Side Effect Monad, its Equational Theory and Applications

O. Shkaravska

Institute of Cybernetics at TUT

Seminar, 2005

Outline

1. Motivation
   - Adding Imperative Features to Functional Programs
   - Previous Works

2. Our Results
   - Categorical Semantics Of View-Update Problem

Outline

Pure Languages

Pure functional languages do not subsume:

- variable assignments $x := 2$,
- field updates $x.tail := another\_list$

The example stolen from G. Plotkin's talk

$$\texttt{function} \quad Sq(x : int) : int$$
$$\texttt{return} \quad x * x$$
$$\texttt{end}$$

Meaning of Sq

$$[\![int]\!] = \mathbb{N}$$
$$[\![Sq]\!] = \mathbb{N} \rightarrow \mathbb{N}$$

Pure Languages

Pure functional languages do not subsume:

- variable assignments $x := 2$,
- field updates $x.tail := another\_list$

The example stolen from G. Plotkin's talk

$$\begin{array}{ll} \texttt{function} & Sq(x : int) : int \\ \texttt{return} & x * x \\ \texttt{end} & \end{array}$$

Meaning of Sq

$$[\![int]\!] = \mathbb{N}$$
$$[\![Sq]\!] = \mathbb{N} \to \quad \mathbb{N}$$

Pure Languages

Pure functional languages do not subsume:

- variable assignments $x := 2$,
- field updates $x.tail := another\_list$

The example stolen from G. Plotkin's talk

$$\begin{array}{ll} \texttt{function} & Sq(x : int) : int \\ \texttt{return} & x * x \\ \texttt{end} & \end{array}$$

Meaning of Sq

$[\![int]\!] = \mathbb{N}$
$[\![Sq]\!] = \mathbb{N} \rightarrow \quad \mathbb{N}$

Pure Languages

Absence of side-effects: Advantages

Convenient reasoning about pure FL, especially CBV.
Example: heap-aware type systems.
We use functional structures to verify heap consumption by a bytecode.

Absence of side-effects: Disadvantages

- One often needs to update fields ...
- CBV: unefficient usage of heap space

To Combination!

Pure Languages

Absence of side-effects: Advantages

Convenient reasoning about pure FL, especially CBV.
Example: heap-aware type systems.
We use functional structures to verify heap consumption by a bytecode.

Absence of side-effects: Disadvantages

- One often needs to update fields ...
- CBV: unefficient usage of heap space

To Combination!

Pure Languages

Absence of side-effects: Advantages

Convenient reasoning about pure FL, especially CBV.
Example: heap-aware type systems.
We use functional structures to verify heap consumption by a bytecode.

Absence of side-effects: Disadvantages

- One often needs to update fields ...
- CBV: unefficient usage of heap space

To Combination!

Pure Languages

Absence of side-effects: Advantages

Convenient reasoning about pure FL, especially CBV.
Example: heap-aware type systems.
We use functional structures to verify heap consumption by a bytecode.

Absence of side-effects: Disadvantages

- One often needs to update fields ...
- CBV: unefficient usage of heap space

To Combination!

Impure Language = Pure Language + Side Effects

Another example stolen from G. Plotkin's talk

$$\begin{array}{ll} \texttt{function} & Sq(x : int) : int \\ y := 3 & \\ \texttt{return} & x * x \\ \texttt{end} & \end{array}$$

Meaning of Sq II

$$[\![Sq]\!] = \mathbb{N} \times S \rightarrow \mathbb{N} \times S$$
$$\text{where} \quad S = \mathbb{N}^{Loc}$$

Equivalently $[\![Sq]\!] = \mathbb{N} \rightarrow (\mathbb{N} \times S)^S$

Impure Language = Pure Language + Side Effects

Another example stolen from G. Plotkin's talk

$$\texttt{function} \quad Sq(x : int) : int$$
$$y := 3$$
$$\texttt{return} \quad x * x$$
$$\texttt{end}$$

Meaning of Sq II

$$[\![Sq]\!] = \mathbb{N} \times S \to \mathbb{N} \times S$$
$$\text{where} \quad S = \mathbb{N}^{Loc}$$

Equivalently $[\![Sq]\!] = \mathbb{N} \to (\mathbb{N} \times S)^S$

Impure Language = Pure Language + Side Effects

Another example stolen from G. Plotkin's talk

```
function   Sq(x : int) : int
y := 3
return     x * x
end
```

Meaning of Sq II

$$[\![Sq]\!] = \mathbb{N} \times S \rightarrow \mathbb{N} \times S$$
$$\text{where} \quad S = \mathbb{N}^{Loc}$$

Equivalently $[\![Sq]\!] = \mathbb{N} \rightarrow (\mathbb{N} \times S)^S$

Impure Languages for Databases?

