15-210 Library Documentation

Draft Last Compiled: 23rd September 2011

Contents

Part I

Behavioural Specifications

Chapter 1

Conventions

For concision, the specifications in this document are typically phrased as implications so that they only apply to expressions that terminate. It is not the case that all expressions terminate. If a higher order function is applied to arguments including a function and input for that function on which it does not terminate, then the application will likely not terminate.

The specifications in this document often contain data type definitions and code fragments in SML-like syntax. These fragments exist to clearly and carefully specify the behaviour of the value being documented, not to restrict its possible implementations. Structures do *not* need to implement the signature with exactly these code fragments; they are typically extremely inefficient. They also contain a mixture of SML syntax and notation from the abstractions in play, and therefore do not immediately compile.

Two functions f and g are said to be *logically equivalent* if they give equal results on equal arguments. It follows that if f is logically equivalent to g, and f does not terminate on some input, then g does not terminate on that input.

We use math and monospace fonts to indicate the difference between the abstraction implemented by an abstract type and SML representations of that abstract notion, respectively. If the same letter or identifier appears in both fonts, it should be understood to mean either the abstraction or representation of the same object.

Types are always typeset in a math font for readability, corresponding to the typical pronunciation of concrete SML syntax. For example, the type of a function

```
f : 'a -> ('a * 'b) -> 'c
```

will be written

$$\texttt{f} : \alpha \to \alpha * \beta \to \gamma$$

When talking about a polymorphic expressions in prose, we will often omit the type variable that they are polymorphic over. For example, we may say that "`l` is a list" to mean "`l` has type α `list`".

The specifications often use the phrase "`x` is a t value", where t is some type. For example, when we say "`i` is an `int` value" we mean that `i` is an expression with type `int` that cannot be evaluated further, like 7 but not like `((fn x => x) 7)`.

Combining combing the above two conventions, the statement

> "`s` is a sequence value"

should be taken to mean

> "`s` has type α `seq`, and `s` is a value"

or, more specifically,

> "`s` has type α `seq`, and for every valid index i into s, s_i is a value"

Chapter 2

SEQUENCE Signature

2.1 Overview

2.1.1 Abstract Sequences

We define an abstract mathematical notion of a sequence. The documentation for the signature that follows states the behaviour of implementations in terms of this abstraction.

- A *sequence* is an ordered finite list of elements of some type, indexed by the natural numbers.
- The *length* of a sequence is the number of elements in that sequence. If s is a sequence, its length is denoted $|s|$.
- A natural number i is said to be a *valid index* into the sequence s if and only if $0 \leq i < |s|$.
- If s is any seqence and i is a valid index into s, then s_i denotes the i^{th} element of s.
- If s is a particular sequence with n elements, we may denote s with the notation

  $$\langle s_0, s_1, \ldots, s_{n-1} \rangle$$

  For example,

  $$\langle \rangle$$

  denotes the empty sequence and

  $$\langle 4, 2, 3 \rangle$$

  denotes a particular sequence of natural numbers with length three.

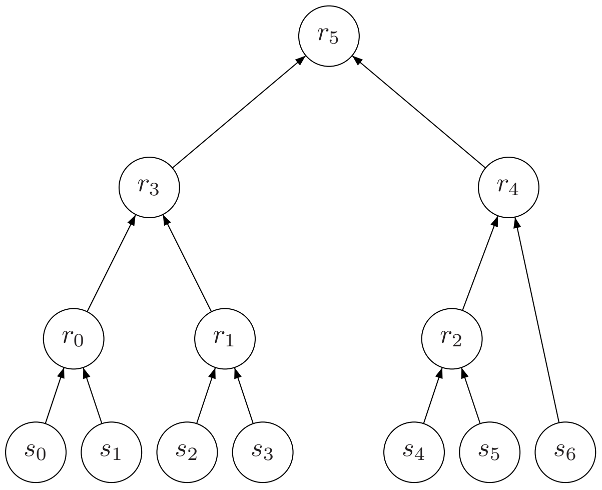

Figure 2.1: Reduce tree structure for any sequence s with $|s| = 7$

- A sequence s' is said to be a *subsequence* of a sequence s if there is a strictly increasing, possibly empty, sequence I of valid indices into s such that $s'_i = s_{I_i}$.
- Sequences can only be ordered if their elements can be ordered. If that condition is met, sequences are ordered lexicographically.

