Programmation Avancée, Cours 5

Les Monades

David Baelde

ENS Paris-Saclay

19 février, 2021

La dernière fois

Monades et transformations de programmes

Une structure abstraite pour encoder diverses notions de calcul :

- $T(A)$ type des calculs retournant des valeurs de type A
- **return** : $A \to T(A)$ injecte les valeurs dans les calculs
- **bind** : $T(A) \to (A \to T(B)) \to T(B)$ permet de composer les calculs

Enrichir un langage de programmation avec un nouvel effet,
encoder des effets dans un calcul pur,
etc.

La dernière fois

Monades et transformations de programmes

Une structure abstraite pour encoder diverses notions de calcul :

- $T(A)$ type des calculs retournant des valeurs de type A
- **return** : $A \to T(A)$ injecte les valeurs dans les calculs
- **bind** : $T(A) \to (A \to T(B)) \to T(B)$ permet de composer les calculs

Enrichir un langage de programmation avec un nouvel effet,
encoder des effets dans un calcul pur,
etc.

Programmer (avec) des monades

Exemples en Haskell (type classes) et OCaml (foncteurs).

La dernière fois

Monades et transformations de programmes

Une structure abstraite pour encoder diverses notions de calcul :

- $T(A)$ type des calculs retournant des valeurs de type A
- **return** : $A \to T(A)$ injecte les valeurs dans les calculs
- **bind** : $T(A) \to (A \to T(B)) \to T(B)$ permet de composer les calculs

Enrichir un langage de programmation avec un nouvel effet,
encoder des effets dans un calcul pur,
etc.

Programmer (avec) des monades

Exemples en Haskell (type classes) et OCaml (foncteurs).

Example (`haskell/valrestr.hs`)

Pour illustrer l'intérêt de la distinction valeur/calcul.

Let's backtrack

Example (`ocaml/nondet.ml` et `ocaml/permutations.ml`)

- Monade de non-détermisme sur '*a* **list**.
- Spécialisation pour l'itération, passage en CPS.
- La monade de non-détermisme à deux continuations.

Monade de continuations

Example (`ocaml/cont.ml`)

- La monade de continuation $M(A) = (A \to R) \to R$.

Monade de continuations

Example (`ocaml/cont.ml`)

- La monade de continuation $M(A) = (A \to R) \to R$.
- L'opération de capture de la continuation courante

$$\textbf{callcc} : \big((A \to M(B)) \to M(A)\big) \to M(A)$$

$$\textbf{callcc}(\lambda k. \quad \ldots \textbf{return} \quad v$$

Monade de continuations

Example (`ocaml/cont.ml`)

- La monade de continuation $M(A) = (A \to R) \to R$.
- L'opération de capture de la continuation courante

$$\textbf{callcc} : \big((A \to M(B)) \to M(A)\big) \to M(A)$$

$$\textbf{callcc}(\lambda k. \quad \ldots \textbf{return} \quad v \ldots k \; v \ldots k \; v' \ldots)$$

- Implémentation des exemples du cours 3 sur les effets.

Modularité et monades

Example (`haskell/eval.hs`)

Write a simple, pure evaluator for your favorite kind of expression :

$$\textbf{eval} :: \textbf{Expr} \to \textbf{Int}$$

Some natural extensions :

- raise exceptions for error cases. . .

Modularité et monades

Example (`haskell/eval.hs`)

Write a simple, pure evaluator for your favorite kind of expression :

$$\textbf{eval} :: \textbf{Expr} \to \textbf{Int}$$

Some natural extensions :

- raise exceptions for error cases. . .
  $\rightsquigarrow$ use some exception monad !

Modularité et monades

Example (`haskell/eval.hs`)

Write a simple, pure evaluator for your favorite kind of expression :

$$\textbf{eval} :: \textbf{Expr} \to \textbf{Int}$$

Some natural extensions :

- raise exceptions for error cases. . .
  $\rightsquigarrow$ use some exception monad !
- increment a counter for each arithmetic operation. . .
  $\rightsquigarrow$ use some state monad !