Intuition behind this Idea

The current content of the data Base is a state.

Programming with D-Bases

is a functional programming with side effects:
select and *update* operations and functions on data.

Impure Languages for Databases?

Intuition behind this Idea

The current content of the data Base is a state.

Programming with D-Bases

is a functional programming with side effects:
select and *update* operations and functions on data.

Outline

1. Motivation
   - Adding Imperative Features to Functional Programs
   - Previous Works
2. Our Results
   - Categorical Semantics Of View-Update Problem

E. Moggi: Programs with Monads

Kleisli Kategory

$Sq : \mathbb{N} \longrightarrow \mathbb{N}$
becomes
$Sq : \mathbb{N} \longrightarrow T_{state}(\mathbb{N})$,
with $T_{state}(\mathbb{N}) = (\mathbb{N} \times S)^S$

$\texttt{Div} : \mathbb{N} \longrightarrow \mathbb{N}$
becomes
$\texttt{Div} : \mathbb{N} \longrightarrow T_{Exception}(\mathbb{N})$,
with $T_{Exception}(\mathbb{N}) = \mathbb{N} + E$

$\mathrm{P} : A \longrightarrow B$
becomes
$\mathrm{P} : A \longrightarrow T(B)$

E. Moggi: Programs with Monads

Kleisli Kategory

$Sq : \mathbb{N} \rightarrow \mathbb{N}$
becomes
$Sq : \mathbb{N} \rightarrow T_{state}(\mathbb{N})$,
with $T_{state}(\mathbb{N}) = (\mathbb{N} \times S)^S$

$\mathtt{Div} : \mathbb{N} \rightarrow \mathbb{N}$
becomes
$\mathtt{Div} : \mathbb{N} \rightarrow T_{Exception}(\mathbb{N})$,
with $T_{Exception}(\mathbb{N}) = \mathbb{N} + E$

$\mathrm{P} : A \rightarrow B$
becomes
$\mathrm{P} : A \rightarrow T(B)$

E. Moggi: Programs with Monads

Kleisli Kategory

$Sq : \mathbb{N} \to \mathbb{N}$
becomes
$Sq : \mathbb{N} \to T_{state}(\mathbb{N})$,
with $T_{state}(\mathbb{N}) = (\mathbb{N} \times S)^S$

$\texttt{Div} : \mathbb{N} \to \mathbb{N}$
becomes
$\texttt{Div} : \mathbb{N} \to T_{Exception}(\mathbb{N})$,
with $T_{Exception}(\mathbb{N}) = \mathbb{N} + E$

$\mathrm{P} : A \to B$
becomes
$\mathrm{P} : A \to T(B)$

E. Moggi: Programs with Monads

Kleisli Kategory

$Sq : \mathbb{N} \to \mathbb{N}$
becomes
$Sq : \mathbb{N} \to T_{state}(\mathbb{N})$,
with $T_{state}(\mathbb{N}) = (\mathbb{N} \times S)^S$

$\mathtt{Div} : \mathbb{N} \to \mathbb{N}$
becomes
$\mathtt{Div} : \mathbb{N} \to T_{Exception}(\mathbb{N})$,
with $T_{Exception}(\mathbb{N}) = \mathbb{N} + E$

$\mathrm{P} : A \to B$
becomes
$\mathrm{P} : A \to T(B)$

Composition for Programs with Monads

Composition for "pure" programs

$P1 : A \to B \qquad P2 : B \to C$

compose to

$P1; P2 = P2 \circ P1 : \; A \to C$

Composition for monadic programs

Monadic programs = Kleisli arrows.

$P1 : A \to T(B) \qquad P2 : B \to T(C)$

compose to

$P1; P2^* = P2 \bullet P1 : \; A \to T(C)$

Additional machinery

$\_^* :: \left(f : A \to T(B)\right) \mapsto \left(f^* : T(A) \to T(B)\right)$

Composition for Programs with Monads

Composition for “pure” programs

$P1 : A \to B$ $\quad$ $P2 : B \to C$

compose to

$P1; P2 = P2 \circ P1 : \ A \to C$

Composition for monadic programs

Monadic programs = Kleisli arrows.

