Regular Paper

A Type Safe Access to Key-value Stores from Functional Languages

Katsuhiro Ueno[1,a) Atsushi Ohori[1,b)

Received: April 6, 2015, Accepted: July 28, 2015

Abstract: This paper presents a scheme comprising a type system and a type-directed compilation method that enables users to integrate high-level key-value store (KVS) operations into statically typed polymorphic functional languages such as Standard ML. KVS has become an important building block for cloud applications because of its scalability. The proposed scheme will enhance the productivity and program safety of KVS by eliminating the need for low-level string manipulation. A prototype that demonstrates its feasibility has been implemented in the SML# language and clarifies issues that need to be resolved in further development towards better practical performance.

Keywords: key-value store, functional programming, SML#

1. Introduction

Cloud storage, such as Google BigTable [2] and Amazon S3 [3], has been attracting attention as a data access framework for highly reliable data storage infrastructures. Such storages usually organize large amounts of data as *Key-value stores* (KVSs), in which keys are mapped to values. Every value in a KVS is paired with a unique key. Users search for a key in a KVS and obtain a value corresponding to that key. Because all key-value pairs in a KVS are naturally independent of each other, data stored in a KVS can be distributed and replicated over a cluster of network nodes. As a result of this property, KVS is widely used in data storages that require high performance, scalability and fault tolerance.

Whereas studies have been conducted on KVS with the objective of achieving efficient data distribution and replication, to the best of our knowledge, high-level programming techniques using KVS have not been elucidated. KVS-based storage servers provide network protocols and APIs for data access. However, both of the protocols and APIs only support string data. Consequently, in order to store data other than strings in KVS, programmers have to first serialize those data in a network-safe format that can interact with KVS through the string-based APIs. This simple approach works effectively when the program in question dealing with values of basic types such as integers and Booleans; however, it does not scale up to programs that use large and complex data such as nested records and arrays because of the need to write low-level string manipulation codes for cumbersome data serialization. This style of programming also undermines the benefit of type checking in a strongly typed programming language that ensures the type safety of a program. Thus, if a technique that facilitates safe and effective utilization of KVS is estabilished, then programmers would be able to develop highly scalable cloud applications using KVS without the above disadvantages.

Our general objective is to develop a framework and compilation method that realizes high-level and type-safe access to KVS in an ML-style functional language. Pursuant to this goal, this paper proposes a *high-level KVS* scheme that allows users to store compound data such as arrays and tuples in a KVS through type-consistent operations in ML, without any string manipulation. The high-level KVS scheme comprises a language extension of ML and a type-directed compilation algorithm for the extension. Both the language extension and compilation algorithm can be straightforwardly merged into an ML compiler equipped with record polymorphism [9]. In this paper, we use SML#, a variant of Standard ML, as an example of such an ML compiler. SML# has already achieved type-safe and easy-to-use access to relational databases [10]. Consequently, combining the proposed scheme with the database access of SML# facilitates the development of applications that can seamlessly integrate local databases and distributed data storages.

The proposed scheme is closely related to data serialization methods in typed functional languages in the sense that both the techniques and KVS externalize internal data structures without losing their structures and type consistency. Examples of such serialization methods include user-level combinator libraries for serialization [4], [7], memory dumps annotated with type information [1], [8], and type inference on memory graphs [6]. These research works focus on serialization of the entire data all at once and effcient checking of the type safety of the data. In contrast, we focus on utilization of string-based KVS in typed functional languages; the objective is to realize features important in practice, such as portable data representation and partial access to large data structures in KVS.

[1] Research Institute of Electrical Communication, Tohoku University, Sendai, Miyagi 980–8577, Japan
a) katsu@riec.tohoku.ac.jp
b) ohori@riec.tohoku.ac.jp

The remainder of this paper is organized as follows. Section 2 discusses the problems to be solved and outlines our strategy. Section 3 defines the type system of KVS that yields high-level KVS. Section 4 presents the language extension for high-level KVS and their implementation strategy. Section 5 outlines our prototype implementation of high-level KVS in SML#. Section 6 discusses issues identified that need to be resolved for better practical performance. Section 7 concludes this paper.

2. Problems and Our Strategy

With the exception of the implementation details of KVS, such as key-value pair distribution and fault tolerance, a KVS can be regarded as an abstract data structure characterized by the following operations in ML:

$\texttt{new} : \texttt{string} \times \texttt{string} \to \texttt{unit}$

$\texttt{get} : \texttt{string} \to \texttt{string}$

$\texttt{put} : \texttt{string} \times \texttt{string} \to \texttt{unit}$

The type `string` denotes the set of keys and values. $\texttt{new}(k,v)$ creates a new pair comprising a key k and value v in KVS. If k already exists in KVS, then $\texttt{new}(k,v)$ raises an exception. $\texttt{get}(k)$ searches for k in KVS and returns the value that is currently paired with k. $\texttt{put}(k,v)$ updates the value of k with v. Both $\texttt{get}(k)$ and $\texttt{put}(k,v)$ raise an exception if k does not exist in KVS. This abstract view of KVS is sufficient to understand the problem. Note that this simple outline does not directly correspond to KVS implementations in the real world, which organize key-value pairs in more sophisticated ways; for example, BigTable uses a triple of row, column and timestamp as a key; and Amazon S3 organizes key-value pairs in a set of *buckets*, each of which is a set of key-value pairs. These methodologies can be regarded as variations of the key structure and therefore are covered by the above abstraction.

Obviously, the above operations can be implemented as library functions in any programming language; therefore, there is no technical issue with integration of these operations into a language. As stated in Section 1, KVS implementations provide APIs corresponding to the above abstraction. While these operations can be used easily for easy-to-serialize data such as integers, they are not suitable for dealing with complex and large data structures.

A natural way to represent a complex data structure in KVS through the above string-based operations is to encode the data structure into key strings using a character that is not used by the user. Let / be such a reserved character. For example, the following set of key-value pairs represents a pair of key A and an array of 10 integers:

Key	Value
$A/\overline{0}$	$\overline{\text{1st element}}$
···	···
$A/\overline{9}$	$\overline{\text{10th element}}$

where $\overline{n}$ is the string representation of integer n, and $A/\overline{i}$ is the string A followed by / and $\overline{i}$. Because the length of a key is not limited, array structures of any length can be encoded in key strings. In this encoding, $\texttt{get}(A/\overline{i})$ is the operation used to obtain the i-th element of array A.

Nested data structures can also be represented in KVS by defining similar encoding rules inductively. For example, a 10×10 matrix M can be represented by including two indices in key strings as follows:

Key	Value
$M/\overline{0}/\overline{0}$	$\overline{(1,1)^{\text{th}}\text{ entry}}$
···	···
$M/\overline{9}/\overline{9}$	$\overline{(10,10)^{\text{th}}\text{ entry}}$

To obtain the $(i+1, j+1)^{\text{th}}$ entry of matrix M, the operation $\texttt{get}(M/\overline{i}/\overline{j})$ is performed. This key structure can also be regarded as a one-dimentional array, with each element also a one-dimentional array; $M/\overline{0}, \ldots, M/\overline{9}$ are the keys of the 1st, ..., 10th element of M, and the value of each of these keys has a similar key structure to M.

As seen in the above examples, compound data structures can be represented in KVS by encoding their structures in key strings. However, implementing such encodings using the string-based APIs would make programs complicated and vulnerable because such implementations usually consist of large amounts of untyped code, which may include potential errors that the type checker of a compiler cannot detect statically. In addition, it would make it difficult to interoperate data stored in KVS with other data in the heap.

