Preventing runtime errors of Redis at compile time

Abstract

Programmers often interact with database systems, by sending queries through libraries or packages in some programming languages. But people make mistakes, while some of the syntactic and semantic errors can be prevented by the language at compile time, or caught by the package at runtime, most semantic errors are just being ignored, causing problems at the database system.

In this paper, we demonstrates how to prevent those runtime errors at compile time, allowing users to write more reliable database queries without runtime overhead, by exploiting type-level programming techniques such as indexed monad, type-level literals and closed type families in Haskell.

The database system and the package we are targeting are *Redis* and *Hedis* respectively, and our implementation is available as *Edis* on *Hackage*.

Categories and Subject Descriptors

Keywords type-level programming; Haskell; database query language; Redis;

1. Introduction

1.1 Redis

Redis is an open source, in-memory data structure store, used as database, cache and message broker. Each value is associated with a binary-safe string key to identify and manipulate with. Redis supports many different kind of values, such as strings, hashes, lists and sets.

To manipulate these values, Redis comes with a set of atomic operations, called *commands*.

For example, if we want to add a bunch of strings to 2 sets, and intersect them, we could input the following sequence of commands into Redis's client interactively.

```
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"
```

Keys such as `some-set` and `SADD` are created on site, with the command `SADD`, which returns the size of set after completed.

1.2 Hedis

Redis can also be used in most programming languages, with 3rd party libraries or packages that talk with Redis' TCP protocol. In Haskell, the most popular package is Hedis.

The previous example in Redis's client would look something like this with Hedis in Haskell.

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

Function `sadd` takes a key and a list of values as arguments, and returns an `Integer`, wrapped in `Either` to indicate possible failures in the context `Redis`[1]of command execution.

```
sadd :: ByteString      -- key
    -> [ByteString]     -- values
    -> Redis (Either Reply Integer)
```

Keys and values are of type `ByteString`, since they are all just binary strings in Redis. If a user wants to store values of arbitrary types, he or she will have to encode and decode them as `ByteString`s.

[1] Hedis provides another kind of context, `RedisTx`, for *transactions*, united with `Redis` under the class of `RedisCtx`. We use `Redis` only here for brevity.

1.3 Problems

"All binary strings are equal, but some binary strings are more equal than others."

Although everything in Redis is essentially a binary string, they are treated differently. Redis supports many different kind of data structures, such as strings, hashes, lists, etc. These values have different datatypes, and most commands only work with certain types of them, much like how C language treats a piece of data.

Problem 1 Consider the following example. `SET` is a command that only works on strings. The key `some-string` is now related with a string. If we treat it as a set and add an element to it with `SADD`, a runtime error arises.

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

Problem 2 Even worse, not all strings are equal! `INCR` parses the string value as an integer, increments it by one, and store it back as a string. If a string value can't be parse as an integer, another runtime error arises

```
redis> SET some-string foo
OK
redis> INCR some-string
(error) ERR value is not an integer or out
 of range
```

In Hedis The same goes for Hedis, since it's only a simple wrapping on top of Redis's protocol written in Haskell.

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

It yields the same error as in Redis's client.

```
Left (Error "WRONGTYPE Operation against a
 key holding the wrong kind of value")
```

The Cause These problems arise from the absence of type checking, with respects to **the type of a value that associates with a key**.

1.4 Hedis as an embedded DSL

Haskell makes it easy to build and use Domain Specific Languages (DSLs), and Hedis can be regarded as one of them. What makes Hedis special is that, it has variable bindings (between keys and values), but with very little or no semantic checking, neither dynamically nor statically.

We began with making Hedis a dynamically checked embedded DSL, and implemented a runtime type checker that keeps track of everything. But then we found that things can be a lot easier, by leveraging the host language's type checker, which also make it statically type-checked!

1.5 Contributions

To summarize our contributions:

- We make Hedis statically type-checked, without runtime overhead.
- We demonstrates how to model variable bindings of an embedded DSL with language extensions like type-level literals and data kinds.
- We provide (yet another) an example of encoding effects and constraints of an action in types, with indexed monad[1] and other language extensions such as closed type-families[**?**] and constraints kinds[11].
- Edis, a package we built for programmers. This package helps programmers to write more reliable Redis codes, and also makes Redis polymorphic by automatically converting back and forth from values of any types and boring ByteStrings.

