Cours N+1 : Algorithmique élémentaire (en Erlang)

March 14, 2024

Séance 8 (S+1) ; Cours N+1 : Algorithmique élémentaire (en Erlang)

0) Introduction
   - Histoire de Erlang (et de qlq langages de programmation),
   - Objectifs de Erlang
   - Réflexions sur les sources de bug (et gestion des bugs dans Erlang)
   - Solution
   - Utilisation de Erlang
1) Eléments de base Erlang séquentiel (syntaxe)
   - Structures de données (cst, Var, fonction, liste)
   - Structures de contrôle (séquence "," , raisonnement par cas ";" , récursivité)
   - Fonctions et gardes (when)
2) Sémantique (opérationnelle) : Modèle d'exécution
   - arithmétique
   - biff dans garde
   - unification asymétrique (pas de réversibilité) Motif = Expression (connue à l'exécution)
   - exécution (sans backtrack, pas de non-déterminisme)
3) Erlang vs Prolog
   - Propriétés algorithmiques classiques (Correction, Complétude, Terminaison (+Complexité))
   - Déclarativité ? (Réversibilité, Non-Déterminisme)
   - Conclusion : Prolog était simple, correct, déclaratif mais pas efficace ; erlang sera ++ simple, correct (?), ++ efficace, et parallèle (…)

1 Introduction

1.1 Aspects Historiques

Développements, par Joe Armstrong et son équipe chez Ericson, suite à la demande d'un évolution du logiciel des standards téléphoniques

- 1983, première réflexion, extension de ProLog pour la programmation concurrente dans les télécoms, évaluations de la gestion du parallélisme dans les langages de programmation
- 1986, première définition d'Erlang, emprunte aux langages fonctionnels (ML, CAML, etc.), à ProLog, et qlq autres (Pascal, Modula, Miranda), mais pas à la vague naissante des langages OO ; 1er interpréteur en ProLog, interpréteur suivant en C
- 1996, prototype d'une machine Erlang en FGPA
- 1998, abandon par Ericson, accord pour une version Open Source

En même temps, l'histoire des langages et des machines évolue vers l'objet et le parallélisme

- ~1970, Pascal, Prolog
- ~1980, Ada, C++
- ~1990, Python, Delphi
- ~2000, Processeurs grand public multicoeurs (Intel 2005, dual core)

1.2 Objectifs du langage

- Produire un (**gros**, 100 000 lignes ?) logiciel de **qualité** (industrielle, 99.999% de Hte Disponibilité, MAJ à chaud) pour le **parallélisme** (10..0 opérations simultanées) dans les télécommunications (bas/haut niveau)

1.3 Analyse des sources de bug

- parallélisme (synchrone ?? vs asynchrone)
- gestion des pointeurs ?? (pointeur null, arithmétique)
- complexité des gros programmes (programmation spaghetti ??, structures imbriquées ??, trop long à lire/écrire, typage statique ?? vs dynamique)
- effets de bords ?? (variables globales, mémoire partagée) vs principe de localité
- objets dynamiques ?? héritage ?? POO ?? vs Design Pattern (modèle/patron de conception)
- et Gestion des erreurs ! (capturer ?? vs laisser faire)

1.4 Solution Erlang

- Pas d'objet (ni héritage, ni dynamique), mais des modèles de conception
- Typage dynamique (et écriture fonctionnelle) pour améliorer la productivité (et la qualité) du code (fonctionnel à titre secondaire, car d'usage naturel)
- Programmation déclarative (raisonnement par cas, unification (expressive), récursivité), pas de mémoire partagée
- Langage orienté concurrence (à la base du langage) à base de processus légers asynchrones communicant par envoie de message
- Gestion des erreurs à part

1.5 Utilisation de Erlang

- Whatsapp
- github (erlang+python)
- gaming (chat : league of legends, call of duty)
- bd (couchDB, Amazon simpleDB)

2 Eléments de base Erlang séquentiel (syntaxe)

Rappels (pour quelques détails de plus, voir le mini-cours Erlang en 5 épisodes, sur youtube, chaine (EDenisBA) ou github en particulier les trois premiers cours sur la syntaxe, le modèle d'exécution et l'algorithmique classique)

2.1 Structures de données

- Variables (ex. V, _MaVariable, ZXY), constantes (10, "Bonjour", faux),

- Unification unique (pas de N=N+1 !)
- Listes (vide [ ], en extension [1,2,3], par premier élément et fin de liste, [E | L])
- Nouveauté (par rapport à ProLog), il y a aussi des fonctions (elles remplacent les prédicats), non seulement pour définir du code (voir la suite), mais aussi comme structure de données : une variable peut référencer une liste (V = fun () .. end), un entier ou une fonction (voir le cours sur la programmation fonctionnelle pour en savoir plus)

