VMλ: A Functional Calculus for Scientific Discovery

Eijiro Sumii
Hideo Bannai
University of Tokyo

Outline of the Talk

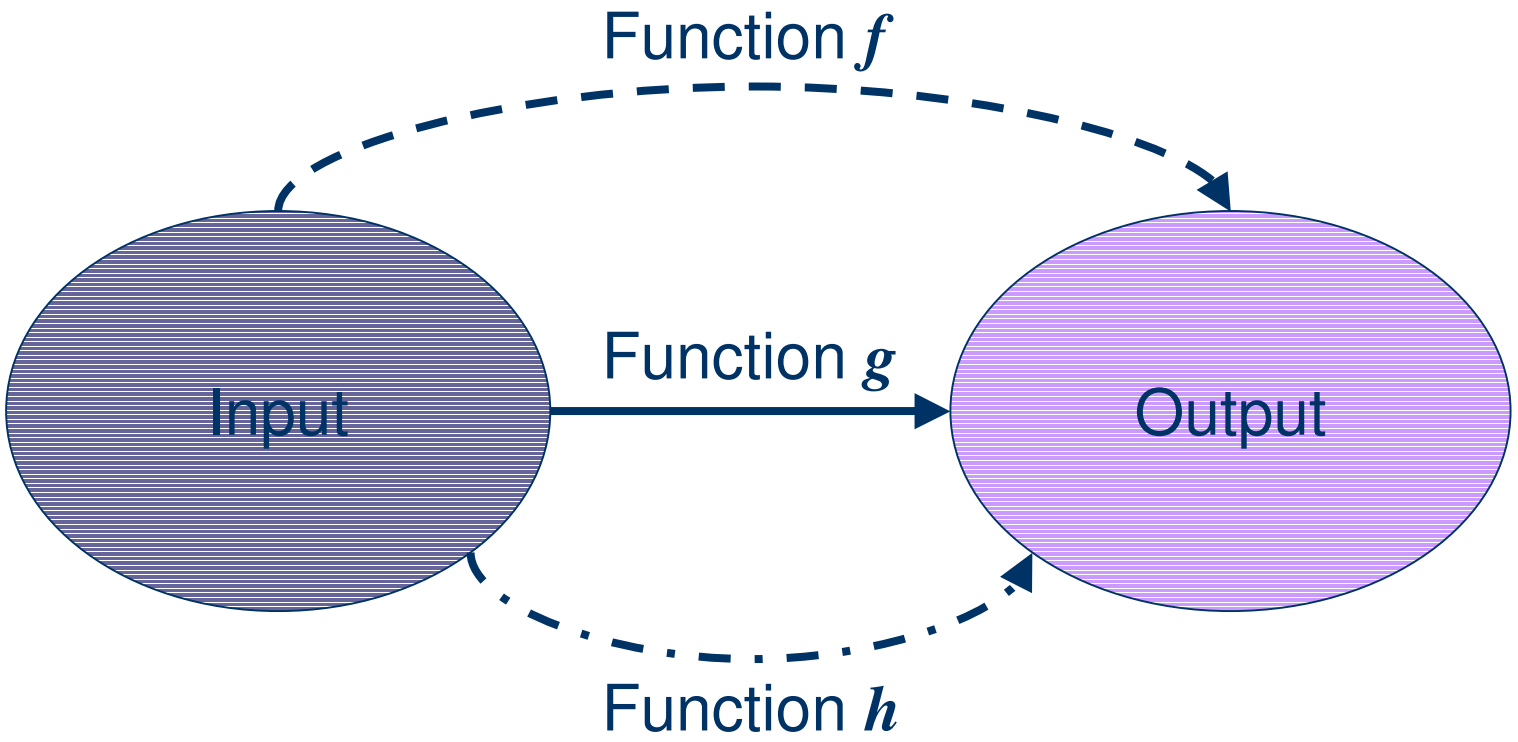
- Background
  - Discovery science and functional programming
- Simple VMλ
- VMλabl

Discovery Science [LNCS/LNAI 1532, 1721, 1967, 2226]

- A new area of computer science and artificial intelligence
- Originates in a project in Japan (http://www.i.kyushu-u.ac.jp/~arikawa/discovery/)

- Aims to carry out a unified study of computer-aided *knowledge discovery*
- Based on formal logic, machine learning, data mining, etc.