2. Type-level dictionaries

To check the bindings between keys and values, we need a *dictionary-like* structure, and encode it as a *type* somehow.

2.1 Datatype promotion

Normally, at the term level, we could express the datatype of dictionary with *type synonym* like this.[2]

```
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"
```

To encode this in the type level, everything has to be *promoted*[13] one level up. From terms to types, and from types to kinds.

Luckily, with recently added GHC extension *data kinds*. With data kinds we can define our own kinds just like we can with datatypes, and every suitable datatype will be promoted to be a kind, and its value constructors promoted to be type constructors, automatically.

```
data List a = Nil | Cons a (List a)
```

[2] `some-set` here represents term-level representations of datatypes, available in `Data.Typeable`

Give rise to the following kinds and type constructors:[3][4]

```
List k :: BOX
Nil  :: List k
Cons :: k -> List k -> List k
```

Haskell sugars lists `[1, 2, 3]` and tuples `(1, 'a')` with brackets and parentheses. Promoted lists and tuples are also sugared with single quote prefixed. For example `'[Int, Char]`, `'(Int, Char)`.

2.2 Type-level literals

Now we have type-level lists and tuples to construct the dictionary. For keys, `String` also has a type-level correspondence: `Symbol`.

```
data Symbol
```

Symbol is defined without a value constructor, because it's intended to be used as a promoted kind.

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

Nonetheless, it's still useful to have a term-level value that links with a Symbol, when we want to retreive type-level informations at runtime (but not the other way around!).

2.3 Putting everything together

With all of these ingrediants ready, let's build some dictionaries!

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

They are defined with *type synonym*, since they are *types* , not *terms*. If we ask `GHCi` what is the kind of `Dict1`, we get `Dict1 :: [ (Symbol, *) ]`

The star `*` is a kind, it stands for any kind of types, such as `Int`, `Char` or even a symbol `"symbol"`, while `Symbol` stands only for all symbols.

3. Indexed monads

Redis commands are *actions*. We could capture the effects caused by an action, by expressing the states it affects, before and after. That is, the *preconditions* and *postconditions* of an action. In such way, we could also impose constraints on the preconditions.

Indexed monads (or *monadish*, *parameterised monad*)[1] can be used[8][7] to model such preconditions and postconditions in types. An indexed monad is a type constructor that takes three arguments: an intitial state, a final state, and the type of a value that an action computes, which can be read like a Hoare triple[10].

```
class IMonad m where
    unit :: a -> m p p a
    bind :: m p q a -> (a -> m q r b) -> m p r b
```

Class `IMonad` comes with two operations: `unit` for identities and `bind` for compositions, as in Monad.

We define a new datatype `Edis`, which is basically just `Redis` indexed with more informations in types. And we make it an instance of `IMonad`.

```
newtype Edis p q a = Edis { unEdis :: Redis a }

instance IMonad Edis where
    unit = Edis . return
    bind m f = Edis (unEdis m >>= unEdis . f )
```

The first and second argument of type `Edis` is where the dictionaries going to stay.

To execute a `Edis` program, simply apply it to `unEdis` to get an ordinary `Hedis` program, with type informations erased.

3.1 `PING`: the first attempt

In Redis, `PING` does nothing but replies with `PONG` if the connection is alive. In Hedis, `PING` has type:

```
ping :: Redis (Either Reply Status)
```

Now we have `Edis`, let's make our own version of `ping`[5]

```
ping :: Edis xs xs (Either Reply Status)
ping = Edis Hedis.ping
```

Dicionary `xs` in the type remain unaffected after the action, because `ping` does not affect any key-value bindings. To encode other commands that alters key-value bindings, we need type-level functions to annotate those effects on the dictionary.

4. Type-level functions

4.1 Closed type families

Type families have a wide variaty of applications. They can appear inside type classes[3][2], or at toplevel. Toplevel type families can be used to compute over types, they come in two forms: open[12] and closed [5].