2.2 Signature Definition

```
signature SEQUENCE =
sig
  type 'a seq
  datatype 'a treeview = EMPTY
                       | ELT of 'a
                       | NODE of ('a seq * 'a seq)
  datatype 'a listview = NIL
                       | CONS of ('a * 'a seq)
  type 'a ord = 'a * 'a -> order
  exception Range
  val empty : unit -> 'a seq
  val singleton : 'a -> 'a seq
  val length : 'a seq -> int
  val nth : 'a seq -> int -> 'a
  val tabulate : (int -> 'a) -> int -> 'a seq
  val fromList : 'a list -> 'a seq
  val collate : 'a ord -> 'a seq ord
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val map2 : (('a * 'b) -> 'c) -> 'a seq -> 'b seq -> 'c seq
  val reduce : (('a * 'a) -> 'a) -> 'a -> 'a seq -> 'a
  val scan : (('a * 'a) -> 'a) -> 'a -> 'a seq -> ('a seq * 'a)
  val filter : ('a -> bool) -> 'a seq -> 'a seq
  val iter : ('b * 'a -> 'b) -> 'b -> 'a seq -> 'b
  val iterh : ('b * 'a -> 'b) -> 'b -> 'a seq -> ('b seq * 'b)
  val flatten : 'a seq seq -> 'a seq
  val partition : int seq -> 'a seq -> 'a seq seq
  val inject : (int*'a) seq -> 'a seq -> 'a seq
  val append : 'a seq * 'a seq -> 'a seq
  val take : 'a seq * int -> 'a seq
  val drop : 'a seq * int -> 'a seq
  val rake : 'a seq -> (int * int * int) -> 'a seq
  val subseq : 'a seq -> (int * int) -> 'a seq
  val splitMid : 'a seq * int -> ('a seq * 'a * 'a seq) option
  val sort : 'a ord -> 'a seq -> 'a seq
  val merge : 'a ord -> 'a seq -> 'a seq -> 'a seq
  val collect : 'a ord -> ('a * 'b) seq -> ('a * 'b seq) seq
  val toString : ('a -> string) -> 'a seq -> string
  val tokens : (char -> bool) -> string -> string seq
  val fields : (char -> bool) -> string -> string seq
  val showt : 'a seq -> 'a treeview
  val showti : 'a seq -> (int -> int) -> 'a treeview
  val hidet : 'a treeview -> 'a seq
  val showl : 'a seq -> 'a listview
  val hidel : 'a listview -> 'a seq
  val % : 'a list -> 'a seq
end
```

2.3 Details of Types

2.3.1 α *seq*

This is the abstract type that represents the notion of a sequence described in section 2.1.1.

2.3.2 α *treeview*

α *treeview* provides a view of the abstract α *seq* type as a binary tree.

2.3.3 α *listview*

α *listiew* provides a view of the abstract α *seq* type as a list.

2.3.4 α *ord*

The type α *ord* represents an ordering on the type α as a function from pairs of elements of α to *order*.

2.4 Details of Exceptions

2.4.1 `Range`

`Range` is raised whenever an invalid index into a sequence is used. The specifications for the individual functions state when this will happen more precisely.

This is the only exception that the functions in a module ascribing to `SEQUENCE` raise. An expression applying such a function to appropriate arguments may raise other exceptions, but it will do so only because one of the arguments in that application raised the other exception.

2.5 Details of Values

2.5.1 `empty:` $unit \rightarrow \alpha\ seq$

`(empty ())` evaluates to $\langle\rangle$.

2.5.2 `singleton:` $\alpha \rightarrow \alpha\ seq$

If `x` is a value, then `(singleton x)` evaluates to $\langle x\rangle$.

2.5.3 `length:` $\alpha\ seq \rightarrow int$

If `s` is a sequence value, then `(length s)` evaluates to $|s|$.

2.5.4 `nth:` $\alpha\ seq \rightarrow int \rightarrow \alpha$

If `s` is a sequence value and `i` is an `int` value and i is a valid index into s, then `(nth s i)` evaluates to s_i.

This application raises `Range` if i is not a valid index.

2.5.5 `tabulate:` $(int \rightarrow \alpha) \rightarrow int \rightarrow \alpha\ seq$

If `f` is a function and `n` is an `int` value, then `(tabulate f n)` evaluates to a sequence s such that $|s| = n$ and, for all valid indicies i into s, s_i is the result of evaluating `(f i)`.

Note that the evaluation of this application will only terminate if f terminates on all valid indices into the result sequence s.

2.5.6 `fromList`: $\alpha\ list \rightarrow \alpha\ seq$

If `l` is a list value, then `(fromList l)` evaluates to the index preserving sequence representation of l. That is to say, `fromList` is logically equivalent to

```
fn l => tabulate (fn i => List.nth(l,i)) (List.length l)
```

2.5.7 `collate`: $\alpha\ ord \rightarrow \alpha\ seq\ ord$

If `ord` is an ordering on the type α, `collate ord` evaluates to an ordering on the type $\alpha\ seq$ derived lexicographically from `ord`.

