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. People, however, 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 demonstrate how to prevent those runtime errors at compile time, thus 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 *Popcorn*[1]on *Hackage*.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Specification techniques; H.2.3 [*Database Management*]: Languages

Keywords Haskell; type families; type-level literals; indexed monad; Redis; database query language

[1] Popcorn, a temporary codename for the purposes of double-blind reviewing

1. Introduction

1.1 Redis

Redis[2] is an open source, in-memory data structure store, often used as database, cache and message broker. A Redis datatype can be think of as a set of key-value pairs, where 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, etc, and provides a collection of of atomic *commands* to manipulate these values.

For an example, consider the following sequence of commands, entered through the interactive interface of Redis. The keys `some-set` abd `another-set` are both associated to a set. The two call to command `SADD` respectively adds three and two values 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"
```

Note that the keys `some-set` and `another-set`, if not existing before the call to `SADD`, are created on site. The call to `SADD` returns the size of the set after completion of the command.

1.2 Hedis

Many third party library exist to allow general purpose programmings to access Redis databases through its TCP protocol. The most popular such library for Haskell is Hedis[3].

The following program is how the previous example looks like in Hedis:

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

[2] https://redis.io

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

```
    sadd "another-set" ["a", "b", "c"]
    sinter ["some-set", "another-set"]
```

The function sadd takes a key and a list of values as arguments, and returns an Integer on success, or returns a Reply, a low-level representation of replies from the Redis server, in case of failures. All wrapped in Redis[4], the context of command execution.

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

Note that keys and values, being nothing but binary strings in Redis, are represented using Haskell ByteString. Values of other types must be encoded as ByteStrings before being written to the database, and decoded after being read back.

2. Motivation

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

Although everything in Redis is essentially a binary string, these strings are treated differently. Redis supports many different kind of data structures, such as strings, hashes, lists, etc. While they are all encoded as binary strings before being written to the databse, most commands, much like how the C language treats a piece of data, only works with data of certain types.

Problem 1 The command SET, by its definition, associates a key to a string. In the following example, the key some-string is associated to string foo. Subsequent calls to SADD causes runtime errors, since the value of some-string is not a set, but a string.

```
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! The call INCR some-string parses the string associated with key some-string to an integer, increments it by one, and store it back as a string. If the string can not be parse as an integer, a runtime error is raised.

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

[4] Hedis provides another kind of context, RedisTx, for *transactions*, united with Redis under the class of RedisCtx. For brevity, we demonstrate only Redis in this paper.

In Hedis Hedis, being only a simple wrapper on top of the TCP protocol of Redis, inherits all the problems mentioned above. The following program yields the same error as that in the Redis client.

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

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

The Cause Every key is associated with a value, and every value has it's own type. But most commands in Redis only work with a certain type of value. When a command is used on a wrong type of key, a runtime error occurs. The problems illustrated above arise from the absence of type checking, with respects to **the type of a value that associates with a key**. These problems could have been avoided, if we could know the type every key associates with in advance, and prevent programs with invalid commands from executing.

2.1 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 peculiar 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 type-checked embedded DSL, and implemented a runtime type checker that keeps track of types of all the variable. But then we found that things can be a lot easier, by leveraging the host language's type checker. We encode variable bindings with *type-level lists* and *strings*, and control the effects on the bindings with *indexed monad*. In contrast to the former approach, we **embedded** our type checker into Haskell's type system, without having to build a **standalone** one on the term level.

2.2 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[14].
- Popcorn, a package we built for programmers. This package helps programmers to write more reliable

Redis programs, and also makes Redis polymorphic by automatically converting back and forth from values of arbitrary types and boring ByteStrings.

3. Type-level dictionaries

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

3.1 Datatype promotion

Normally, at the term level, we could express the datatype of dictionary with *type synonym* like this.[5]

```
type Key = String
type Dictionary = [(Key, TypeRep)]
```

To encode this in the type level, everything has to be *promoted*[18] one level up. From terms to types, and from types to kinds.

