fonction, sans aucun besoin de référer à l'environnement non-local.

Maintenant que nos divers types de données ont été définis, nous allons présenter, dans les sections qui suivent, les spécifications des *processus* modélisant les différentes formes d'évaluation, processus qui utiliseront et manipuleront des valeurs des types présentés plus haut.

3 La modélisation de l'évaluation stricte

3.1 Gestion de l'environnement pour l'évaluation non-paresseuse

La figure 6 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:

- `demander`: Indique une demande pour obtenir la valeur d'un argument. Une telle demande, lorsqu'acceptée, ne peut qu'être immédiatement suivie

```
type Const is Integer renamedby
  sortnames
    Const for Int
endtype

type VIdent is Natural renamedby
  sortnames
    VIdent for Nat
endtype

type Op is
  sorts Op
  opns
    Plus: -> Op
    Moins: -> Op
    ...
endtype

type ArgsEffectifs is Sequence actualizedby Expression using
  sortnames
    Expr for Element
    ArgsEffectifs for Sequence
endtype
```

Figure 3: Définitions auxiliaires pour la spécification du langage

de l'envoi de la valeur appropriée. Dans le cas non-paresseux, cette demande ne peut être acceptée que lorsque l'élément est déjà défini.

- `obtenir`: Permet au "client" (le corps de la fonction appelée) d'obtenir la valeur d'un argument formel *après* l'avoir demandé.
- `definir`: Permet au client, dans ce cas-ci l'appelant de la fonction, de définir la valeur d'un argument (association de la valeur d'une expression effective à un argument formel).

Graphiquement, ces ports peuvent donc être modélisés tels que présentés à la figure 7.

3.2 Modélisation de l'évaluation stricte

Le processus de haut niveau d'évaluation est modélisée à la figure 8. Le processus `Eval` reçoit en entrée un programme, i.e., une expression (`Expr`) et un ensemble

```
type Expression is Const, VIdent, FIdent, Op, ArgsEffectifs
  sorts SorteExpr, Expr
  opns
     exprConst: Const -> Expr
     exprVar: VIdent -> Expr
     exprBinOp: Op, Expr, Expr -> Expr
     exprSi: Expr, Expr, Expr -> Expr
     exprAppel: FIdent, ArgsEffectifs -> Expr
     sorte: Expr -> SorteExpr
     constante: Expr -> Const
     variable: Expr -> VIdent
     operateur: Expr -> Op
     operande1, operande2: Expr -> Expr
     condition, partie_si, partie_sinon: Expr -> Expr
     fonction: Expr -> FIdent
     arguments: Expr -> ArgsEffectifs
  eqns
    forall c: Const, x: VIdent, f: FIdent, op: Op,
           e, e0, e1: Expr, args: ArgsEffectifs
    ofsort SorteExpr
     sorte( exprConst(c) ) = Constante
     sorte( exprVar(x) ) = Var
     sorte( exprBinOp(op, e1, e2) ) = BinOp
     sorte( exprSi(e0, e1, e2) ) = Si
     sorte( exprAppel(f, args) ) = Appel
    ofsort Const
     constante( exprConst(c) ) = c
    ofsort VIdent
     variable( exprVar(x) ) = x
    ofsort FIdent
     fonction( exprAppel(f, args) ) = f
    ofsort Op
     operateur( exprBinOp(op, e1, e2) ) = op
    ofsort Expr
     operande1( exprBinOp(op, e1, e2) ) = e1
     operande2( exprBinOp(op, e1, e2) ) = e2
     condition( exprSi(e0, e1, e2) ) = e0
     partie_si( exprSi(e0, e1, e2) ) = e1
     partie_sinon( exprSi(e0, e1, e2) ) = e2
    ofsort ArgsEffectifs
     arguments( exprAppel(f, args) ) = args
endtype
```