2.2 Structures de contrôle

- unification (matching) : attention, l'unification est asymétrique, la partie *droite* doit être connue, ou doit pouvoir être évaluée et donner un résultat connu, i.e. : elle ne doit pas contenir de variable après évaluation (la partie droite de X=Expr, ou les paramètres lors d'appels de fonction f(Exp) lorsqu'ils doivent s'unifier avec une définition de fonction f(X) -> …).
  Par suite, il n'y aura **pas de réversibilité en Erlang** (sauf à l'introduire et à le gérer explicitement/"à la main"/par programme).
- raisonnement par cas (permet de simuler les conditionnelles de l'algorithmique classique)
  - séparés par des point virgules ";" (au lieu du point Prolog) mais le dernier cas terminé par un point "." (comme en Prolog)
  - avec des conditions d'application (des **gardes**) introduites par le mot clé "when" (entre la tête de la fonction et la flèche)
  - parmi les expressions autorisées dans les gardes, certaines BIF (builtin function) et des expressions : voir doc erlang (qlq ex. : is_integer(X), abs(X), hd(L), tl(L))
  - exemple : `maximum(A,B) when A>B -> A; ...`
- récursivité (permet de simuler les boucles de l'algorithmique classique ; dans les détails, on peut ajouter la récursivité terminale, la notion d'accumulateur, mais attention à ne pas vouloir à tout pris revenir à l'algorithmique classique, on risque d'y perdre le sens des programmations déclaratives)
- décomposition de problèmes en sous-problèmes (s'ils sont isomorphes au problème initial, cela correspond à une récursivité ; sinon, cela nécessite d'introduire des fonctions supplémentaires/annexes (pour résoudre ces sous-problèmes, que l'on espère/souhaite plus petits et plus faciles à résoudre)

Exemples en fin de document

2.3 Programmes = Fonctions

En Erlang, les programmes sont des fonctions.

En prolog, c'était des prédicats, ils étaient vrais et s'exécutait avec succès (ou échouaient), mais ne donnaient pas vraiment de résultat (leur exécution avec succès était une forme de résultat positif, leur exécution avec échec, une forme de résultat négatif, mais globalement, ce n'était pas un résultat au sens des fonctions)

En Erlang, les fonctions s'exécutent (sans erreur, si possible), et donnent un résultat (résultat de la fonction).

3 Sémantique (opérationnelle) : Modèle d'exécution

- Evaluation des paramètres des expressions et fonctions en 1er

- Evaluation du corps des fonctions ensuite
- Pas de Backtrack (pas de point de choix) : le premier cas qui s'unifie avec l'appel demandé, dans l'ordre d'écriture dans le fichier, est utilisé, c'est le seul cas qui sera utilisé.
  Par suite, il n'y aura **pas de non-déterminisme en Erlang** (sauf à l'introduire et à le gérer explicitement/"à la main"/par programme)

4 Erlang vs Prolog

4.1 Propriétés algorithmiques classiques

- Correction : preuve plus compliquée car le choix d'un cas à appliquer dépend des contraintes sur les cas précédents non applicable, les pré-conditions dépendent donc d'une analyse du programme complet (ce n'est plus une preuve locale !)
- Complétude : plus facile à obtenir (trop facile), il suffit d'un cas final sans condition. Autre façon de le dire : avec un cas final non contraint, la complétude est acquise (trop facilement, attention à la correction pour des cas où le programme ne devraient pas fournir de solution à tout prix)
- Terminaison : idem ProLog (même technique, même difficulté)

4.2 Déclarativité ?

- Réversibilité : Erlang ne prévoit pas cela à la base, si c'est nécessaire, c'est à ajouter "à la main"/par programme
- Non-Déterminisme : Erlang ne prévoit pas cela à la base, si c'est nécessaire, c'est à ajouter "à la main"/par programme

4.3 Conclusion

Prolog était simple, correct, déclaratif mais pas efficace ;
Erlang sera simple++, correct (?), efficace++, et parallèle (…)

Plus de détails dans la suite du cours.

5 Exemples

Pour rappel, en ProLog, la concaténation est donnée par (par exemple) :

```
/* concatene(D,F,L) est vrai ssi la concaténation des listes D et F corespond à la liste L. */
concatene([],F,F).
concatene([E|L],F,[E|R]):-concatene(L,F,R).
```

En Erlang (avec la longueur et le maximum de deux nombres en prime) :

[20]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

%% maximum(A,B:entiers) -> M:entier, où M est le plus grand entre A et B
maximum(A,B) when A>B -> A;
maximum(_A,B) -> B.
```

```
%% longueur(L:liste) -> N:entier, où N est la longueur de L
longueur([])->0;
longueur([_E|L])->1+longueur(L).

%% concatener(A,B:listes) -> R:liste, où R est le résultat de la concatenation␣
  ↪de A et de B.
concatener([],B)->B;
concatener([E|L],B)->[E|concatener(L,B)].

main([]) ->
  io:format('~nResultat : ~p ~n',[maximum(3,1)]),
  io:format('~nResultat : ~p ~n',[longueur([1,2,3,4,5])]),
  io:format('~nResultat : ~p ~n',[concatener([1,2],[3,4,5])]).
```

```
Overwriting prog.erl
```

[19]:
```
!escript prog.erl
```

```
Resultat : 10

Resultat : 5

Resultat : [1,2,3,4,5]
```