Luckily, with recently added GHC extension *data kinds*, suitable datatype will be automatically promoted to be a kind, and its value constructors to be type constructors. The following type `List`

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

Give rise to the following kinds and type constructors:[6] [7]

```
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. We could also express promoted lists and tuples in types like this with a single quote prefixed. For example: `'[Int, Char]`, `'(Int, Char)`.

3.2 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
```

[5] `TypeRep` supports term-level representations of datatypes, available in `Data.Typeable`

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

[7] All kinds have *sort* BOX in Haskell[7]

3.3 Putting everything together

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

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

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

The kind `*` (pronounced "star") stands for the set of all concrete type expressions, such as `Int`, `Char` or even a symbol `"symbol"`, while `Symbol` is restricted to all symbols only.

4. 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[9, 11] to model such preconditions and postconditions in types. An indexed monad is a type constructor that takes three arguments: an initial state, a final state, and the type of a value that an action computes, which can be read like a Hoare triple[13].

```
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 monads.

We define a new datatype `Popcorn`, which is basically just the context `Redis` indexed with more information in types. We make it an instance of `IMonad`.

```
newtype Popcorn p q a =
    Popcorn { unPopcorn :: Redis a }

instance IMonad Popcorn where
    unit = Popcorn . return
    bind m f =
        Popcorn (unPopcorn m >>= unPopcorn . f )
```

The first and second argument of type `Popcorn` is where the dictionaries going to be.

To execute a `Popcorn` program, simply apply it to `unPopcorn` to get an ordinary Hedis program back, with type information erased.

4.1 `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 with Popcorn, we could make our own version of ping[8]

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

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

5. Type-level functions

5.1 Closed type families

Type families have a wide variety of applications. They can appear inside type classes[3, 4], or at toplevel. Toplevel type families can be used to compute over types, they come in two forms: open[16] and closed [6].

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

5.2 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 benefit 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 a type checker 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 fashion.

```
-- 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
```

[8] ping from Hedis is qualified with Hedis to prevent function name clashing in our code.

5.3 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
    -> Popcorn xs (Del xs s) (Either Reply Integer)
del key = Popcorn $ Hedis.del (encodeKey key)
```

`KnownSymbol` is a class that gives the string associated with a concrete type-level symbol, which can be retrieved with `symbolVal`.[9]Where `encodeKey` converts `Proxy s` to `ByteString`.

```
encodeKey :: KnownSymbol s => Proxy s -> ByteString
encodeKey = encode . symbolVal
```

Since Haskell has a *phase distinction, phasedistinction*, types are erased before runtime. It's impossible to obtain information directly from types, we can only do this indirectly, with *singleton types, singletons*.

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 variable `s` is a `Symbol` that is decided by the argument of type `Proxy s`. To use `del`, we would have to apply it with a clumsy term-level proxy like this:

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

6. 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.

6.1 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 types for the universe of containers.

```
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
    -> Popcorn xs (Set xs s Strings) (Either Reply Status)
set key val = Popcorn $ Hedis.set (encodeKey key) val
```

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

6.2 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 can do 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`.[10]

6.3 Extending container types

We rename container types and extend it with an extra type argument, to indicate what kind of encoded value 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
    -> Popcorn xs (Set xs s (StringOf x)) (Either Reply Status)
set key val = Popcorn $ Hedis.set (encodeKey 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.

6.4 Handling `INCR`

Commands such as `INCR` and `INCRBYFLOAT`, are not only container-specific, they also have some requirements on what types of value they could operate with.

We could handle this by mapping Redis's strings of integers and floats to Haskell's `Integer` and `Double`.

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

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

```
incr :: (KnownSymbol s
       , Get xs s ~ Just (StringOf Integer))
      => Proxy s
      -> Popcorn xs xs (Either Reply Integer)
incr key = Popcorn $ Hedis.incr (encodeKey key)

incrbyfloat :: (KnownSymbol s
      , Get xs s ~ Just (StringOf Double))
     => Proxy s
     -> Double
     -> Popcorn xs xs (Either Reply Double)
incrbyfloat key n =
    Popcorn $ Hedis.incrbyfloat (encodeKey key) n
```

7. Imposing constraints

To rule out programs with undesired properties, certain constraints must be imposed, on what arguments they can take, or what preconditions they must hold.

Consider the following example: `LLEN` returns the length of the list associated with a key, else raises a type error.

