The Extension of ML with Hypothetical Views for Discovery Science: Formalization and Implementation*

Eijiro Sumii[1] and Hideo Bannai[2]

[1] Department of Computer and Information Science, University of Pennsylvania
sumii@saul.cis.upenn.edu
[2] Human Genome Center, Institute of Medical Science, University of Tokyo
bannai@ims.u-tokyo.ac.jp

Abstract. We present VMλ, a formalization and implementation of the functional language VML.

VML is a programming language proposed by discovery scientists for the purpose of assisting the process of knowledge discovery. It is a non-trivial extension of ML with *hypothetical views*. Operationally, a hypothetical view is a value with a representation that indicates how the value was created. The notion of hypothetical views has already been successful in the domain of genome analysis, and known to be useful in the process of knowledge discovery. However, VML as a programming language was only informally defined in English prose, and indeed found problematic both in theory and in practice. Thus, a proper definition and implementation of VML with formal foundations would be of great help to discovery science and hence corresponding domain sciences.

This paper gives a solid foundation of VML by extending the standard simply typed call-by-value λ-calculus. Although this extension, VMλ, is simple and clear, its design required much care to find and fix problems of the original VML. We also present a real implementation of VMλ, written in Camlp4 as a conservative translator into OCaml. This implementation makes extensive use of labeled arguments and polymorphic variants – two advanced features of OCaml that originate in OLabl.

1 Introduction

Functional Programming for Scientific Discovery: Approaches and Problems. Higher-order functional programming languages are known to be good for complex applications such as theorem proving, artificial intelligence, program generation, database querying, and genome analysis [17, 18, 29, 34, 38, 40]. Given the success of functional languages in these domains, it is natural to consider the use of functional languages in the field of *discovery science* [3–5], an area of

* This work was carried out while the first author was in Department of Computer Science, Graduate School of Information Science and Technology, University of Tokyo.

information science that aims to develop systematic methods of knowledge discovery. Indeed, the features of functional languages—in particular, first-class functions—seem to help much to create and evaluate scientific hypotheses.

For instance, suppose that we have the data of some people's heights and weights, and would like to find the relationship between each person's height and weight. The data can be represented by a list of pairs of two floating-point numbers, one for height in centimeters and the other for weight in kilograms. (For the sake of concreteness, we adopt OCaml-like syntax [27]. Readers who are familiar with other functional languages should not have much difficulty in understanding it.)

```
# let data =
    [(175.4, 73.9); (167.6, 66.1); (180.8, 81.2); ...] ;;
val data : (float * float) list =
  [(175.4, 73.9); (167.6, 66.1); (180.8, 81.2); ...]
```

A hypothesis for explaining this data can be modeled by a function that estimates each person's weight from the person's height, for example by subtracting 100. (The operator `-.` is the OCaml syntax of subtraction for floating-point numbers.)

```
# let simple_hypothesis h = h -. 100.0 ;;
val simple_hypothesis : float -> float = <fun>
```

Then, the appropriateness of this hypothesis can be evaluated by some statistical method implemented by a higher-order function that takes the hypothesis as an argument.

```
# (* fitness : (float -> float) ->
       (float * float) list -> float *)
  fitness simple_hypothesis data ;; (* 1.0 means a perfect fit *)
- : float = 0.81
```

Of course, there may well be other functions that fit the data better, for example:

```
# fitness (fun h -> h -. 101.0) data ;;
- : float = 0.83
```

Rather than trying the infinite possibilities of such functions one by one, it is nicer to have another higher-order function that returns the function (of a certain class) that fits the data best.

```
# let create_hypothesis data =
    let (a, b) = compute such a and b that f(x) = ax + b is
                 the affine function f that fits the data best in
    fun h -> a *. h +. b ;;
val create_hypothesis :
  (float * float) list -> float -> float = <fun>
```

Then, by using this hypothesis-creating function, we can automatically obtain a hypothesis that explains the data well.

```
# let good_hypothesis = create_hypothesis data ;;
val good_hypothesis : float -> float = <fun>
# fitness good_hypothesis data ;;
- : float = 0.95
```

Or can we? Not really – we cannot actually see what the function *is*, because we have access only to the *value* of the function. That is, we have no access to the *representation* of the hypothesis. Thus, there is no way for the user to interpret the meaning of this hypothesis and evaluate it under domain experience. This significant limitation makes the present system far less useful for knowledge discovery.

An obvious solution for this problem is to have the user modify the program and manipulate the representation by hand. For example, the hypothesis-creating function above can be rewritten as

```
# let create_hypothesis data =
    let (a, b) = ... in ((fun h -> a *. h +. b), (a, b)) ;;
val create_hypothesis : (float * float) list ->
  (float -> float) * (float * float) = <fun>
```

so that it returns the pair of the function and its parameters. In real programs, however, this approach is much more troublesome and error-prone than it may seem: the user must take care not to confuse the representations of a class of functions with those of another; furthermore, in typed languages, some trick (e.g., to use exceptions as an extensible data type) is necessary to unify the types of functions whose values have the same type but whose representations may have different types. These difficulties spoil the utility and simplicity of functional languages in this application.

Another naive solution is to remember or reconstruct the source code of a function, that is, taking the source code as the representation of a hypothesis. However, doing so is inefficient or even impossible (e.g., for preserving type abstraction) in many languages, though it is possible in a few situations (e.g., by *get-lambda-expression* in certain dialects of Lisp or by type-directed partial evaluation [9, 10]). Furthermore, the source code of functions can be rather complex and therefore is not very useful as representation of hypotheses. In addition, a similar problem arises in first-order values as well: in knowledge discovery, it is often necessary to know not only a value itself but also how the value was computed [6]; however, remembering the history of computation is even more expensive than having the source code of a function.

VML: A Functional Language with Hypothetical Views. To address the issues above, a group of discovery scientists have recently proposed a functional language VML [6],[3] an extension of ML with *hypothetical views* or just *views* in short. (Note that they are different from views for abstract data types [39].) Intuitively, a view is the pair of a value and its representation that remembers how

[3] It has nothing to do with the Vector Markup Language [25].

the value was computed. Views are constructed by defining a *view constructor* via the keyword `view` and by applying the view constructor to an argument. A view thus constructed can then be pattern-matched via the keyword `vmatch`.

For instance, in the example above, let us define the hypotheses as views rather than as ordinary functions. (For the sake of clarity, we use a little different syntax and semantics from the original VML.)

```
# view AffineFun(a, b) = fun h -> a *. h +. b ;;
view AffineFun of float * float : float -> float
```

The view constructor `AffineFun` takes a pair of two floating-point numbers `a` and `b`, and returns a function of type `float -> float` with the representation `AffineFun(a, b)` of the function.

```
# AffineFun(1.0, -100.0) ;;
- : (float -> float) view = <fun> as AffineFun(1.0, -100.0)
```

By using this view constructor, the hypothesis-creating function above can be rewritten as:

```
# let create_hypothesis data =
    let (a, b) = ... in AffineFun(a, b) ;;
val create_hypothesis :
  (float * float) list -> (float -> float) view = <fun>
```

Then, by applying this view-returning function and by pattern-matching the returned view, we can finally see what the automatically obtained good hypothesis is.

```
# let good_hypothesis = create_hypothesis data ;;
- : (float -> float) view = <fun> as AffineFun(1.03, -102.8)
# vmatch good_hypothesis with AffineFun(a, b) -> (a, b) ;;
- : float * float = (1.03, -102.8)
```

From the data, the hypothesis-creating function found the function $f(x) = 1.03x - 102.8$ to be the best affine function f that estimates each person's weight from his/her height. Of course, it is also possible to use the value part of the view (i.e., the function `fun h -> 1.03 *. h -. 102.8`) as well as its representation part.

```
# (valof good_hypothesis) 175.4 ;;
- : float = 77.862
```

Our Contributions. The notion of views itself has already been proved to be useful by several applications in the domain of genome analysis [7, 22–24]. However, the semantics and even the syntax of VML were only informally presented in English prose [6] and theoretically unclear as well as practically problematic. As a result, VML was never successfully implemented.