2.5.8 `map`: $(\alpha \rightarrow \beta) \rightarrow \alpha\ seq \rightarrow \beta\ seq$

If `f` is a function and `s` is a sequence value such that $|s| = n$, then `(map f s)` evaluates to the sequence r such that $|r| = n$ and, for all valid indicies i into s, r_i is the result of evaluating `(f` s_i`)`.

Note that the evaluation of this application will only terminate if `f` terminates on s_i for all valid indicies i.

2.5.9 `map2`: $((\alpha \times \beta) \rightarrow \gamma) \rightarrow \alpha\ seq \rightarrow \beta\ seq \rightarrow \gamma\ seq$

If `f` is a function and s_1 and s_2 are sequence values, then `(map2 f` s_1 s_2`)` evaluates to the sequence r such that r_i is the result of evaluating $f\ (s_{1i}, s_{2i})$ for all i that are valid indices into both s_1 and s_2.

It follows from the definition of a valid index and the above specification that

$$|r| = \min(|s_1|, |s_2|)$$

Note that the evaluation of this application will only terminate if f terminates on (s_{1i}, s_{2i}) for all $0 \leq i < |r|$.

2.5.10 `reduce`: $((\alpha \times \alpha) \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha\ seq \rightarrow \alpha$

To define the behaviour of `reduce`, we'll first define a type of non-empty binary trees, then a mapping from non-empty sequences to those trees, then an analog to `reduce` on trees, and finally `reduce` on sequences.

The type of non-empty trees we'll use is

```
datatype 'a tree = Leaf of 'a
                 | Node of ('a tree * 'a tree)
```

Assume that `prevpow2` is a function with type `int` → `int` such that if `x` is an `int` value then `prevpow2 x` evaluates to the maximum element of the set

$$\{y | y < x \wedge \exists i \in \mathbb{N}. y = 2^i\}$$

With these two assumptions, we define a mapping from non-empty sequences to trees as

```
fun toTree s =
   case |s|
    of 1 => Leaf(s_0)
     | n => Node(toTree (take (s, prevpow2 |s|)),
                 toTree (drop (s, prevpow2 |s|)))
```

The result of this is a nearly-balanced tree where the number of leaves to the left of any internal node is the greatest power-of-two less than the total number of leaves below that node. The structure of such trees depends only on the length of the input sequence. An example tree is shown in Figure 2.1.

We'll now define the function `reducet` for the tree type. `reducet` has type

$$((\alpha \times \alpha) \to \alpha) \to \alpha \ \mathit{tree} \to \alpha$$

and is defined as

```
fun reducet f (Leaf x) = x
  | reducet f (Node(l,r)) = f(reducet l, reducet r)
```

Finally, if `f` is a function, `b` a value, and `s` a sequence value, there are two cases:

- If $|s| = 0$ then `(reduce f b s)` evaluates to `b`.
- If $|s| > 0$, and `(reducet f (toTree s))` evaluates to some value `v`, then `(reduce f b s)` evaluates to $f(b, v)$.

Note that this definition does *not* require that `f` is associative. The transformation to trees and `reduce` on trees are both well-defined without respect to any associativity of `f`. The tree structure defined by `toTree` defines a particular association of `f` on any sequence: if we use $\oplus$ as infix notation for `f`, the tree corresponds to exactly one of the many ways to parenthesize the expression

$$\left(s_0 \oplus s_1 \oplus \ldots \oplus s_{|s|-1}\right) \oplus b$$

If `f` happens to be associative, all of the possible ways to parenthesize this expression result in the same computation.

It follows that if `f` is associative and `b` is a value such that b is the identity of f, `reduce f b` is logically equivalent to `iter f b`.

2.5.11 `scan:` $((\alpha \times \alpha) \to \alpha) \to \alpha \to \alpha \ \mathit{seq} \to (\alpha \ \mathit{seq} \times \alpha)$

If `f` is an associative function, and `b` a value such that b is an identity of f, `(scan f b)` is logically equivalent to

```
fn s =>
 (tabulate (fn i => reduce f b (take(s,i))) (length s),
  reduce f b s)
```

2.5.12 filter: $(\alpha \to bool) \to \alpha\ seq \to \alpha\ seq$

If p is a predicate and s is a sequence value, then `(filter p s)` evaluates to the longest subsequence s' of s such that p holds for every element of s'.

