Type Safe Redis Queries: A Case Study of Type-Level Programming in Haskell

Ting-Yan Lai*
Institute of Information Science
Academia Sinica
Nangang, Taipei City, Taiwan
banacorn@gmail.com

Tyng-Ruey Chuang
Institute of Information Science
Academia Sinica
Nangang, Taipei City, Taiwan
trc@iis.sinica.edu.tw

Shin-Cheng Mu
Institute of Information Science
Academia Sinica
Nangang, Taipei City, Taiwan
scm@iis.sinica.edu.tw

Abstract

REDIS is an in-memory data structure store, often used as a database, with a Haskell interface HEDIS. REDIS is dynamically typed — a key can be discarded and re-associated to a value of a different type, and a command, when fetching a value of a type it does not expect, signals a runtime error. We develop a domain-specific language that, by exploiting Haskell type-level programming techniques including indexed monad, type-level literals and closed type families, keeps track of types of values in the database and statically guarantees that type errors cannot happen for a class of REDIS programs.

CCS Concepts • **Software and its engineering → Functional languages**; *Domain specific languages*; • **Information systems →** *Query languages*;

Keywords Haskell, Redis, Type-Level Programming, Type Safety, Key-Value Store

Note from the Authors:

This work is to be presented at the *2nd Workshop on Type-Driven Development (TyDe 2017), September 3, 2017, Oxford, UK.* This paper is not included in the workshop proceedings published by the ACM. The authors choose to place it in the public domain.
https://arxiv.org/abs/1708.xxxxx

1 Introduction

REDIS[1] is an open source, in-memory data structure store that can be used as database, cache, and message broker. A REDIS data store can be thought of as a set of key-value pairs. The value can be a string, a list of strings, a set of strings, or a hash table of strings, etc. However, string is the only primitive datatype. Numbers, for example, are serialized to strings before being saved in the data store, and parsed back to numbers to be manipulated with. While the concept is simple, REDIS is used as an essential component in a number of popular, matured services, including Twitter, GitHub, Weibo, StackOverflow, and Flickr.

For an example, consider the following sequence of commands, entered through the interactive interface of REDIS. The keys `some-set` and `another-set` are both assigned a set of strings. The two calls to command `SADD` respectively add three and two strings to the two sets, before `SINTER` takes their intersection:

```
redis> SADD some-set a b c
(integer) 3
redis> SADD another-set a b
(integer) 2
redis> SINTER some-set another-set
1) "a"
2) "b"
```

Notice that the keys `some-set` and `another-set`, if not existing before the call to `SADD`, are created on site. The calls to `SADD` return the sizes of the resulting sets.

Many third party libraries provide interfaces for general purpose programming languages to access REDIS through its TCP protocol. For Haskell, the most popular library is HEDIS.[2] A (normal) REDIS computation returning a value of type a is represented in Haskell by Redis (Either Reply a), where the type Redis is a monad, while Either Reply a indicates that the computation either returns a value of type a, or fails with an error message represented by type Reply. The following program implements the previous example:

program :: Redis (Either Reply [ByteString])
program = **do** *sadd* "some-set" ["a", "b"]
    sadd "another-set" ["a", "b", "c"]
    sinter ["some-set", "another-set"] .

The function

sadd :: ByteString → [ByteString] →
    Redis (Either Reply Integer)

*Ting-Yan Lai was a summer intern from the Inst. of Computer Science and Engineering, National Chiao Tung Univ., Taiwan, when part of this work was developed.

[1] https://redis.io

[2] https://hackage.haskell.org/package/hedis

TyDe'17, September 3, 2017, Oxford, UK

https://arxiv.org/abs/1708.xxxxx

takes a key and a list of values as arguments, and returns a Redis computation yielding Integer. Keys and values, being nothing but binary strings in Redis, are represented using Haskell ByteString.

The Problems Most commands work only with data of certain types. In the following example, the key `some-string` is assigned a string `foo` — the command `SET` always assigns a string to a key. The subsequent call to `SADD`, which adds a value to a set, thus raises a runtime error.

```
redis> SET some-string foo
OK
redis> SADD some-string bar
(error) WRONGTYPE Operation against a key
holding the wrong kind of value
```

For another source of type error, the command `INCR key` increments the value associated with `key` by one. With strings being the only primitive type, Redis parses the stored string to an integer and, after incrementation, stores a string back. If the string cannot be parsed as an integer, a runtime error is raised.

We point out a peculiar pattern of value creation and updating in Redis: the same command is used both to create a key-value pair and to update them. Similar to `SADD`, the command `LPUSH` appends a value (a string) to a list, or creates one if the key does not exist:

```
redis> LPUSH some-list bar
(integer) 1
```

Another command `LLEN` returns the length of the list, and signals an error if the key is not associated with a list:

```
redis> LLEN some-list
(integer) 1
redis> SET some-string foo
OK
redis> LLEN some-string
(error) WRONGTYPE Operation against a key
holding the wrong kind of value
```

Curiously, however, when applied to a key not yet created, Redis designers chose to let `LLEN` return `0`:

```
redis> LLEN nonexistent
(integer) 0
```

Being a simple wrapper on top of the TCP protocol of Redis, Hedis inherits all the behaviors. Executing the following program yields the same error, but wrapped in a Haskell constructor: Left (Error `"WRONGTYPE Operation against a key holding the wrong kind of value"`).

```
program :: Redis (Either Reply Integer)
program = do set "some-string" "foo"
             sadd "some-string" ["a"] .
```

Such a programming model is certainly very error-prone. Working within Haskell, a host language with a strong typing system, one naturally wishes to build a domain-specific embedded language (DSEL) that exploits the rich type system of Haskell to not only ensure the absence of Redis type errors (at least in the simplified situation where there is one client accessing the store), but also provides better documentations. We wish to be sure that a program calling `INCR`, for example, can be type checked only if we can statically guarantee that the value to be accessed is indeed an integer. We wish to see from the type of operators such as `LLEN` when it can be called, and allow it to be used only in contexts that are safe. We may even want to explicitly declare a fresh key and its type, to avoid reusing an existing key by accident, and to prevent it from unexpectedly being used as some other type.

This paper discusses the techniques we used and the experiences we learned from building such a language, Edis. We constructed an *indexed monad*, on top of the monad Redis, which is indexed by a dictionary that maintains the set of currently defined keys and their types. To represent the dictionary, we need to encode variable bindings with *type-level* lists and strings. And to manipulate the dictionary, we applied various type-level programming techniques. To summarize our contributions:

- We present Edis, a statically typed domain-specific language embedded in Haskell (with extension provided by the Glasgow Haskell Compiler) and built on Hedis. Serializable Haskell datatypes are automatically converted before being written to Redis data store. Available keys and their types are kept track of in type-level dictionaries. The types of embedded commands state clearly their preconditions and postconditions on the available keys and types, and a program is allowed to be constructed only if it is guaranteed not to fail with a type error, assuming that it is the only client accessing the store.
- We demonstrate the use of various type-level programming techniques, including data kinds, singleton types and proxies, closed type families, etc., to define type-level lists and operations that observe and manipulate the lists.
- This is (yet another) example of encoding effects and constraints of programs in types, using indexed monad [1], closed type-families [5] and constraint kinds [16].