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

Such constraint could be expressed in types with *equality constraints*[17].

```
llen :: (KnownSymbol s
       , Get xs s ~ Just (ListOf x))
      => Proxy s
      -> Popcorn xs xs (Either Reply Integer)
llen key =
      Popcorn $ Hedis.llen (encodeKey key)
```

Where (`~`) denotes that `Get xs s` and `Just (ListOf x)` needs to be the same.

The semantics of `LLEN` defined above is actually not complete. `LLEN` also accepts keys that do not exist, and replies with `0`.

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

In other words, we require that the key to be associated with a list, **unless** it doesn't exist at all.

7.1 Expressing constraint disjunctions

Unfortunately, expressing disjunctions in constraints is much more difficult than expressing conjunctions, since the latter could be easily done by placing constraints in a tuple (at the left side of `=>`).

There are at least three ways to express type-level constraints [5]. Luckily we could express constraint disjunctions with type families in a modular way.

The semantics we want could be expressed informally like this:

`Get xs s` $\equiv$ `Just (ListOf x)` $\vee \neg$ `(Member xs s)`

We could achieve this simply by translating the semantics we want to the domain of Boolean, with type-level boolean functions such as (`&&`), (`||`), `Not`, (`==`), etc.[11]To avoid

```
Get xs s == Just (ListOf x) || Not (Member xs s)
```

To avoid addressing the type of value (as it may not exist at all), we defined an auxiliary predicate `IsList :: Maybe * -> Bool` to replace the former part.

```
IsList (Get xs s) || Not (Member xs s)
```

The type expression above has kind `Bool`, we could make it a type constraint by asserting equality.

```
(IsList (Get xs s) || Not (Member xs s)) ~ True
```

With *constraint kind*, a recent addition to GHC, type constraints now has its own kind: `Constraint`. That means type constraints are not restricted to the left side of a `=>` anymore, they could appear in anywhere that accepts something of kind `Constraint`, and any type that has kind `Constraint` can also be used as a type constraint. [12]

As many other list-related commands also have this "List or nothing" semantics, we could abstract the lengthy type constraint above and give it an alias with type synonym.

```
ListOrNX xs s =
    (IsList (Get xs s) || Not (Member xs s)) ~ True
```

The complete implementation of `LLEN` with `ListOrNX` would become:

```
llen :: (KnownSymbol s, ListOrNX xs s)
        => Proxy s
        -> Popcorn xs xs (Either Reply Integer)
llen key = Popcorn $ Hedis.llen (encodeKey key)
```

8. Assertions

Users may need to make assertions about the status of some key-type bindings in a Redis program. Specifically, when declaring or renouncing the existence of a key and the type of its associating value. We provide these functions, which do nothing but fiddle with types.

[11] Available in `Data.Type.Bool` and `Data.Type.Equality`

[12] See https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/constraint-kind.html.

```
declare :: (KnownSymbol s, Member xs s ~ False)
        => Proxy s
        -> Proxy x  -- type of value
        -> Popcorn xs (Set xs s x) ()
declare s x = Popcorn $ return ()

renounce :: (KnownSymbol s, Member xs s ~ True)
        => Proxy s -> Popcorn xs (Del xs s) ()
renounce s = Popcorn $ return ()

-- to be used at the beginning of a program
start :: Popcorn '[] '[] ()
start = Popcorn $ return ()
```

8.1 A complete example

The following program increases the value of "A" as an integer, push the result of the increment to list "L", and then pops it out.

```
main :: IO ()
main = do
    conn <- connect defaultConnectInfo
    result <- runRedis conn $ unPopcorn $ start
        `bind` \_ ->  declare
                        (Proxy :: Proxy "A")
                        (Proxy :: Proxy Integer)
        `bind` \_ ->  incr      (Proxy :: Proxy "A")
        `bind` \n ->  case n of
            Left  err -> lpush (Proxy :: Proxy "L") 0
            Right n   -> lpush (Proxy :: Proxy "L") n
        `bind` \_ ->  lpop      (Proxy :: Proxy "L")
    print result
```

The syntax is pretty heavy, like the old days when there's no *do-notation*[8]. But if we don't need any variable bindings between operations, we could compose these commands with a sequencing operator (>>>).

```
(>>>) :: IMonad m => m p q a -> m q r b -> m p r b

program = start
    >>> declare
            (Proxy :: Proxy "A")
            (Proxy :: Proxy Integer)
    >>> incr    (Proxy :: Proxy "A")
    >>> lpush   (Proxy :: Proxy "L") 0
    >>> lpop    (Proxy :: Proxy "L")
```

9. Discussions

Syntax No one could ignore the glaring shortcoming of the syntax, which occurs mainly in two places: *symbol singletons* and *indexed monad*. We are hoping that these issues could be resolved with future syntactic extensions.