2.5.13 iter: $(\beta \times \alpha \to \beta) \to \beta \to \alpha\ seq \to \beta$

`iter` is logically equivalent to `iterate`, defined below.

```
fun iterate f b s =
    case showl s
     of NIL => b
      | CONS (x,xs) => iter f (f(b,x)) xs
```

Less formally, if `f` is a function, `b` is a value, and `s` is a sequence value, then `(iter f b s)` computes the iteration of `f` on `s` with left-association and `b` as the base case. We can write this iteration as

$$f(f(\ldots f(b, s_0), \ldots s_{|s|-2}), s_{|s|-1})$$

or, using $\oplus$ as infix notation for f,

$$(\ldots(((b \oplus s_0) \oplus s_1) \oplus s_2) \oplus \ldots \oplus s_{|s|-1})$$

2.5.14 iterh: $(\beta \times \alpha \to \beta) \to \beta \to \alpha\ seq \to (\beta\ seq \times \beta)$

`iterh` is a generalization of `iter` that also computes the sequence of all partial results produced by the iterated application of the functional argument. Specifically, `(iterh f b)` is logically equivalent to

```
fn s => (tabulate (fn i => iter f b (take (i,s))) (|s|),
         iter f b s)
```

2.5.15 flatten: $\alpha\ seq\ seq \to \alpha\ seq$

`flatten` is logically equivalent to `(iter append (empty ()))`.

Less formally, if `s` is a sequence value of sequence values, then `(flatten s)` evaluates to the concatenation of the sequences in s in the order that they appear in s.

2.5.16 `partition:` $int\ seq \to \alpha\ seq \to \alpha\ seq\ seq$

If `I` is an `int` sequence value and `s` is a sequence value, then `(partition I s)` evauluates to a sequence of sequences `p` such that $|p| = |I|$ and, for every i that's a valid index of I, p_i is a subsequence of s of length I_i starting at index

$$\sum_{j=0}^{i-1} I_j$$

in s.

That is to say, `partition` produces a sequence of adjacent subsequence of s with lengths specified by the elements of I.

Let

$$l := \sum_i I_i$$

`partition` has the property that for any sequence s,

```
(flatten (partition I s))
```

is the first subsequence of s of length l. In particular, if $l = |s|$, this means that

```
fn ss => (partition (map length ss) (flatten ss))
```

is functionally equivalent to the identity function on $tseq\alpha$ `seq`.

2.5.17 `inject:` $(int \times \alpha)\ seq \to \alpha\ seq \to \alpha\ seq$

Let `ind` and `s` be sequence values and let

$$occ(i) := \{j | ind_j = (i, x) \text{ for some x}\}$$

`(inject ind s)` evaluates to the sequence s' with length $|s|$, where for all valid indicies i into s

$$s'_i = \begin{cases} s_i & occ(i) = \{\} \\ x & j = \max(occ(i)) \wedge ind_j = (i, x) \end{cases}$$

This application will raise `Range` if any element of ind has a first component that is not a valid index into s.

2.5.18 `append:` $\alpha\ seq \times \alpha\ seq \to \alpha\ seq$

If `s1` and `s2` are sequence values, then `(append (s1, s2))` evaluates to a sequence s with length $|s_1| + |s_2|$ such that the subsequence of s starting at index 0 with length $|s_1|$ is s_1 and the subsequence of s starting at index $|s_1|$ with length $|s_2|$ is s_2.

2.5.19 `take:` $\alpha\ seq \times int \to \alpha\ seq$

If `s` is a sequence value and `n` is an integer, then `(take (s,n))` evaluates to the first subsequence of s of length n. This application will raise `Range` if $n > |s|$.

2.5.20 `drop:` $\alpha\ seq \times int \to \alpha\ seq$

If `s` is a sequence value and `n` is an integer, then `(drop (s,n))` evaluates to the last subsequence of s of length $|s| - n$. This application will raise `Range` if $n > |s|$.

2.5.21 `subseq:` $\alpha\ seq \to (int \times int) \to \alpha\ seq$

If `s` is a sequence value and `j` and `len int` are values such that $j + len < |s|$, then `(subseq s (j, len))` evaluates to the subsequence s' of s of length len such that $s'_i = s_{i+j}$ for all `i` < `len`.

This application will raise `Range` if the subsequence specification is invalid.

2.5.22 `splitMid:` $\alpha\ seq \times int \to (\alpha\ seq \times \alpha \times \alpha\ seq)\ option$

Let s be a sequence value and i an `int` value.

- If $|s| = 0$, then `(splitMid(s,i))` evaluates to `NONE`.
- If $|s| > 0$, `(splitMid(s,i))` evaluates to `(SOME (l,`s_i`,r))` where l is the first subsequence of s of length $i - 1$ and r is the last subsequence of s of length $|s| - 1 - i$.