$P1 : A \to T(B)$ $\quad$ $P2 : B \to T(C)$

compose to

$P1; P2^{*} = P2 \bullet P1 : \ A \to T(C)$

Additional machinery

$\_^{*} :: \left(f : A \to T(B)\right) \mapsto \left(f^{*} : T(A) \to T(B)\right)$

Composition for Programs with Monads

Composition for “pure” programs

$P1 : A \rightarrow B \qquad P2 : B \rightarrow C$

compose to

$P1; P2 = P2 \circ P1 : \; A \rightarrow C$

Composition for monadic programs

Monadic programs = Kleisli arrows.

$P1 : A \rightarrow T(B) \qquad P2 : B \rightarrow T(C)$

compose to

$P1; P2^{*} = P2 \bullet P1 : \; A \rightarrow T(C)$

Additional machinery

$\_* :: \left(f : A \rightarrow T(B)\right) \mapsto \left(f^{*} : T(A) \rightarrow T(B)\right)$

Composition for Programs with Monads

Associativity

$$P3 \bullet (P2 \bullet P1) = (P3 \bullet P2) \bullet P1$$

means

$$(P1; P2^*); P3^* = P1; (P2; P3^*)^*$$

This condition is assured by

$$(f^*; g^*) = (f; g^*)^*$$
$$P1; (P2; P3^*)^* = P1; (P2^*; P3^*) = (P1; P2^*); P3^*$$

Composition for Programs with Monads

Associativity

$$P3 \bullet (P2 \bullet P1) = (P3 \bullet P2) \bullet P1$$

means

$$(P1; P2^*); P3^* = P1; (P2; P3^*)^*$$

This condition is assured by

$$(f^*; g^*) = (f; g^*)^*$$
$$P1; (P2; P3^*)^* = P1; (P2^*; P3^*) = (P1; P2^*); P3^*$$

Composition for Programs with Monads

Associativity

$$P3 \bullet (P2 \bullet P1) = (P3 \bullet P2) \bullet P1$$

means

$$(P1; P2^*); P3^* = P1; (P2; P3^*)^*$$

This condition is assured by

$$(f^*; g^*) = (f; g^*)^*$$

$$P1; (P2; P3^*)^* = P1; (P2^*; P3^*) = (P1; P2^*); P3^*$$

Composition for Programs with Monads

Associativity

$$P3 \bullet (P2 \bullet P1) = (P3 \bullet P2) \bullet P1$$

means

$$(P1; P2^*); P3^* = P1; (P2; P3^*)^*$$

This condition is assured by

$$(f^*; g^*) = (f; g^*)^*$$
$$P1; (P2; P3^*)^* = P1; (P2^*; P3^*) = (P1; P2^*); P3^*$$

Identities

Why do we need the following map?

$$\eta_A : A \to T(A)$$

(BTW, an element of $T(A)$ is called a *computation*)

As a respectable programming language our "pure", original, one, has a program-which-do-nothing:

$$\begin{array}{lll} P : A \to B & & \\ P \circ \mathbf{id}_A = P & \text{that is} & \mathbf{id}_A ; P = P \\ \mathbf{id}_B \circ P = P & \text{that is} & P ; \mathbf{id}_B = P \end{array}$$

What should be identities for the monadic langauge?

$A \rightsquigarrow A$ is $A \to T(A)$

$$\begin{array}{lll} P : A \to T(B) & & \\ P \bullet \eta_A = P & \text{that is} & \eta_A ; P^* = P \\ \eta_B \bullet P = P & \text{that is} & P ; (\eta_B)^* = P \end{array}$$

Identities

Why do we need the following map?

$$\eta_A : A \to T(A)$$

(BTW, an element of $T(A)$ is called a *computation*)
As a respectable programming language our "pure", original, one, has a program-which-do-nothing:

$$\begin{array}{lll} P : A \to B & & \\ P \circ \mathbf{id}_A = P & \text{that is} & \mathbf{id}_A ; P = P \\ \mathbf{id}_B \circ P = P & \text{that is} & P ; \mathbf{id}_B = P \end{array}$$

What should be identities for the monadic langauge?
$A \rightsquigarrow A$ is $A \to T(A)$

$$\begin{array}{lll} P : A \to T(B) & & \\ P \bullet \eta_A = P & \text{that is} & \eta_A ; P^* = P \\ \eta_B \bullet P = P & \text{that is} & P ; (\eta_B)^* = P \end{array}$$

Identities

Why do we need the following map?

$$\eta_A : A \to T(A)$$

(BTW, an element of $T(A)$ is called a *computation*)
As a respectable programming language our "pure", original, one, has a program-which-do-nothing:

$$\begin{array}{lll} P : A \to B & & \\ P \circ \mathbf{id}_A = P & \text{that is} & \mathbf{id}_A ; P = P \\ \mathbf{id}_B \circ P = P & \text{that is} & P ; \mathbf{id}_B = P \end{array}$$

