

Compiling Monads

Olivier Danvy, Jürgen Koslowski, Karoline Malmkjær.
Paper presentation by Pablo Rauzy.

École normale supérieure

Categories and Lambda-Calculus presentation, February 21, 2011

λ -interpreters

- ▶ λ -interpreters have a long standing tradition in the Lisp world.
- ▶ Interpreters have a runtime price. This lead to research on how to transform interpreters to compilers.

Denotational semantics directed compiler

- ▶ Formal semantics of PL gave birth to PL mimicking them.
- ▶ Categorical semantics gave birth to monads which can be used for a modular approach to denotational semantics.
- ▶ Monads are able to model amazingly many PL features.

Monadic λ -interpreter

- ▶ Parametrized using monads.
- ▶ Able to remove imperative overhead using *partial evaluation*.
- ▶ Work on a fixed BNF.
- ▶ Extensible by introducing procedures in the initial environment.

This allow to keep the same monadic interpreter for all possible monads.

Two representations of procedures

- ▶ Kleisli interpreter
- ▶ Reynolds interpreter

- ▶ Moggi showed that formal semantics of PL could be structured using monads.
- ▶ The following instantiations are interesting:
 - ▶ a λ -expression in monadic form instantiated with the continuation monads yields the CPS counterpart of the λ -expression
 - ▶ a monadic λ -interpreter instantiated with the continuation monad yields a continuation-passing λ -interpreter.
- ▶ However, such instantiations of monads require tedious simplification steps \Rightarrow partial evaluation can help!

Definition

It's a program transformation that specialize the program with respect to part of its input, producing specialized faster programs.

$$prog : I_{static} \times I_{dynamic} \rightarrow O.$$

The partial evaluator transforms $\langle prog, I_{static} \rangle$ into $prog^* : I_{dynamic} \rightarrow O$.

Futamura projections

1. Specializing an interpreter for given source code, yielding an executable
2. Specializing the specializer for the interpreter (as applied in #1), yielding a compiler
3. Specializing the specializer for itself (as applied in #2), yielding a tool that can convert any interpreter to an equivalent compiler

Partial evaluation usage

- ▶ instantiate an interpreter with a monad,
- ▶ specialize an interpreter with respect to a program,
- ▶ transforming an interpreter from one monadic form to another

The interpreter evaluate Scheme-like expressions specified by the following BNF:

$$e ::= c \mid i \mid l \mid e_0(e_1, \dots, e_n) \mid e_1 \rightarrow e_2, e_3 \\ \mid \text{let } (i_1, \dots, i_n) = (e_1, \dots, e_n) \\ \mid \text{letrec } (i_1, \dots, i_n) = (l_1, \dots, l_n) \text{ in } e_0 \\ \mid e_0 ; \dots ; e_m$$
$$l ::= \lambda(i_1, \dots, i_n).e$$

- ▶ The interpreter represent λ -abstractions of type $A \rightarrow B$ as elements of the exponent $[A \rightarrow TB]$
- ▶ The meaning of each syntactic construct is determined using a combination of monadic construct

Kleisli interpreter

State monad

```
> (boot-kleisli state-monad)
Kleisli interpreter -- join mode
state> (add1 (get))
(1 . 0)
state> (set 3)
(void . 3)
state> (begin (set 3) (let ([x (get)]) (begin (set 4) (* x x))))
(9 . 4)
state> ^D
```

Kleisli interpreter

Continuation monad

```
> (boot-kleisli continuation-monad)
Kleisli interpreter -- join mode
continuation> (+ 10 (call/cc (lambda (k) (add1 (k 1)))))
11
continuation> ^D
```

Kleisli interpreter

Continuation + state monad

```
> (boot-kleisli continuation+state-monad)
Kleisli interpreter -- join mode
continuation+state> (set (+ 10 (call/cc (lambda (k) (add1 (k 1))))))
(void . 11)
continuation+state> (begin (set 3)
                           (call/cc (lambda (k)
                                       (begin (set 4)
                                             (k 9)))))
(9 . 4)
continuation+state> ^D
```

- ▶ The interpreter represent λ -abstractions of type $A \rightarrow B$ as elements of the exponent $[TA \rightarrow TB]$

Reynolds interpreter

Exception monad

```
> (boot-reynolds exception-monad)
Reynolds interpreter -- join mode
exception> (+ 10 20)
(expected . 30)
exception> (+ 10 (raise))
(excepted . void)
exception> (+ 10 (begin (catch (/ (raise) 0)) 20))
(expected . 30)
exception> ^D
```

Instantiating the λ -interpreters

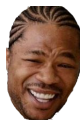
- ▶ With a monad.
- ▶ With a program ($PE\langle M, p \rangle = p_m$).
- ▶ With a λ -interpreter ($PE\langle M, I \rangle = I_m$).

Instantiating the λ -interpreters

- ▶ With a monad.
- ▶ With a program ($PE\langle M, p \rangle = p_m$).
- ▶ With a λ -interpreter ($PE\langle M, I \rangle = I_m$).

Instantiating the λ -interpreters

- ▶ With a monad.
- ▶ With a program ($PE\langle M, p \rangle = p_m$).
- ▶ With a λ -interpreter ($PE\langle M, I \rangle = I_m$).



Yo dawg!

Questions?

For teh lulz: *"It soon became clear that monads are in fact a 2-categorical concept, not restricted to the 2-category of categories, functors and natural transformations"*.