This paper formalizes the syntax, the type system, and the dynamic semantics of VML by extending the standard simply typed call-by-value λ-calculus.

$$
\begin{array}{rcll}
M \text{ (term)} & ::= & x & \text{(variable)} \\
& | & \lambda x.\, M & (\lambda\text{-abstraction}) \\
& | & M_1 M_2 & \text{(function application)} \\
& | & \texttt{view } V\{x\} = M_1 \texttt{ in } M_2 & \text{(view definition)} \\
& | & V & \text{(view constructor)} \\
& | & M_1\{M_2\} & \text{(view application)} \\
& | & \texttt{vmatch } M_1 \texttt{ with } V\{x\} \Rightarrow M_2 \texttt{ else } M_3 & \text{(view matching)} \\
& | & \texttt{valof } M & \text{(view destruction)}
\end{array}
$$

Fig. 1. Syntax of Simple VMλ

This formalization, VMλ, reveals and fixes problems in the original VML. Furthermore, we give a translation of VMλ into ordinary OCaml without views. The translation is fully implemented in Camlp4 [11] and is conservative – that is, features of the original OCaml are available for free.

2 Simple VMλ

First, we present the simplest version of VMλ, where every view constructor takes just one argument. Later in the next section, we will present a more sophisticated version of VMλ, where view constructors may take any number of arguments in any order, with partial application of view constructors supported as well as pattern matching against such partially applied view constructors.

2.1 Syntax and Informal Semantics

The syntax of VMλ is given in Figure 1. It assumes two countably infinite disjoint sets *Var* of variables x, y, z, $\ldots$ and *Name* of view constructors V, W, $\ldots$. In addition to standard λ-terms, there are five kinds of terms involving views. Recall that a view is the pair of a value v_1 and its representation $V\{v_2\}$.

- A view definition $\texttt{view } V\{x\} = M_1 \texttt{ in } M_2$ first defines the view constructor V and then evaluates the body M_2, where the view constructor V takes an argument v, evaluates the term M_1 with the variable x bound to the value v, and returns the result with its representation $V\{v\}$. The name V is called *bound* in the term M_2 and can be implicitly renamed by α-conversion.
- A view constructor V refers to its own definition as above.
- A view application $M_1\{M_2\}$ first evaluates the term M_1 to a view constructor and the term M_2 to a value v, and then applies the view constructor to the argument v.
- A view matching $\texttt{vmatch } M_1 \texttt{ with } V\{x\} \Rightarrow M_2 \texttt{ else } M_3$ first evaluates the term M_1 to a view and then matches its representation part $W\{v\}$ against the pattern $V\{x\}$. If $V = W$, then the term M_2 is evaluated with the variable x bound to the value v. Otherwise, the term M_3 is evaluated.

$$
\begin{array}{rll}
v \text{ (value)} ::= & \langle \mathcal{E}; \lambda x.\, M \rangle & \text{(ordinary function closure)} \\
\mid & \langle \mathcal{E}; V\{x\} = M \rangle & \text{(view constructor closure)} \\
\mid & V\{v_1\} = v_2 & \text{(view)}
\end{array}
$$

$$
\frac{\mathcal{E}(x) = v}{\mathcal{E} \vdash x \Downarrow v}\text{(E-Var)} \qquad \mathcal{E} \vdash \lambda x.\, M \Downarrow \langle \mathcal{E}; \lambda x.\, M \rangle \quad \text{(E-Lam)}
$$

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

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

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

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

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

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

Fig. 2. Semantics of Simple VMλ

– A view destruction `valof` M evaluates the term M to a view and extracts its value part v.

For example, assuming primitives for integers and tuples, the term

$$
\begin{array}{l}
\texttt{view } V\{x\} = x + 1 \texttt{ in let } v = V\{2\} \texttt{ in} \\
\langle \texttt{valof } v, \texttt{vmatch } v \texttt{ with } V\{y\} \Rightarrow y \texttt{ else } -1 \rangle
\end{array}
$$

evaluates to the tuple $\langle 3, 2 \rangle$. Here, `let` $x = M_1$ `in` M_2 is the syntax sugar of $(\lambda x.\, M_2)M_1$.

2.2 Operational Semantics

The semantics of VMλ is formalized by the *evaluation* relation $\mathcal{E} \vdash M \Downarrow v$, where v is a *value* denoting the result of evaluation and $\mathcal{E}$ is an *environment* mapping free variables (and free view constructors) of M to their values. Intuitively, the relation $\mathcal{E} \vdash M \Downarrow v$ means that the term M evaluates to the value v under the environment $\mathcal{E}$. Formally, $\mathcal{E} \vdash M \Downarrow v$ is the least relation over $\mathcal{E}$, M, and v that satisfies the rules in Figure 2.

A value v is either a closure $\langle\mathcal{E}; \lambda x.\, M\rangle$ of an ordinary function $\lambda x.\, M$, a closure $\langle\mathcal{E}; V\{x\} = M\rangle$ of a view constructor V defined as $V\{x\} = M$, or a view $V\{v_1\} = v_2$ of a value v_2 represented as $V\{v_1\}$.

The rules (E-Var), (E-Lam), and (E-FApp) are standard. The other rules formalize the intuitive semantics above of terms involving views. In rule (E-VDef), the premise that V is bound to a closure of fresh V' reflects the fact that view constructors are treated as *generative* because they are α-convertible but may escape their syntactic scopes. Accordingly, in the rules (E-VMatch-Succ) and (E-VMatch-Fail), this freshly generated V' is looked up in $\mathcal{E}$ by V and used for the pattern matching. Here, _ denotes the "don't-care" meta-variable.

For example, assuming primitives for integers and booleans, let M_1 be the term:

$$\texttt{view}\ V\{x\} = \texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else}\ 0\ \texttt{in}\ V\{\texttt{true}\}$$

Then, under the empty environment, M_1 evaluates to the value $V'\{\texttt{true}\} = 1$ for a fresh view constructor V'. That is, $\vdash M_1 \Downarrow V'\{\texttt{true}\} = 1$. Let furthermore M_2 be the term:

$$\texttt{view}\ V\{y\} = y + 1\ \texttt{in}\ \texttt{vmatch}\ M_1\ \texttt{with}\ V\{z\} \Rightarrow z - 2\ \texttt{else}\ -1$$

Since the V' above is fresh, the pattern matching in this term fails and M_2 evaluates to the integer -1 rather than causing the runtime type error $\texttt{true} - 2$.

2.3 Type System

The type system of VMλ, given in Figure 3, is an extension of the simple type system of the standard λ-calculus.[4] In addition to standard types, there are two kinds of types involving views.