Figure 4: Spécification abstraite des expressions pour le langage fonctionnel de premier ordre

```
type Valeur is Integer renamedby
  sortnames
    Valeur for Int
endtype

type Environnement is Sequence actualizedby Valeur using
  sortnames
    Valeur for Element
    Env for Sequence
endtype
```

Figure 5: Environnement (bloc d'activation) d'une fonction

```
process GererEnv[definir, demander, obtenir](env: Env) : exit :=
  -- Une demande n'est acceptee que si 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 6: Processus de gestion de l'environnement pour l'évaluation non-paresseuse

de définitions de fonctions (`FEnv`) à utiliser pour évaluer cette expression, et produit en sortie la valeur appropriée (via l'appel à `EvalExpr`). Dans le cas d'une expression représentant le programme de plus haut niveau, on sait qu'une telle expression ne pourra pas référer à aucun paramètre formel. On peut donc l'évaluer dans un environnement vide (`GererEnv[...](creer(0))`).

Le coeur du processus d'évaluation, l'évaluation d'une expression, est présenté à la figure 9 (`EvalExpr`). Ce processus est paramétrisé par les ports d'accès à l'environnement approprié (`demander` et `obtenir`) et par le port de sortie, ainsi que par la valeur de l'expression à évaluer combinée avec le programme associé (`fenv`).

Les éléments à souligner sont les suivants:

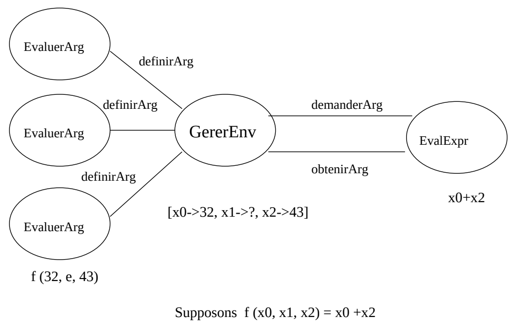

Figure 7: Fonction appelante vs. fonction appelée vs. processus de gestion de l'environnement

```
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 8: Processus de haut niveau pour l'évaluation d'une expression

- Lorsqu'une variable est rencontrée, une demande pour la variable appropriée est envoyée au processus de gestion de l'environnement associé (port `demander`), suivie immédiatement d'une action pour obtenir (i.e., recevoir) cette valeur, laquelle est ensuite simplement transmise en sortie. Si la valeur associée n'est pas encore disponible, ce qui sera possible dans le cas de l'évaluation indulgente, l'action `demander` sera suspendue jusqu'à ce qu'une définition appropriée soit transmise (par l'appelant).

- Dans le cas d'une expression binaire ou conditionnelle, un processus auxiliaire d'évaluation est amorcée pour évaluer les sous-expressions appropriées et le ou les résultats requis sont reçus via les ports de `sortie` appropriés pour être ensuite utilisés, soit pour produire le résultat de l'application de l'opérateur binaire, soit pour sélectionner la branche appropriée de la conditionnelle. (La condition de l'instruction `Si` est traitée "à la `C`", i.e., 0 dénote `Faux` et toute valeur différente de 0 dénote `Vrai`).

```
process EvalExpr[demander, obtenir, sortie](e: Expr, fenv: FEnv) : exit :=
     [sorte(e) = Constante] ->
     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]|
       (EvaluerArgs[demander, obtenir, definirArg](args, fenv, 0)
        >>
        EvalExpr[demanderArg, obtenirArg, sortie]
                (corps(fenv, fonction(e)), fenv)
        )
endprocess
```

Figure 9: Évaluation stricte d'une expression

```
process EvaluerArgs[demander, obtenir, definirArgs]
                  (args: ArgsEffectifs, fenv: Fenv, k: nat) : exit :=
  [k ~= longueur(args)] ->
     EvaluerArg[demander, obtenir, definirArgs](args[k], k, fenv)
     |||
     EvaluerArgs[demander, obtenir, definirArgs](args, fenv, k+1)
  []
  [k = longueur(args)] ->
     exit
endprocess
```