To overcome the above issues, KVS must therefore be seamlessly integrated with the type system of high-level programming languages. The major requirements for the seamless integration of KVS and a programming language are the following:

(1) *High-level data access operations for KVS.* Basic operations on typical compound data structures must be available for data in KVS as well as those in memory without writing cumbersome and unsafe key encoding codes by hand.
(2) *Type-safe access to KVS.* Type safety must hold even in programs that utilize KVS.

Hereafter, we refer to the KVS satisfying the above requirements as *high-level KVS.*

For the first requirement, similar to the above examples, we represent a compound value v paired with a key k as a set of key-value pairs S such that the structure of v is encoded in the keys of S and all keys in S begin with k. In a typed language, the structure of a value v is statically determined in the type of the expression that will evaluate to v; therefore, all keys in S are generated at compile time. For example, the type of M in the above example is τ `array array` in ML, where τ is the type of an entry of the matrix. From the structure of this type term, the ML compiler statically determines that each key in the key-value pairs representing M must include two indices, as stated above. This means that the high-level data access operations can be realized by extending the compiler with a compilation scheme that automatically generates string-based codes that manage the key encoding and data serialization from the type of KVS access operations.

The second requirement is the main issue of the integration. ML compilers compute the types of all data structures that the program potentially generates and consumes. This is the procedure performed by the type checker of ML to ensure the type safety of low-level memory operations. If programs using KVS dealt only with strings, then the above definitions of `new`, `get`, and `put` would be enough for type-safe KVS operations. To extend those operations to types other than strings, however, the type system of ML must be extended so that it can compute the types of data structures stored in KVS.

In the sense of the above key encoding schemes, the string-based KVS can be regarded as a low-level data representation for higher-level compound data structures. This directly corresponds to the fact that all data structures in ML are stored in memory as low-level binary data whose structures are statically determined by types. This immediately yields the following strategy. As in ML, we define a type system for KVS as a set of derivation rules that computes the types of all compound data structures encoded in string-based KVS. The type-safe and high-level KVS operations are realized in ML by extending ML with the type system for KVS.

The only remaining issue is definition of the model of KVS that captures real usage of KVS as well as makes the type system sound. If a KVS is occupied by a program and used as an internal heap area that is allocated and freed at the start and end of the process, we can regard each key-value pair in the KVS as a mutable memory similar to the `ref` type of ML. However, KVS is usually used as a storage that is independent of programs and therefore it may be manipulated by other processes. In addition, because KVS is an untyped storage, the meaning of the values stored in KVS depends on the interpretation of each program. For example, at the same time a program is reading an integer value from key k, another program may delete k and create a new k paired with a Boolean value. Therefore, compile-time type checking of a program cannot ensure that KVS is always consistent with respect to the typing of a program. This issue is not limited to KVS but is in fact a general issue associated with the use of external data in a typed programming language.

One approach to dealing with external persistent data in a typed language is to include type information in the external data and checking the type consistency by comparing the type information to the types of the program at runtime. The type system of dynamic types proposed by Abadi et al. [1] realizes this approach. Their proposed system introduces a special type dynamic that denotes external objects whose type cannot be determined statically and defines rules to create and use the external objects. We summarize the rules as two expressions `create` and `use` whose typing rules are given as follows:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{create}(e) : \mathsf{dynamic}}$$

$$\frac{\Gamma \vdash e : \mathsf{dynamic}}{\Gamma \vdash \texttt{use}\ e\ \texttt{as}\ \tau : \tau}$$

where dynamic is the type of the externalized data, `create`(e) externalizes the value of e coupled with metadata representing the type information τ of e, and `use` e `as` τ reads externalized data e as a value of τ if the metadata of e is equivarent to τ. This equivalence check is performed at runtime. If the check fails, this expression raises a runtime error. If the check passes, the externalized data are read as a value of type τ and therefore the system holds the type safety.

We combine the system of dynamic and the above key encoding schemes to realize high-level KVS independent of programs and type-safe operations for high-level KVS data access in ML. These high-level operations allow programmers to enjoy KVS programming with the same type safety as string-based KVS but without cumbersome string manipulations.

3. A Type System for High-Level KVS

This section defines the data model of high-level KVS and corresponding type system that are independent of any specific programming language. The type system yields a strategy for realizing high-level KVS on the top of string-based KVS as well as a foundation for interoperation between KVS and typed langauges. On the basis of this type system, in the next section, we design a polymorphic interface from ML to high-level KVS.

3.1 Structure of String-based KVS

To define the model and type system of KVS, we introduce the following assumptions and notations. We consider a string-based KVS as a finite map over strings. Let S be a meta-variable indicating a string-based KVS. Let Σ and Σ^+ be the given set of characters and strings. Let k and v range over the set of keys and values in string-based KVS, i.e., Σ^+. $\{k_1 \mapsto v_1, \dots, k_n \mapsto v_n\}$ is the extensive notation of a KVS. $\mathrm{dom}(S)$ is the set of keys in S, and $S(k)$ is a value corresponding to k in S if $k \in \mathrm{dom}(S)$. We sometimes consider S as a set of key-value pairs. We define $S_1 \cup S_2$ as the *union* of S_1 and S_2; that is, the union set of key-value pairs of S_1 and S_2. $S_1 \cup S_2$ is defined only if $S_1(k) = S_2(k)$ for any $k \in \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2)$. For simplicity, we implicitly assume this condition when we write $S_1 \cup S_2$.

We additionally define the following related to keys:

- Let `/` be a reserved character used to encode the data structure in key strings. As we shall present in Section 4, the user is not allowed to include `/` in key strings of high-level KVS. k_1/k_2 is the concatenation of k_1, `/` and k_2.
- We write the string representation of integer n and type σ as $\overline{n}$ and $\overline{\sigma}$. The function $\overline{n}$ and $\overline{\sigma}$ are injective. These representations are used for the data structure encoding and meta-information of externalized data.

3.2 Types and Their Semantics

Let b range over the set of basic types such as integers (`int`), Booleans (`bool`) and floating-point numbers (`real`). The set of data types stored in high-level KVS (ranged over by σ) is given by the following syntax:

$$\sigma ::= b \mid \mathit{Pair}(\sigma, \sigma) \mid \mathit{Array}(\sigma)$$

$\mathit{Pair}(\sigma_1, \sigma_2)$ is the type of pairs of values of type σ_1 and σ_2. $\mathit{Array}(\sigma)$ is the type of arrays of values of type σ. This set of types is large enough to analyze the issues of the type system of the high-level KVS.

We define the data representations in KVS of the above types in the following strategy. In general, the data representation of a

$$\begin{aligned}
\mathcal{R}(b) &= \Big\{(k, \{k \mapsto v\}) \,\Big|\, k \in \Sigma^{+}, v \in \mathcal{E}(b)\Big\} \\
\mathcal{R}(Pair(\sigma_1, \sigma_2)) &= \Big\{(k, \{k \mapsto \overline{Pair(\sigma_1, \sigma_2)}\} \cup S_1 \cup S_2) \\
&\qquad \Big|\, (k/\overline{1}, S_1) \in \mathcal{R}(\sigma_1), (k/\overline{2}, S_2) \in \mathcal{R}(\sigma_2)\Big\} \\
\mathcal{R}(Array(\sigma)) &= \bigcup_{n \geq 0} \mathcal{R}'(Array(\sigma), n)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{R}'(Array(\sigma), n) = \\
&\quad \Big\{(k, \left\{\begin{array}{lll} k & \mapsto & \overline{Array(\sigma)}, \\ k/\texttt{len} & \mapsto & \overline{n}, \end{array}\right\} \cup S_0 \cup \cdots \cup S_n) \\
&\qquad \Big|\, (k/\overline{0}, S_0) \in \mathcal{R}(\sigma), \ldots, (k/\overline{n-1}, S_n) \in \mathcal{R}(\sigma)\Big\}
\end{aligned}$$

