Compiling Monads

Alex AUVOLAT

January 12th, 2015

Outline

1 Introduction

2 Kleisli Interpreter

3 Example Monads

4 Reynolds Interpreter

5 Bind Form Reformulation

Section 1

Introduction

Objective

- Monads present an elegant formulation of programs with side-effects in terms of pure programs.
- We compile impure programs into pure monadic-form programs.

Cartesian Closed Categories

Definition

A *cartesian category* is a category $\mathcal{X}$ having a tensor product operator $\otimes : \mathcal{X} \times \mathcal{X} \to \mathcal{X}$ and an identity element I.

Definition

A *cartesian closed category* is a cartesian category $(\mathcal{X}, \otimes, I)$ equipped, for each $A \in \mathcal{X}$ of a functor $[A \to \_] : \mathcal{X} \to \mathcal{X}$ such that

$$A \times \_ \quad \dashv \quad A \to \_$$

ie there is a natural isomorphism between $\mathcal{X}\langle X \otimes Y, Z\rangle$ and $\mathcal{X}\langle Y, [X \to Z]\rangle$. Application is the counit:

$$[X \to Y] \otimes X \xrightarrow{@_Y^X} Y$$

Closed Endofunctor

In a CCC, composition and identity can be internalized as $\odot$ and 1_X.

Definition

A *closed endofunctor* is an endofunctor $T : \mathcal{X} \to \mathcal{X}$ with a family of arrows:

$$[X \to Y] \xrightarrow{map^T_{X \to Y}} [TX \to TY]$$

compatible with internal composition and internal identity:

$$\begin{aligned} map^T_{X \to Z} \circ \odot &= \odot \circ (map^T_{Y \to Z} \otimes map^T_{X \to Y}) \\ map^T_{X \to X} \circ 1_X &= 1_{TX} \end{aligned}$$

Monads

Definition

A *monad* is a triple $\mathbf{T} = (T, unit^T, join^T)$ consisting of:

- A constructor $T : \mathbf{Ob}(\mathcal{X}) \to \mathbf{Ob}(\mathcal{X})$
- Two natural transformations $id_{\mathcal{X}} \overset{unit^T}{\Longrightarrow} T$ and $TT \overset{join^T}{\Longrightarrow} T$

verifying the associativity law and the right and left unit laws.

Strong monads

Definition

A *strong monad* is a quadruple $\mathbf{T} = (T, map^T, unit^T, join^T)$ consisting of a monad $(T, unit^T, join^T)$ and a family of $\mathcal{X}$-arrows map^T that make T into a strong endofunctor, subject to the following axioms:

$$\begin{aligned} join^T_X \circ join^T_{TX} &= join^T_X \circ map^T_{TTX \to TX}(join^T_X) \\ join^T_X \circ unit^T_{TX} &= 1_{TX} \\ &= join^T_X \circ map^T_{X \to TX}(unit^T_X) \end{aligned}$$

Strong Monads: the Intuition

Strong monads can be used as a programming paradigm for the Expression of calculations with side-effects: a monadic value of type TX is a calculation that returns a value of type X but may also have side-effects such as storing state or raising exceptions.

- $unit_X^T$ transforms a pure value X into a monadic value TX.
- $map_{X \to Y}^T f$ applies a pure function $f : X \to Y$ inside the monad.
- $join^T$ is used to collapse several levels of nested monads into one single level, thus sequencing the side-effects.

The Compiled λ-Language

We are interested in compiling a small language with some basic constructs:

$$
\begin{array}{lcl}
l & ::= & \lambda(i_1, \ldots, i_n).e \\
e & ::= & c \mid i \mid l \mid e_0(e_1, \ldots, e_n) \mid \\
 & & \text{let } (i_1, \ldots, i_n) = (e_1, \ldots, e_n) \text{ in } e_0 \mid \\
 & & \text{letrec } (i_1, \ldots, i_n) = (e_1, \ldots, e_n) \text{ in } e_0
\end{array}
$$

In this language, when instantiating the interpreter with specific monads, we will add predefined monadic operators, such as get and set for the state monad.

Two Types of Interpreters

We will look at two types of interpreters:

- Kleisli interpreter: a function f whose λ-calculs type is $A \to B$ is compiled into an element of $[A \to TB]$
- Reynolds interpreter: a function $f : A \to B$ is compiled into an element of $[TA \to TB]$

Section 2

Kleisli Interpreter

The Principle

A term of λ-calculus type:

$$e \quad : \quad A \to B$$

becomes an element :

$$[\![e]\!] \quad : \quad [A \to TB]$$

for a particular monad T (parameter of the interpreter).