Returns only determined datatypes All other data structures in Redis also follow the semantics similar to "List or nothing". Take the case of GET, which also shares a "String or nothing" semantics that should be typed:

```
get :: (KnownSymbol s, Serialize x, StringOrNX xs s)
    => Proxy s -> Popcorn xs xs (Either Reply (Maybe x))
```

Since the key may not exist, we don't know what x would be. We could left x ambiguous, and let it be decided by the caller. But users will then be forced to spell out the complete type signature of everything, including the dictionaries, only to specify the desired resulting type.

Instead of allowing the key to be non-existent, we require that the key must exist and it's associating type to be determined at compile time. So our version of get has a stricter semantics:

```
get :: (KnownSymbol s, Serialize x
      , Just (StringOf x) ~ Get xs s)
    => Proxy s -> Popcorn xs xs (Either Reply (Maybe x))
```

Commands with multiple inputs or outputs Some command may take a variable number of arguments as inputs, and returns more than one value as outputs. To illustrate this, consider sinter in Hedis:

```
Hedis.sinter :: [ByteString]                        -- keys
             -> Redis (Either Reply [ByteString]) -- values
```

In Hedis such command could easily be expressed with lists of ByteStrings. But in Popcorn, things escalate quickly, as the keys and values will have to be expressed with *heterogeneous lists*[10], which would be pratically infeasible, considering the cost, if not impossible.

Most importantly, the keys will all have to be constrained by KnownSymbol, which enforces these type literals to be concrete and known at compile time. It's still unclear whether this is possible.

So instead, we are offering commands that only has a single input and output.

```
sinter :: ByteString                      -- single key
       -> Redis (Either Reply ByteString) -- single value
```

Not all Redis programs can be typechecked (even if they might turn out to be type safe). We opted for type safety rather than expressiveness.

Redis Transactions Redis has *transactions*, another context for executing commands. Redis transactions are atomic in the sense that, all commands in a transaction will executed sequentially, and no other requests issued by other clients will be served **in the middle.**[13] In contrast, we cannot make such a guarantee in the ordinary context, which may destroy the assertions we made in types.

At this point of writing, transactions are not supported in our implementation. We are planning to add it in the future, and we are expecting that there wouldn't be much difficulty, since we've implemented a runtime type checker specifically targeting Redis transactions once, before we moved on to the types.

10. Related Work

While Redis can be viewed as a non-relational database system (although Redis seems reluctant to admit this in recent years), there also are similar goals on relational database systems, trying to achieve a safer database interface by making them statically checked. *HaskellDB*[2, 12] expresses quires with relational algebra-like combinators, wrapped in phantom types. Or as examples, at first to demonstrate the power of dependent types in Agda[15], and then with singletons in Haskell[5].

11. Conclusions

We have demonstrated how a DSL with variable bindings could be embedded in types. With more and more extensions added to the language, Haskell is gradually becoming a dependently-typed language,[14] but it's not that dreadful as many (including us) would have thought.

Acknowledgements [15]

[14] Why not be dependently typed? `http://stackoverflow.com/questions/12961651/why-not-be-dependently-typed`

[15] Hidden for the purposes of double-blind reviewing

References

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

[2] B. Bringert, A. Höckersten, C. Andersson, M. Andersson, M. Bergman, V. Blomqvist, and T. Martin. Student paper: Haskelldb improved. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 108–115. ACM, 2004.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.

[9] 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.

[10] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM, 2004.

[11] 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.

[12] D. Leijen and E. Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.

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

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

[15] N. Oury and W. Swierstra. The power of pi. In *ACM Sigplan Notices*, volume 43, pages 39–50. ACM, 2008.

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

[17] M. Sulzmann, M. M. Chakravarty, S. P. Jones, and K. Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.

[18] 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.