In Section 2 we introduce indexed monads, to reason about pre and postconditions of stateful programs, and in Section 3 we review the basics of type-level programming in Haskell that allows us to build type-level dictionaries to keep track of keys and types. Embeddings of Redis commands are presented in Section 4. In Section 5 we discuss various issues regarding design choices, limitations of this approach, as well as possible future works, before we review related work in Section 6.

2 Indexed Monads

Stateful computations are often reasoned using Hoare logic. A *Hoare triple* $\{P\}S\{Q\}$ denotes the following proposition: if the statement S is executed in a state satisfying predicate P, when it terminates, the state must satisfy predicate Q. Predicates P and Q are respectively called the *precondition* and the *postcondition* of the Hoare triple.

In Haskell, stateful computations are represented by monads. To reason about them within the type system, we wish to label a state monad with its pre and postcondition. An *indexed monad* [1] (also called a *parameterised monad* or *monadish*) is a monad that, in addition to the type of its result, takes two more type parameters representing an initial state and a final state, to be interpreted like a Hoare triple:

$$\begin{array}{l}\textbf{class}\ \mathsf{IMonad}\ m\ \textbf{where}\\ \quad \mathit{unit} :: a \to m\ p\ p\ a\\ \quad \mathit{bind} :: m\ p\ q\ a \to (a \to m\ q\ r\ b) \to m\ p\ r\ b\ .\end{array}$$

The intention is that a computation of type $m\ p\ q\ a$ is a stateful computation such that, if it starts execution in a state satisfying p and terminates, it yields a value of type a, and the new state satisfies q. The operator *unit* lifts a pure computation to a stateful computation that does not alter the state. In x `*bind*` f, a computation $x :: m\ p\ q\ a$ is followed by $f :: a \to m\ q\ r\ b$ — the postcondition of x matches the precondition of the computation returned by f. The result is a computation $m\ p\ r\ b$.

We define a new indexed monad Edis. At term level, the *unit* and *bind* methods are not interesting: they merely make calls to *return* and ($\gg\!=$) of Redis, and extract and re-apply the constructor Edis when necessary. With Edis being a **newtype**, they can be optimized away in runtime. The purpose is to add the pre/postconditions at type level:

$$\begin{array}{l}\textbf{newtype}\ \mathsf{Edis}\ p\ q\ a = \mathsf{Edis}\ \{\mathit{unEdis} :: \mathsf{Redis}\ a\}\ ,\\ \textbf{instance}\ \mathsf{IMonad}\ \mathsf{Edis}\ \textbf{where}\\ \quad \mathit{unit} = \mathsf{Edis} \cdot \mathit{return}\\ \quad \mathit{bind}\ m\ f = \mathsf{Edis}\ (\mathit{unEdis}\ m \gg\!= \mathit{unEdis} \cdot f)\ .\end{array}$$

The properties of the state we care about are the set of currently allocated keys and types of their values. We will present, in Section 3, techniques that allow us to specify properties such as "the keys in the data store are "A", "B", and "C", respectively assigned values of type Int, Char, and Bool." For now, however, let us look at the simplest Redis command.

Redis commands can be executed in two contexts: the normal context, or a *transaction*. In Hedis, a command yielding value of type a in the normal case is represented by Redis (Either Reply a), as mentioned in Section 1; in a transaction, the command is represented by two other datatypes RedisTx (Queued a). In this paper we focus on the former case. For brevity we abbreviate Either Reply a to $\mathsf{R} \uplus a$.

The command PING in Redis does nothing but replying a message PONG if the connection is alive. In Hedis, *ping* has type Redis ($\mathsf{R} \uplus$ Status). The Edis version of *ping* simply applies an additional constructor (functions from Hedis are qualified with Hedis to prevent name clashing):

$$\begin{array}{l}\mathit{ping} :: \mathsf{Edis}\ \mathit{xs}\ \mathit{xs}\ (\mathsf{R} \uplus \mathsf{Status})\\ \mathit{ping} = \mathsf{Edis}\ \mathsf{Hedis}.\mathit{ping}\ .\end{array}$$

Since *ping* does not alter the data store, the postcondition and precondition are the same. Commands that are more interesting will be introduced after we present our type-level encoding of constraints on states.

3 Type-Level Dictionaries

One of the challenges of statically ensuring type correctness of stateful languages is that the type of the value of a key can be altered by updating. In Redis, one may delete an existing key and create it again by assigning to it a value of a different type. To ensure type correctness, we keep track of the types of all existing keys in a *dictionary*. A dictionary is a finite map, which can be represented by an associate list, or a list of pairs of keys and some encoding of types. For example, we may use the dictionary [("A", Int), ("B", Char), ("C", Bool)] to represent a predicate, or a constraint, stating that "the keys in the data store are "A", "B", and "C", respectively assigned values of type Int, Char, and Bool." (This representation will be refined in the next section.)

The dictionary above mixes values (strings such as "A", "B") and types. Furthermore, as mentioned in Section 2, the dictionaries will be parameters to the indexed monad Edis. In a dependently typed programming language (without the so-called "phase distinction" — separation between types and terms), this would pose no problem. In Haskell however, the dictionaries, to index a monad, has to be a type as well.

In this section we describe how to construct a type-level dictionary, to be used with the indexed monad in Section 2.

3.1 Datatype Promotion

Haskell maintains the distinction between values, types, and kinds: values are categorized by types, and types are categorized by kinds. The kinds are relatively simple: $*$ is the kind of all *lifted* types, while type constructors have kinds such as $* \to *$, or $* \to * \to *$, etc.[3] Consider the datatype definitions below:

$$\textbf{data}\ \mathsf{Nat} = \mathsf{Zero} \mid \mathsf{Suc}\ \mathsf{Nat}\ , \qquad \textbf{data}\ [a] = [\,] \mid a : [a]\ .$$

The left-hand side is usually seen as having defined a type Nat :: $*$, and two value constructors Zero :: Nat and Suc :: Nat $\to$ Nat. The right-hand side is how Haskell lists are understood. The *kind* of $[\cdot]$ is $* \to *$, since it takes a lifted type a to a lifted type $[a]$. The two value constructors respectively have types $[\,] :: [a]$ and $(:) :: a \to [a] \to [a]$, for all types a.

The GHC extension *data kinds* [20], however, automatically promotes certain "suitable" types to kinds.[4] With the extension, the **data** definitions above has an alternative reading: Nat is a new kind, Zero :: Nat is a type having kind Nat, and Suc :: Nat → Nat is a type constructor, taking a type in kind Nat to another type in Nat. When one sees a constructor in an expression, whether it is promoted can often be inferred from the context. When one needs to be more specific, prefixing a constructor with a single quote, such as in 'Zero and 'Suc, denotes that it is promoted.

The situation of lists is similar: for all kinds k, $[k]$ is also a kind. For all kinds k, '[ ] is a type of kind $[k]$. Given a type a of kind k and a type as of kind $[k]$, a ': as is again a type of kind $[k]$. Formally, $(':) :: k \to [k] \to [k]$. For example, Int ': (Char ': (Bool ': '[ ])) is a type having kind $[*]$ — it is a list of (lifted) types. The optional quote denotes that the constructors are promoted. The same list can be denoted by a syntax sugar '[Int, Char, Bool].

