1. Hash tables and hash functions

1.1. Hash tables

1.1.1. Constructors

`(make-eq-hash-table)`
`(make-eq-hash-table` *k*`)`

Returns a newly allocated hash table that accepts arbitrary objects as keys, and compares those keys with eq? If an argument is given, then the initial capacity of the hash table is set to *k* elements.

`(make-eqv-hash-table)`
`(make-eqv-hash-table` *k*`)`

Returns a newly allocated hash table that accepts arbitrary objects as keys, and compares those keys with eqv? If an argument is given, then the initial capacity of the hash table is set to *k* elements.

`(make-hash-table` *procedure*$_1$ *procedure*$_2$`)`
`(make-hash-table` *procedure*$_1$ *procedure*$_2$ *k*`)`

Returns a newly allocated mutable hash table using *procedure*$_1$ as the hash function and *procedure*$_2$ as the procedure used to compare keys. The hash function must accept a key and return a non-negative exact integer. If a third argument is given, then the initial capacity of the hash table is set to *k* elements.

1.1.2. Procedures

`(hash-table?` *hash-table*`)`
Returns `#t` if *hash-table* was created by one of the hash table constructors, otherwise returns `#f`.

`(hash-table-size` *hash-table*`)`
Returns the number of keys contained in *hash-table* as an exact integer.

`(hash-table-ref` *hash-table key*`)`
Returns the value in the hash-table associated with *key*. If the hash table does not contain *key*, an exception is raised. [TODO: hash table exception types]

`(hash-table-ref/default` *hash-table key default*`)`

Returns the value in the hash-table associated with the *key*. If the hash table does not contain *key*, returns *default*.

`(hash-table-ref/call` *hash-table key f*`)`
Returns the value associated with key in the hash-table if the hash table contains key; otherwise tail-calls f on key.

`(hash-table-ref/thunk` *hash-table key thunk*`)`
Returns the value associated with key in the hash-table if the hash table contains key; otherwise tail-calls thunk.

`(hash-table-get` *hash-table key*`)`
Equivalent to

```
(hash-table-ref/default hash-table key #f)
```

`(hash-table-set!` *hash-table key value*`)`
Changes the hash-table to associate key with value, replacing any existing association for key. Returns the unspecified value.

`(hash-table-delete!` *hash-table key*`)`
Removes any association for key within the hash-table.

`(hash-table-contains?` *hash-table key*`)`
Returns true iff the hash table contains an entry for key.

`(hash-table-update!` *hash-table procedure*`)`
Equivalent to:

```
(hash-table-set!
 hash-table key
 (procedure (hash-table-ref hash-table key)))
```

Raises an exception if the hash table does not contain an entry for key.

`(hash-table-update!/default` *hash-table procedure default*`)`

Equivalent to, but potentially more efficient than:

```
(hash-table-set!
 hash-table key
 (procedure (hash-table-ref/default
             hash-table key default)))
```

(`hash-table-update!/thunk` *hash-table procedure thunk*)

Equivalent to, but potentially more efficient than:

```
(hash-table-set!
 hash-table key
 (procedure (hash-table-ref/thunk
             hash-table key thunk)))
```

(`hash-table-update!/call` *hash-table procedure f*)

Equivalent to, but potentially more efficient than:

```
(hash-table-set!
 hash-table key
 (procedure (hash-table-ref/call
             hash-table key f)))
```

(`hash-table-fold` *hash-table procedure init*)
For every association in hash-table, calls procedure with three arguments: the association key, the association value, and an accumulated value. The accumulated value is init for the first invocation of procedure, and for subsequent invocations of procedure, the return value of the previous invocation of procedure. The return value of hash-table-fold is the value of the last invocation of procedure.

(`hash-table-copy` *hash-table*)
Returns a copy of hash-table.

(`hash-table-clear!` *hash-table*)
Removes all associations from hash-table. Returns the unspecified value.

(`hash-table-for-each` *procedure hash-table*)
The procedure must accept two arguments, a key and the value associated with that key. The procedure is called once for each key-value association in hash-table. The order of these calls is indeterminate. Equivalent to:

```
(hash-table-fold hash-table
                 (lambda (k v a)
                   (procedure k v))
                 (unspecified))
```

(`hash-table->alist` *hash-table*)
Equivalent to:

```
(hash-table-fold hash-table
                 (lambda (k v a)
                   (cons (cons k v) a))
                 '())
```

(`hash-table-keys` *hash-table*)
Equivalent to:

```
(hash-table-fold hash-table
                 (lambda (k v a) (cons k a))
                 '())
```

(`hash-table-values` *hash-table*)
Equivalent to:

```
(hash-table-fold hash-table
                 (lambda (k v a) (cons v a))
                 '())
```

1.1.3. Reflection

(`hash-table-equivalence-predicate` *hash-table*)

Returns the equivalence predicate used by *hash-table* to compare keys. For hash tables created with `make-eq-hash-table` and `make-eqv-hash-table`, returns `eq?` and `eqv?` respectively.

`(hash-table-hash-function` *hash-table*`)`
Returns the hash function used by hash-table. For hash tables created by `make-eq-hash-table` or `make-eqv-hash-table`, the unspecified value is returned.

`(hash-table-mutable?` *hash-table*`)`
Returns mutability status of hash-table.

1.2. Hash functions

`(equal-hash` *obj*`)`
Returns an integer hash value for *obj*, based on its contents.

`(string-hash` *string*`)`
Returns an integer hash value for *string*, based on its contents.

`(string-ci-hash` *string*`)`
Returns an integer hash value for *string* based on its contents, ignoring case.

`(symbol-hash` *string*`)`
Returns an integer hash value for symbol, based on its print name.

1.3. Issues

1.3.1. Limit proposal scope

Should the specification be limited to eq and eqv hash tables, since those are the only kind which may not be implementable as a portable library?

1.3.2. Immutability

The proposal should probably support immutable hash tables, which can be constructed from a provided assoc list. The appropriate constructor(s) need to be defined, supporting the various hash table options, preferably without doubling the number of constructors.

1.3.3. Complexity

It may be appropriate to specify constraints on complexity, such as constant time for hash-table-size, or an appropriate constraint on accessor procedures.

1.3.4. Concurrency

R6RS does not deal with concurrency. Even if this proposal does not say anything about that, the issue should be considered. Any implementation that supports concurrency will have to implement some kind of mutual exclusion for operations that have side effects, and some will need mutual exclusion even for `hash-table-ref`. As specified, the updating operations are not atomic, so they create no new problems. The `hash-table-fold` and `hash-table-for-each` procedures already have a problem, even without concurrency.

1.3.5. Omission of procedures

Some procedures could perhaps be omitted, e.g. there are four retrieval procedures. Which one(s) should be omitted, if any? Procedures which take failure thunks are candidates for omission.

1.3.6. Side-effects

There is a potential problem with the higher-order procedures, if a procedure argument mutates the hash table being operated on. This should be addressed somehow, if only by a statement that the behavior caused by such procedures is unspecified.

1.3.7. hash-table-map

This has been omitted because a single appropriate specification is not obvious, and any reasonable specification can easily be implemented in terms of `hash-table-fold`.

1.3.8. Names

Should the type name be `hash-table`, `hashtable`, or something else?