Fig. 1 Data model of high-level KVS.

new type is introduced as a new data constructor combined with values of existing types as with the variant types in ML; however, we cannot follow this approach because the data model of our high-level KVS must be constructed on top of the existing model of string-based KVS, which allows us to use only string pairs and the key encoding. We therefore derive the model of the types directly from the string-based low-level representation. Each of the types of KVS is either a basic type or a possibly nested compound type. For data of a basic type b, we use their string representations as the model of b because this is the commonly accepted way to store basic values in KVS. Consequently, we assume that there is a system-independent and network-safe standard string data representation for each b, denoted by $\mathcal{E}(b)$. For data of a compound type, we represent their structures as the a set of key-value pairs. In the heap memory, the model of a compound type is tree-structured data consisting of pointers and memory blocks that represent the (possibly recursive) structure of the model. In KVS, we design similar data representation to the heap memory by encoding the recursive structures in key strings and pairing the keys with values that the structure contains.

Following the above strategy, we define the data model $\mathcal{R}(\sigma)$ of type σ as a set of pairs (k, S) of key k and KVS S that realizes the structure of values of type σ. **Figure 1** shows the definition of $\mathcal{R}(\sigma)$. In the case of basic types, the value paired with k is in the standard string representation. We assume that the standard string representations of basic types are different from each other. In the case of compound types, k represents the root of the data structure that S realizes. k is paired with $\overline{\sigma}$, which is the string representation of type σ, for runtime type checking. We also assume that the string representation of types is not identical to the standard string representation of any data. By this assumption, the type information of a basic type value is uniquely determined from its standard string representation. Each element of a compound type data is stored in a key containing the identifier of the compound data. For example, in Fig. 1, the structure of a value of type $\mathcal{R}(Pair(\sigma_1, \sigma_2))$ is represented in the following three keys: k for its type information, $k/\overline{1}$ for its left element, and $k/\overline{2}$ for its right element. The data structures of $k/\overline{1}$ and $k/\overline{2}$ are recursively represented by S_1 and S_2, respectively. The KVS representing the entire pair is the union of S_1, S_2 and $\{k \mapsto \overline{Pair(\sigma_1, \sigma_2)}\}$.

We say that key k has type σ under S, denoted by $S \vdash k : \sigma$, if there is some S' such that $S' \subseteq S$ and $(k, S') \in \mathcal{R}(\sigma)$. The following is an instance of k and S satisfying $S \vdash k : Pair(\texttt{int}, Pair(\texttt{real}, \texttt{bool}))$:

$$\begin{array}{llll}
S = \{ & k & \mapsto & \overline{Pair(\texttt{int}, Pair(\texttt{real}, \texttt{bool}))}, \\
& k/\overline{1} & \mapsto & \overline{n}, \\
& k/\overline{2} & \mapsto & \overline{Pair(\texttt{real}, \texttt{bool})}, \\
& k/\overline{2}/\overline{1} & \mapsto & \overline{r}, \\
& k/\overline{2}/\overline{2} & \mapsto & \overline{l}\ \}
\end{array}$$

where n, r and l are values of `int`, `real`, and `bool`, respectively. The type check of KVS can be carried out independently from any program; therefore, this model enables the runtime type check performed at the `use` expression in the dynamic type system.

4. Type-safe Access to High-level KVS

Access operations from a program to a high-level KVS must be functions that realize the data encoding presented in Section 3. Those functions must also be used polymorphically for any data types that can be stored in the high-level KVS. In this section, we define the functions in the following three steps. Firstly, we define monomorphic functions for each type σ that reads and writes a value of σ in KVS. Secondly, we extend the functions to polymorphic ones based on the idea of the type-directed compilation for polymorphic records [9] and present a compilation scheme for them. Lastly, we refine the presented scheme to ones that can be used in practice in ML.

4.1 Monomorphic Access Operations

The host language has its own set of types including basic types, tuples and arrays. For simplicity, we use the type terms of KVS defined in Section 3 as those of the host language. We write $\tau_1 \times \tau_2$ instead of $Pair(\tau_1, \tau_2)$ to represent the fact that pairs of $\tau_1 \times \tau_2$ are consumed in the host language and are not relevant to KVS. In the discussion below, we let σ also range over the set of types of the host language except for type variables.

The type-safe access functions from the host language to high-level KVS must be the following:

$$\begin{aligned}
\texttt{create}_\sigma &: \texttt{string} \times \sigma \to \texttt{unit} \\
\texttt{update}_\sigma &: \texttt{string} \times \sigma \to \texttt{unit} \\
\texttt{find}_\sigma &: \texttt{string} \to \sigma
\end{aligned}$$

These three functions correspond to the three low-level KVS access functions presented in Section 2. $\texttt{create}_\sigma$ and $\texttt{update}_\sigma$ create the structure defined through $\mathcal{R}(\sigma)$ in KVS from the given key and value of type σ. $\texttt{find}_\sigma$ reads a value of σ from the given key k if k exists in KVS and the meta-information of k stored in KVS is equivalent to σ. To preserve the consistency of high-level KVS, keys given to these functions never include the / character, which is reserved for internal use by high-level KVS. For simplicity, we omit the check for / in the given keys in the discussion below.

The function $\texttt{update}_\sigma$ performs runtime type checking similar to $\texttt{find}_\sigma$ to prevent overwriting of the value of a key with a value of a different type from the type of the key. As discussed in Section 2, KVS is an untyped storage thus the user can change the type of the keys without any restrictions. In contrast, as seen in the `ref` type in ML, destructive update operations in a typed functional language do not usually change the type of the updated value. Introduction of an update operation that may change the

```
$\texttt{create}_b = \lambda(k,v).\,\texttt{new}(k, \texttt{toString}_b\ v)$

$\texttt{create}_{Pair(\sigma_1,\sigma_2)} = \lambda(k,(v_1,v_2)).\,(\texttt{new}(k, \overline{Pair(\sigma_1,\sigma_2)});$
                                   $\texttt{create}_{\sigma_1}\ (k/\overline{1}, v_1);$
                                   $\texttt{create}_{\sigma_2}\ (k/\overline{2}, v_2))$

$\texttt{create}_{Array(\sigma)} = \lambda(k,a).\,(\texttt{new}(k, \overline{Array(\sigma)});$
                          $\texttt{new}(k/\texttt{len}, \texttt{toString}_{\texttt{int}}\ |a|);$
                          for each $v_i \in a = [v_0,\ldots,v_{n-1}]$,
                              $\texttt{create}_\sigma\ (k/\overline{i}, v_i))$

$\texttt{update}_b = \lambda(k,v).\,(\texttt{fromString}_b\ (\texttt{get}(k));$
                 $\texttt{put}(k, \texttt{toString}_b\ v))$

$\texttt{update}_{Pair(\sigma_1,\sigma_2)} = \lambda(k,(v_1,v_2)).$
                     $(\texttt{mustEqual}\ (\texttt{get}(k), \overline{Pair(\sigma_1,\sigma_2)});$
                      $\texttt{update}_{\sigma_1}\ (k/\overline{1}, v_1);$
                      $\texttt{update}_{\sigma_2}\ (k/\overline{2}, v_2))$

$\texttt{update}_{Array(\sigma)} = \lambda(k,a).\,(\texttt{mustEqual}\ (\texttt{get}(k), \overline{Array(\sigma)});$
                          $\texttt{put}(k/\texttt{len}, \texttt{toString}_{\texttt{int}}\ |a|);$
                          for each $v_i \in a = [v_0,\ldots,v_{n-1}]$,
                              $\texttt{update}_\sigma\ (k/\overline{i}, v_i))$

$\texttt{find}_b = \lambda k.\,\texttt{fromString}_b\ (\texttt{get}(k))$

$\texttt{find}_{Pair(\sigma_1,\sigma_2)} = \lambda k.\,(\texttt{mustEqual}\ (\texttt{get}(k), \overline{Pair(\sigma_1,\sigma_2)});$
                     $(\texttt{find}_{\sigma_1}\ (k/\overline{1}), \texttt{find}_{\sigma_2}\ (k/\overline{2})))$

$\texttt{find}_{Array(\sigma)} = \lambda k.\,(\texttt{mustEqual}\ (\texttt{get}(k), \overline{Array(\sigma)});$
                 let $n$ be $\texttt{fromString}_{\texttt{int}}\ (\texttt{get}(k/\texttt{len}))$ in
                 $[\texttt{find}_\sigma\ (k/\overline{0}), \ldots, \texttt{find}_\sigma\ (k/\overline{n-1})]$
```