Tuples are also promoted. Thus we may put two types in a pair to form another type, such as in '(Int, Char), a type having kind $(*, *)$.

Strings in Haskell are nothing but lists of Chars. Regarding promotion, however, a string can be promoted to a type having kind Symbol. In the expression:

`"this is a type-level string literal"` :: Symbol ,

the string on the left-hand side of (::) is a type whose kind is Symbol.

With all of these ingredients, we are ready to build our dictionaries, or type-level associate lists:

type DictEmpty = '[ ] ,
type Dict0 = '['("key", Bool)] ,
type Dict1 = '['("A", Int), '("B", "A")] .

All the entities defined above are types, where Dict0 has kind $[(\text{Symbol}, *)]$. In Dict1, while Int has kind $*$ and "A" has kind Symbol, the former kind subsumes the later. Thus Dict1 also has kind $[(\text{Symbol}, *)]$.

3.2 Type-Level Functions

Now that we can represent dictionaries as types, the next step is to define operations on them. A function that inserts an entry to a dictionary, for example, is a function from a type to a type. While it was shown that it is possible to simulate type-level functions using Haskell type classes [13], in recent versions of GHC, *indexed type families*, or *type families* for short, are considered a cleaner solution.

[3]In Haskell, the opposite of *lifted* types are *unboxed* types, which are not represented by a pointer to a heap object, and cannot be stored in a polymorphic data type.
[4]It is only informally described in the GHC manual what types are "suitable".

For example, compare disjunction (∨) and its type-level counterpart Or:

$(\vee) :: \text{Bool} \to \text{Bool} \to \text{Bool}$
True ∨ b = True
a ∨ b = b ,

type *family* Or (a :: Bool) (b :: Bool) :: Bool
  where 'True ‘Or‘ b = 'True
    a ‘Or‘ b = b .

The first is a typical definition of (∨) by pattern matching. In the second definition, Bool is not a type, but a type lifted to a kind, while True and False are types of kind Bool. The declaration says that Or is a family of types, indexed by two parameters a and b of kind Bool. The type with index 'True and b is 'True, and all other indices lead to b. For our purpose, we can read Or as a function from types to types — observe how it resembles the term-level (∨). We present two more type-level functions about Bool — negation, and conditional, that we will use later:

type *family* Not a **where**
  Not 'False = 'True
  Not 'True = 'False ,

type *family* If (c :: Bool) (t :: a) (f :: a) :: a **where**
  If 'True *tru* *fls* = *tru*
  If 'False *tru* *fls* = *fls* .

As a remark, type families in GHC come in many flavors. One can define families of **data** types, as well as families of **type** synonyms. They can appear inside type classes [3, 4] or at toplevel. Top-level type families can be open [18] or closed [5]. The flavor we choose is top-level, closed type synonym families, since it allows overlapping instances, and since we need none of the extensibility provided by open type families. Notice that the instance 'True ‘Or‘ b could be subsumed under the more general instance, a ‘Or‘ b. In a closed type family we may resolve the overlapping in order, just like how cases overlapping is resolved in term-level functions.

Now we can define operations on type-level dictionaries. Let us begin with:

type *family* Get (xs :: $[(\text{Symbol}, *)]$) (k :: Symbol) :: $*$**where**
  Get ('(k, x) ': xs) k = x
  Get ('(t, x) ': xs) k = Get xs k .

Get xs k returns the entry associated with key k in the dictionary xs. Notice, in the first case, how type-level equality can be expressed by unifying type variables with the same name. Note also that Get is a partial function on types: while Get '['("A", Int)] "A" evaluates to Int, when Get '['("A", Int)] "B" appears in a type expression, there are no applicable rules to reduce it. The expression thus stays un-reduced.

```
-- inserts or updates an entry
type family Set (xs :: [(Symbol, *)]) (k :: Symbol) (x :: *)
                :: [(Symbol, *)] where
  Set '[ ]            k x = '['(k, x)]
  Set ('(k, y) ': xs) k x = '(k, x) ': xs
  Set ('(t, y) ': xs) k x = '(t, y) ': Set xs k x
-- removes an entry
type family Del (xs :: [(Symbol, *)]) (k :: Symbol)
                :: [(Symbol, *)] where
  Del '[ ]            k = '[ ]
  Del ('(k, y) ': xs) k = xs
  Del ('(t, y) ': xs) k = '(t, y) ': Del xs k
-- membership
type family Member (xs :: [(Symbol, *)]) (k :: Symbol)
                   :: Bool where
  Member '[ ]            k = 'False
  Member ('(k, x) ': xs) k = 'True
  Member ('(t, x) ': xs) k = Member xs k
```

Figure 1. Some operations on type-level dictionaries.

Some other dictionary-related functions are defined in Figure 1. The function Set either updates an existing entry or inserts a new entry, Del removes an entry matching a given key, while Member checks whether a given key exists in the dictionary.

4 Embedding Hedis Commands

Having the indexed monads and type-level dictionaries, in this section we present our embedding of Hedis commands into Edis, while introducing necessary concepts when they are used.

4.1 Proxies and Singleton Types

The Hedis function $del :: [\text{ByteString}] \rightarrow \text{Redis}\ (\text{R} \uplus \text{Integer})$ takes a list of keys (encoded to ByteString) and removes the entries having those keys in the database. For some reason to be explained later, we consider an Edis counterpart that takes only one key. A first attempt may lead to something like the following:

```
del :: String → Edis xs (Del xs ?) (R ⊎ Integer)
del key = Edis (Hedis.del [encode key]) ,
```

where the function *encode* converts String to ByteString. At term-level, our *del* merely calls Hedis.*del*. At type-level, if the status of the database before *del* is called meets the constraint represented by the dictionary *xs*, the status afterwards should meet the constraint Del *xs* ?. The question, however, is what to fill in place of the question mark. It cannot be Del *xs key*, since *key* is a runtime value and not a type. How do we smuggle a runtime value to type-level?

In a language with phase distinction like Haskell, it is certainly impossible to pass the value of *key* to the type checker if it truly is a runtime value, for example, a string read from the user. If the value of *key* can be determined statically, however, *singleton types* [6] can be used to represent a type as a value, thus build a connection between the two realms.

A singleton type is a type that has only one term. When the term is built, it carries a type that can be inspected by the type checker. The term can be thought of as a representative of its type at the realm of runtime values. For our purpose, we will use the following type Proxy:

```
data Proxy t = Proxy .
```

For every type t, Proxy t is a type that has only one term: Proxy.[5] To call *del*, instead of passing a key as a String value, we give it a proxy with a specified type:

```
del (Proxy :: Proxy "A") ,
```

where "A" is not a value, but a string lifted to a type (of kind Symbol). Now that the type checker has access to the key, the type of *del* can be Proxy $k \rightarrow$ Edis *xs* (Del *xs* k) (R ⊎ Integer).

The next problem is that, *del*, at term level, gets only a value constructor Proxy without further information, while it needs to pass a ByteString key to Hedis.*del*. Every concrete string literal lifted to a type, for example, "A", belongs to a type class KnownSymbol. For all types k in KnownSymbol, the function *symbolVal*:

```
symbolVal :: KnownSymbol k ⇒ proxy k → String ,
```

retrieves the string associated with a type-level literal that is known at compile time. In summary, *del* can be implemented as:

```
del :: KnownSymbol k ⇒
       Proxy k → Edis xs (Del xs k) (R ⊎ Integer)
del key = Edis (Hedis.del [encodeKey key]) ,
```

where *encodeKey* = *encode* · *symbolVal*.