- A view constructor type $\texttt{view}\{\tau_1\}\tau_2$ denotes the type of a view constructor that takes an argument of type τ_1 and returns the view for a value of type τ_2.
- A view type $\texttt{view}\{\}\tau$ denotes the type of a view for a value of type τ.

Thus, in this version of VMλ, a view constructor type $\texttt{view}\{\tau_1\}\tau_2$ is actually equivalent to the function type $\tau_1 \rightarrow \texttt{view}\{\}\tau_2$. However, these types are distinguished for the sake of presentation consistent with the next section.

The typing rules are straightforward, given the semantics of VMλ and the meanings of types above. Thanks to the generativity of view constructors, there is no worry about cases where two definitions of a syntactically identical view constructor expect different types of argument, as in the example above.

The soundness of this type system is formally proved as follows. First, the typing rules are naturally extended for values and environments as in Figure 4. Next, the relation $\mathcal{E} \vdash M \Downarrow \mathit{error}$, denoting a runtime error in the evaluation

[4] $\Gamma, x : \tau$ is the type environment such that $(\Gamma, x : \tau)(y) = \Gamma(y)$ for $y \in \mathit{dom}(\Gamma)$ and $(\Gamma, x : \tau)x = \tau$ where $x \notin \mathit{dom}(\Gamma)$. If it happens that $x \in \mathit{dom}(\Gamma)$, we assume implicit α-conversion of x to some $z \notin \mathit{dom}(\Gamma)$.

$$
\begin{array}{rcll}
\tau \ (\text{type}) & ::= & b & (\text{base type}) \\
& | & \tau_1 \to \tau_2 & (\text{function type}) \\
& | & \texttt{view}\{\tau_1\}\tau_2 & (\text{view constructor type}) \\
& | & \texttt{view}\{\}\tau & (\text{view type})
\end{array}
$$

$$
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}(\text{T-Var}) \qquad \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2}(\text{T-Lam})
$$

$$
\frac{\Gamma \vdash M_1 : \tau \to \tau' \quad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 M_2 : \tau'}(\text{T-FApp})
$$

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

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

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

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

Fig. 3. Type System of Simple VMλ

$$
\frac{\Gamma' \vdash \mathcal{E} \quad \Gamma' \vdash \lambda x.\, M : \tau_1 \to \tau_2}{\Gamma \vdash \langle \mathcal{E}; \lambda x.\, M \rangle : \tau_1 \to \tau_2}(\text{T-FunClos})
$$

$$
\frac{\Gamma \vdash V : \texttt{view}\{\tau\}\tau' \quad \Gamma' \vdash \mathcal{E} \quad \Gamma', x : \tau \vdash M : \tau'}{\Gamma \vdash \langle \mathcal{E}; V\{x\} = M \rangle : \texttt{view}\{\tau\}\tau'}(\text{T-VConClos})
$$

$$
\frac{\Gamma \vdash V : \texttt{view}\{\tau\}\tau' \quad \Gamma \vdash v_1 : \tau \quad \Gamma \vdash v_2 : \tau'}{\Gamma \vdash V\{v_1\} = v_2 : \texttt{view}\{\}\tau'}(\text{T-View})
$$

$$
\frac{\Gamma \vdash \mathcal{E}(x) : \Gamma(x) \text{ for each } x \in \mathit{dom}(\mathcal{E})}{\Gamma \vdash \mathcal{E}}(\text{T-Env})
$$

Fig. 4. Typing Rules for Values and Environments in Simple VMλ

of the term M under the environment $\mathcal{E}$, is defined as the least relation that satisfies rules such as:

$$\frac{V \notin dom(\mathcal{E})}{\mathcal{E} \vdash V \Downarrow error} \qquad \frac{\mathcal{E} \vdash M \Downarrow v \quad v \text{ is not a view}}{\mathcal{E} \vdash \mathtt{valof}\ M \Downarrow error} \qquad \frac{\mathcal{E} \vdash M \Downarrow error}{\mathcal{E} \vdash \mathtt{valof}\ M \Downarrow error}$$

The other rules are similar. (In the present formalization, we could actually define $\mathcal{E} \vdash M \Downarrow error$ just as "there exists no such v that $e \vdash M \Downarrow v$." However, this approach does not scale to cases where evaluation may diverge, e.g., because of recursion.) Then, the type soundness is proved as follows.

Lemma 1 (Weakening). *Let Γ, Γ' be the type environment such that $(\Gamma, \Gamma')(x) = \Gamma(x)$ for $x \in dom(\Gamma)$ and $(\Gamma, \Gamma')(x) = \Gamma'(x)$ for $x \in dom(\Gamma')$ where $dom(\Gamma) \cap dom(\Gamma') = \emptyset$.*

1. *If $\Gamma \vdash M : \tau$, then $\Gamma, \Gamma' \vdash M : \tau$ for any Γ' with $dom(\Gamma) \cap dom(\Gamma') = \emptyset$.*
2. *If $\Gamma \vdash v : \tau$, then $\Gamma, \Gamma' \vdash v : \tau$ for any Γ' with $dom(\Gamma) \cap dom(\Gamma') = \emptyset$.*
3. *If $\Gamma \vdash \mathcal{E}$, then $\Gamma, \Gamma' \vdash \mathcal{E}$ for any Γ' with $dom(\Gamma) \cap dom(\Gamma') = \emptyset$.*

Proof. Straightforward induction on the derivation of $\Gamma \vdash M : \tau$, $\Gamma \vdash v : \tau$, and $\Gamma \vdash \mathcal{E}$.

Theorem 1 (Type Soundness). *If $\Gamma \vdash M : \tau$ and $\Gamma \vdash \mathcal{E}$, then $\mathcal{E} \vdash M \not\Downarrow error$. Furthermore, if $\mathcal{E} \vdash M \Downarrow v$, then $\Gamma, \Gamma' \vdash v : \tau$ for some Γ' with $dom(\Gamma) \cap dom(\Gamma') = \emptyset$.*

Proof. By induction on the structure of M using Lemma 1. The only non-trivial cases are the following two.

Case $M = (\mathtt{view}\ V\{x\} = M_1\ \mathtt{in}\ M_2)$**.** Suppose $\Gamma, x : \tau' \vdash M_1 : \tau_1'$ and $\Gamma, V : \mathtt{view}\{\tau'\}\tau_1' \vdash M_2 : \tau$. By Lemma 1, $\Gamma, V : \mathtt{view}\{\tau'\}\tau_1', V' : \mathtt{view}\{\tau'\}\tau_1' \vdash M_2 : \tau$ for any fresh V'. On the other hand, $\Gamma, V : \mathtt{view}\{\tau'\}\tau_1', V' : \mathtt{view}\{\tau'\}\tau_1' \vdash \langle \mathcal{E}; V'\{x\} = M_1 \rangle : \mathtt{view}\{\tau'\}\tau_1'$ by (T-VConsClos) with the assumption that $\Gamma \vdash \mathcal{E}$, so $\Gamma, V : \mathtt{view}\{\tau'\}\tau_1', V' : \mathtt{view}\{\tau'\}\tau_1' \vdash \mathcal{E}, V \mapsto \langle \mathcal{E}; V'\{x\} = M_1 \rangle$ by (T-Env) with the inversion of $\Gamma \vdash \mathcal{E}$. Therefore, by the induction hypothesis, $\mathcal{E}, V \mapsto \langle \mathcal{E}; V'\{x\} = M_1 \rangle \vdash M_2 \not\Downarrow error$. Suppose $\mathcal{E}, V \mapsto \langle \mathcal{E}; V'\{x\} = M_1 \rangle \vdash M_2 \Downarrow v$. The theorem then follows by (E-VDef) and the induction hypothesis.