Knowledge as Functions



Knowledge discovery = finding a "good" function

Knowledge Discovery by Functional Programming

- Fully automatic knowledge discovery is too difficult
  - $\Rightarrow$ Human interaction is essential
- What kind of interface is good for manipulating functions? (simple, expressive, fast, ...)
  - — Functional programming!

Example

- let data : (input × output) list =
  [(175.4, 73.9); (167.6, 66.1); (180.8, 81.2); ...]
  - List of pairs of two data (e.g., people's height and weight)
- let fitness :
  (input → output) → (input × output) list → float = ...
  - Tells how well a given function fits given data (according to some statistical criterion)
- let affine_approx :
  (input × output) list → (input → output) = ...
  - Creates the affine function $f(x) = ax + b$ that fits given data best

How it works...

```
# let f = affine_approx data ;;
val f : input → output = <fun>
# fitness data f ;;
- : float = 0.98
```

How it works...or does it?

```
# let f = affine_approx data ;;
val f : input → output = <fun>
# fitness data f ;;
- : float = 0.98
```

Not really helpful – what *is* the function **`f`** ???

Naive Solutions

- Show the source code
  - Not very nice, because it can be too complex
- Pair the function with its representation

```
# let f' = affine_approx' data ;;
val f' :
  (float → float) × repr =
  <fun>, AffineFun(1.03, -102.8)
```

  - Works, but too troublesome to do by hand
    - In particular because of a typing problem: functions with *different representations* may need to have the *same type*

Our Solution: "Views"

Pair of a value and its representation (of an extensible data type) that remembers "how the value was created"

(≠ views for abstract types [Wadler 87])

```
# view AffineFun(a, b) = fun x → a × x + b ;;
view AffineFun of float * float : float -> float
# let v = affine_approx' data ;;
- : (float -> float) view =
  <fun> as AffineFun(1.03, -102.8)
# vmatch v with AffineFun(a, b) → (a, b) else ... ;;
- : float * float = (1.03, -102.8)
```

VML: ML Extended with Views

- Originally proposed in [Bannai et al. 2001]

- Defined in English prose only :-(

⇓

- Had problematic syntax and semantics :-(:-(
- Never implemented successfully :-(:-(:-(

VMλ: λ-calculus extended with views

- Simple VMλ: every view must take just one argument
- VMλabl: views may take any number of arguments *in any order*
  - Implemented as an extension of OCaml/OLabl

Outline of the Talk

- Background
  - Discovery science and functional programming
- Simple VMλ
- VMλabl

Syntax of Simple VMλ

M (term) ::= ... (standard λ-terms)

| view V{x} = M_1 in M_2 (view definition)

| V (view constructor)

| M_1{M_2} (view application)

| vmatch M_1 with V{x} ⇒ M_2 else M_3 (view matching)

| valof M (view destruction)

Semantics of Simple VMλ (1/2)

v (value) ::= ... (standard λ-values)
| ⟨ε; V{x} = M⟩ (view constructor closure)
| $V\{v_1\} = v_2$ (view)

$$\frac{V' \text{ fresh} \quad \mathcal{E}, V \mapsto \langle \mathcal{E}; V'\{x\} = M_1 \rangle \vdash M_2 \Downarrow v}{\mathcal{E} \vdash \texttt{view}\ V\{x\} = M_1\ \texttt{in}\ M_2 \Downarrow v}\text{(E-VDef)}$$

$$\frac{\mathcal{E} \vdash M_1 \Downarrow \langle \mathcal{E}'; V\{x\} = M' \rangle \quad \mathcal{E} \vdash M_2 \Downarrow v \quad \mathcal{E}', x \mapsto v \vdash M' \Downarrow v'}{\mathcal{E} \vdash M_1\{M_2\} \Downarrow V\{v\} = v'}\text{(E-VApp)}$$

