Lazi / développement : translate et les notations

Emmanuel Chantréau

8 décembre 2016

Table des matières

1 Introduction

Le but de ce texte est présenter la gestion des notations du point de vue développement.

2 Définition des notations

Ce que l'on appelle « notations » est défini dans le texte des fondations mathématiques Lazi au chapitre « Lazi-Nota : les notations ».

3 Une traduction bidirectionnelle

Il est nécessaire de traduire le code source en code (pré)compilé, mais il est beaucoup moins usuel de traduire le code pré-compilé en code source. Cette fonctionnalité des logiciels Lazi est ici indispensable, voici pourquoi : Lazi est un langage qui inclut en lui sa propre définition, les manipulations et traductions du langage sont un élément naturel de Lazi, comme le sont les nombres ou les listes pour d'autres langages. Il devient alors indispensable de pouvoir convertir le langage Lazi de bas niveau en celui de plus haut niveau, ne serait-ce que pour écrire le résultat de calculs.

4 Les langages de compute et translate

4.1 Présentation

Le logiciel translate traite les langages suivants :

ComputeFm : C'est l'extensions par notations de Lazi qui est utilisée dans le logiciel compute.

SourceFm C'est l'extension par notations de Lazi qui se rapproche le plus possible du langage du code source.

OppExpression Ce n'est pas une extension de Lazi, mais un langage permettant de représenter les opérateurs et les parenthèses d'un langage basé sur le lambda calcul. Il est utilisé pour traiter la traduction des parenthèses et opérateurs.

PrintText Ce n'est pas une extension de Lazi, mais un langage du même ordre que l'html et qui permet de représenter le formatage des sources.

le langage des sources C'est le langage source Lazi écrit par l'utilisateur.

Le logiciel compute ne connaît que computeFm, sous forme de texte XML ou sous forme de structure interne. La structure interne ajoute des optimisations (partage de pointeurs, pattern des variables) qui ne changent rien au sens des données.

4.2 ComputeFm

Ce type (voir `LaziTranslate.Formulas.ComputeFm.ComputeFm` dans le code source de translate) représente le langage Lazi (Word WkBasic/Apply) plus les fonctons à une variable (Function), les mots définis (Word WkDef), les variables (Word WkVar) et la représentation des mots (WordRepr).

```
data ComputeFm =
-- ^ A Word
Word WordKind String |
-- ^ Application of a function to an argument
Apply ComputeFm ComputeFm |
-- ^ A representation of a Lazi' Word (representable by String).
WordRepr String |
-- ^ A function definition with the variable and the body.
Function Var ComputeFm
```

Var est l'information d'une String (le nom de la variable) plus l'information si le calcul de la valeur de la variable doit être partagé entre les différentes copies crées par le remplacement de la variable par la valeur. Par défaut la valeur est partagée ce qui optimise le calcul, mais dans certains cas exceptionnels (probablement jamais rencontrés en pratique) le partage engendre un calcul plus long (voir infini ?).

4.3 SourceFm

SourcesFm ressemble à ComputeFm avec des notations en plus.

```
data SourcesFm =
-- ^ like in ComputeFm
Word WordKind String |
-- ^ like in ComputeFm
Apply SourcesFm SourcesFm |
```

```
-- ^ like in ComputeFm
WordRepr String |
-- ^used to have the "Apply" like an operator (for "." in Sources)
ApplyNota |
-- ^ multi-variables function
FunctionM [VarExp] SourcesFm |
-- ^ $Let var = value, body
Let VarExp SourcesFm SourcesFm |
-- ^ A list
List [SourcesFm] |
-- ^ A dictionnary. The first String is the name of the variable representing
-- the whole dictionnary (generaly "this" is chosen).
Dict String ( M.Map String SourcesFm ) |
-- ^ A quantifier on a list
ListQuantification  Quantifier Var SourcesFm
```

Ici les fonctions sont multi-variables, plus précisément des VarExp. Un VarExp permet de représenter une correspondance par motif. Nous détaillerons les VarExp ci-dessous.

WordKind peut prendre ici une valeur supplémentaire par rapport à ComputeFm : `WkUnknown`. Elle symbolise une sorte inconnue de mot. Le code source Lazi ne spécifie pas systématiquement la sorte des mots, les règles de détermination de la sorte du mot dans le code source sont :

- Si `$Kw[x]` signifie x en tant que mot clé basique ou non défini. On détermine alors sa sorte suivant x.
- Si un mot x se trouve dans le corps d'une expression où x est une variable alors c'est une variable, sinon c'est un mot clé.

La sorte `WkUnknown` est utilisée dans les étapes intermédiaires, juste après la lecture des sources ou juste avant l'écriture.

4.4 OppExpression

Les OppExpression sont traités dans une librairie indépendante, afin d'isoler le traitement des parenthèses et des opérateurs. Du fait de la priorité et des opérateurs il est intéressant d'isoler la complexité des conversions « avec opérateurs et parenthèses/sans ».