Fig. 2 Monomorphic KVS access operations.

type requires careful and detailed consideration in a type theory, which is beyond the scope of this paper. For this reason also, we do not consider delete operations. Hereafter, for simplicity, we assume that the user never changes the types of existing keys and therefore omit their check.

The implementation of these functions are naturally derived from the structure of σ because the structure $\mathcal{R}(\sigma)$ is defined inductively on the structure of σ. **Figure 2** shows the implementations of $\texttt{create}_\sigma$, $\texttt{update}_\sigma$, and $\texttt{find}_\sigma$ in an ML-like pseudo language that includes the following syntax:

- $|a|$ denotes the number of elements in array a.
- $[e_1, \ldots, e_n]$ constructs an array with elements $e_1,\ldots,e_n$.
- let n be e_1 in e_2 is the expression that evaluates e_1, binds n to the value of e_1, and evaluates e_2.
- $(e_1; e_2)$ is sequential execution.

The following auxiliary functions are used in the figure:

- `new`, `put`, and `get` are the string-based access operations presented in Section 2.
- $\texttt{toString}_b$ and $\texttt{fromString}_b$ convert between the value of b and its standard string representation. Whereas $\texttt{toString}_b$ always succeeds, $\texttt{fromString}_b$ raises an exception if the conversion fails.
- `mustEqual` raises an exception if the two given strings are not equal.

$\texttt{update}_\sigma$ and $\texttt{find}_\sigma$ carry out the runtime type checking using $\texttt{fromString}_b$ for basic types and `mustEqual` for compound types. This runtime type checking is performed recursively on the structure of the data stored in KVS. This recursive behavior of $\texttt{update}_\sigma$ and $\texttt{find}_\sigma$ corresponds to the procedure that derives type judgment $S \vdash k : \sigma$ from KVS S and the given key k.

The definition outlined in Fig. 2 is inductive on the structure of σ with the exception of the generation of $\overline{\sigma}$. If there exists a function *PairTy* and *ArrayTy* on strings such that

$$\overline{Pair(\sigma_1,\sigma_2)} = PairTy(\overline{\sigma_1}, \overline{\sigma_2})$$
$$\overline{Array(\sigma)} = ArrayTy(\overline{\sigma}),$$

then the entire definition in Fig. 2 is inductive on the structure of σ. This premise of *PairTy* and *ArrayTy* is natural if we implement them in a functional programming language. Henceforth, we assume that both $\overline{Pair(\sigma_1,\sigma_2)}$ and $\overline{Array(\sigma)}$ satisfy the above equations. The inductive property of the high-level KVS access operations is required for extending them to polymorphic functions in the next subsection.

4.2 Polymorphic Access Operations

To execute the monomorphic version of $\texttt{create}_\sigma$, $\texttt{update}_\sigma$, and $\texttt{find}_\sigma$ presented in the previous subsection, the user must specify σ explicitly. However, as seen in the definition of these functions, they are generic operations for any type that can be stored in KVS. In ML, such generic functions are usually defined as polymorphic functions. For example, `foldr` is a typical polymorphic function that can be applied to any list of any element type. Because the three functions above are generic operations, they should also be defined as polymorphic functions in ML. The polymorphic version of the KVS access operations should look like functions of the following types:

$$\begin{aligned} \texttt{create} &: \forall t.\,\texttt{string} \times t \to \texttt{unit} \\ \texttt{update} &: \forall t.\,\texttt{string} \times t \to \texttt{unit} \\ \texttt{find} &: \forall t.\,\texttt{string} \to t \end{aligned}$$

However, because the behavior of each of the three functions varies as a result of the type instances of t, none of these functions can be implemented as a single code instance. Therefore, there is no parametric polymorphic function that realizes their behaviors.

We observe that this problem is in essence the same as the case of the record field selectors in the polymorphic record calculus proposed by Ohori [9]. In general, field selectors are defined only for those records whose label sets are statically determined. For example, the semantics of field selector `#L2` depends on the type of the given record. If the given record is {`L1: int, L2: int`}, then `#L2` reads the second word of the record. If the record is {`L2: int, L3: int`}, then `#L2` reads the first word of the record. In the polymorphic record calculus, to realize these various behaviors of field selectors without losing polymorphism, the compiler generates codes that pass field index information determined at type instantiation to polymorphic functions. The field index information is computed statically by introducing a *singleton type* whose value is uniquely determined. We refer to this compilation strategy as type-directed compilation. For example, the polymorphic record calculus gives `#L2` the following type:

$$\forall t_1.\forall t_2 :: \{\!\{\texttt{L2} : t_1\}\!\}.t_2 \to t_1$$

where $t_2 :: \{\!\{ \texttt{L2 : } t_1 \}\!\}$ signifies that t_2 may be any record type that has at least L2 field. Hence, t_2 may be instantiated to a record type such as {L1: int, L2: int} and {L2: int, L3: int}. The field index for #L2 is generated by the compiler from the instance of t_2.

We apply this approach to the KVS access operations to make them polymorphic. In the sense of KVS, the field selector and field index correspond to the access functions and a meta-object specifying a variation of the behavior of the access functions, respectively. We introduce a new singleton type to generate the meta-object from type instantiation.

Following the above strategy, we present the polymorphic KVS access operations and a compilation scheme for them as follows. For simplicity, in this subsection, we consider them based on a language with explicit type abstractions and type instantiations.

The KVS access operations must behave polymorphically only for types that can be stored in KVS; therefore, the type system must disallow their application to other data types such as first-class functions. To represent the type of such polymorphic behaviors, we introduce a type system with *type kinds* that restrict the set of type instances of a type variable. The set of monomorphic types (ranged over by τ) in the host language is given as follows:

$$\tau ::= t \mid b \mid \mathit{Pair}(\tau, \tau) \mid \mathit{Array}(\tau) \mid \cdots$$

where t ranges over the given set of type variables. We omit types used only in the host language such as record and function types. In contrast to σ, in τ, type variables may appear in $\mathit{Pair}(\tau_1, \tau_2)$ and $\mathit{Array}(\tau)$. A polymorphic type in the host language is of the form $\forall t_1 :: d_1. \ldots \forall t_n :: d_n. \tau$, which indicates that bound type variables $t_1, \ldots, t_n$ have type kinds $d_1, \ldots, d_n$ respectively. A type kind is either the universal kind U denoting the set of all types or *KVS* denoting the set of types compatible with KVS. A *kind assignment* ranged over by $\mathcal{K}$ is a finite map from type variables to type kinds. We say that type τ has kind d under $\mathcal{K}$, denoted by $\mathcal{K} \vdash \tau :: d$, if it is derivable by the following kinding rules:

- $\mathcal{K} \vdash \tau :: U$ for any τ.
- $\mathcal{K} \vdash b :: \mathit{KVS}$.
- $\mathcal{K} \vdash t :: \mathit{KVS}$ iff $\mathcal{K}(t) = \mathit{KVS}$.
- $\mathcal{K} \vdash \mathit{Pair}(\tau_1, \tau_2) :: \mathit{KVS}$ iff $\mathcal{K} \vdash \tau_1 :: \mathit{KVS}$ and $\mathcal{K} \vdash \tau_2 :: \mathit{KVS}$.
- $\mathcal{K} \vdash \mathit{Array}(\tau) :: \mathit{KVS}$ iff $\mathcal{K} \vdash \tau :: \mathit{KVS}$.