Case $M = (\mathtt{vmatch}\ M_1\ \mathtt{with}\ V\{x\} \Rightarrow M_2\ \mathtt{else}\ M_3)$**.** Suppose $\Gamma(V) = \mathtt{view}\{\tau'\}\tau_1'$ and $\Gamma \vdash M_1 : \mathtt{view}\{\}\tau_1'$ with $\Gamma, x : \tau' \vdash M_2 : \tau$ and $\Gamma \vdash M_3 : \tau$. Since $\Gamma \vdash \mathcal{E}$, $\Gamma \vdash \mathcal{E}(V) : \mathtt{view}\{\tau'\}\tau_1'$. Thus, by inversion of (T-VConsClos), $\mathcal{E}(V)$ is of the form $\langle \_; V'\{\_\} = \_ \rangle$ and $\Gamma \vdash V' : \mathtt{view}\{\tau'\}\tau_1'$. On the other hand, $\mathcal{E} \vdash M_1 \not\Downarrow error$ by the induction hypothesis. Suppose $\mathcal{E} \vdash M_1 \Downarrow v_1$. Again by the induction hypothesis, $\Gamma, \Gamma' \vdash v_1 : \mathtt{view}\{\}\tau_1'$ for some Γ'. Thus, by inversion of (T-View), v_1 is of the form $W\{v'\} = \_$. If $W \neq V'$, the theorem follows by (E-VMatch-Fail) and the induction hypothesis. Suppose $W = V'$. Again by inversion of (T-View), $\Gamma, \Gamma' \vdash v' : \tau'$. Therefore, by (T-Env), $\Gamma, \Gamma', x : \tau' \vdash \mathcal{E}, x \mapsto v'$. On the other hand, $\Gamma, \Gamma', x : \tau' \vdash M_2 : \tau$ by Lemma 1. The theorem then follows by (E-VMatch-Succ) and the induction hypothesis.

$$
\begin{array}{rll}
M \text{ (term)} ::= & \ldots & \text{(same as before)} \\
| & \texttt{view } V\{l^+ = x^+\} = M_1 \texttt{ in } M_2 & \text{(view definition)} \\
| & M_1\{l^+ = M_2^+\} & \text{(view application)} \\
| & \texttt{vmatch } M_1 \texttt{ with } V\{l^* = x^*\} \Rightarrow M_2 \texttt{ else } M_3 & \text{(view matching)}
\end{array}
$$

Fig. 5. Syntax of VMλabl

3 VMλabl: An Extension of Simple VMλ with Labeled Arguments

In this section, we present VMλabl, a more sophisticated version of VMλ extended with *labeled arguments* [1, 13, 16].

Partial application of functions is a convenient feature of functional languages: it allows one to create a special function by fixing part of the arguments of a generic function, without having to name and define the special function explicitly and separately; furthermore, λ-abstraction enables giving any (rather than only the first) of the arguments of a function in advance, for example like $\lambda x.\, \lambda z.\, f(x, 123, z)$.

Unfortunately, however, in the simple VMλ in the previous section, this convenient feature is not available for views: even if $V\{(x, 123, z)\}$ is a view, $\lambda x.\, \lambda z.\, V\{(x, 123, z)\}$ is not – it is a mere ordinary function that does not have a representation and cannot be pattern-matched.

To overcome this limitation, the original VML allowed pattern matching over λ-abstracted views [6], like:

```
# vmatch (fun x -> fun z -> V(x, 123, z)) with V(_, y, _) -> y
- : int = 123
```

However, it is rather problematic both theoretically and practically, because it requires evaluation inside functions and breaks the standard weak normalization strategy of most functional languages. For instance, pattern-matching $\lambda x.\, \lambda z.\, V\{(x, g(456), z)\}$ against $V(\_, y, \_)$ forces the quite unnatural evaluation of $g(456)$, which may diverge, have a side effect, or take a long time. (It may take more than an hour or even a week if the data is large.)

We solve the problems above by allowing partial and commutative application of view constructors via *labeled arguments* [1, 13, 16]. For this purpose, we define VMλabl, an extension of VMλ with labeled arguments of view constructors. With this extension, the example above can be rewritten like `vmatch` $V\{l_2 = g(456)\}$ `with` $V\{l_2 = y\} \Rightarrow y$, in which it is natural to evaluate $g(456)$.

3.1 Syntax

The abstract syntax of VMλabl is given in Figure 5. It is the same as VMλ except for three kinds of terms, namely, view definition, view application, and view matching, where the arguments of view constructors are labeled. We assume yet another countably infinite set *Lab* of *labels* l, m, n, ..., distinct from variables and view constructors. The order of labeled arguments does not matter: for example, $\{l_1 = M_1, l_2 = M_2\}$ is the same as $\{l_2 = M_2, l_1 = M_1\}$.

For the sake of brevity, we use $+$ and $*$ to abbreviate sequences: X^+ denotes $X_1, \dots, X_n$ where $n > 0$; X^* denotes $X_1, \dots, X_n$ where $n \geq 0$ (i.e., the sequence may be empty). The notations X^+ *op* Y^+ and X^* *op* Y^* mean the sequence X_1 *op* $Y_1, \dots, X_n$ *op* Y_n for any binary operator *op*. For example, $\{l^+ = M^+\}$ means $\{l_1 = M_1, \dots, l_n = M_n\}$ where $n > 0$.

Also, for the sake of syntactic convenience, the view application $V\{l^+ = v^+\}$ can take more than one argument at a time. It is semantically equivalent to the sequence of view applications $V\{l_1 = v_1\} \dots \{l_n = v_n\}$.

3.2 Semantics

The evaluation semantics of VMλabl is given in Figure 6. Its difference from the semantics of the simple VMλ is that view constructors may be partially applied, and also, that their arguments are labeled. The result of the partial application $V\{l^+ = v^+\}$ of a view constructor defined as $V\{l^+ = y^+, m^+ = x^+\} = M$ is written $V\{l^+ = v^+, m^+ = x^+\} = M$. Note that the labels m^+ of the yet unknown arguments x^+ are non-empty, which makes this application partial. The body M is evaluated when all of the arguments are given, that is, when the view constructor V is fully applied. Thus, the rules (E-VApp), (E-VMatch-Succ) and (E-VMatch-Fail) are divided into two cases (...-Part) and (...-Full) each, according to whether the view constructor of concern is partially applied (or not applied at all) or fully applied.

3.3 Type System

Having introduced partial application of view constructors, types in VMλabl are actually simpler than those in the simple VMλ, because the type $\texttt{view}\{\}\tau$ of views are integrated into the type $\texttt{view}\{l^* : \tau^*\}\tau$ of view constructors that are possibly partially applied, as a special case where the yet unknown arguments l^* are empty. The typing rules of VMλabl, given in Figure 7, are straightforward adaptation of those of the simple VMλ. We conjecture that the type soundness proof of VMλabl should also be similar to that of the simple VMλ.