A final note: the function *encode*, from the Haskell library cereal, helps to convert certain datatypes that are *serializable* into ByteString. The function and its dual *decode* will be used more later.

```
encode :: Serialize a ⇒ a → ByteString ,
decode :: Serialize a ⇒ ByteString → Either String a .
```

4.2 Automatic Serialization

As mentioned before, while Redis provide a number of container types including lists, sets, and hash, etc., the primitive type is string. Hedis programmers manually convert data of other types to strings before saving them into the data store.

[5]While giving the same name to both the type and the term can be very confusing, it is unfortunately a common practice in the Haskell community.

In Edis, we wish to save some of the effort for the programmers, as well as keeping a careful record of the intended types of the strings in the data store.

To keep track of intended types of strings in the data store, we define the following types (that have no terms):

$$\begin{aligned}
&\textbf{data}\ \mathsf{StringOf} :: * \to * \ ,\\
&\textbf{data}\ \mathsf{ListOf} \quad :: * \to * \ ,\\
&\textbf{data}\ \mathsf{SetOf} \quad :: * \to * \ldots
\end{aligned}$$

If a key is associated with, for example, StringOf Int in our dictionary, we mean that its value in the data store was serialized from an Int and should be used as an Int. Types ListOf a and SetOf a, respectively, denotes that the value is a list or a set of type a.

While the *set* command in Hedis always writes a string to the data store, the corresponding *set* in Redis applies to any serializable type (those in the class Serialize), and performs the encoding for the user:

$$\begin{aligned}
&set :: (\mathsf{KnownSymbol}\ k, \mathsf{Serialize}\ a) \Rightarrow \mathsf{Proxy}\ k \to a \to\\
&\qquad \mathsf{Edis}\ xs\ (\mathsf{Set}\ xs\ k\ (\mathsf{StringOf}\ a))\ (\mathsf{Either\ Reply\ Status})\\
&set\ key\ v = \mathsf{Edis}\ (\mathsf{Hedis}.set\ (encodeKey\ key)\ (encode\ v))\ ,
\end{aligned}$$

For example, executing *set* (Proxy :: Proxy "A") True updates the dictionary with an entry '("A", StringOf Bool). If "A" is not in the dictionary, this entry is added; otherwise the old type of "A" is updated to StringOf Bool.

Redis command INCR reads the (string) value of the given key, parses it as an integer, and increments it by one, before storing it back. The command INCRBYFLOAT increments the floating point value of a key by a given amount. They are defined in Edis below:

$$\begin{aligned}
&incr :: (\mathsf{KnownSymbol}\ k, \mathsf{Get}\ xs\ k \sim \mathsf{StringOf\ Integer}) \Rightarrow\\
&\qquad \mathsf{Proxy}\ k \to \mathsf{Edis}\ xs\ xs\ (\mathsf{R} \uplus \mathsf{Integer})\\
&incr\ key = \mathsf{Edis}\ (\mathsf{Hedis}.incr\ (encodeKey\ key))\ ,\\
&incrbyfloat :: (\mathsf{KnownSymbol}\ k, \mathsf{Get}\ xs\ k \sim \mathsf{StringOf\ Double})\\
&\quad \Rightarrow \mathsf{Proxy}\ k \to \mathsf{Double} \to \mathsf{Edis}\ xs\ xs\ (\mathsf{R} \uplus \mathsf{Double})\\
&incrbyfloat\ key\ eps =\\
&\quad \mathsf{Edis}\ (\mathsf{Hedis}.incrbyfloat\ (encodeKey\ key)\ eps)\ .
\end{aligned}$$

Notice the use of ($\sim$), *equality constraints* [19], to enforce that the intended type of the value of k must respectively be Integer and Double. The function *incr* is only allowed to be called in a context where the type checker is able to reduce Get xs k to StringOf Integer — recall that when k is not in xs, Get xs k cannot be fully reduced. The type of *incrbyfloat* works in a similar way.

4.3 Disjunctive Constraints

Recall, from Section 1, that commands `LPUSH key val` and `LLEN key` succeed either when *key* appears in the data store and is assigned a list, or when *key* does not appear at all. What we wish to have in their constraint is thus a predicate equivalent to Get xs k $=\!=$ ListOf $a \lor \neg$ (Member xs k).

$$\begin{aligned}
&\textbf{type}\ \textit{family}\ \mathsf{IsList}\ (t :: *) :: \mathsf{Bool}\ \textbf{where}\\
&\quad \mathsf{IsList}\ (\mathsf{ListOf}\ a) = \text{'}\mathsf{True}\\
&\quad \mathsf{IsList}\ t \qquad\quad\ = \text{'}\mathsf{False}\\
&\textbf{type}\ \textit{family}\ \mathsf{IsSet}\ (t :: *) :: \mathsf{Bool}\ \textbf{where}\\
&\quad \mathsf{IsSet}\ (\mathsf{SetOf}\ a) = \text{'}\mathsf{True}\\
&\quad \mathsf{IsSet}\ t \qquad\quad\ = \text{'}\mathsf{False}\\
&\textbf{type}\ \textit{family}\ \mathsf{IsString}\ (t :: *) :: \mathsf{Bool}\ \textbf{where}\\
&\quad \mathsf{IsString}\ (\mathsf{StringOf}\ a) = \text{'}\mathsf{True}\\
&\quad \mathsf{IsString}\ t \qquad\qquad = \text{'}\mathsf{False}\\
&\textbf{type}\ \mathsf{ListOrNX} \quad xs\ k =\\
&\quad (\mathsf{IsList} \quad (\mathsf{Get}\ xs\ k)\ \text{`}\mathsf{Or}\text{`}\ \mathsf{Not}\ (\mathsf{Member}\ xs\ k)) \sim \text{'}\mathsf{True}\\
&\textbf{type}\ \mathsf{SetOrNX} \quad xs\ k =\\
&\quad (\mathsf{IsSet} \quad (\mathsf{Get}\ xs\ k)\ \text{`}\mathsf{Or}\text{`}\ \mathsf{Not}\ (\mathsf{Member}\ xs\ k)) \sim \text{'}\mathsf{True}\\
&\textbf{type}\ \mathsf{StringOrNX}\ xs\ k =\\
&\quad (\mathsf{IsString}\ (\mathsf{Get}\ xs\ k)\ \text{`}\mathsf{Or}\text{`}\ \mathsf{Not}\ (\mathsf{Member}\ xs\ k)) \sim \text{'}\mathsf{True}
\end{aligned}$$

Figure 2. The "well-typed, or non-existent" constraints.