Modularité et monades

Example (`haskell/eval.hs`)

Write a simple, pure evaluator for your favorite kind of expression :

$$\textbf{eval} :: \textbf{Expr} \to \textbf{Int}$$

Some natural extensions :

- raise exceptions for error cases. . .
  $\rightsquigarrow$ use some exception monad !
- increment a counter for each arithmetic operation. . .
  $\rightsquigarrow$ use some state monad !

How to combine both extensions ?

How to combine a state and exceptions

We know how to separately implement each monad, using for instance

$$\textbf{Maybe} = A \mapsto 1 + A \quad \texttt{and} \quad \textbf{State Int} = A \mapsto \textbf{Int} \to A \times \textbf{Int}$$

See `haskell/evalE.hs` and `haskell/evalS.hs` for details.

Which new monad ?

How to combine a state and exceptions

We know how to separately implement each monad, using for instance

$$\textbf{Maybe} = A \mapsto 1 + A \quad \texttt{and} \quad \textbf{State Int} = A \mapsto \textbf{Int} \to A \times \textbf{Int}$$

See `haskell/evalE.hs` and `haskell/evalS.hs` for details.

Which new monad ?

Two possibilities :

- $M_1(A) = \textbf{Maybe}\ (\textbf{State Int}\ A)$
- $M_2(A) = \textbf{State Int}\ (\textbf{Maybe}\ A)$

How to combine a state and exceptions

We know how to separately implement each monad, using for instance

$$\textbf{Maybe} = A \mapsto 1 + A \quad \texttt{and} \quad \textbf{State Int} = A \mapsto \textbf{Int} \to A \times \textbf{Int}$$

See `haskell/evalE.hs` and `haskell/evalS.hs` for details.

Which new monad ?

Two possibilities :

- $M_1(A) = \textbf{Maybe}\ (\textbf{State Int}\ A)$
- $M_2(A) = \textbf{State Int}\ (\textbf{Maybe}\ A)$

Two different semantics for M_2 :

- $\textbf{bind}_a\ m\ n = \lambda s.\ \ \textbf{match}\ m\ s\ \textbf{with}\ (\textbf{Just}\ v, s') \mapsto n\ v\ s'$
  $\mid (\textbf{Nothing}, s') \mapsto (\textbf{Nothing}, s')$

How to combine a state and exceptions

We know how to separately implement each monad, using for instance

$$\textbf{Maybe} = A \mapsto 1 + A \quad \texttt{and} \quad \textbf{State Int} = A \mapsto \textbf{Int} \to A \times \textbf{Int}$$

See `haskell/evalE.hs` and `haskell/evalS.hs` for details.

Which new monad ?

Two possibilities :

- $M_1(A) = \textbf{Maybe}\ (\textbf{State Int}\ A)$
- $M_2(A) = \textbf{State Int}\ (\textbf{Maybe}\ A)$

Two different semantics for M_2 :

- $\textbf{bind}_a\ m\ n = \lambda s.\ \ \textbf{match}\ m\ s\ \textbf{with}\ (\textbf{Just}\ v, s') \mapsto n\ v\ s'$
  $\mid (\textbf{Nothing}, s') \mapsto (\textbf{Nothing}, s')$
- $\textbf{bind}_b mn = \lambda s.\ \ \ldots (\textbf{Nothing}, s)$

A more systematic approach

Combining monads by hand is tedious and repetitive. It is not modular.

Idea :

A more systematic approach

Combining monads by hand is tedious and repetitive. It is not modular.

Idea : instead of implementing monads, implement monad transformers.

- From any monad M build a monad on $T\ M = A \mapsto T(M(A))$, e.g.

$$\textbf{Monad}\ m \Rightarrow \textbf{Monad}\ (\textbf{Maybe}\ m)$$

- Show that any computation in M can be lifted to TM :

$$\textbf{lift} :: \textbf{Monad}\ m \Rightarrow ma \rightarrow tma$$

Example (`transformers.hs`, `evalSE.hs`)

Doing this ourselves, then using the standard library.

Pour conclure

Résumé

- Des exemples avancés : non-déterminisme et continuations.
- Transformations de monades : composition modulaire de monades. Au passage, un excellent exemple d'utilisation des type classes.

Questions en suspens

- Quel rapport avec Leibniz ? la théorie des catégories ?
- Comment traiter les opérations spécifiques à une monade.
- ...