By definition, $\mathcal{K} \vdash \sigma :: \mathit{KVS}$ holds for any σ.

The type of the polymorphic KVS access operations are given as follows:

$$\begin{aligned} \texttt{create} &: \forall t :: \mathit{KVS}. \texttt{string} \times t \to \texttt{unit} \\ \texttt{update} &: \forall t :: \mathit{KVS}. \texttt{string} \times t \to \texttt{unit} \\ \texttt{find} &: \forall t :: \mathit{KVS}. \texttt{string} \to t \end{aligned}$$

We realize the semantics of the above polymorphic functions in the following strategy. Based on the idea of the type-directed compilation, we introduce a singleton type $M(\tau)$ and its unique value $\mathcal{M}(\tau)$ for any τ of *KVS* kind. $\mathcal{M}(\tau)$ is a meta-object containing essential information that signifies the behaviors of the above three functions. For any σ, $\mathcal{M}(\sigma)$ is a unique value of $M(\sigma)$. By the type-directed compilation, the compiler compiles a type abstraction term

$$(\Lambda t :: \mathit{KVS}. e_1) : \forall t :: \mathit{KVS}. \tau_1$$

to a function with an extra meta-object parameter

$$(\Lambda t :: \mathit{KVS}. \lambda I : M(t). e_1{}') : \forall t :: \mathit{KVS}. M(t) \to \tau_1,$$

and a type instantiation term for a type variable of *KVS* kind

$$(e_2 : \forall t :: \mathit{KVS}. \tau_2)\ \{\sigma\} : \tau_2[\sigma/t]$$

to a function application term with a meta-object corresponding to the type instance

$$(e_2{}' : \forall t :: \mathit{KVS}. M(t) \to \tau_2)\ \{\sigma\}\ \mathcal{M}(\sigma) : \tau_2[\sigma/t].$$

If a bound type variable t_2 with *KVS* kind is instantiated to another type variable t_1 with *KVS* kind, as in the following example,

$$\Lambda t_1 :: \mathit{KVS}. \cdots (e_3 : \forall t_2 :: \mathit{KVS}. \tau_3)\ \{t_1\}) \cdots,$$

the compiler searches for a bound variable I_1 whose value is the meta-object corresponding to the instance of t_1 from the context and generates codes that pass I_1 as follows:

$$\begin{aligned} &\Lambda t_1 :: \mathit{KVS}. \lambda I_1 : M(t_1). \\ &\quad \cdots (e_3{}' : \forall t_2 :: \mathit{KVS}. M(t_2) \to \tau_3)\ \{t_1\}\ I_1) \cdots \end{aligned}$$

See [9] for the details of how the compiler searches for I_1. In the above compilation scheme, the polymorphic version of create, update, and find must be the functions that perform the following steps: (1) Obtain a meta-object $\mathcal{M}(\tau)$, where τ is the instance of the bound type variable t, through the extra parameter I inserted by the type-directed compilation; (2) extract information from the meta-object that corresponds to the behavior of $\texttt{create}_\tau$, $\texttt{update}_\tau$, and $\texttt{find}_\tau$; and (3) evaluate the information.

The following is required to implement the above behavior:

(1) $\mathcal{M}(\tau)$ must include sufficient information for the polymorphic access operations to behave similar to the corresponding monomorphic version.

(2) The compiler must be able to generate $\mathcal{M}(\tau)$ from any τ of *KVS* kind.

(3) $\mathcal{M}(\tau)$ must be inductive on the structure of τ so that beta reduction on terms including $\mathcal{M}(\tau)$ preserves types.

If τ is limited to σ, which contains no type variable, then the above requirements are satisfied by defining the meta-object as a 4-tuple of $\texttt{create}_\sigma$, $\texttt{update}_\sigma$, $\texttt{find}_\sigma$, and $\overline{\sigma}$ as the following:

$$\begin{aligned} M(\sigma) = \{\ & \texttt{create} &&: \texttt{string} \times \sigma \to \texttt{unit}, \\ & \texttt{update} &&: \texttt{string} \times \sigma \to \texttt{unit}, \\ & \texttt{find} &&: \texttt{string} \to \sigma, \\ & \texttt{ty} &&: \texttt{string}\ \} \\ \mathcal{M}(\sigma) = \{\ & \texttt{create} &&= \texttt{create}_\sigma, \\ & \texttt{update} &&= \texttt{update}_\sigma, \\ & \texttt{find} &&= \texttt{find}_\sigma, \\ & \texttt{ty} &&= \overline{\sigma}\ \} \end{aligned}$$

The ty field is needed for inductive construction of the three other fields. From this definition and Fig. 2, the meta-objects of compound types are derived from the meta-objects of their element types. For example, create of $\mathcal{M}(\mathit{Pair}(\sigma_1, \sigma_2))$ is computed from $\mathcal{M}(\sigma_1)$ and $\mathcal{M}(\sigma_2)$ as follows:

$$
\begin{aligned}
&\texttt{\#create}\ \mathcal{M}(\mathit{Pair}(\sigma_1,\sigma_2)) = \\
&\quad \lambda(k,(v_1,v_2)). \\
&\qquad (\texttt{new}(k,\mathit{PairTy}(\texttt{\#ty}\ \mathcal{M}(\sigma_1),\texttt{\#ty}\ \mathcal{M}(\sigma_2))); \\
&\qquad\ \texttt{\#create}\ \mathcal{M}(\sigma_1)\ (k/\overline{1},v_1); \\
&\qquad\ \texttt{\#create}\ \mathcal{M}(\sigma_2)\ (k/\overline{2},v_2))
\end{aligned}
$$

This construction of meta-objects for type σ is naturally extended to type τ by adding the case for type variable t to $\overline{\sigma}$ and the definitions shown in Fig. 2. As stated above, the type-directed compilation algorithm is able to search for a variable I of type $M(t)$ from the context. Therefore, the cases for t are given by using this I as follows:

$$
\begin{aligned}
&\texttt{create}_t = \texttt{\#create}\ I \\
&\texttt{update}_t = \texttt{\#update}\ I \\
&\texttt{find}_t = \texttt{\#find}\ I \\
&\overline{t} = \texttt{\#ty}\ I
\end{aligned}
$$

where $\#l$ selects the l field of the given record. The compilation algorithm $\mathcal{T}$ for `create`, `update` and `find` is given below:

$$
\begin{aligned}
&\mathcal{T}(\texttt{create}) = \Lambda t{::}\mathit{KVS}.\,\lambda I{:}M(t).\,\texttt{\#create}\ I \\
&\mathcal{T}(\texttt{update}) = \Lambda t{::}\mathit{KVS}.\,\lambda I{:}M(t).\,\texttt{\#update}\ I \\
&\mathcal{T}(\texttt{find}) = \Lambda t{::}\mathit{KVS}.\,\lambda I{:}M(t).\,\texttt{\#find}\ I
\end{aligned}
$$

Figure 3 shows an example how `create` and its application are compiled. The underlined part in Fig. 3 is the difference from the previous compilation step.