4 Translation of VMλabl into OCaml

This section explains an implementation[5] of VMλabl, specifically, its translation into OCaml [27]. The translation itself is implemented by using Camlp4 [11], a pre-processor (and pretty printer) for OCaml. The translated code uses labeled arguments and polymorphic variants [14, 30] (so the target language may well be

[5] Available at: `http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pub/vml.tar.gz`

$$
\begin{array}{lll}
v \text{ (value)} ::= \ldots & & \text{(same as before)} \\
\quad \mid & \langle \mathcal{E}; V\{l^* = v^*, m^+ = x^+\} = M \rangle & \text{(view constructor closure)} \\
\quad \mid & V\{l^+ = v^+\} = v & \text{(view)}
\end{array}
$$

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

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

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

$$
\frac{\begin{array}{c}\mathcal{E} \vdash M_1 \Downarrow \langle \_; V'\{l^* = v^*, m^+ = y^+\} = \_ \rangle \\ \mathcal{E}(V) = \langle \_; V'\{\ldots\} = \_ \rangle \quad \mathcal{E}, x^* \mapsto v^* \vdash M_2 \Downarrow v\end{array}}{\mathcal{E} \vdash \texttt{vmatch } M_1 \texttt{ with } V\{l^* = x^*\} \Rightarrow M_2 \texttt{ else } M_3 \Downarrow v} \text{(E-VMatch-Succ-Part)}
$$

$$
\frac{\begin{array}{c}\mathcal{E} \vdash M_1 \Downarrow \langle \_; V'\{\ldots\} = \_ \rangle \quad \mathcal{E}(V) = \langle \_; W\{\ldots\} = \_ \rangle \\ W \neq V' \quad \mathcal{E} \vdash M_3 \Downarrow v\end{array}}{\mathcal{E} \vdash \texttt{vmatch } M_1 \texttt{ with } V\{l^* = x^*\} \Rightarrow M_2 \texttt{ else } M_3 \Downarrow v} \text{(E-VMatch-Fail-Part)}
$$

$$
\frac{\begin{array}{c}\mathcal{E} \vdash M_1 \Downarrow V'\{l^+ = v^+\} = \_ \quad \mathcal{E}(V) = \langle \_; V'\{\ldots\} = \_ \rangle \\ \mathcal{E}, x^+ \mapsto v^+ \vdash M_2 \Downarrow v\end{array}}{\mathcal{E} \vdash \texttt{vmatch } M_1 \texttt{ with } V\{l^+ = x^+\} \Rightarrow M_2 \texttt{ else } M_3 \Downarrow v} \text{(E-VMatch-Succ-Full)}
$$

$$
\frac{\begin{array}{c}\mathcal{E} \vdash M_1 \Downarrow V'\{\ldots\} = \_ \quad \mathcal{E}(V) = \langle \_; W\{\ldots\} = \_ \rangle \\ W \neq V' \quad \mathcal{E} \vdash M_3 \Downarrow v\end{array}}{\mathcal{E} \vdash \texttt{vmatch } M_1 \texttt{ with } V\{l^+ = x^+\} \Rightarrow M_2 \texttt{ else } M_3 \Downarrow v} \text{(E-VMatch-Fail-Full)}
$$

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

Fig. 6. Semantics of VMλabl

$$\begin{array}{rlr} \tau \text{ (type)} ::= & \ldots & \text{(same as before)} \\ | & \texttt{view}\{l^* : \tau^*\}\tau & \text{(view and view constructor type)} \end{array}$$

$$\frac{\Gamma, x^+ : \tau^+ \vdash M_1 : \tau_1 \quad \Gamma, V : \texttt{view}\{l^+ : \tau^+\}\tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \texttt{view } V\{l^+ = x^+\} = M_1 \texttt{ in } M_2 : \tau_2} \text{(T-VDef')}$$

$$\frac{\Gamma \vdash M_1 : \texttt{view}\{l^+ : \tau^+, l_0^* : \tau_0^*\}\tau \quad \Gamma \vdash M_2^+ : \tau^+}{\Gamma \vdash M_1\{l^+ = M_2^+\} : \texttt{view}\{l_0^* : \tau_0^*\}\tau} \text{(T-VApp')}$$

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

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

Fig. 7. Type System of VMλabl

called OLabl [13] rather than OCaml). Although we have adopted OCaml as the actual target language, observations in this section would also apply to other typed languages extended with labeled arguments and polymorphic variants. (We are interested in typed languages because views are originally typed [7, 22–24].) For a brief introduction to labeled arguments and polymorphic variants, see `http://caml.inria.fr/ocaml/htmlman/manual006.html`.

The main ideas of the translation are as follows. A possibly partially applied view $V\{l^* = v^*, m^* = x^*\} = M$, including a non-applied view $V\{m^+ = x^+\} = M$ and a fully applied view $V\{l^+ = v^+\} = v$, is represented as a record with three fields `valu`, `repr`, and `addargs`. The field `valu` holds a function `fun` m_1`:`x_1 `->` ... `-> fun` m_n`:`x_n `->` M with its arguments x^* labeled as m^*. The field `repr` keeps the polymorphic variant `` ` ``V `` [`$l_1$( ``v_1`); ...; `` ` ``l_n`(`v_n`)]` of a list of polymorphic variants `` ` ``l^*`(`v^*`)`. The field `addargs` is an auxiliary function for allowing the translation of a view application to apply the view constructor without knowing its label. Generic rules of the translation based on these ideas are given in Figure 8.

Why do we use polymorphic variants? The reason is twofold: one is to unify the types of two views whose values have the same type but whose representations have different types (this demand precludes encodings using tuples, records, or objects); the other is to avoid unnecessary type declaration (this demand precludes encodings using exceptions or modules). The types of the arguments of polymorphic variants need not be declared, and the types `` [> ` ``V_1 `of` τ_1`]` and `` [> ` ``V_2 `of` τ_2`]` of different polymorphic variants V_1 and V_2 can naturally be unified to `` [> ` ``V_1 `of` τ_1 `|` `` ` ``V_2 `of` τ_2`]`, which is indeed the main point of polymorphic variants.

```
exception VMatchFailure
type ('v, 'r, 'a) view =
 { valu : 'v; repr : 'r; addargs : 'r -> 'a list -> 'r }

⟦view V{l^+ = x^+} = M_1 in M_2⟧ =
 let W_i = generate_fresh_name_from V in
 let vml_V =
  { valu = fun ~l_1:x_1 -> ... -> fun ~l_n:x_n -> ⟦M_1⟧;
    repr = `W_i [];
    addargs = fun newargs ->
               (function `W_i oldargs -> `W_i (oldargs @ newargs)
                       | _ -> failwith "a bug in VMλ") } in
 let vml_V_getargs = (function `W_i args -> args
                             | _ -> raise VMatchFailure) in ⟦M_2⟧

⟦V⟧ = vml_V

⟦M_1{l^+ = M_2^+}⟧ =
 let { valu = vml_valu; repr = vml_repr; addargs = vml_addargs } = ⟦M_1⟧ in
 let vml_arg_l_1 = ⟦M_21⟧ in
 ...
 let vml_arg_l_n = ⟦M_2n⟧ in
 { valu = vml_valu ~l_1:vml_arg_l_1 ... ~l_n:vml_arg_l_n;
   repr = vml_addargs [`l_1(vml_arg_l_1); ...; `l_n(vml_arg_l_n)] vml_repr;
   addargs = vml_addargs }

⟦vmatch M_1 with V{l^* = x^*} ⇒ M_2 else M_3⟧ =
 let { valu = vml_valu; repr = vml_repr; addargs = vml_addargs } = ⟦M_1⟧ in
 let _ = fun () -> TypeCheck in
 try let vml_repr' = vml_V_getargs vml_repr in
     let x_1 = search_l_1 vml_repr' in
     ...
     let x_n = search_l_n vml_repr' in ⟦M_2⟧
 with VMatchFailure -> ⟦M_3⟧
where
 let rec search_l_i = (function [] -> failwith "a bug in VMλ"
                             | `l_i(x) :: _ -> x
                             | _ :: r -> search_l_i r)
 TypeCheck = (* a trick to ensure that M_1 and V{l^* = x^*} have the same type *)
             fun ~l_1:vml_arg_l_1 -> ... -> ~l_n:vml_arg_l_n ->
             vml_valu == vml_V.valu ~l_1:vml_arg_l_1 ... ~l_n:vml_arg_l_n

⟦valof M⟧ = ⟦M⟧.valu
```

