Le langage Erlang

Robotique et Systèmes Embarqués

Samuel Tardieu

samuel.tardieu@enst.fr

École Nationale Supérieure des Télécommunications

Institut de la Francophonie pour l'Informatique

Erlang

- Langage de programmation créé par Éricsson
  - Pour les autocommutateurs (centraux téléphoniques)
  - Adapté pour les applications réparties temps-réel mou
  - Doit permettre une mise à jour de l'application sans interruption
  - Doit permettre une répartition de la charge et une migration de service
- Disponibilité
  - Open Source depuis fin 1998
  - Fonctionne sur Unix, Windows et systèmes embarqués
  - Utilise un bytecode compact et efficace et/ou du code natif

Caractéristiques

- Caractéristiques du langage
  - Langage fonctionnel
  - Parallélisation grâce à des processus concurrents
  - Faiblement typé
  - Basé sur l'unification (*pattern matching*)
- Interpréteur et outils
  - Interpréteur interactif
  - Outils de gestion graphiques
  - Base de données intégrée avec langage de requête
  - Permet de booter un nœud sur le réseau

Langage fonctionnel

- Pas de « procédure », uniquement des fonctions renvoyant quelque chose
- Les fonctions sont des objets du langage à part entière et peuvent être manipulées
- Aucune structure de boucle impérative n'existe
- Pas de limite sur la taille des entiers manipulés
- L'unification permet une expression claire:

```
fact (0) -> 1;
fact (N) -> N * fact (N-1).
```

Types de données

- Types de base
  - Atomes (commencent par une minuscule ou entre guillements simples): `foo`, `'Bar'`, `'123'`
  - Entiers: 1, 2
  - Flottants: 1.0, 2.5
- Types composites
  - Tuples: {2, 1.0, a}
  - Listes: [2.4, 1, 2, 3, [xyz, "foobar"]]
- Un caractère est représenté par un entier, une chaîne de caractères par une liste d'entiers: "abc" = [97, 98, 99]

Variables

- Une variable
  - a un nom qui commence par une majuscule
  - est accessible dans les scopes imbriqué
  - n'est liée qu'une fois
- L'unifi cation permet
  - d'affecter une valeur à une variable
  - de tester la valeur d'une variable
- Exemple:

```
A = 3.  % Affectation
A = 3.  % Unification: succès
A = 4.  % Échec
```

Modules et fonctions

- Un module
  - est déclaré par la construction `-module (`*`name`*`).`
  - déclare les fonctions exportées avec leur arité:
    `-export([f/1, g/2]).`
  - seules les fonctions exportées peuvent être appelées de l'extérieur du module

```
-module (math).
-export ([fact/1]).

fact (0) -> 1;
fact (N) -> N * fact (N-1).
```

Construction d’une fonction

- Une fonction possède
  - une partie gauche (clause) qui sera unifi ée lors de l’appel
  - une partie droite (corps)
- Une déclaration se termine par un `.` (point), une clause par un `;` (point-virgule) si d’autres clauses suivent

```
fact (0) -> 1;
fact (N) -> N * fact (N-1).
```

Récursivité terminale

- Erlang gère la récursivité terminale
- La récursivité terminale est préférable à la récursivité non-terminale (mémoire consommée en $O(1)$)
- Exemple:

```
-module (math).
-export ([fact/1]).

fact (0, A) -> A;
fact (N, A) -> fact (N-1, A*N).

fact (N)    -> fact (N, 1).
```

Gardes

- Les gardes permettent de tester les paramètres
- Une garde ne peut contenir que des opérations simples

```
-module (math).
-export ([fact/1]).

fact (0, A) -> A;
fact (N, A) -> fact (N-1, A*N).

fact (N) when integer (N), N >= 0 ->
  fact (N, 1).
```

Lambda expressions

- Une expression lambda est une fonction anonyme
- L'environnement (fermeture) est capturé
- Exemple:

```
-module (lambda).
-export ([adder/1]).

adder (N) ->
    fun (X) ->
            N + X   % "Capture" N
    end.
```

Listes

- Une liste non-vide possède
  - une tête
  - une queue (qui est une liste)
- `[1, 2, 3] = [1 | [2 | [3 | []]]]`
- Exemple:

```
length ([])      -> 0;
length ([_ | T]) -> 1 + length (T).
```

- Dans toute unification, _ peut être utilisé pour une correspondance réussissant toujours

Compréhensions

- Permet d'exprimer un filtrage sur les éléments d'une liste
- Exemple: `[X || X <- L, X > 5]` représente la liste des éléments de L supérieurs à 5
- Exemple: `[(X,Y) || X <- L1, Y <- L2]` construit le produit cartésien des listes L1 et L2
- Tri rapide (`quicksort`):

```
sort ([])     -> [];
sort ([P | L]) ->
  sort ([X || X <- L, X < P]) ++
  [P] ++
  sort ([X || X <- L, X >= P]).
```

Processus

- Un processus Erlang est une entité autonome doté d'un PID unique
- Il peut avoir un ou plusieurs noms symboliques
- Il possède une boîte à lettres
  - tout processus peut envoyer un message à un autre processus en plaçant l'information dans la boîte du destinataire
  - tout processus peut attendre, dans sa boîte, avec un *timeout* éventuel, un message qui vérifierait certaines propriétés

Exemple de processus

Compteur sérialisé:

```
start () ->
    spawn (?MODULE, counter, [1]).

counter (N) ->
    receive
        Sender -> Sender ! N
    end,
    counter (N+1).

get (PID) ->
    PID ! self (),
    receive
        Answer -> Answer
    end.
```

Processus nommé

```
start () ->
    register (cnt, spawn (?MODULE, counter, [1])).

counter (N) ->
    receive
        Sender -> Sender ! N
    end,
    counter (N+1).

get () ->
    cnt ! self (),
    receive
        Answer -> Answer
    end.
```

Limite de temps

```
start () ->
    register (cnt, spawn (?MODULE, counter, [1])).

counter (N) ->
    receive
        Sender -> Sender ! N
    after 1000 ->
            counter (N*2)
    end,
    counter (N+1).

get () ->
    cnt ! self (),
    receive
        Answer -> Answer
    end.
```