4.3 Practical KVS Access Operations

The KVS access operations presented above have the following two issues in practice.

The first issue is that the implementation of the polymorphic KVS access operations in ML without any modification is not straightforward. One reason for this is that, in ML, type abstractions and type instantiations are implicit and are inferred automatically. To implement the polymorphic KVS access operations in ML, we need to realize the semantics of `create`, `update`, and `find` only with inferred type abstractions and type instantiations. From this point of view, there is a problem in `find`. To clarify the problem, consider the following example. A user writes a `copy` function that copies the value of `k1` to `k2` as follows:

$$\texttt{copy} = \lambda().\,\texttt{create}\ (\texttt{"k1"}, \texttt{find}\ \texttt{"k2"})$$

This `copy` function appears to be a polymorphic function that copies the values of any type. This is true if the type inference algorithm inserts type abstractions and instantiations as follows:

$$\Lambda t{::}\mathit{KVS}.\,\lambda().\,\texttt{create}\ \{t\}\ (\texttt{"k1"}, \texttt{find}\ \{t\}\ \texttt{"k2"})$$

and infers the type of `copy` as $\forall t{::}\mathit{KVS}.\,\texttt{unit} \rightarrow \texttt{unit}$. However, the ML compiler infers it as $\texttt{unit} \rightarrow \texttt{unit}$ and therefore `copy` is not a polymorphic function. Thus, the compiler cannot compile `copy` to codes that behave as we expected. In this case, the type-directed compilation algorithm chooses a certain type, such as `int`, for the type of `create` and `find`. This behavior of the compiler is rather different from what the user would expect from the source code.

```
val f = fn x => create ("key", x)
val y = f "foo"
          ↓ Type inference
val f : ∀t₁ :: KVS. t₁ → unit =
    Λt₁ :: KVS.
    fn x : t₁ => (create : ∀t₂ :: KVS. string × t₂ → unit)
                 {t₁}
                 ("key", x)
val y : unit = (f : ∀t₁ :: KVS. t₁ → unit) {string} "foo"
          ↓ Type-directed compilation
val f : ∀t₁ :: KVS. M(t₁) → t₁ → unit =
    Λt₁ :: KVS. λI : M(t₁).
    fn x : t₁ =>
       (create : ∀t₂ :: KVS. M(t₂) → string × t₂ → unit)
         {t₁}
         (I : M(t₁))
         ("key", x)
val y : unit = (f : ∀t₁ :: KVS. M(t₁) → t₁ → unit)
                 {string}
                 (𝓜(string) : M(string))
                 "foo"
          ↓ Expanding create and 𝓜(σ)
val f : ∀t₁ :: KVS. M(t₁) → t₁ → unit =
    Λt₁ :: KVS. λI : M(t₁).
    fn x : t₁ => #create I ("key", x)
val y : unit = (f : ∀t₁ :: KVS. M(t₁) → t₁ → unit)
                 {string}
                 ({create = create_string,
                   udpate = update_string,
                   find = find_string,
                   ty = string} : M(string))
                 "foo"
```

Fig. 3 Sample compilation of the polymorphic KVS access operations.

The second issue is that the access operations do not allow us to obtain or update a part of the data stored in KVS. KVS usually contains a huge volume of data that is much larger than heap memory. In our high-level KVS, such a huge volume of data should be a huge array of huge tuples. However, the access operations presented thus far only allow us to copy the whole of the data between KVS and heap memory at once. Hence, we cannot deal with such huge data in ML using only those access operations.

To rectify the above two issues, we replace the polymorphic `find` operation with the following syntax:

$$\texttt{find}\ k\ \texttt{as}\ \sigma$$

that forces us to specify the type σ of key k and returns a handle that allows us partial access to the compound data. The typing rule of this syntax is given as a derivation rule that derives a type judgment $\Gamma \vdash e : \tau$ indicating that expression e has type τ under kind assignment $\mathcal{K}$ and type assignment Γ as follows:

$$\frac{\mathcal{K},\Gamma \vdash e : \texttt{string}}{\mathcal{K},\Gamma \vdash \texttt{find}\ e\ \texttt{as}\ \sigma : \mathit{Obj}(\sigma)}$$

where $\mathit{Obj}(\sigma)$ is the type of the handle.

$$
\begin{array}{l}
Obj(b) = b \\[4pt]
Obj(Pair(\sigma_1, \sigma_2)) = \\
\quad \begin{array}{llll}
\{ & \texttt{first} & : & \texttt{unit} \to Obj(\sigma_1), \\
 & \texttt{second} & : & \texttt{unit} \to Obj(\sigma_2), \\
 & \texttt{updateFirst} & : & \sigma_1 \to \texttt{unit}, \\
 & \texttt{updateSecond} & : & \sigma_2 \to \texttt{unit}\ \}
\end{array} \\[4pt]
Obj(Array(\sigma)) = \\
\quad \begin{array}{llll}
\{ & \texttt{getLength} & : & \texttt{unit} \to \texttt{int}, \\
 & \texttt{getElem} & : & \texttt{int} \to Obj(\sigma), \\
 & \texttt{putElem} & : & \texttt{int} \times \sigma \to \texttt{unit}\ \}
\end{array}
\end{array}
$$

Fig. 4 Type of the partial access handle.

$$
\begin{array}{l}
\mathcal{C}(\texttt{find}\ e\ \texttt{as}\ b) = \texttt{find}_b\ \mathcal{C}(e) \\[4pt]
\mathcal{C}(\texttt{find}\ e\ \texttt{as}\ Pair(\sigma_1, \sigma_2)) = \\
\quad \text{let } k \text{ be } \mathcal{C}(e) \text{ in} \\
\quad (\texttt{mustEqual}\ (\texttt{get}(k), \overline{Pair(\sigma_1, \sigma_2)}); \\
\quad\ \{\ \texttt{first} = \lambda().\, \mathcal{C}(\texttt{find}\ k/\overline{1}\ \texttt{as}\ \sigma_1), \\
\quad\quad \texttt{second} = \lambda().\, \mathcal{C}(\texttt{find}\ k/\overline{2}\ \texttt{as}\ \sigma_2), \\
\quad\quad \texttt{updateFirst} = \lambda v.\, \texttt{update}_{\sigma_1}\ (k/\overline{1}, v), \\
\quad\quad \texttt{updateSecond} = \lambda v.\, \texttt{update}_{\sigma_2}\ (k/\overline{2}, v)\ \})
\end{array}
$$

Fig. 5 Compilation algorithm for `find`.

The handle for partial access should be designed as a high-level data structure considering the user's convenience. There is flexibility in the design; in this work, we define the handle as a record of functions in the object-oriented style. As with the monomorphic access operations presented in Section 4.1, the implementations of *Obj*(σ) are specific for each σ. **Figure 4** defines the definition of *Obj*(σ). In this definition, we include only primitive operations; other access functions may be added to the handle for the user's convenience. This handle object allows us to read and write a part of the compound data without copying the entire data to the memory. The following `swap` function is an example of the usage of this handle that swaps the `i`-th element of an integer array of the given key `arrayKey` with `new`:

```
fun swap i new =
    let
        val m = find "arrayKey" as Array(int)
        val old = #getElem m i
    in
        #putElem m (i, new); old
    end
```

Variable `m` in the above example has the following record type:

$$
\begin{array}{lllll}
\texttt{m} : \{ & \texttt{getLength} & : & \texttt{unit} \to \texttt{int}, \\
 & \texttt{getElem} & : & \texttt{int} \to \texttt{int}, \\
 & \texttt{putElem} & : & \texttt{int} \times \texttt{int} \to \texttt{unit}\ \}
\end{array}
$$

Each function in record `m` is an operation on the array stored in KVS with key `arrayKey`.