The Structure

$$
\begin{aligned}
\mathcal{M} &: Exp \otimes Env \to T(Val) \\
\mathcal{W} &: List(Exp) \otimes Env \to T(List(Val)) \\
Env &= Id \to Val \\
Val &= Bool + Num + String + (Val \times Val) + Proc \\
Proc &= [List(Val) \to T(Val)]
\end{aligned}
$$

The Translations

$$
\begin{aligned}
\mathcal{M}\,[\![i]\!]\,\rho &= \mathit{unit}(\rho i) \\
\mathcal{M}\,[\![\lambda(i_1,\ldots,i_n).e]\!]\,\rho &= \mathit{unit}(\lambda(v_1,\ldots,v_n).\\
&\qquad \mathcal{M}\,[\![e]\!]\,\rho[i_1 \mapsto v_1,\ldots,i_n \mapsto v_n]) \\
\mathcal{M}\,[\![e_0(e_1,\ldots,e_n)]\!]\,\rho &= \mathit{join}(\mathit{map}(\lambda f.\mathit{join}(\mathit{map}\ f\,(\mathcal{W}\,[\![e_1,\ldots,e_n]\!]\,\rho)))\\
&\qquad (\mathcal{M}\,[\![e_0]\!]\,\rho))
\end{aligned}
$$

Section 3

Example Monads

State Monad

Monad definition:

$$
\begin{aligned}
SA &= [State \to A \times State] \\
map^S_{A\to B}\ f\ \hat{a} &= \lambda\sigma.\ \textbf{let}\ (a, \sigma') = \hat{a}\,\sigma\ \textbf{in}\ (f\ a, \sigma') \\
unit^S_A\ a &= \lambda\sigma.\ (a, \sigma) \\
join^S_A\ \hat{\hat{a}} &= \lambda\sigma.\ \textbf{let}\ (\hat{a}, \sigma') = \hat{\hat{a}}\,\sigma\ \textbf{in}\ \hat{a}\,\sigma'
\end{aligned}
$$

State manipulation operators:

$$
\begin{aligned}
get, set &: List(A) \to SA \\
get\,() &= \lambda\sigma.\ (\sigma, \sigma) \\
set\ a &= \lambda\sigma.\ (nil, a)
\end{aligned}
$$

State Monad Example

```
> (add1 (get))
(1 . 0)
> (set 3)
(void . 3)
> (begin (set 3)
         (let ([x (get)])
              (begin (set 4) (* x x))))
(9 . 4)
```

Continuation Monad

Monad definition:

$$\begin{aligned} CA &= [[A \to Answer] \to Answer] \\ map^C_{A\to B}\ f\ \hat{a} &= \lambda k.\ \hat{a}\ (\lambda a.k(f\ a)) \\ unit^C_A\ a &= \lambda k.\ k\ a \\ join^C_A\ \hat{\hat{a}} &= \lambda k\ \hat{\hat{a}}\ (\lambda \hat{a}.\ \hat{a}\ k) \end{aligned}$$

Call/CC operator:

$$\begin{aligned} call/cc &: List(A) \to CA \\ call/cc\ f &= \lambda k.\ f(\lambda a.\lambda k'.\ k\ a)\ k \end{aligned}$$

State+Continuation Monad

We refine the *Answer* domain:

$$Answer = [State \to Answer']$$

New definition of get and set:

$$\begin{aligned} get\,() &= \lambda k.\lambda\sigma.\ k\ \sigma\ \sigma \\ set\ a &= \lambda k.\lambda\sigma.\ k\ nil\ a \end{aligned}$$

State+Continuation Monad Example

```
> (set (+ 10 (call/cc (lambda (k) (add1 (k 1))))))
(void . 11)
> (begin (set 3)
         (call/cc (lambda (k)
                            (begin (set 4) (k 9)))))
(9 . 4)
```

Exception Monad

Monad definition:

$$
\begin{aligned}
EA &= A + Ex \\
map^E_{A\to B}\ f\ \hat{a} &= \textbf{case}\ \hat{a}\ \textbf{of} \\
&\quad \mid Ok(a) \to Ok(f\ a) \\
&\quad \mid Ex \to Ex \\
unit^E_A\ a &= Ok(a) \\
join^E_A\ \hat{\hat{a}} &= \textbf{case}\ \hat{\hat{a}}\ \textbf{of} \\
&\quad \mid Ok(\hat{a}) \to \hat{a} \\
&\quad \mid Ex \to Ex
\end{aligned}
$$

Raise operator:

$$
\begin{aligned}
raise &: List(A) \to EA \\
raise\ () &= Ex
\end{aligned}
$$

Exception Monad Problems

We cannot implement a catch operator that has the following type:

$$catch \quad : \quad List(A) \rightarrow EA$$

Section 4

Reynolds Interpreter

The Principle

A term of λ-calculus type:

$$e \quad : \quad A \to B$$

becomes an element :

$$[\![e]\!] \quad : \quad [TA \to TB]$$

for a particular monad T (parameter of the interpreter).