To impose a conjunctive constraint P ∧ Q, one may simply put them both in the type: (P, Q) ⇒ .... Expressing disjunctive constraints is only slightly harder, thanks to our type-level functions. We may thus write the predicate as:

$$\mathsf{Get}\ xs\ k \sim \mathsf{ListOf}\ a\ \text{`}\mathsf{Or}\text{`}\ \mathsf{Not}\ (\mathsf{Member}\ xs\ k)\ .$$

To avoid referring to a, which might not exist, we define an auxiliary predicate IsList :: $* \to$ Bool such that IsList t reduces to 'True only if t = ListOf a. As many Redis commands are invokable only under such "well-typed, or non-existent" precondition, we give names to such constraints, as seen in Figure 2.

The Edis counterpart of LPUSH and LLEN are therefore:

$$\begin{aligned}
&lpush :: (\mathsf{KnownSymbol}\ k, \mathsf{Serialize}\ a, \mathsf{ListOrNX}\ xs\ k) \Rightarrow\\
&\qquad \mathsf{Proxy}\ k \to a \to\\
&\qquad\quad \mathsf{Edis}\ xs\ (\mathsf{Set}\ xs\ k\ (\mathsf{ListOf}\ a))\ (\mathsf{R} \uplus \mathsf{Integer})\\
&lpush\ key\ val =\\
&\qquad \mathsf{Edis}\ (\mathsf{Hedis}.lpush\ (encodeKey\ key)\ [\,encode\ val\,])\ ,\\
&llen :: (\mathsf{KnownSymbol}\ k, \mathsf{ListOrNX}\ xs\ k) \Rightarrow\\
&\qquad \mathsf{Proxy}\ k \to \mathsf{Edis}\ xs\ xs\ (\mathsf{R} \uplus \mathsf{Integer})\\
&llen\ key = \mathsf{Edis}\ (\mathsf{Hedis}.llen\ (encodeKey\ key))\ .
\end{aligned}$$

Similarly, the type of *sadd*, a function we have talked about a lot, is given below:

$$\begin{aligned}
&sadd :: (\mathsf{KnownSymbol}\ k, \mathsf{Serialize}\ a, \mathsf{SetOrNX}\ xs\ k) \Rightarrow\\
&\qquad \mathsf{Proxy}\ k \to a \to\\
&\qquad\quad \mathsf{Edis}\ xs\ (\mathsf{Set}\ xs\ k\ (\mathsf{SetOf}\ a))\ (\mathsf{R} \uplus \mathsf{Integer})\\
&sadd\ key\ val =\\
&\quad \mathsf{Edis}\ (\mathsf{Hedis}.sadd\ (encodeKey\ key)\ [\,encode\ val\,])\ ,
\end{aligned}$$

To see a command with a more complex type, consider *setnx*, which uses the type-level function If defined in Section 3.2:

$$\begin{aligned}&\mathit{setnx} :: (\mathsf{KnownSymbol}\ k, \mathsf{Serialize}\ a) \Rightarrow \mathsf{Proxy}\ k \to a \to\\ &\qquad \mathsf{Edis}\ \mathit{xs}\ (\mathsf{If}\ (\mathsf{Member}\ \mathit{xs}\ k)\ \mathit{xs}\ (\mathsf{Set}\ \mathit{xs}\ k\ (\mathsf{StringOf}\ a)))\\ &\qquad\qquad (\mathsf{Either}\ \mathsf{Reply}\ \mathsf{Bool})\\ &\mathit{setnx}\ \mathit{key}\ \mathit{val} =\\ &\quad \mathsf{Edis}\ (\mathsf{Hedis}.\mathit{setnx}\ (\mathit{encodeKey}\ \mathit{key})\ (\mathit{encode}\ \mathit{val}))\ .\end{aligned}$$

From the type one can see that *setnx key val* creates a new entry (*key*, *val*) in the data store only if *key* is fresh. The type of *setnx* computes a postcondition for static checking, as well as serving as a good documentation for its semantics.

4.4 Hashes

Hash is a useful datatype supported by Redis. While the Redis data store can be seen as a set of key/value pairs, a hash is itself a set of field/value pairs. The following commands assigns a hash to key `user`. The fields are `name`, `birthyear`, and `verified`, respectively with values `banacorn`, `1992`, and `1`.

```
redis> hmset user name banacorn
       birthyear 1992 verified 1
OK
redis> hget user name
"banacorn"
redis> hget user birthyear
"1992"
```

For a hash to be useful, we should allow the fields to have different types. To keep track of types of fields in a hash, HashOf takes a list of (Symbol, $*$) pairs:

$$\mathbf{data}\ \mathsf{HashOf} :: [(\mathsf{Symbol}, *)] \to *\ .$$

By having an entry (k, HashOF ys) in a dictionary, we denote that the value of key k is a hash whose fields and their types are specified by ys, which is also a dictionary.

Figure 3 presents some operations we need on dictionaries when dealing with hashes. Let xs be a dictionary, GetHash $xs\ k\ f$ returns the type of field f in the hash assigned to key k, if both k and f exists. SetHash $xs\ k\ f\ a$ assigns the type a to the field f of hash k; if either f or k does not exist, the hash/field is created. Del $xs\ k\ f$ removes a field, while MemHash $xs\ k\ f$ checks whether the key k exists in xs, and its value is a hash having field f. Their definitions make use of functions Get, Set, and Member defined for dictionaries.

Once those type-level functions are defined, embedding of Hedis commands for hashes is more or less routine. For example, functions *hset* and *hget* are shown below. Note that, instead of *hmset* (available in Hedis), we provide a function *hset* that assigns fields and values one pair at a time.

$$\begin{aligned}&\mathit{hset} :: (\mathsf{KnownSymbol}\ k, \mathsf{KnownSymbol}\ f,\\ &\qquad \mathsf{Serialize}\ a, \mathsf{HashOrNX}\ \mathit{xs}\ k)\\ &\qquad \Rightarrow \mathsf{Proxy}\ k \to \mathsf{Proxy}\ f \to a\\ &\qquad \to \mathsf{Edis}\ \mathit{xs}\ (\mathsf{SetHash}\ \mathit{xs}\ k\ f\ (\mathsf{StringOf}\ a))\ (\mathsf{R} \uplus \mathsf{Bool})\\ &\mathit{hset}\ \mathit{key}\ \mathit{field}\ \mathit{val} =\\ &\quad \mathsf{Edis}\ (\mathsf{Hedis}.\mathit{hset}\ (\mathit{encodeKey}\ \mathit{key})\\ &\qquad\qquad (\mathit{encodeKey}\ \mathit{field})\ (\mathit{encode}\ \mathit{val}))\ ,\\ &\mathit{hget} :: (\mathsf{KnownSymbol}\ k, \mathsf{KnownSymbol}\ f, \mathsf{Serialize}\ a,\\ &\qquad \mathsf{StringOf}\ a \sim \mathsf{GetHash}\ \mathit{xs}\ k\ f) \Rightarrow\\ &\qquad \mathsf{Proxy}\ k \to \mathsf{Proxy}\ f \to \mathsf{Edis}\ \mathit{xs}\ \mathit{xs}\ (\mathsf{R} \uplus \mathsf{Maybe}\ a)\\ &\mathit{hget}\ \mathit{key}\ \mathit{field} =\\ &\quad \mathsf{Edis}\ (\mathsf{Hedis}.\mathit{hget}\ (\mathit{encodeKey}\ \mathit{key})\ (\mathit{encodeKey}\ \mathit{field}) \ggg\!=\\ &\qquad \mathit{decodeAsMaybe})\ ,\end{aligned}$$

where

$$\begin{aligned}&\mathit{decodeAsMaybe} :: \mathsf{Serialize}\ a \Rightarrow (\mathsf{R} \uplus \mathsf{Maybe}\ \mathsf{ByteString}) \to\\ &\quad \mathsf{Redis}\ (\mathsf{R} \uplus \mathsf{Maybe}\ a)\ ,\end{aligned}$$

using the function *decode* mentioned in Section 4.1, parses the ByteString in R $\uplus$ Maybe _ to type a. The definition is a bit tedious but routine.