The compiler compiles the `find` expression to codes that perform the runtime type check and constructs a record of type *Obj*(σ). As with $\texttt{find}_\sigma$ and $\texttt{update}_\sigma$ in Fig. 2, the implementations of the functions in *Obj*(σ) are derived inductively on the structure of σ. We refer to this compilation algorithm as $\mathcal{C}$. **Figure 5** shows a part of the definition of $\mathcal{C}$. The cases for *Array*(σ), which we omit in Fig. 5, can be defined similarly according to Fig. 2.

5. Prototype Implementation

To demonstrate the feasibility of the proposed scheme, we implemented a prototype in SML#. As we shall discuss in Section 6, further issues still remain for better practical performance, such as mutual exclusion. To clarify those issues through the prototype implementation, we did not implement the type-directed compilation algorithm in the SML# compiler, which requires significant implementation effort, in order to facilitate flexibility in system design modification. Instead, we implemented a library that provides functions to construct $\mathcal{M}(\sigma)$ by hand. We also designed a common signature for the string-based KVS access operations and implemented two KVS modules of that signature: a toy implementation for test use that simulates KVS with a binary search tree, and a binding for the Web APIs of Riak CS [11] servers. Using the latter module, we successfully connected our implementation to an actual KVS server.

Figure 6 shows the actual output of an interactive session carried out using this prototype. The first part of the output shows the signatures of the functions we implemented. The polymorphic `create` and `find` functions are implemented in SML# user-level codes; hence, they are record-polymorphic functions. `intMeta` and `arrayMeta` are implementations of the base and array case of algorithm $\mathcal{C}$. In this prototype, instead of `find` e `as` σ syntax, the user calls the `find` function with a meta-object corresponding to type σ. The remaining parts demonstrate the usage of this prototype library. As can be seen in the output, the user successfully created an array in KVS and partially updated the array through the handle. The runtime type check also works as we expected.

6. Further Issues to be Resolved for Better Practical Performance

Through the above prototype implementation, we discovered several issues that are important for the realization of a practical high-level KVS system. These issues include those that require further development beyond the type system and compilation technique. We believe that those issues can be resolved and practical high-level KVS can be achieved by applying technologies widely used for database transaction management and data access control to our method. In this section, we discuss each of the issues identified in each respective subsection.

6.1 Access Control and Transaction Management

To maintain the consistency of the huge volume of data stored in KVS, a mechanism and algorithm for systematic access control is essential. In string-based KVS, management of data consistency is carried out only for each string key-value pair in each network node; maintaining consistency between keys is the programmer's responsibility. In high-level KVS, because the user can store compound data as a value, the system must guarantee the consistency of the compound data represented in the set of keys. A major technical issue associated with this requirement is establishment of a mechanism for transaction management that automatically guarantees the consistency of arrays and tuples when creating and updating them without negatively af-

```
# create;
val create =
    fn : ['a#{create: string * 'b -> unit}, 'b.
          'a -> string * 'b -> unit]
# find;
val find =
    fn : ['a#{find: string -> 'b}, 'b.
          'a -> string -> 'b]
# intMeta;
val intMeta =
    {create = fn, find = fn, type = "int"}
    : {create: string * int -> unit,
       find: string -> int,
       type: string}
# arrayMeta;
val arrayMeta =
    fn
    : ['a, 'b.
       {create: string * 'a -> unit,
        find: string -> 'b,
        type: string}
       -> {create: string * 'a array -> unit,
           find: string -> {getElem: int -> 'b,
                            getLength: unit -> int,
                            putElem: int * 'a -> unit},
           type: string}]
```

Storing an array in KVS and obtaining the handle to the array:

```
# create (arrayMeta intMeta)
         ("arrayKey", Array.array (78, 0));
val it = () : unit
# val intArrayObj =
      find (arrayMeta intMeta) "arrayKey";
val intArrayObj =
    {getElem = fn, getLength = fn, putElem = fn}
    : {getElem: int -> int,
       getLength: unit -> int,
       putElem: int * int -> unit}
```

Accessing to the array through the handle:

```
# #getLength intArrayObj ();
val it = 78 : int
# #getElem intArrayObj 0;
val it = 0 : int
# #putElem intArrayObj (0,888);
val it = () : unit
# #getElem intArrayObj 0;
val it = 888 : int
```

Referring to a non-existing key:

```
# find (arrayMeta intMeta) "bogusKey";
uncaught exception: TypedKVS.KeyNotFound
```

Accessing a key of a different type:

```
# find intPtrMeta "arrayKey";
uncaught exception: TypedKVS.TypeMismatch
```

Fig. 6 Interactive SML# session using the prototype implementation.

fecting the scalability and performance of KVS.

6.2 Shareability of Data Structures and Garbage Collection

In high-level KVS, the update operation may produce some garbage keys — that is, keys that were a part of a compound data structure but are currently not reachable from any other key structures. For example, consider a case in which a user overwrites an array of two elements stored in k with another one-element array. In this case, k, $k/\texttt{len}$, and $k/\overline{0}$ are overwritten by this update operation, but $k/\overline{1}$ may remain as garbage. This garbage can be eliminated by removing keys consisting of the overwritten data before writing the new data. This operation is safe because no key is shared among two distinct data structures in our high-level KVS.

The model of the high-level KVS can be naturally extended to facilitate sharing of data components among multiple data structures. This extension can be realized by including a key string to a data structure as a pointer to a data component. This extension would make the high-level KVS more efficient and enable representation of more complex data structures such as cyclic graph structures. However, updating a pointer may produce unreachable components that cannot be eliminated by the above simple strategy. A garbage collection technique for KVS should therefore be investigated with the objective of overcoming this problem.

6.3 Dealing with Algebraic Data Types

While we only considered arrays and tuples for KVS, there is a variety of types in functional languages, such as records, lists, and trees. For high-level KVS to be more practical, these data types should be supported. Record types can be supported similar to $Pair(\sigma_1, \sigma_2)$ type. For lists and trees, we need to extend the data model $\mathcal{R}$ and the access handle Obj with support for algebraic data types. Here, we only mention possible support for lists in high-level KVS and will leave the detailed analysis and complete design of the algebraic data type support for future work. We can add a list type $List(\sigma)$ to the KVS types as follows. The data model of $List(\sigma)$ is given below:

$$\begin{aligned}
\mathcal{R}(List(\sigma)) &= \Big\{ (k, \left\{ \begin{array}{lcl} k & \mapsto & \overline{List(\sigma)} \\ k/\texttt{tag} & \mapsto & c \end{array} \right\} \cup S) \\
&\qquad \Big| \ (c, S) \in \mathcal{R}_{\mathrm{cons}}(k, \sigma) \cup \{(\texttt{"nil"}, \{\})\} \Big\} \\
\mathcal{R}_{\mathrm{cons}}(k, \sigma) &= \Big\{ (\texttt{"cons"}, S) \\
&\qquad \Big| \ (k/\texttt{val}, S) \in \mathcal{R}(Pair(\sigma, List(\sigma))) \Big\}
\end{aligned}$$

The implementation of `create`, `update`, and `find` can be obtained by extending their definitions based on this model. The partial access handle $Obj(List(\sigma))$ can be implemented in the following strategy. In ML, a value of an algebraic data type is eliminated by a `case` branch. High-level KVS should provide users with a similar case branch feature for lists stored in KVS. The most primitive operations for the case branch is to match the given data with a specific data constructor. Hence, $Obj(List(\sigma))$ is given as follows:

$$\begin{aligned}
& Obj(List(\sigma)) = \\
& \quad \{\, \texttt{getCons} : \texttt{unit} \to Obj(Pair(\sigma, List(\sigma)))\ \texttt{option}, \\
& \quad\ \ \texttt{getNil} : \texttt{unit} \to \texttt{unit option} \,\}
\end{aligned}$$

where `getCons` and `getNil` return `SOME` if the value of the key k`/tag` is `cons` and `nil`, respectively; otherwise, they return `NONE`.