Figure 10: Évaluation des arguments effectifs

- Dans le cas d'un appel de fonction, un nouvel environnement est créé pour gérer l'association des arguments effectifs de l'appel à leur arguments formels (`GererEnv[...](creer(n))`). Cet environnement est utilisé comme suit: dans un premier temps, les arguments effectifs sont évalués et définis en parallèle via le processus `EvaluerArgs` (fig. 10); comme on veut modéliser une évaluation stricte, qui ne produira un résultat que si tous les arguments s'évaluent complètement et correctement, l'évaluation du corps de la fonction appelée ne se fera *qu'après* que les arguments effectifs auront tous été évalués. Cette séquentialisation est modélisée par l'utilisation de l'opérateur `>>`, qui assure que dans un processus composé de la forme `P >> Q`, le processus `Q` ne pourra débuter que lorsque `P` aura complètement terminé sa tâche.

  La figure 11 présente le processus permettant d'évaluer un argument effectif, i.e., d'évaluer une expression utilisée dans un appel de fonction pour transmettre une valeur à la fonction appelée. Comme pour les expressions binaires et conditionnelles, un processus auxiliaire d'évaluation est créé; quant à la valeur obtenue via le port `sortie`, elle est ensuite utilisée pour définir la valeur du paramètre formel associé dans l'environnement approprié via le port `definir`.

3.3 Modélisation de l'évaluation indulgente

Dans le cadre de l'approche utilisée jusqu'à présent, la modélisation de l'évaluation indulgente est très simple et consiste simplement à modifier la façon dont l'appel de fonction est traité. Cette modification est présentée à la figure 12. En fait, le seul changement consiste à remplacer l'opérateur "`>>`" par l'opérateur "`|||`" dans l'expression qui combine les processus d'évaluation des arguments (`EvaluerArgs`) et de la fonction (appel à `EvalExpr`). En d'autres mots, dans

```
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 11: 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]|
       (EvaluerArgs[demander, obtenir, definirArg](args, fenv, 0)
        |||
        EvalExpr[demanderArg, obtenirArg, sortie]
                (corps(fenv, fonction(e)), fenv)
        )
```

Figure 12: Traitement d'un appel de fonction pour l'évaluation indulgente

un contexte d'évaluation parallèle tel que celui modélisé ici, l'évaluation indulgente est essentiellement caractérisée par le fait que l'évaluation du corps d'une fonction appelée peut s'amorcer *en parallèle* avec l'évaluation des arguments. Si un argument n'est pas encore disponible au moment où il est requis lors de l'évaluation du corps de la fonction, le processus d'évaluation (via l'action `demander` lorsque la sorte de l'expression est une `Variable`) sera bloqué jusqu'à ce que l'argument soit défini et que la garde pour l'action `demander` soit vraie (cf. `GererEnv`).