We will talk about difficulties of implementing *hmset* in Section 5.

4.5 Assertions

Finally, the creation/update behavior of Redis functions is, in our opinion, very error-prone. It might be preferable if we can explicit declare some new keys, after ensuring that they do not already exist (in our types). This can be done below:

$$\begin{aligned}&\mathit{declare} :: (\mathsf{KnownSymbol}\ k, \mathsf{Member}\ \mathit{xs}\ k \sim \mathsf{False}) \Rightarrow\\ &\qquad \mathsf{Proxy}\ k \to \mathsf{Proxy}\ a \to \mathsf{Edis}\ \mathit{xs}\ (\mathsf{Set}\ \mathit{xs}\ k\ a)\ ()\\ &\mathit{declare}\ \mathit{key}\ \mathit{typ} = \mathsf{Edis}\ (\mathit{return}\ ())\ .\end{aligned}$$

The command *declare key typ*, where *typ* is the proxy of a, adds a fresh *key* with type a into the dictionary. Notice that *declare* does nothing at term level, but simply returns (), since it only has effects on types.

Another command for type level assertion, *start*, initializes the dictionary to the empty list, comes in handy when starting a series of Edis commands:

$$\begin{aligned}&\mathit{start} :: \mathsf{Edis}\ '[\,]\ '[\,]\ ()\\ &\mathit{start} = \mathsf{Edis}\ (\mathit{return}\ ())\ .\end{aligned}$$

4.6 A Slightly Larger Example

We present a slightly larger example as a summary. The task is to store a queue of messages in Redis. Messages are represented by a ByteString and an Integer identifier:[6]

$$\begin{aligned}&\mathbf{data}\ \mathsf{Message} = \mathsf{Msg}\ \mathsf{ByteString}\ \mathsf{Integer}\\ &\quad \mathbf{deriving}\ (\mathsf{Show}, \mathsf{Generic})\ ,\\ &\mathbf{instance}\ \mathsf{Serialize}\ \mathsf{Message}\ \mathbf{where}\ \ .\end{aligned}$$

In the data store, the queue is represented by a list. Before pushing a message into the queue, we increment *counter*,

[6]Message is made an instance of Generic in order to use the generic implementation of methods of Serialize.

$$
\begin{array}{l}
\textbf{type}\ \textit{family}\ \mathrm{GetHash}\ (xs :: [(\mathrm{Symbol}, *)])\ (k :: \mathrm{Symbol})\ (f :: \mathrm{Symbol}) :: *\,\textbf{where} \\
\quad \mathrm{GetHash}\ ('(k, \mathrm{HashOf}\ hs)\ ': xs)\ k\ f = \mathrm{Get}\ hs\ f \\
\quad \mathrm{GetHash}\ ('(l, y)\ ': xs)\ k\ f = \mathrm{GetHash}\ xs\ k\ f \\
\textbf{type}\ \textit{family}\ \mathrm{SetHash}\ (xs :: [(\mathrm{Symbol}, *)])\ (k :: \mathrm{Symbol})\ (f :: \mathrm{Symbol})\ (a :: *) :: [(\mathrm{Symbol}, *)]\ \textbf{where} \\
\quad \mathrm{SetHash}\ '[\,]\ k\ f\ a = '(k, \mathrm{HashOf}\ (\mathrm{Set}\ '[\,]\ f\ a))\ ': '[\,] \\
\quad \mathrm{SetHash}\ ('(k, \mathrm{HashOf}\ hs)\ ': xs)\ k\ f\ a = '(k, \mathrm{HashOf}\ (\mathrm{Set}\ hs\ f\ a))\ ': xs \\
\quad \mathrm{SetHash}\ ('(l, y)\ ': xs)\ k\ f\ a = '(l, y)\ ': \mathrm{SetHash}\ xs\ k\ f\ a \\
\textbf{type}\ \textit{family}\ \mathrm{DelHash}\ (xs :: [(\mathrm{Symbol}, *)])\ (k :: \mathrm{Symbol})\ (f :: \mathrm{Symbol}) :: [(\mathrm{Symbol}, *)]\ \textbf{where} \\
\quad \mathrm{DelHash}\ '[\,]\ k\ f = '[\,] \\
\quad \mathrm{DelHash}\ ('(k, \mathrm{HashOf}\ hs)\ ': xs)\ k\ f = '(k, \mathrm{HashOf}\ (\mathrm{Del}\ hs\ f))\ ': xs \\
\quad \mathrm{DelHash}\ ('(l, y)\ ': xs)\ k\ f = '(l, y)\ ': \mathrm{DelHash}\ xs\ k\ f \\
\textbf{type}\ \textit{family}\ \mathrm{MemHash}\ (xs :: [(\mathrm{Symbol}, *)])\ (k :: \mathrm{Symbol})\ (f :: \mathrm{Symbol}) :: \mathrm{Bool}\ \textbf{where} \\
\quad \mathrm{MemHash}\ '[\,]\ k\ f = '\mathrm{False} \\
\quad \mathrm{MemHash}\ ('(k, \mathrm{HashOf}\ hs)\ ': xs)\ k\ f = \mathrm{Member}\ hs\ f \\
\quad \mathrm{MemHash}\ ('(k, x)\ ': xs)\ k\ f = '\mathrm{False} \\
\quad \mathrm{MemHash}\ ('(l, y)\ ': xs)\ k\ f = \mathrm{MemHash}\ xs\ k\ f
\end{array}
$$

Figure 3. Type-level operations for dictionaries with hashes.

a key storing a counter, and use it as the identifier of the message:

$$
\begin{array}{l}
\textit{push} :: (\mathrm{StringOfIntegerOrNX}\ xs\ \texttt{"counter"}, \\
\qquad \mathrm{ListOrNX}\ xs\ \texttt{"queue"}) \Rightarrow \\
\qquad \mathrm{ByteString} \rightarrow \mathrm{Edis}\ xs\ (\mathrm{Set}\ xs\ \texttt{"queue"} \\
\qquad\quad (\mathrm{ListOf\ Message}))\ (\mathrm{R} \uplus \mathrm{Integer}) \\
\textit{push}\ \textit{msg} = \textit{incr}\ \textit{kCounter}\ `\textit{bind}`\ \lambda i \rightarrow \\
\qquad\qquad \textit{lpush}\ \textit{kQueue}\ (\mathrm{Msg}\ \textit{msg}\ (\textit{fromRight}\ i))\ ,
\end{array}
$$

where $\textit{fromRight} :: \mathrm{Either}\ a\ b \rightarrow b$ extracts the value wrapped by constructor Right, and the constraint StringOfIntegerOrNX $xs\ k$ holds if either k appears in xs and is converted from an Integer, or k does not appear in xs. For brevity, the proxies are given names:

$$
\begin{array}{l}
\textit{kCounter} :: \mathrm{Proxy}\ \texttt{"counter"} \\
\textit{kCounter} = \mathrm{Proxy}\ , \\
\\
\textit{kQueue} :: \mathrm{Proxy}\ \texttt{"queue"} \\
\textit{kQueue} = \mathrm{Proxy}\ .
\end{array}
$$