While the above construction is a possible candidate for list support, it may limit the usability of KVS to implement all possible data types in ML in this way owing to the following reasons. Firstly, the above encoding would be inefficient in dealing with a linear linked list because the length of the key increases with the length of the list. In the above list support construction, because the length of the keys is proportional to the number of cons cells in the list, it requires $O(n^2)$ space to store a list of n elements and $O(n^2)$ steps to enumerate its elements. Secondly, certain data structures can be implemented directly by using features that KVS naturally provides. A typical example of such a data structure is a map structure, which is usually implemented using a binary search tree in ML. Because KVS is in essence a map structure, it is straightforward and efficient to store the map structure directly in KVS rather than to encode the binary search tree structure in some key encoding. For more practical high-level KVS, we have to consider introducing efficient data models specific to each data type as well as a generic model for glgebraic data types.

6.4 More Flexible Type Specification in `find`

The `find` k `as` σ expression presented in Section 4.3 requires the user to specify a concrete type σ. On the other hand, as stated in Section 2, KVS is in essence an untyped storage; therefore, there may be a multiplicity of possible choices for the type of k. If the user could specify the type choices in the `find` expression, the usability of high-level KVS would increase. An approach to realizing this is to replace the runtime type check mechanism of `find` with the `typecase` presented in the dynamic type system [1]. Using `typecase`, a program that reads either an integer or a Boolean can be written as follows:

```
typecase find k of
   (x : int) => x
   (x : bool) => if x then 1 else 0
  else raise Fail "unexpected type"
```

Another possible extension of the `find` syntax is to allow its type specification to include a type variable, i.e., to use τ as its type specification instead of σ. A serious technical issue in the realization of this extension is the definition of the equivalence relationship with respect to $Obj(t)$ and the semantics of a program including $Obj(t)$ in a style that can be seamlessly integrated with ML. This issue may be solved by generalizing the idea of polymorphic record compilation [9] or applying the theory of intensional polymorphism [5], which is closely related to the type-directed compilation; however, these approaches may significantly change the type-directed compilation scheme.

7. Conclusion

We proposed a scheme for type-safe access to KVS in an ML-style polymorphic functional language. The proposed scheme comprises a type system for KVS and a language extension that enables the storing of compound data structures in KVS without any string manipulation. The access operations are provided as polymorphic functions and are realized based on the idea of the type-directed compilation scheme of the polymorphic record calculus. In addition, we implemented a prototype of the proposed scheme and demonstrated its feasibility. Through the prototype, we briefly explored further developments that would facilitate better practical performance of the proposed scheme.

The efficacy and practicability of the proposed scheme should be evaluated following further development of the high-level KVS. We are currently developing an extension to the SML# compiler based on the scheme proposed in this paper. One of the major problems with this development is the question of how to provide a common infrastructure that allows us to connect various KVS servers with high-level access control, as discussed in Section 6. Performance evaluation, including the scale-out performance evaluation, will be conducted following the completion of this development. We believe that the proposed scheme will be beneficial to big data analysis when the issues discussed in this paper are resolved and a practical functional language equipped with the proposed scheme is realized.

Acknowledgments The authors thank Masanori Endo and Yuto Mukade for their cooperation in implementing the prototype reported in Section 5. The authors also thank some members of the project "Research and Development on Highly Functional and Highly Available Information Storage Technology" sponsored by the Ministry of Education, Culture, Sports, Science and Technology in Japan for discussions. This work has been partially supported by JSPS KAKENHI Grant Number 25280019.

References

[1] Abadi, M., Cardelli, L., Pierce, B.C. and Plotkin, G.D.: Dynamic Typing in a Statically Typed Language, *ACM Trans. Program. Lang. Syst.*, Vol.13, No.2, pp.237–268 (1991).

[2] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E.: BigTable: A Distributed Storage System for Structured Data, *ACM Trans. Comput. Syst.*, Vol.26, No.2, pp.4:1–4:26 (online), DOI: 10.1145/1365815.1365816 (2008).

[3] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon's highly available key-value store, *Proc. 21st ACM SIGOPS symposium on Operating systems principles*, *SOSP '07*, New York, NY, USA, pp.205–220, ACM (online), DOI: 10.1145/1294261.1294281 (2007).

[4] Elsman, M.: Type-specialized serialization with sharing, *TFP'05: Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming*, pp.47–62 (2005).

[5] Harper, R. and Morrisett, G.: Compiling Polymorphism Using Intensional Type Analysis, *Proc. 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, New York, NY, USA, pp.130–141, ACM (online), DOI: 10.1145/199448.199475 (1995).

[6] Henry, G., Mauny, M., Chailloux, E. and Manoury, P.: Typing unmarshalling without marshalling types, *Proc. 17th ACM SIGPLAN international conference on Functional programming*, *ICFP '12*, New York, NY,USA, pp.287–298, ACM (online), DOI: 10.1145/2364527.2364569 (2012).

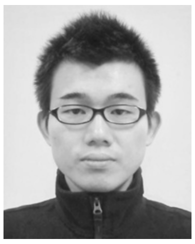
[7] Kennedy, A.J.: Pickler combinators, *J. Funct. Program.*, Vol.14, No.6, pp.727–739 (online), DOI: 10.1017/S0956796804005209 (2004).

[8] Leroy, X. and Mauny, M.: Dynamics in ML, *J. Funct. Program.*, Vol.3, No.4, pp.431–463 (1993).

[9] Ohori, A.: A polymorphic record calculus and its compilation, *ACM Trans. Programming Languages and Systems*, Vol.17, No.6, pp.844–895 (1995).

[10] Ohori, A. and Ueno, K.: Making Standard ML a practical database programming language, *Proc. 16th ACM SIGPLAN international conference on Functional programming*, *ICFP '11*, New York, NY, USA,

pp.307–319, ACM (online), DOI: 10.1145/2034773.2034815 (2011).
[11] Riak CS, available from ⟨http://basho.com/riak-cloud-storage/⟩.



Katsuhiro Ueno was born in 1981. He received his Doctor of Philosophy (Information Sciences) from Tohoku University in 2009. He is currently an assistant professor at Research Institute of Electrical Communication, Tohoku University. He is interested in functional programming languages.

Atsushi Ohori was born in 1957. He received his B.A. in Philosophy from the University of Tokyo in 1981, and his Ph.D. in Computer and Information Science from University of Pennsylvania in 1989. He is a professor of Research Institute of Electrical Communication, Tohoku University. He worked at Oki Electric Industry, Co. Ltd. (1981–1993), Research Institute for Mathematical Sciences, Kyoto University (1993–2000), and School of Information Science, Japan Advanced Institute of Science and Technology (2000–2005) before he moved to Tohoku University in 2005. He is interested in programming languages and database systems.