We choose *closed type families*, because it allows overlapping instances, and we need none of the extensibility provided by open type families. For example, consider both term-level and type-level `&&`:

```
(&&) :: Bool -> Bool -> Bool
True && True = True
```

[4] To distinguish between types and promoted constructors that have ambiguous names, prefix promoted constructor with a single quote like `'Cons` and `Integer`

[4] All kinds have *sort* BOX in Haskell[6]

[5] ping from Hedis is qualified with `Hedis` to prevent function name clashing

```
a     && b     = False

type family And (a :: Bool) (b :: Bool) :: Bool where
    And True True = True
    And a    b    = False
```

The first instance of `And` could be subsumed under the more general instance `And a b`. But the closedness allows these instances to be resoluted in order, just like how cases are resoluted in term-level functions. Also notice that how much `And` resembles to it's term-level sibling.

4.2 Functions on type-level dictionaries

With closed type families, we could define functions on the type level. Let's begin with dictionary lookup.

```
type family Get
    (xs :: [(Symbol, *)])    -- dictionary
    (s :: Symbol)            -- key
    :: * where               -- type

    Get ('(s, x) ': xs) s = x
    Get ('(t, x) ': xs) s = Get xs s
```

Another benifit of closed type families is that type-level equality can be expressed by unifying type variables with the same name. `Get` takes two type arguments, a dictionary and a symbol. If the key we are looking for unifies with the symbol of an entry, then `Get` returns the corresponding type, else it keeps searching down the rest of the dictionary.

`Get '[ '("A", Int) ] "A"` evaluates to `Int`.

But `Get '[ '("A", Int) ] "B"` would get stuck. That's because `Get` is a partial function on types, and these types are computed at compile-time. It wouldn't make much sense for type checkers to crash and throw a "Non-exhaustive" error or be non-terminating.

We could make `Get` total, as we would at the term level, with `Maybe`.

```
type family Get
    (xs :: [(Symbol, *)])    -- dictionary
    (s :: Symbol)            -- key
    :: Maybe * where         -- type

    Get '[]               s = Nothing
    Get ('(s, x) ': xs) s = Just x
    Get ('(t, x) ': xs) s = Get xs s
```

Other dictionary-related functions are defined in a similar way.

```
-- inserts or updates an entry
type family Set
    (xs :: [(Symbol, *)])    -- old dictionary
    (s :: Symbol)            -- key
    (x :: *)                 -- type
    :: [(Symbol, *)] where   -- new dictionary

    Set '[]              s x = '[ '(s, x) ]
    Set ('(s, y) ': xs) s x = ('(s, x) ': xs)
    Set ('(t, y) ': xs) s x =
        '(t, y) ': (Set xs s x)

-- removes an entry
type family Del
    (xs :: [(Symbol, *)])    -- old dictionary
    (s :: Symbol)            -- key
    :: [(Symbol, *)] where   -- new dictionary

    Del '[] s              = '[]
    Del ('(s, y) ': xs) s = xs
    Del ('(t, y) ': xs) s = '(t, y) ': (Del xs s)

-- membership
type family Member
    (xs :: [(Symbol, *)])    -- dictionary
    (s :: Symbol)            -- key
    :: Bool where            -- exists?

    Member '[]              s = False
    Member ('(s, x) ': xs) s = True
    Member ('(t, x) ': xs) s = Member xs s
```

4.3 Proxies and singleton types

Now we could annotate the effects of a command in types. `DEL` removes a key from the current database, regardless of its type.

```
del :: KnownSymbol s
    => Proxy s
    -> Edis xs (Del xs s) (Either Reply Integer)
del key = Edis $ Hedis.del (symbolVal key)
```

`KnownSymbol` is a class that gives the string associated with a concrete type-level symbol, which can be retreived with `symbolVal`.[6]

Since Haskell has a *phase distinction*[9], types are erased before runtime. It's impossible to obtain informations directly from types, we can only do this indirectly, with *singleton types* [4].

A singleton type is a type that has only one instance, and the instance can be think of as the representative of the type at the realm of runtime values.

`Proxy`, as its name would suggest, can be used as singletons. It's a phantom type that could be indexed with any type.

```
data Proxy t = Proxy
```

In the type of `del`, the type varabile `s` would be a `Symbol` that is decided by the argument of type `Proxy s`.