To pop a message we use the function *rpop* which, given a key associated with a list, extracts the rightmost element of the list

$$
\begin{array}{l}
\textit{pop} :: (\mathrm{Get}\ xs\ \texttt{"queue"} \sim \mathrm{ListOf\ Message}) \Rightarrow \\
\qquad \mathrm{Edis}\ xs\ xs\ (\mathrm{R} \uplus \mathrm{Maybe\ Message}) \\
\textit{pop} = \textit{rpop}\ \textit{kQueue}\ , \\
\textit{rpop} :: (\mathrm{KnownSymbol}\ k, \mathrm{Serialize}\ a, \mathrm{Get}\ xs\ k \sim \mathrm{ListOf}\ a) \Rightarrow \\
\qquad \mathrm{Proxy}\ k \rightarrow \mathrm{Edis}\ xs\ xs\ (\mathrm{R} \uplus \mathrm{Maybe}\ a) \\
\textit{rpop}\ \textit{key} = \mathrm{Edis}\ (\mathrm{Hedis}.\textit{rpop}\ (\textit{encodeKey}\ \textit{key}) \gg= \\
\qquad\qquad\qquad \textit{decodeAsMaybe})\ .
\end{array}
$$

Our sample program is shown below:

$$
\begin{array}{l}
\textit{prog} = \textit{declare}\ \textit{kCounter}\ (\mathrm{Proxy} :: \mathrm{Proxy\ Integer}) \\
\qquad \ggg \textit{declare}\ \textit{kQueue}\ \ (\mathrm{Proxy} :: \mathrm{Proxy}\ (\mathrm{ListOf\ Message})) \\
\qquad \ggg \textit{push}\ \texttt{"hello"} \\
\qquad \ggg \textit{push}\ \texttt{"world"} \\
\qquad \ggg \textit{pop}\ ,
\end{array}
$$

where the monadic sequencing operator ($\ggg$) is defined by:

$$
\begin{array}{l}
(\ggg) :: \mathrm{IMonad}\ m \Rightarrow m\ p\ q\ a \rightarrow m\ q\ r\ b \rightarrow m\ p\ r\ b \\
m_1 \ggg m_2 = m_1\ `\textit{bind}`\ (\lambda\_ \rightarrow m_2)\ .
\end{array}
$$

Use of *declare* in *prog* ensures that neither "counter" nor "queue" exist before the execution of *prog*. The program simply stores two strings in "queue", before extracting the first string. GHC is able to infer the type of *prog*:

$$
\begin{array}{l}
\textit{prog} :: \mathrm{Edis}\ '[\,]\ '['(\texttt{"counter"}, \mathrm{Integer}), \\
\qquad\qquad\qquad '(\texttt{"queue"}, \mathrm{ListOf\ Message})] \\
\qquad\qquad (\mathrm{R} \uplus \mathrm{Maybe\ Message})\ .
\end{array}
$$

To get things going, the main program builds a connection with the Redis server, runs *prog*, and prints the result:

$$
\begin{array}{l}
\textit{main} :: \mathrm{IO}\ () \\
\textit{main} = \textbf{do}\ \textit{conn} \leftarrow \textit{connect}\ \textit{defaultConnectInfo} \\
\qquad\qquad\ \textit{result} \leftarrow \textit{runRedis}\ \textit{conn}\ (\textit{unEdis}\ (\textit{start} \ggg \textit{prog})) \\
\qquad\qquad\ \textit{print}\ \textit{result}\ .
\end{array}
$$

The command *start* in *main* guarantees that the program is given a fresh run without previously defined keys at all. All type-level constraints in *start* $\ggg$ *prog* are stripped away by *unEdis*. The untyped program stored in Edis, of type Redis (R $\uplus$ Maybe Message), is passed to the Redis function *runRedis*, of type Connection $\rightarrow$ Redis a $\rightarrow$ IO a. In this case the output is Right (Just "hello").

5 Discussion

Returning Inferrable Types. `GET` is yet another command that is invokable only under a "well-typed or non-existent" precondition, mentioned in Section 4.3. It fetches the value of a key and, if the key does not exist, returns a special value `nil`. An error is raised if the value is not a string. In Edis the situation is made slightly complex, since we parse the string to the type it was supposed to have encoded from. The Edis version of *get* could be typed:

$$\begin{aligned}&\mathit{get} :: (\mathsf{KnownSymbol}\ k, \mathsf{Serialize}\ a, \mathsf{StringOrNX}\ \mathit{xs}\ k) \Rightarrow\\ &\quad \mathsf{Proxy}\ k \to \mathsf{Edis}\ \mathit{xs}\ \mathit{xs}\ (\mathsf{R} \uplus \mathsf{Maybe}\ a)\ .\end{aligned}$$

where StringOrNX is defined in Figure 2.

The problem with such typing, however, is that a cannot be inferred from xs and k when k does not appear in xs. In such situations, to avoid Haskell complaining about ambiguous type, a has to be specified by the caller of *get*. The user will then be forced to spell out the complete type signature, only to make a explicit.

We think it is more reasonable to enforce that, when *get* is called, the key should exist in the data store. Thus *get* in Redis has the following type:

$$\begin{aligned}&\mathit{get} :: (\mathsf{KnownSymbol}\ k, \mathsf{Serialize}\ a, \mathsf{Get}\ \mathit{xs}\ k \sim \mathsf{StringOf}\ a) \Rightarrow\\ &\quad \mathsf{Proxy}\ k \to \mathsf{Edis}\ \mathit{xs}\ \mathit{xs}\ (\mathsf{R} \uplus \mathsf{Maybe}\ a)\ ,\end{aligned}$$

which requires that (k, a) presents in xs and thus a is inferrable from xs and k.

Variable Number of Input/Outputs. Recall that, in Section 4.1, the Redis command `DEL` takes a variable number of keys, while our Edis counterpart takes only one. Some Redis commands take a variable number of arguments as inputs, and some returns multiple results. Most of them are accurately implemented in Hedis. For another example of a variable-number input command, the type of *sinter* in Hedis is shown below:

$$\mathsf{Hedis}.\mathit{sinter} :: [\mathsf{ByteString}] \to \mathsf{Redis}\ (\mathsf{R} \uplus [\mathsf{ByteString}])\ .$$

It takes a list of keys, values of which are all supposed to be sets, and computes their intersection (the returned list is the intersected set).