Fig. 8. Translation of VMλabl into OCaml with Labeled Arguments and Polymorphic Variants

One drawback of this approach is that labels of polymorphic variants are not generative, so view constructors are not actually generative either; rather, they are *applicative* [20] in the present implementation. Fortunately, however, this problem is not so significant: since view constructors can be renamed by α-conversion at compile time, the conflict of view types occurs only in subtle cases involving polymorphic functions, for example as below, and incurs only static (rather than dynamic) type errors. (As for pattern matching between different instances of the same view constructor, it may actually be even better to have applicativity rather than generativity.)

```
# let f x = (view V{l = _} = 123 in V{l = x}) ;;
val f : 'a -> int view = <fun>
# if true then f 4.56 else f "abc" ;;
  (* This causes a mysterious static type error
     because the then-clause gives V{l = 4.56}
     while the else-clause gives V{l = "abc"}.
     It would not occur if the two V's were fresh and different. *)
Characters 25-32:
This expression has type
  [> `V_1 of [> `l of string] list]
but is here used with type
  [> `V_1 of [> `l of float] list]
```

A simple way of avoiding such unfortunate cases is to apply view constructors to monomorphic values only. (This restriction would be compulsory in encodings using exceptions, since they cannot carry polymorphic values.)

Another drawback is that the notorious *value restriction* interferes with separate compilation of modules that export (possibly partially) applied view constructors, since their representations have non-generalizable types (i.e., types with free monomorphic type variables) such as

```
_[> `V of _[> `l of int | `m of bool ] list]
```

which are not allowed in module interfaces. This problem is solved if a weaker restriction on polymorphism (e.g. [15, 28]) is adopted, which makes the types polymorphic like

```
[> `V of [> `l of int | `m of bool ] list]
```

as desired.

5 A Real Example

As a test case to show the utility of VMλabl, we applied it to the following problem. (This explanation is simplified for the sake of informal presentation and not quite precise from the viewpoint of molecular biology. See [2, Chapter 12] and [8] for more detailed description.) In a living cell, genes coded in DNA are copied into RNA, materialized as a chain of amino acids and folded to a piece of protein. Those pieces of protein are then biochemically transported to specific places in a cell to fulfill their function. The information which determines the destination of a protein is typically contained in a portion of its amino-acid sequence, and the problem is to discover the rule that determines what kind of protein is transported to which places from the sequence information only. This problem is important because: since the biological function of the protein is related to where it is carried to, being able to guess where a (presently uncharacterized) protein would go can help reduce the cost incurred in experimentally confirming its function.

Firstly, we considered the set of proteins that are known (via biological experiments) to be transported to a place called endoplasmic reticulum (ER for short). Specifically, we looked at the amino-acid sequences of 269 proteins (in the cells of plants) that are transported to the ER and 671 proteins that are not.

On the basis of observation by experts, we designed the following hypothetical view to explain the distinction between those two sets of amino-acid sequences.

$$\begin{gathered}\texttt{view}\ V_1\{\mathsf{sequence} = s; \mathsf{position} = p; \mathsf{length} = l;\\ \mathsf{aa\_index} = i; \mathsf{threshold} = t\} =\\ t \leq \mathit{average}\ (\mathit{apply\_aa\_index}\ i\ (\mathit{substring}\ p\ l\ s))\end{gathered}$$

Each amino-acid sequence is represented as a string s. The function *substring* p l takes a string and gives its substring of length l from position p. The function *apply_aa_index* i also takes a string, maps each character to a floating-point number, and returns its list. This mapping (taken from a database called AAindex [19]) represents various characteristics of amino acids, where the identifier i (just a string, actually) determines which characteristic to use. Finally, the function *average* takes a list of floating-point numbers and returns their arithmetic means. Thus, the view V_1 as a whole has type:

$$\begin{gathered}\texttt{view}\{\mathsf{sequence} : \mathit{string}; \mathsf{position} : \mathit{int}; \mathsf{length} : \mathit{int};\\ \mathsf{aa\_index} : \mathit{string}; \mathsf{threshold} : \mathit{float}\}\,\mathit{bool}\end{gathered}$$

It can be seen as a function from an amino-acid sequence to a boolean value, with four auxiliary parameters. The boolean value means whether the protein made from the sequence would be transported to the ER.

Then, our task is to find the "best" values for position, length, aa_index and threshold that explain the experimental facts. By means of a brute-force search, we found:

$$\begin{aligned}&\mathit{good\_views\_for\_er} : \texttt{view}\{\mathsf{sequence} : \mathit{string}\}\,\mathit{bool}\ \mathit{list} =\\ &\quad[V_1\{\mathsf{position} = 5; \mathsf{length} = 20; \mathsf{aa\_index} = \texttt{"NADH010103"}; \mathsf{threshold} = 635\};\\ &\quad\ V_1\{\mathsf{position} = 5; \mathsf{length} = 20; \mathsf{aa\_index} = \texttt{"KYTJ820101"}; \mathsf{threshold} = 18.5\};\\ &\quad\ V_1\{\mathsf{position} = 5; \mathsf{length} = 20; \mathsf{aa\_index} = \texttt{"JURD980101"}; \mathsf{threshold} = 15.68\}; \ldots]\end{aligned}$$

We evaluated these hypothetical views (according to a certain criterion called MCC [26], in which 1.0 means the best and 0.0 the worst) by implementing a function $\mathit{evaluate\_view\_for\_er} : \texttt{view}\{\mathsf{sequence} : \mathit{string}\}\mathit{bool} \rightarrow \mathit{float}$.

```
> List.map evaluate_view_for_er good_views_for_er ;;
- : float list = [0.874716968026; 0.873939179445; 0.861257759764; ...]
```

We confirmed that these numbers are comparable to those in other work [12] and our hypotheses are indeed reasonable from biological viewpoints [36] (e.g., all three AA-indices shown above are related to the hydropathy of the amino acids, and the hypotheses capture the hydrophobic h-region common to signal peptides).

Secondly, we applied the same approach to characterize the amino-acid sequence of proteins that are transported to mitochondria and chloroplasts (both of them are a kind of "power stations" in living cells). The data set is the amino-acid sequences of 509 proteins (in plant cells) that are carried to mitochondria or chloroplasts, and 162 proteins that are not.