Semantics of Simple VMλ (2/2)

v (value) ::= ... (standard λ-values)

| ⟨ε; V{x} = M⟩ (view constructor closure)

| $V\{v_1\} = v_2$ (view)

$$\frac{\mathcal{E} \vdash M_1 \Downarrow V'\{v'\} = _ \quad \mathcal{E}(V) = \langle _; V'\{_\} = _ \rangle \quad \mathcal{E}, x \mapsto v' \vdash M_2 \Downarrow v}{\mathcal{E} \vdash \texttt{vmatch}\ M_1\ \texttt{with}\ V\{x\} \Rightarrow M_2\ \texttt{else}\ M_3 \Downarrow v}\text{(E-VMatch-Succ)}$$

$$\frac{\mathcal{E} \vdash M \Downarrow _\{_\} = v}{\mathcal{E} \vdash \texttt{valof}\ M \Downarrow v}\text{(E-ValOf)}$$

Type System of Simple VMλ (1/2)

τ (type) ::= ... (standard λ-types)
| view{τ_1}τ_2 (view constructor type)
| view{ }τ (view type)

$$\frac{\Gamma, x : \tau \vdash M_1 : \tau_1 \quad \Gamma, V : \mathtt{view}\{\tau\}\tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \mathtt{view}\ V\{x\} = M_1\ \mathtt{in}\ M_2 : \tau_2}(\text{T-VDef})$$

$$\frac{\Gamma \vdash M_1 : \mathtt{view}\{\tau\}\tau' \quad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1\{M_2\} : \mathtt{view}\{\}\tau'}(\text{T-VApp})$$

Type System of Simple VMλ (2/2)

τ (type) ::= ... (standard λ-types)
| view{τ_1}τ_2 (view constructor type)
| view{ }τ (view type)

$$\frac{\Gamma(V) = \texttt{view}\{\tau\}\tau_1 \quad \Gamma \vdash M_1 : \texttt{view}\{\}\tau_1 \quad \Gamma, x : \tau \vdash M_2 : \tau_2 \quad \Gamma \vdash M_3 : \tau_2}{\Gamma \vdash \texttt{vmatch}\ M_1\ \texttt{with}\ V\{x\} \Rightarrow M_2\ \texttt{else}\ M_3 : \tau_2}(\text{T-VMatch})$$

$$\frac{\Gamma \vdash M : \texttt{view}\{\}\tau}{\Gamma \vdash \texttt{valof}\ M : \tau}(\text{T-ValOf})$$

Type Soundness

If $\vdash M : \tau$,

then $\vdash M \not\Downarrow error$.

Outline of the Talk

- Background
  - Discovery science and functional programming
- Simple VMλ
- VMλabl

Partial Application of Multiple-Argument Views: The Problem

- *Partial application* of functions is a convenient feature of higher-order functional languages

...but does not extend to views *in a naive way*

Example (the originally proposed approach):

```
view V{x, y, z} = ... in
   let v' = fun x → fun z → V{x, 1 + 2, z} in
   vmatch v' with V{_, y', _} → ...
         (* forces unnatural evaluation of 1 + 2 *)
```

Our Solution: VMλabl

Use *labeled arguments* [Garrigue & Ait-Kaci 94]

view V{ℓ_x = x; ℓ_y = y; ℓ_z = z} = ... in
  let v' = V{ℓ_y = 1 + 2} in
    (* natural to evaluate 1 + 2 here *)
  vmatch v' with V{ℓ_y = y'} → ...

Syntax of VMλabl

M (term) ::= ... (same as before)

| view $V\{\ell^+ = x^+\} = M_1$ in M_2 (view definition)

| $M_1\{\ell^+ = M_2^+\}$ (view application)

| vmatch M_1 with $V\{\ell^* = x^*\} \Rightarrow M_2$ else M_3 (view matching)

- X^* and X^+ are abbreviations for $X_1, ..., X_n$ where $n \geq 0$ or $n > 0$, respectively

Semantics of VMλabl (1/3)

v (value) ::= ... (same as before)

| $\langle \varepsilon; V\{\ell^* = v^*, m^+ = x^+\} = M \rangle$ (view constructor closure)

| $V\{\ell^+ = v_1{}^+\} = v_2$ (view)

$$\frac{V' \text{ fresh} \qquad \mathcal{E}, V \mapsto \langle \mathcal{E}; V'\{l^+ = x^+\} = M_1 \rangle \vdash M_2 \Downarrow v}{\mathcal{E} \vdash \texttt{view}\ V\{l^+ = x^+\} = M_1\ \texttt{in}\ M_2 \Downarrow v}\ \text{(E-VDef)}$$