In Edis, for a function to accept a list of keys as input, we have to specify that all the keys are in the class KnownSymbol. It can be done by defining a datatype, indexed by the keys, serving as a witness that they are all in KnownSymbol. We currently have not implemented such feature and leave it as a possible future work. For now, we offer commands that take fixed numbers of inputs. The Edis version of *sinter* has type:

$$\begin{aligned}&\mathit{sinter} :: (\mathsf{KnownSymbol}\ k_1, \mathsf{KnownSymbol}\ k_2, \mathsf{Serialize}\ a,\\ &\qquad \mathsf{SetOf}\ x \sim \mathsf{Get}\ \mathit{xs}\ k_1, \mathsf{SetOf}\ x \sim \mathsf{Get}\ \mathit{xs}\ k_2) \Rightarrow\\ &\qquad \mathsf{Proxy}\ k_1 \to \mathsf{Proxy}\ k_2 \to \mathsf{Edis}\ \mathit{xs}\ \mathit{xs}\ (\mathsf{R} \uplus [a])\ .\end{aligned}$$

The function *hmset* in Hedis allows one to set the values of many fields in a hash, while *hgetall* returns all the field-value pairs of a hash. They have the following types:

$$\begin{aligned}&\mathsf{Hedis}.\mathit{hmset} :: \mathsf{ByteString} \to\\ &\quad [(\mathsf{ByteString}, \mathsf{ByteString})] \to \mathsf{Redis}\ (\mathsf{R} \uplus \mathsf{Status})\ ,\\ &\mathsf{Hedis}.\mathit{hgetall} :: \mathsf{ByteString} \to\\ &\quad \mathsf{Redis}\ (\mathsf{R} \uplus [(\mathsf{ByteString}, \mathsf{ByteString})])\ .\end{aligned}$$

Proper implementations of them in Edis should accept or return a *heterogeneous list* [9] — a list whose elements can be of different types. We also leave such functions as a future work.

Not All Safe Redis Programs Can Be Typechecked. Enforcing a typing discipline rules out some erroneous programs, and reduces the number of programs that are allowed. Like all type systems, our type system takes a conservative estimation: there are bound to be some Redis programs that are not typable in our type system, but do not actually throw a type error. We demand that elements in our lists must be of the same type, for example, while a Redis program could store in a list different types of data, encoded as strings, and still work well.

One innate limitation is that we cannot allow dynamic generation of keys. In Hedis, the Haskell program is free to generate arbitrary sequence of keys to be used in the data store, which is in general not possible due to the static nature of Edis.

Transactions. Commands in Redis can be wrapped in *transactions*. Redis offers two promises regarding commands in a transaction. Firstly, all commands in a transaction are serialized and executed sequentially, without interruption from another client. Secondly, either all of the commands or none are processed.

Support of transactions in Edis is a future work. We expect that there would not be too much difficulty — in an early experiment, we have implemented a runtime type checker specifically targeting Redis transactions, and we believe that the experience should be applicable to static type checking as well.

6 Conclusions and Related Work

By exploiting various recent extensions and type-level programming techniques, we have designed a domain-specific embedded language Edis which enforces typing disciplines that keep track of available keys and their types. The type of a command clearly specifies which keys must or must not present in the data store, what their types ought to be, as well as how the keys and types are updated after the execution. A program can be constructed only if it does not throw a runtime type error when it is run with a store whose status matches its precondition and when it is the sole client interacting with the store. The type also serves as documentation of the commands. We believe that it is a neat case study of application of type-level programming.

REDIS identifies itself as a data structure store/server, rather than a database. Nevertheless, there has been attempts to design DSELs for relational databases that guarantee all queries made are safe. Among them, HASKELLDB [2, 11] dynamically generates, from monad comprehensions, SQL queries to be executed on ODBC database servers. With the expressiveness of dependent types, Oury and Swierstra [17] build a DSEL in Agda for relational algebra. Eisenberg and Weirich [6] ported the result to Haskell using singleton types, after GHC introduced more features facilitating type-level programming.

None of the type-level programming techniques we used in this paper are new. Indexed monads (also called parameterized monads) have been introduced by Atkey [1]. McBride [14] showed how to construct indexed free-monads from Hoare Logic specifications. Kiselyov et al. [8] used indexed monads to track the locks held among a given finite set. The same paper also demonstrated implementation of a variety of features including memorization, generic maps, session types, typed `printf` and `sprintf`, etc., by type-level programming. Before the introduction of type families, Kiselyov and Shan [10] used type classes and functional dependencies to implement type-level functions, and showed that they are sufficient to track resources in device drivers.

Lindley and McBride [12] provided a thorough analysis and summary of the dependent-type-like features currently in Haskell, and compared them with dependently typed languages without phase distinction such as Agda. It turns out that GHC's constraint solver works surprisingly well as an automatic theorem prover.

References

[1] Robert Atkey. 2009. Parameterised notions of computation. *Journal of Functional Programming* 19, 3-4 (2009), 335–376.

[2] Björn Bringert, Anders Höckersten, Conny Andersson, Martin Andersson, Mary Bergman, Victor Blomqvist, and Torbjörn Martin. 2004. Student paper: HaskellDB improved, See [15], 108–115.

[3] Manuel MT Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated type synonyms. In *International Conference on Functional Programming*, Benjamin C. Pierce (Ed.). ACM Press, 241–253.

[4] Manuel MT Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated types with class. In *Symposium on Principles of Programming Languages*, Jens Palsberg and Martin Abadi (Eds.). ACM Press, Long Beach, California, USA, 1–13.

[5] Richard A Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed type families with overlapping equations. In *Symposium on Principles of Programming Languages*, Peter Sewell (Ed.). ACM Press, 671–683.

[6] Richard A Eisenberg and Stephanie Weirich. 2012. Dependently typed programming with singletons. In *Symposium on Haskell*, Robby Findler (Ed.). ACM Press, 117–130.

[7] James Hook and Peter Thiemann (Eds.). 2008. *International Conference on Functional Programming*. ACM Press.

[8] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. 2010. Fun with type functions. In *Reflections on the Work of C.A.R. Hoare*, A. William Roscoe, Cliff B. Jones, and Kenneth R. Wood (Eds.). Springer, 301–331.

[9] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections, See [15], 96–107.

[10] Oleg Kiselyov and Chung-chieh Shan. 2007. Lightweight Static Resources: Sexy types for embedded and systems programming. In *Symposium on Trends in Functional Programming*.

[11] Daan Leijen and Erik Meijer. 1999. Domain specific embedded compilers. In *Domain-Specific Languages*, Thomas Ball (Ed.). ACM Press, 109–122.

[12] Sam Lindley and Conor McBride. 2013. Hasochism: the pleasure and pain of dependently typed Haskell programming. In *Symposium on Haskell*, Chung-chieh Shan (Ed.). ACM Press, 81–92.

[13] Conor McBride. 2002. Faking it: simulating dependent types in Haskell. *Journal of Functional Programming* 12, 5 (July 2002), 375–392.

[14] Conor McBride. 2011. Functional pearl: Kleisli arrows of outrageous fortune. (2011).

[15] Henrik Nilsson (Ed.). 2004. *Haskell Workshop*. ACM Press.

[16] Dominic Orchard and Tom Schrijvers. 2010. Haskell type constraints unleashed. In *Functional and Logic Programming*. Springer, 56–71.

[17] Nicolas Oury and Wouter Swierstra. 2008. The power of Pi, See [7], 39–50.

[18] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type checking with open type functions, See [7], 51–62.

[19] Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*. ACM, 53–66.

[20] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon L. Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Types in Language Design and Implementation*, Benjamin C. Pierce (Ed.). ACM Press, 53–66.