What should be identities for the monadic langauge?
$A \rightsquigarrow A$ is $A \to T(A)$

$$\begin{array}{lll} P : A \to T(B) & & \\ P \bullet \eta_A = P & \text{that is} & \eta_A ; P^* = P \\ \eta_B \bullet P = P & \text{that is} & P ; (\eta_B)^* = P \end{array}$$

Kleisli Triple

Definition

$$(T,\ \eta,\ \_^*):\quad \begin{aligned} \eta_A^* &= \mathbf{id}_{T(A)} \\ \eta_A; f^* &= f \\ (f^*; g^*) &= (f; g^*)^* \end{aligned}$$

Example: Side-Effects

$T(A) = (A \times S)^S$
$\eta_A : a \mapsto \lambda s : S.\ (a, s)$
$(f : A \to T(B)) \mapsto (f^* : T(A) \to T(B))$
s. t. $f^*(c) == \lambda s : S.\ \mathtt{let}\ (a, s') = c(s)\ \mathtt{in}\ f(a)(s')$

Kleisli Triple

Definition

$$(T,\ \eta,\ \_^*):\quad \begin{array}{l} \eta_A^* = \mathbf{id}_{T(A)} \\ \eta_A; f^* = f \\ (f^*; g^*) = (f; g^*)^* \end{array}$$

Example: Side-Effects

$$T(A) = (A \times S)^S$$
$$\eta_A : a \mapsto\ \lambda s : S.\,(a, s)$$
$$(f : A \to T(B)) \mapsto (f^* : T(A) \to T(B))$$
$$\text{s. t. } f^*(c) == \lambda s : S.\ \mathtt{let}\ (a,\ s') = c(s)\ \mathtt{in}\ f(a)(s')$$

Strength

$$t_{A,\,B} : A \times T(B) \to T(A \times B)$$

Compare a “simple” `let`
and a `let` with nonlinear usage of variables.

Example: Side-Effects

$$T(A) = (A \times S)^S$$

$$t(a,\, c) == \lambda\, s : S.\ \mathtt{let}\ (b,\, s') = c(s)\ \mathtt{in}\ \big((a,\, b),\, s'\big)$$

Strength

$$t_{A,\,B} : A \times T(B) \to T(A \times B)$$

Compare a “simple” `let`
and a `let` with nonlinear usage of variables.

Example: Side-Effects

$$T(A) = (A \times S)^S$$
$$t(a,\ c) == \lambda\, s : S.\ \texttt{let}\ (b,\ s') = c(s)\ \texttt{in}\ \big((a,\ b),\ s'\big)$$

Axiomatics

Side-Effects

$S = V^{Loc}$

$$sel(upd(a,\ loc,\ v),\ loc) = v$$
$$upd(a,\ loc,\ sel(a,\ loc)) = a$$
$$upd(upd(a,\ loc,\ v),\ loc,\ v') = upd(a,\ loc,\ v')$$
$$upd(upd(a,\ loc,\ v),\ loc',\ v') = upd(upd(a,\ loc',\ v'),\ loc,\ v),$$
$$\text{where } loc \neq loc'$$

Positive Subtyping

$$get(put(c,\ a)) = a$$
$$put(c,\ get(c)) = c$$
$$put(put(c,\ a),\ a') = put(c,\ a')$$

Axiomatics

Side-Effects

$S = V^{Loc}$

$$sel(upd(a,\ loc,\ v),\ loc) = v$$
$$upd(a,\ loc,\ sel(a,\ loc)) = a$$
$$upd(upd(a,\ loc,\ v),\ loc,\ v') = upd(a,\ loc,\ v')$$
$$upd(upd(a,\ loc,\ v),\ loc',\ v') = upd(upd(a,\ loc',\ v'),\ loc,\ v),$$
$$\textit{where } loc \neq loc'$$

Positive Subtyping

$$get(put(c,\ a)) = a$$
$$put(c,\ get(c)) = c$$
$$put(put(c,\ a),\ a') = put(c,\ a')$$

View-Update Problem

Views of a database, concrete or abstract, are its **states**.
Sets of views, C and A, determine
the corresponding **state monads**, $(C \times (-))^C$ and $(A \times (-))^A$.
A total lens l is a pair of maps, *get*,

$$l \nearrow : C \to A$$

and *putback*,

$$l \searrow : C \times A \to C.$$

A lens is called *very well behaved* if its components subject to three axioms:

$$
\begin{array}{ll}
l \searrow (l \nearrow c,\ c) = c & \text{(GetPut)} \\
l \nearrow (l \searrow (a, c)) = a & \text{(PutGet)} \\
l \searrow (a', l \searrow (a, c)) = l \searrow (a', c) & \text{(PutPut)}
\end{array}
$$

View-Update Problem

Views of a database, concrete or abstract, are its **states**.