Semantics of VMλabl (2/3)

v (value) ::= ... (same as before)

| $\langle \varepsilon; V\{\ell^* = v^*, m^+ = x^+\} = M \rangle$ (view constructor closure)

| $V\{\ell^+ = v_1{}^+\} = v_2$ (view)

$$\frac{\mathcal{E} \vdash M_1 \Downarrow \langle \mathcal{E}'; V\{l_1^* = v_1^*, l_2^+ = x^+, l_3^+ = y^+\} = M \rangle \qquad \mathcal{E} \vdash M_2^+ \Downarrow v_2^+}{\mathcal{E} \vdash M_1\{l_2^+ = M_2^+\} \Downarrow \langle \mathcal{E}', x^+ \mapsto v_2^+; V\{l_1^* = v_1^*, l_2^+ = v_2^+, l_3^+ = y^+\} = M \rangle} \text{(E-VApp-Part)}$$

Semantics of VMλabl (3/3)

v (value) ::= ... (same as before)

| $\langle \varepsilon; V\{\ell^* = v^*, m^+ = x^+\} = M \rangle$ (view constructor closure)

| $V\{\ell^+ = v_1{}^+\} = v_2$ (view)

$$\frac{\mathcal{E} \vdash M_1 \Downarrow \langle \mathcal{E}'; V\{l_1^* = v_1^*, l_2^+ = x^+\} = M \rangle \quad \mathcal{E} \vdash M_2^+ \Downarrow v_2^+ \quad \mathcal{E}', x^+ \mapsto v_2^+ \vdash M \Downarrow v}{\mathcal{E} \vdash M_1\{l_2^+ = M_2^+\} \Downarrow V\{l_1^* = v_1^*, l_2^+ = v_2^+\} = v} \text{(E-VApp-Full)}$$

Type System of VMλabl

τ (type) ::= ... (same as before)

| view{ℓ^* : τ^*}τ (view / view constructor type)

$$\frac{\Gamma, x^+ : \tau^+ \vdash M_1 : \tau_1 \quad \Gamma, V : \texttt{view}\{l^+ : \tau^+\}\tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \texttt{view}\ V\{l^+ = x^+\} = M_1\ \texttt{in}\ M_2 : \tau_2}\text{(T-VDef)}$$

$$\frac{\Gamma \vdash M_1 : \texttt{view}\{l^+ : \tau^+, l_0^* : \tau_0^*\}\tau \quad \Gamma \vdash M_2^+ : \tau^+}{\Gamma \vdash M_1\{l^+ = M_2^+\} : \texttt{view}\{l_0^* : \tau_0^*\}\tau}\text{(T-VApp)}$$

$$\frac{\begin{array}{c}\Gamma(V) = \texttt{view}\{l^* : \tau^*, l_0^* : \tau_0^*\}\tau \quad \Gamma \vdash M_1 : \texttt{view}\{l_0^* : \tau_0^*\}\tau \\ \Gamma, x^* : \tau^* \vdash M_2 : \tau' \quad \Gamma \vdash M_3 : \tau'\end{array}}{\Gamma \vdash \texttt{vmatch}\ M_1\ \texttt{with}\ V\{l^* = x^*\} \Rightarrow M_2\ \texttt{else}\ M_3 : \tau'}\text{(T-VMatch)}$$

Implementation of VMλabl

Translation by Camlp4 into OCaml/OLabl

- value of view constructor $\Rightarrow$ function with labeled arguments
- representation of view $\Rightarrow$ polymorphic variants
  - Recall "view = pair of a value and its representation (of an extensible data type)"

- Why polymorphic variants? (not abstract types, exceptions, etc.)
  - Allow pattern matching (unlike abstract types)
  - Don't require type declaration (unlike exceptions)

Conclusions

- We have formalized and implemented VML (ML with views), a functional programming language for scientific knowledge discovery
- Real applications are explained in a previous paper [Bannai et al. 2001]
  - Detection of gene regulatory sites
  - Characterization of N-terminal protein sorting signals
- People *do* find functional programming (and its theories) useful, if they open their mind