This time, however, the view V_1 above did not yield any good hypotheses: even the best parameters scored less than 0.7. We therefore defined another view V_2:

$$\begin{array}{l}\texttt{view}\ V_2\{\mathsf{sequence} = s; \mathsf{position} = p; \mathsf{length} = l;\\ \qquad\qquad \mathsf{alphabet\_indexing} = i; \mathsf{pattern} = p; \mathsf{mismatch\_allowance} = m\} =\\ \quad \mathit{astrstr}\ m\ p\ (\mathit{apply\_alphabet\_indexing}\ i\ (\mathit{substring}\ p\ l\ s))\end{array}$$

The first three arguments sequence, position and length are the same as in V_1. Instead of AA-indexing and averaging, however, V_2 uses alphabet indexing [32] (mapping from each amino acid to some small natural number) and approximate string matching with a bounded number of mismatches allowed. Since this hypothetical view neither performed well by itself—in fact, it even tends to give the opposite answers!—we furthermore tried the combination of two views:

$$\begin{array}{l}\texttt{view}\ V_3\{\mathsf{v}_1 = v_1; \mathsf{v}_2 = v_2; \mathsf{sequence} = s\} =\\ \quad \texttt{valof}\ v_1\{\mathsf{sequence} = s\} \wedge \neg(\texttt{valof}\ v_2\{\mathsf{sequence} = s\})\end{array}$$

Then, we obtained the following hypotheses that make biological sense [37] (e.g., the amino acids are seldom negatively charged and the signals are at the N-terminal)

$$
\begin{array}{l}
\mathit{good\_views\_for\_mit\_chl} : \texttt{view}\{\mathsf{sequence} : \mathit{string}\}\mathit{bool\ list} = \\
\quad [V_3\{\mathsf{v}_1 = V_1\{\mathsf{position} = 0; \mathsf{length} = 30; \\
\qquad\qquad\qquad \mathsf{aa\_index} = \texttt{"FAUJ880112"}; \mathsf{threshold} = 3\}; \\
\qquad \mathsf{v}_2 = V_2\{\mathsf{position} = 0; \mathsf{length} = 10; \mathsf{alphabet\_indexing} = \\
\qquad\qquad\qquad [(\texttt{"DE"}, 0); (\texttt{"AR"}, 1); (\texttt{"BCFGHIKLMNPQSTVWXYZ"}, 2)]; \\
\qquad\qquad\qquad \mathsf{pattern} = \texttt{"2022202222"}; \mathsf{mismatch\_allowance} = 3\}\}; \\
\quad V_3\{\mathsf{v}_1 = V_1\{\mathsf{position} = 0; \mathsf{length} = 30; \\
\qquad\qquad\qquad \mathsf{aa\_index} = \texttt{"FAUJ880112"}; \mathsf{threshold} = 3\}; \\
\qquad \mathsf{v}_2 = V_2\{\mathsf{position} = 0; \mathsf{length} = 10; \mathsf{alphabet\_indexing} = \\
\qquad\qquad\qquad [(\texttt{"DE"}, 0); (\texttt{"CR"}, 1); (\texttt{"ABFGHIKLMNPQSTVWXYZ"}, 2)]; \\
\qquad\qquad\qquad \mathsf{pattern} = \texttt{"2002222222"}; \mathsf{mismatch\_allowance} = 3\}\}; \\
\quad V_3\{\mathsf{v}_1 = V_1\{\mathsf{position} = 0; \mathsf{length} = 30; \\
\qquad\qquad\qquad \mathsf{aa\_index} = \texttt{"RICJ880106"}; \mathsf{threshold} = 26.9\}; \\
\qquad \mathsf{v}_2 = V_2\{\mathsf{position} = 0; \mathsf{length} = 10; \mathsf{alphabet\_indexing} = \\
\qquad\qquad\qquad [(\texttt{"DE"}, 0); (\texttt{"AR"}, 1); (\texttt{"BCFGHIKLMNPQSTVWXYZ"}, 2)]; \\
\qquad\qquad\qquad \mathsf{pattern} = \texttt{"2022202222"}; \mathsf{mismatch\_allowance} = 3\}\}; \ldots]
\end{array}
$$

with scores

$$
\begin{array}{l}
\texttt{>}\ \mathit{List.map\ evaluate\_view\_for\_mit\_chl\ good\_views\_for\_mit\_chl}\ ;; \\
\texttt{-}\ :\ \mathit{float\ list} = [0.771504; 0.766387; 0.762190; \ldots]
\end{array}
$$

which are again comparable to the results in other work [12, 37].

Note that the advantages of views—that is, they have visible representations of an extensible data type and can be applied to multiple arguments in an arbitrary order—were of great help in the above process of knowledge discovery. Indeed, it took only a few days for us to obtain these results from the very beginning of this task. Without VMλabl, the process would have been far less pleasant.

As for execution performance, most of the computation time was spent on the approximate string matching during the brute-force search of good V_2. The actual CPU power spent was about 84 processor-hours on UltraSPARC III 900 MHz. Since our implementation is based on the interactive OCaml bytecode interpreter for engineering reasons, there is still much room left for efficiency improvement.

6 Related Work and Future Work

ML-Like Exceptions. The representation part of a view is similar to ML-like exceptions in that they carry arguments, have generative constructors, and can be pattern-matched. Indeed, representations of views can be implemented by exceptions. (In the implementation above, we did not take this approach because of the lack of type information in Camlp4: in OCaml, exception definitions require explicit type annotations.) However, views are different from ML-like exceptions in that a view has its own value part in addition to the representation part. Nevertheless, techniques developed for exceptions, such as static detection of uncaught exceptions [21, 41–44], can perhaps be adapted for views, for example to analyze the flow of view constructors and check the exhaustiveness (or redundancy) of pattern matching.

Views for Abstract Data Types. Originally, hypothetical views had nothing to do with views for abstract data types [39]. Taking a hindsight, however, it would be interesting to consider implementing hypothetical views by using views for abstract data types, because they might enable hiding the type of the representation part of a view, yet allowing pattern matching over this representation.

Generative Names. As stated before, view constructors in (the formalization of) VMλ are generative. In our formal semantics, however, fresh generation of view constructors is assumed *a priori* and left implicit. More rigorous treatment of generative names is already studied in the literature [31, 33, 35, etc.].

Extensions of Hypothetical Views. Since a view is just a pair of a value and its representation, it is straightforward to introduce *recursive views* on the basis of standard recursive types and recursive functions.