[6] They are defined in `GHC.TypeLits`.

To use `del`, we would have to apply it with a clumsy term-level proxy like this:

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

5. Making Redis polymorphic

Redis supports many different datatypes, these datatypes as can be viewed as *containers* of strings. For example, lists (of strings), sets (of strings), and strings themselves.

6. Denoting containers

Most Redis commands only work with a certain type of these containers. To annotate what container a key is associated with, we introduce these container types.

```
data Strings
data Lists
data Sets
...
```

`SET` stores a string, regardless the datatype the key was associated with. Now we could implement `SET` like this:

```
set :: KnownSymbol s
    => Proxy s
    -> ByteString       -- data to store
    -> Edis xs (Set xs s Strings) (Either Reply Status)
set key val = Edis $ Hedis.set (symbolVal key) val
```

After `SET`, the key will be associated with `Strings` in the dictionary, indicating that it's a string.

6.1 Automatic data serialization

But in the real world, raw binary strings are hardly useful, people would usually serialize their data into strings before storing them, and deserialize them back when in need.

Instead of letting users writing these boilerplates, we will be doing these serializations/deserializations for them. With the help from `cereal`, a binary serialization library. `cereal` comes with these two functions:

```
encode :: Serialize a
       => a -> ByteString Source

decode :: Serialize a
       => ByteString -> Either String a
```

Which would do all the works for us, as long as the datatype it's handling is an instance of class `Serialize`.[7]

6.2 Extending container types

We rename container types and extend it with an extra type arguement, to indicate what kind of data it's holding.

```
data StringOf x
data ListOf x
data SetOf x
...
```

`set` reimplemented with extended container types:

```
set :: (KnownSymbol s, Serialize x)
    => Proxy s
    -> x       -- can be anything, as long as it's serializable
    -> Edis xs (Set xs s (StringOf x)) (Either Reply Status)
set key val = Edis $ Hedis.set (symbolVal key) (encode val)
```

For example, if we execute `set (Proxy :: Proxy "A") True`, a new entry `'("A", StringOf Bool)` will be inserted to the dictionary.

6.3 Imposing constraints

To rule out programs with undesired properties, certain constraints must be imposed, on what arguments they can take, and what preconditions they must hold. These constraints are best encoded in types.

6.4 Redis datatypes

Redis supports many different datatypes, and most commands only work with certain types of them. For example: `SET` with strings or `SADD` with sets.

7. Discussions

8. Conclusions and Related Work

[7] The methods of `Serialize` will have default generic implementations for all datatypes with some language extenstions enabled, no sweat!

References

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

[2] M. M. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *ACM SIGPLAN Notices*, volume 40, pages 241–253. ACM, 2005.

[3] M. M. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *ACM SIGPLAN Notices*, volume 40, pages 1–13. ACM, 2005.

[4] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. *ACM SIGPLAN Notices*, 47(12):117–130, 2013.

[5] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. *ACM SIGPLAN Notices*, 49(1):671–683, 2014.

[6] D. Gratzer. Types and kinds and sorts, oh my! Blog, Febuary 2014. URL http://jozefg.bitbucket.org/posts/2014-02-10-types-kinds-and-sorts.html.

[7] O. Kiselyov and C.-c. Shan. Position: Lightweight static resources: Sexy types for embedded and systems programming. In *In TFP'07, the 8 th Symposium on Trends in Functional Programming*. Citeseer, 2007.

[8] O. Kiselyov, S. P. Jones, and C.-c. Shan. Fun with type functions. In *Reflections on the Work of CAR Hoare*, pages 301–331. Springer, 2010.

[9] S. Lindley and C. McBride. Hasochism: the pleasure and pain of dependently typed haskell programming. *ACM SIGPLAN Notices*, 48(12):81–92, 2014.

[10] C. McBride. Functional pearl: Kleisli arrows of outrageous fortune. *Journal of Functional Programming (to appear)*, 2011.

[11] D. Orchard and T. Schrijvers. Haskell type constraints unleashed. In *Functional and Logic Programming*, pages 56–71. Springer, 2010.

[12] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. *ACM Sigplan Notices*, 43(9):51–62, 2008.

[13] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.