Résumé Aladyn

# Réflexion en 3 lips :

**Introduction :**

Introduit par brian smith :

3 domaines definissent la calculabilité :

Domaine syntaxique : le nombre 42

Domaine interne : en bites

Domaine réel : philosophique

Le passage de D1 a D2 ce fait par l’internalisation

Normalisation : D2 a D3

**Reflexion structurelle**

Implémentation : utilisation de Lips 1.5

Des opérateurs …

Réflexions structurelle

Cela nous montre qu’il est possible de modifier la structure du programme.

3 lips = 2lips + une infinité de niveau + lambdas reflexifs

Lambdas reflexif :

Pour le réduire il faut le « dé-reflechire » => prob une reflexion infini

Pour rendre cela efficace sans interpréteur du biveau n :

* Code compilé vers code compilé : jump vers le debut de la fonction
* Code interprété vers compilé

RPP :

Décomposition horizontal :

* Si procédure du noyau => on exécute
* Sinon on décompose en morceau de procédure

Décomposition verticale :

* Shift-up
* Shift-down

évite d’avoir des niveaux ennuyeux => évite les appels aux noyaux

L’interpreteur crée et detruit les niveaux.

3 lips = framework pour ajouter la reflexion à n’importe quel langage.

Il y a des continuation en attente => cf prochain article

Interprete minimaux en scheme

Interpretes repl

Langage scheme

Réflexivité comportemental => cf article 1

On va exécuter un interpréteur et exécuté le langage (et lui mm) donc remonté de façon récursive

L’objectif est de dire que les tours infini c’est bien, mais on montré un model infini avec un model fini.

* On parle de l’interpréteur *I* (il peut s’interpréter lui mm)

Continuation => pile d’exécution d’un programme

Utilisation de la grammaire *I*

Quote => permet d’obtenir l’information d’une valeur /symbole

2 types de procedures :

* Compound : c’est la façon dont l’interpréteur va voir …
* Primitives : les méthodes normals de scheme

Prob c’est lourd d’évaluer I => on doit rechercher le niveau et l’environnement

Amélioration de I :

* Partage de l’environnement
* Procédures réifiées : évalué au niveau de l’interpreteur

Fonction evaluate = pattern matching

Modification de l’implementation de I à l’execution de l’interpreteur

Scheme peut manipuler les continuations => call/cc (ses propres manipulations)

Analogie a setJump/longJump en C

Les avantages :

* Modifier mes implèmentation en cours d’exec
* Approche simple/ facile a utiliser
* Definition des procedures réifiées

Les limites :

* La descente est fini et la monté est lente
* Uniquement pédagogique / pas real world
* Possible car c’est un langage interprété

Meta programmation F#

Méta programmation depuis LISP

D expression : symbolique expression, un programme peut etre representaté en donnée pour etre évalué par la suite.

La transformation ce fait par la méthode « quote »

On cherchera a simplifier ce mécanisme avec les quasi-quotation

3 principes:

* Backquote : expression quasiquote
* Comma : valeur externe dans expression
* Splicing : expression dans expression

S-expression quasiquote = facilité de construction MAIS aucune SURETE

Utilisation d’un langage méta ML

* ML : fonctionnel, statiquement typé, polymorphisme
* Multistage : transformation sur le code
* First-class

Les ajouts :

* Notions de stages : chaque quote imbriqué est une étape
* Quote vérifié syntaxiquement et sémantiquement
* Fermeture des variables dans les quotes

La sureté est grandement augmentée en gardant l’expressivité grace : AVANT FSHARP