4.5 PrintText

Ce langage n'est utilisé que comme étape de conversion entre SourceFm et le langage des sources. Il permet d'exprimer le formatage souhaité et gère les coupures de lignes, les alignements de code, les espacements etc.

4.6 Le langage des sources

Plus Lazi sera développé et plus la syntaxe des sources sera modifiable par des notations complexes. À terme translate devra être défini en Lazi pour directement intégrer la prise en compte des nouvelles notations.

5 Convertir SourceFm en ComputeFm

5.1 Présentation

Chaque notation a sa traduction qu'il suffit d'appliquer. Nous allons donc juste ici donner les traductions des notations, en les ordonnant par leur simplicité. Certaines traductions n'aboutissent pas directement, mais passent par une autre expression SourceFm plus simple.

Il pourra sembler que certaines notations sont traduites avec un affreux manque d'optimisation. En fait c'est le contraire : l'optimisation se fait dans un autre module, cela permet de ne pas s'en soucier et d'optimiser toutes les lourdeurs, peu importe leur provenance.

5.2 List

`List [ x1, ... , xn ]` → `addToList xn . ... . addToList x1 . emptyList`

5.3 Let

Si la valeur assignée à la variable était sans récursion nous pourrions traduire :
`Let ve = r, body` ⟼ `($F ve -> body) r`

Si r utilise les variables de l'expression à variable (varExp) ve, nous devons calculer r par récurrence :
ve est définit par une équation : `ve = ($F ve -> r) ve`
On a déduit : `ve = ($F ve -> r) . ($F ve -> r) . .... ($F ve -> r) . ve`
Ce qui correspond à la fonction de récurrence (voir la définition de « recurse »), on a donc :
`ve = recurse $F ve -> r`

Donc la traduction avec récursivité est :
`Let ve = r, body` ⟼ `($F ve -> body) . recurse $F ve -> r`

5.4 Dict

Remarquons que la notation des dictionnaires permet d'utiliser les variables suivantes dans les expressions

- une variable représentant tout le dictionnaire (donc c'est une définition récursive),
- des variables de même nom que les entrées et représentant les valeurs des entrées correspondantes.

```
$D/t[ a1 = v1, a2 = v2 ... ]
⟼
recurse $F t@$D[a1,a2,...] -> $L[ $P[‘a1’ , v1], $P[‘a2’ , v2] ... ]
```

5.5 ListQuantification

∀`l x/y; z` ⟼ `forAllOnList y $F x -> z`

∃`l x/y; z` ⟼ `existsOnList y $F x -> z`

5.6 Fonctions : multi vers mono variable

`$F x,y -> body` ⟼ `$F x -> $F y -> body`

5.7 Les expressions à variables

Dans les expressions ci-dessous, « v » est une variable non libre dans f. Les expressions à variables sont récursives, et par exemple le « b » ci-dessous est une expression à variable.

5.7.1 Assignation

`$F a@b -> f` ⟼ `$F a -> ($F b -> f) a`

5.7.2 Paire

`$F $P[a1,a2] -> f` ⟼ `$F v -> ($F a1,a2 -> f, pairFirst v, pairSecond v)`

5.7.3 Ajout à une liste

`($F li :l la -> f` ⟼ `$F v -> ($F la,li -> f, listLast v, listInit v)`

5.7.4 Liste

`$F $L[ a1, ... , an ] -> f`

- si n>1 : `$F $L[ a1, ... , a(n-1) ] :l an -> f`
- si n = 1 : `$F v -> ($F a1 -> f, listLast v)`
- si n = 0 : `$constantF f`

Remarquons que l'argument de la fonction peut être une liste plus grande, seuls les n derniers éléments sont utilisés. Ainsi, pour prendre le dernier élément d'une liste il suffit d'utiliser la varExp `$L[last]`.

5.7.5 Dictionnaire

```
$F $D[ n1 = a1,..., nn = an ] -> f
```
⟼
```
$F v -> ($F a1, ..., an -> f, v .d 'n1', ... , v .d 'nn')
```

6 Convertir ComputeFm en SourceFm

Nous devons reconnaître les expressions de ComputeFm décrite ci-dessus dans la conversion inverse. Les règles d'optimisation peuvent simplifier ces expressions, il faut donc aussi en ternir compte.

6.1 Règles d'optimisation

- `$F x -> y x` ⟼ `y`
- Si body utilise 'x' zero fois : `($F x -> body) y` ⟼ `body`
- Si body utilise 'x' une fois et que x n'apparaît pas dans une sous-fonction de body ayant une variable en commun avec y (sinon on aurait une collision de variables) :
  `($F x -> body) y` ⟼ `body2` où body2 = body avec x remplacé par y.
- Si body n'utilise pas 'f' : `recurse $F f -> body` ⟼ `f`
- Factorise les sous-expressions composées (celles qui nécessitent forcément un calculs) identiques (même dans les corps des fonctions) dans une formule. Par exemple :
  `f (g . x y , h . x y)` ⟼ `($F a -> f (g a, h a)) . x y` (≪ x y ≫ est factorisé) où 'a' est une variable inutilisée de l'expression initiale.