Sets of views, C and A, determine the corresponding **state monads**, $(C \times (-))^C$ and $(A \times (-))^A$.

A total lens l is a pair of maps, *get*,

$$l \nearrow : C \to A$$

and *putback*,

$$l \searrow : C \times A \to C.$$

A lens is called *very well behaved* if its components subject to three axioms:

$$l \searrow (l \nearrow c, c) = c \qquad \text{(GetPut)}$$
$$l \nearrow (l \searrow (a, c)) = a \qquad \text{(PutGet)}$$
$$l \searrow (a', l \searrow (a, c)) = l \searrow (a', c) \qquad \text{(PutPut)}$$

View-Update Problem

Views of a database, concrete or abstract, are its **states**.
Sets of views, C and A, determine
the corresponding **state monads**, $(C \times (-))^C$ and $(A \times (-))^A$.
A total lens l is a pair of maps, *get*,

$$l \nearrow : C \to A$$

and *putback*,

$$l \searrow : C \times A \to C.$$

A lens is called *very well behaved* if its components subject to three axioms:

$$l \searrow (l \nearrow c,\ c) = c \qquad \text{(GetPut)}$$
$$l \nearrow (l \searrow (a, c)) = a \qquad \text{(PutGet)}$$
$$l \searrow (a', l \searrow (a, c)) = l \searrow (a', c) \qquad \text{(PutPut)}$$

View-Update Problem

Views of a database, concrete or abstract, are its **states**.
Sets of views, C and A, determine
the corresponding **state monads**, $(C \times (-))^C$ and $(A \times (-))^A$.
A total lens l is a pair of maps, *get*,

$$l \nearrow : C \to A$$

and *putback*,

$$l \searrow : C \times A \to C.$$

A lens is called *very well behaved* if its components subject to three axioms:

$$l \searrow (l \nearrow c,\ c) = c \qquad \text{(GetPut)}$$
$$l \nearrow (l \searrow (a, c)) = a \qquad \text{(PutGet)}$$
$$l \searrow (a', l \searrow (a, c)) = l \searrow (a', c) \qquad \text{(PutPut)}$$

View-Update Problem

Views of a database, concrete or abstract, are its **states**.
Sets of views, C and A, determine
the corresponding **state monads**, $(C \times (-))^C$ and $(A \times (-))^A$.
A total lens l is a pair of maps, *get*,

$$l \nearrow : C \to A$$

and *putback*,

$$l \searrow : C \times A \to C.$$

A lens is called *very well behaved* if its components subject to three axioms:

$$l \searrow (l \nearrow c,\ c) = c \qquad \text{(GetPut)}$$
$$l \nearrow\ (l \searrow (a,\ c)) = a \qquad \text{(PutGet)}$$
$$l \searrow\ (a',\ l \searrow (a,\ c)) = l \searrow (a',\ c) \qquad \text{(PutPut)}$$

Outline

1. Motivation
   - Adding Imperative Features to Functional Programs
   - Previous Works
2. Our Results
   - Categorical Semantics Of View-Update Problem

Categorical Semantics Of View-Update Problem

Theorem. *Given a very well behaved lens l, one can construct a functor from* $Kl(T_A)$ *onto* $Kl(T_C)$.

Conclusions

- Programming over data bases may be considered as functional programming with side effects
- A very-well behavied lens defines a map of Kleisli categories
- Future Work
  - To which extend aur assumption is correct? What can it bring to data base world?
  - Very-well behavied lens defines a monad morphism

Conclusions

- Programming over data bases may be considered as functional programming with side effects
- A very-well behavied lens defines a map of Kleisli categories

- Future Work
  - To which extend aur assumption is correct? What can it bring to data base world?
  - Very-well behavied lens defines a monad morphism

Conclusions

- Programming over data bases may be considered as functional programming with side effects
- A very-well behavied lens defines a map of Kleisli categories

- Future Work
  - To which extend aur assumption is correct? What can it bring to data base world?
  - Very-well behavied lens defines a monad morphism

Conclusions

- Programming over data bases may be considered as functional programming with side effects
- A very-well behavied lens defines a map of Kleisli categories

- Future Work
  - To which extend aur assumption is correct? What can it bring to data base world?
  - Very-well behavied lens defines a monad morphism