The Structure

$$
\begin{aligned}
\mathcal{M} &: Exp \otimes Env \to T(Val) \\
\mathcal{W} &: List(Exp) \otimes Env \to T(List(Val)) \\
Env &= Id \to Val \\
Val &= Bool + Num + String + (Val \times Val) + Proc \\
Proc &= [T(List(Val)) \to T(Val)]
\end{aligned}
$$

The Translation

$$
\begin{aligned}
\mathcal{M}\,[\![i]\!]\,\rho &= \mathit{unit}(\rho i) \\
\mathcal{M}\,[\![\lambda(i_1,\ldots,i_n).e]\!]\,\rho &= \mathit{unit}(\lambda(t_1,\ldots,t_n).\ \mathit{join}(\\
&\qquad \mathit{map}\ (\lambda(v_1,\ldots,v_n). \\
&\qquad\qquad \mathcal{M}\,[\![e]\!]\,\rho[i_1 \mapsto v_1,\ldots,i_n \mapsto v_n]) \\
&\qquad (t_1,\ldots,t_n))) \\
\mathcal{M}\,[\![e_0(e_1,\ldots,e_n)]\!]\,\rho &= \mathit{join}(\mathit{map}\ (\lambda f.\ f(\mathcal{W}\,[\![e_1,\ldots,e_n]\!]\,\rho)) \\
&\qquad (\mathcal{M}\,[\![e_0]\!]\,\rho))
\end{aligned}
$$

New Raise and Catch Constructs

$$
\begin{array}{rcl}
raise, catch & : & E(List(A)) \to EA \\
raise\ () & = & Ex \\
catch\ \hat{a} & = & \textbf{case}\ \hat{a}\ \textbf{of} \\
 & & \mid Ex \to Ok(nil) \\
 & & \mid Ok(a) \to \hat{a}
\end{array}
$$

Exception Monad Example

```
> (+ 10 20)
(expected . 30)
> (+ 10 (raise))
(excepted . void)
> (+ 10 (begin (catch (/ (raise) 0)) 20))
(expected . 30)
```

Section 5

Bind Form Reformulation

Several Reformulations of Monads

The *join* operator can be replaced by several other operators:

$$\begin{aligned} join^T_A &: [TTA \to TA] \\ clone^T_{X,Y,Z} &: [[Y \to TZ] \to [[X \to TY] \to [X \to TZ]]] \\ ext^T_{X,Y} &: [[X \to TY] \to [TX \to TY]] \end{aligned}$$

Each operator is subject to a particular set of axioms.

The Bind Form

$$bind^T_{X,Y} \quad : \quad [TX \to [[X \to TY] \to TY]]$$

This is the form used in Haskell.

Bind Form Monads

State Monad:

$$bind^T_{A\to B}\ \hat{a}\ f \quad = \quad \lambda\sigma.\ \textbf{let}\ (a,\sigma') = \hat{a}\,\sigma\ \textbf{in}\ f\ a\ \sigma'$$

Continuation Monad:

$$bind^T_{A\to B}\ \hat{a}\ f \quad = \quad \lambda k.\ \hat{a}\ (\lambda a.\ f\ a\ k)$$

Exception Monad:

$$\begin{array}{lll} bind^T_{A\to B}\ \hat{a}\ f & = & \textbf{case}\ \hat{a}\ \textbf{of} \\ & & \mid Ok(a) \to f\ a \\ & & \mid Ex \to Ex \end{array}$$