3.4 Modélisation de l'évaluation paresseuse

La modélisation de l'évaluation paresseuse requiert, outre des modifications au niveau de la modélisation d'un appel, des changements à la façon dont est géré l'environnement. Dans un contexte d'évaluation paresseuse, où un argument n'est évalué que si cela est absolument nécessaire pour produire le résultat, un argument doit être associé à une suspension, i.e., à une expression devant être

```
[] -- 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 13: Traitement d'un appel de fonction pour l'évaluation paresseuse

évaluée combinée avec un environnement approprié pour son évaluation. Dans la figure 13, ces références à l'environnement statique approprié se font via les ports `demander` et `obtenir`, ports qui sont hérités par `GererEnv` du contexte de l'appelant, i.e., du contexte associé aux arguments effectifs au moment de l'appel. On remarque dans cette figure qu'aucun processus n'est amorcé pour évaluer les arguments et que l'évaluation du corps de la fonction est amorcée aussitôt que l'appel est fait: ce sont les demandes pour les valeurs des arguments qui, comme on le verra, créeront les processus appropriés pour l'évaluation des expressions utilisées comme arguments effectifs.

La gestion d'un environnement (bloc d'activation) d'une fonction, dans un contexte d'évaluation paresseuse, se fait tel que présenté à la figure 14. Tel qu'indiqué précédemment, des ports supplémentaires sont ici requis (`demander-Nonlocal` et `obtenirNonlocal`), de façon à référer à l'environnement statique approprié.

Contrairement aux approches d'évaluation non-paresseuse, l'action `demander` peut toujours être effectuée. Lorsqu'une telle action est effectuée pour un argument `i` donné, l'effet résultant dépend alors du statut de cet argument:

- Si cet argument a déjà été demandé, la demande n'a aucun effet.
- Si cet argument n'a jamais été demandé, alors un processus parallèle d'évaluation de l'expression appropriée est amorcée (`EvaluerArg`) tout en indiquant qu'une demande a maintenant été faite (appel à `demander`). Le demandeur restera ensuite bloqué sur une action `obtenir` jusqu'à ce qu'une définition ait été effectuée.

La modélisation de l'évaluation paresseuse ne demande aucune modification de la façon dont une référence à un identificateur est traitée dans `EvalExpr`: dans toutes les approches, on doit effectuer une action `demander` suivie d'une

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

action `obtenir`. Les différences de comportement dans le cas de l'évaluation paresseuse sont les suivantes:

- L'action `demander` peut toujours s'effectuer, i.e., ne peut jamais bloquer, contrairement aux approches non paresseuses.
- L'action `obtenir`, par contre, peut bloquer si l'élément n'est pas encore défini, i.e., si le processus d'évaluation amorcée par la première demande n'a pas encore terminé son travail donc n'a pas encore effectué l'action `definir` appropriée.

4 Conclusion

Dans ce document, nous avons tenté de modéliser les différences existant, au niveau du parallélisme, entre les diverses formes d'évaluation rencontrées dans les langages fonctionnels.

Le modèle LOTOS que nous avons présenté permet de mieux voir pourquoi l'évaluation indulgente semble adaptée à la programmation parallèle [5]: 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[3] 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 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.

Soulignons que ce travail s'inscrit dans une démarche plus large où nous tentons de motiver et justifier l'utilisation, pour la programmation parallèle, de langages fonctionnels qui soient non-stricts mais sans être paresseux. Ce choix est motivé par des travaux faits précédemment qui nous ont amené à réaliser que parallélisme et évaluation paresseuse ne font pas bon ménage [9], alors que l'évaluation indulgente semble mieux adaptée à la programmation parallèle.

Bibliographie

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

[2] Jan de Meer, Rudolf Roth, and Son Vuong. Introduction to algebraic specifications based on the language ACT ONE. *Computer Networks and ISDN Systems*, 23(5):363–392, 1992.

[3] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1* — Equations and Initial Semantics. Springer-Verlag, 1985. [QA769.9D35E37].

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

[5] 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.

[6] R.S. Nikhil and Arvind. Id: A language with implicit parallelism. Computation Structures Group Memo 305, Laboratory for Computer Science, MIT, 1990.

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

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

[8] G. Tremblay. Langages fonctionnels non-stricts: De la paresse et de l'indulgence. In *JFLA 97 – Journes Francophones des Langages Applicatifs*, pages 189–208. INRIA #16 (Collection Didactique), 1997.

[9] G. Tremblay and G.R. Gao. The impact of laziness on parallelism and the limits of strictness analysis. In A.P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, pages 119–133. Lawrence Livermore National Laboratory, CONF-9504126, 1995.

[10] K.J. Turner. *Using formal description techniques: an introduction to Estelle, LOTOS, and SDL.* Wiley series in communication and distributed systems, 1993.