It is also straightforward to extend VMλ for remembering and pattern-matching the values of the free variables of a view definition, for example like:

```
# let x = 3 ;;
val x : int = 3
# view V{y = y} = x + y ;; (* x is free in the definition of V *)
view V of { _x : int = 3; y : int } : int
# let v = V{y = 7} ;;
val v : int view = 10 as V{_x = 3; y = 7} ;;
# vmatch v with V{_x = x; y = y} -> (x, y) ;;
- : int * int = (3, 7)
```

Views for views deserve more consideration. For example, suppose that we define a view constructor W applying another view constructor V as follows.

```
# view V{x = x; y = y} = x + y ;;
view V of { x : int; y : int } : int
# view W{z = z} = V{x = 1; y = z + 2} ;;
view W of { z : int } : int view
```

Currently, applying W yields a view whose value is another view.

```
# let v = W{z = 3} ;;
val v : int view view = (6 as V{x = 1; y = 5}) as W{z = 3} ;;
# vmatch v with W{z = z} -> z ;;
- : int = 3
# vmatch v with V{x = x; y = y} -> (x, y) ;;
Uncaught exception: VMatchFailure.
```

```
# vmatch (valof v) with V{x = x; y = y} -> (x, y) ;;
- : int * int = (1, 5)
```

However, it may be useful if one can pattern-match this view both with V and with W, for example as follows.

```
# let v = W{z = 3} ;;
val v : int view = 6 as V{x = 1; y = 5} and W{z = 3} ;;
# vmatch v with W{z = z} -> z ;;
- : int = 3
# vmatch v with V{x = x; y = y} -> (x, y) ;;
- : int * int = (1, 5)
```

As for fully applied views, this extension is straightforwardly realizable by remembering a list of representations. As for partially applied views, however, the extension is more subtle for the same reason as pattern matching over λ-abstracted views was problematic in the original VML (cf. Section 3).

Other Directions. Other examples of scientific knowledge discovery using the notion of views are found in previous work [6, 7, 22–24]. A more user-friendly interface to the language for non-programmers is also a future topic of research interest.

Acknowledgments

We would like to thank the members of Prof. Miyano's group and Prof. Yonezawa's group in University of Tokyo, as well as the anonymous reviewers, for suggestions and encouragements on this work.

References

1. Hassan Aït-Kaci and Jacques Garrigue. Label-selective lambda-calculus: Syntax and confluence. In *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 24–40. Springer-Verlag, 1993.
2. Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, and Peter Walter. *Molecular biology of the cell*. Garland Science, 4th edition, 2002.
3. Setsuo Arikawa and Koichi Furukawa, editors. *Proceedings of the Second International Conference on Discovery Science*, volume 1721 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1999.
4. Setsuo Arikawa and Shinichi Morishita, editors. *Proceedings of the Third International Conference on Discovery Science*, volume 1967 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2000.
5. Setsuo Arikawa and Hiroshi Motoda, editors. *Proceedings of the First International Conference on Discovery Science*, volume 1532 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1998.

6. Hideo Bannai, Yoshinori Tamada, Osamu Maruyama, and Satoru Miyano. VML: A view modeling language for computational knowledge discovery. In *Proceedings of the Fourth International Conference on Discovery Science*, volume 2226 of *Lecture Notes in Artificial Intelligence*, pages 30–44, 2001.
7. Hideo Bannai, Yoshinori Tamada, Osamu Maruyama, Kenta Nakai, and Satoru Miyano. Views: Fundamental building blocks in the process of knowledge discovery. In *Proceedings of the 14th International FLAIRS Conference*, pages 233–238. AAAI Press, 2001.
8. Hideo Bannai, Yoshinori Tamada, Osamu Maruyama, Kenta Nakai, and Satoru Miyano. Extensive feature detection of n-terminal protein sorting signals. *Bioinformatics*, 18(2):298–305, 2002.
9. Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 242–257, 1996.
10. Olivier Danvy. Type-directed partial evaluation. In *Partial Evaluation – Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 367–411. Springer-Verlag, 1999.
11. Daniel de Rauglaudre. Camlp4. `http://caml.inria.fr/camlp4/`.
12. O. Emanuelsson, H. Nielsen, S. Brunak, and G. von Heijne. Predicting subcellular localization of proteins based on their N-terminal amino acid sequence. *J. Mol. Biol.*, 300(4):1005–1016, July 2000.
13. Jacques Garrigue. OLabl. `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/`.
14. Jacques Garrigue. Programming with polymorphic variants. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.
15. Jacques Garrigue. Relaxing the value restriction. `http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/`, 2003.
16. Jacques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective lambda-calculus. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 35–47, 1994.
17. Haskell in practice (applications written in Haskell). `http://www.haskell.org/practice.html`.
18. John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.
19. S. Kawashima and M. Kanehisa. AAindex: Amino Acid index database. *Nucleic Acids Res.*, 28(1):374, 2000.
20. Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.
21. Xavier Leroy and Francois Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000. An extended abstract appeared in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999, pages 276–290.
22. Osamu Maruyama and Satoru Miyano. Design aspects of discovery systems. *IEICE Transactions in Information and Systems*, E83-D(1):61–70, 2000.
23. Osamu Maruyama, Tomoyuki Uchida, Takayoshi Shoudai, and Satoru Miyano. Toward genomic hypothesis creator: View designer for discovery. In Arikawa and Motoda [5], pages 105–116.
24. Osamu Maruyama, Tomoyuki Uchida, Kim Lan Sim, and Satoru Miyano. Designing views in hypothesiscreator: System for assisting in discovery. In Arikawa and Furukawa [3], pages 115–127.

25. Brian Mathews, Daniel Lee, Brian Dister, John Bowler, Howard Cooperstein, Ajay Jindal, Tuan Nguyen, Peter Wu, and Troy Sandal. VML - the vector markup language. `http://www.w3.org/TR/NOTE-VML`.
26. B. W. Matthews. Comparison of predicted and observed secondary structure of t4 phage lysozyme. *Biochim. Biophys. Acta*, 405:442–451, 1975.
27. Objective Caml. `http://caml.inria.fr/`.
28. Atsushi Ohori and Nobuaki Yoshida. Type inference with rank 1 polymorphism for type-directed compilation of ML. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 160–171, 1999.
29. Our users' achievements (significant applications written in Caml). `http://caml.inria.fr/users_programs-eng.html`.
30. Didier Rémy. Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 77–88, 1989.
31. Claudio V. Russo. *Types For Modules.* PhD thesis, University of Edinburgh, 1998. `http://www.dcs.ed.ac.uk/home/cvr/ECS-LFCS-98-389.html`.
32. Shinichi Shimozono. Alphabet indexing for approximating features of symbols. *Theoretical Computer Science*, 210(2):245–260, 1999.
33. Ian Stark. *Names and Higher-Order Functions.* PhD thesis, University of Cambridge, 1994. `http://www.dcs.ed.ac.uk/home/stark/publications/thesis.html`.
34. Gerd Stolpmann. The O'Caml link database. `http://www.npc.de/ocaml/linkdb/`.
35. Eijiro Sumii and Benjamin Pierce. Logical relations for encryption. In *14th IEEE Computer Security Foundations Workshop*, pages 256–269, 2001.
36. G. von Heijne. The signal peptide. *J. Membr. Biol.*, 115:195–201, 1990.
37. G. von Heijne, J. Steppuhn, and R. G. Herrmann. Domain structure of mitochondrial and chloroplast targeting peptides. *Eur. J. Biochem.*, 180:535–545, 1989.
38. Philip Wadler. Functional programming in the real world. `http://www.research.avayalabs.com/user/wadler/realworld/`.
39. Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 307–313, 1987.
40. Philip Wadler. Functional programming: An angry half-dozen. *SIGPLAN Notices*, 33(2):25–30, 1998.
41. Kwangkeun Yi. Compile-time detection of uncaught exceptions in Standard ML programs. In *Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 238–254. Springer-Verlag, 1994.
42. Kwangkeun Yi. An abstract interpretation for estimating uncaught exceptions in Standard ML programs. *Science of Computer Programming*, 31(1):147–173, 1998.
43. Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 98–113. Springer-Verlag, 1997.
44. Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in SML programs. *Theoretical Computer Science*, 277(1–2):185–217, 2002.