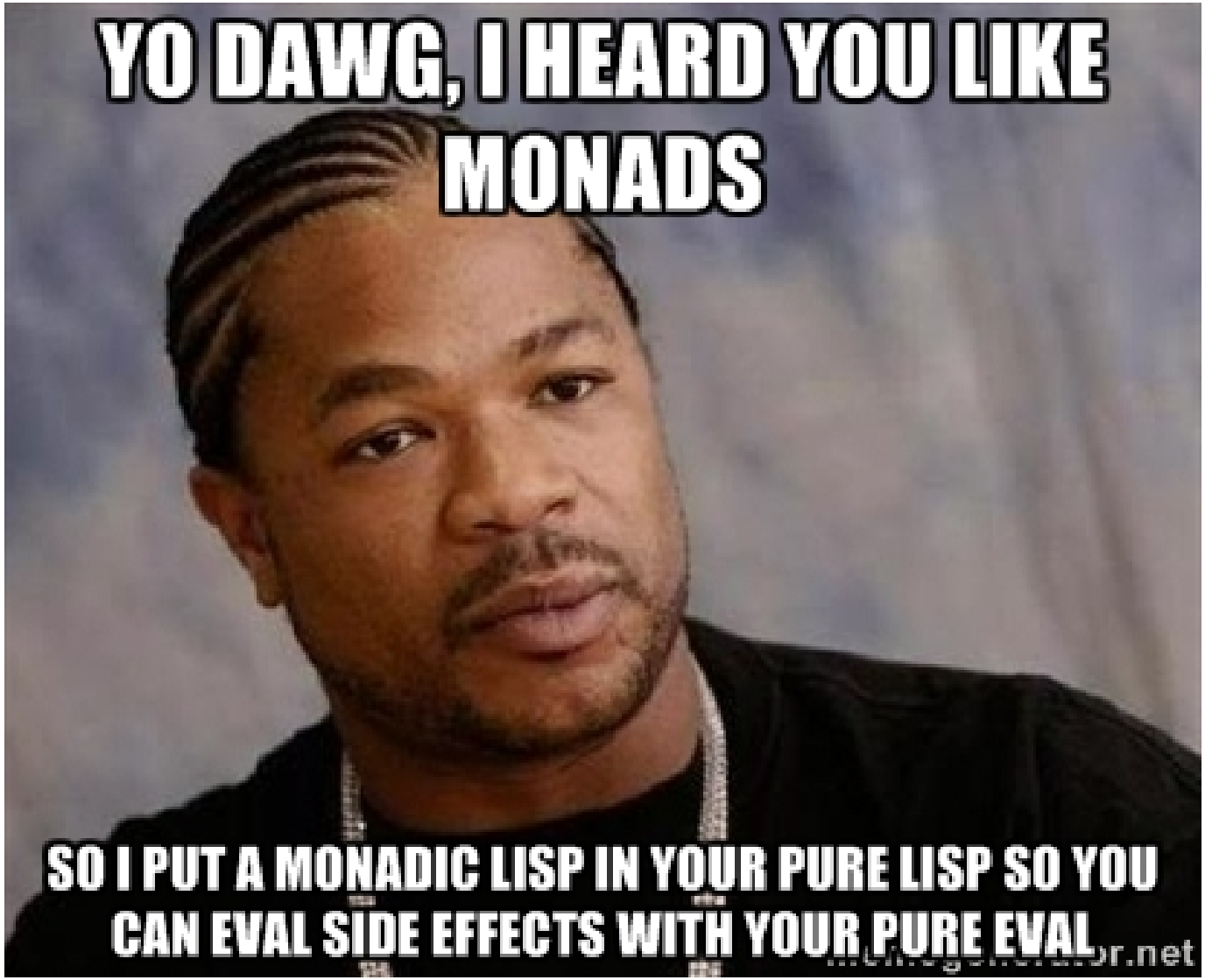
Bind Form Reynolds Interpreter

$$
\begin{aligned}
\mathcal{M}\,[\![\lambda(i_1,\ldots,i_n).e]\!]\,\rho &= \mathit{unit}(\lambda(t_1,\ldots,t_n).\ \mathit{bind}\ (t_1,\ldots,t_n) \\
&\qquad (\lambda(v_1,\ldots,v_n). \\
&\qquad\quad \mathcal{M}\,[\![e]\!]\,\rho[i_1 \mapsto v_1,\ldots,i_n \mapsto v_n])) \\
\mathcal{M}\,[\![e_0(e_1,\ldots,e_n)]\!]\,\rho &= \mathit{bind}\ (\mathcal{M}\,[\![e_0]\!]\,\rho) \\
&\qquad (\lambda f.\ f\ @\ (\mathcal{W}\,[\![e_1,\ldots,e_n]\!]\,\rho))
\end{aligned}
$$

Section 6

Conclusion

Conclusion


YO DAWG, I HEARD YOU LIKE MONADS
SO I PUT A MONADIC LISP IN YOUR PURE LISP SO YOU CAN EVAL SIDE EFFECTS WITH YOUR PURE EVAL

References

- *Compiling Monads*
  Olivier Danvy, Jürgen Koslowski, and Karoline Malmkjær, December 1991
  Department of Computing and Information Sciences, Kansas State University