This application will raise `Range` if $i > |s|$.

2.5.23 `sort:` $\alpha\ ord \to \alpha\ seq \to \alpha\ seq$

If `ord` is an ordering and `s` is a sequence, `(sort ord s)` evaluates to a rearrangement of the elements of `s` that is sorted with respect to `ord`.

2.5.24 `merge:` $\alpha\ ord \to \alpha\ seq \to \alpha\ seq \to \alpha\ seq$

2.5.25 `collect:` $\alpha\ ord \to (\alpha \times \beta)\ seq \to (\alpha \times \beta\ seq)\ seq$

Let `ord` be an ordering and s be a sequence of pairs. `(collect ord s)` evaluates to a sequence of sequences where each unique first coordinate of elements of s is paired with the sequence of second coordinates of elements of s. The resultant sequence is sorted by the first coordinates, according to `ord`. The elements in the second coordinates appear in their original order in s.

For example, if

$$s = \langle (5, "b"), (1, "a"), (1, "b"), (1, "b") \rangle$$

and *ord* is the usual ordering on integers, then `(collect ord s)` will evaluate to

$$\langle(1,\langle"a","b","b"\rangle),(5,\langle"b"\rangle)\rangle$$

2.5.26 `toString:` $(\alpha \rightarrow string) \rightarrow \alpha\ seq \rightarrow string$

If `f` is a function and `s` is a sequence value, `(toString f s)` evaluates to a string representation of s. This representation begins with "⟨", which is followed by the results of applying f to each element of s, in left-to-right order, interleaved with ",", and ends with "⟩".

2.5.27 `tokens:` $(char \rightarrow bool) \rightarrow string \rightarrow string\ seq$

Let `p` be a predicate on characters. A token is a non-empty maximal substring of a string not containing any character that satisfies p. If `p` is a predicate on characters and `s` is a string, `(tokens p s)` evaluates to the sequence of tokens of s in left to right order.

For example, `tokens Char.isPunct ‘‘the,,,horse’’` evaluates to

$$\langle\text{“}the'', \text{“}horse''\rangle$$

and `tokens Char.isPunct ‘‘the,horse’’` evaluates to

$$\langle\text{“}the'', \text{“}horse''\rangle$$

.

2.5.28 `fields:` $(char \rightarrow bool) \rightarrow string \rightarrow string\ seq$

Let `p` be a predicate on characters. A field is a possibly empty maximal substring of not containing any character that satisfies p. If `p` is a predicate on characters and `s` is a string, `(fields p s)` evaluates to the sequence of tokens of s in left to right order.

For example, `fields Char.isPunct ‘‘the,,,horse’’` evaluates to

$$\langle\text{“}the'', \text{“}'', \text{“}'', \text{“}horse''\rangle$$

and `fields Char.isPunct ‘‘the,horse’’` evaluates to

$$\langle\text{“}the'', \text{“}horse''\rangle$$

.

2.5.29 `showt:` $\alpha\ seq \rightarrow \alpha\ treeview$

Let `s` be a sequence value.

- If $|s| = 0$, `(showt s)` evaluates to `EMPTY`.

- If $|s| = 1$, `(showt s)` evaluates to `ELT(`s_0`)`.
- If $|s| > 1$, and `NODE(take (s, `$(|s|/2)$`), drop (s, `$(|s|/2)$`))` evalautes to some value `v`, `(showt s)` evaluates to `v`.

2.5.30 `showti:` α *seq* $\rightarrow$ $(int \rightarrow int) \rightarrow \alpha$ *treeview*

Let `s` be a sequence value.

- If $|s| = 0$, `(showti s f)` evaluates to `EMPTY`.
- If $|s| = 1$, `(showti s f)` evaluates to `ELT(`s_0`)`.
- If $|s| > 1$, f : `int` $\rightarrow$ `int` is a function, and `NODE(take (s, f `$|s|$`), drop (s, f `$|s|$`))` evalautes to some value `v`, `(showti s f)` evaluates to `v`.

2.5.31 `hidet:` α *treeview* $\rightarrow$ α *seq*

Let `tv` be a *treeview* value.

- If `tv` is `EMPTY`, then `(hidet tv)` evaluates to $\langle\rangle$.
- If `tv` is `(ELT x)`, then `(hidet tv)` evaluates to $\langle x \rangle$.
- If `tv` is `NODE (l,r)`, then `(hidet tv)` evaluates to the same value as `append (l,r)`.

2.5.32 `showl:` α *seq* $\rightarrow$ α *listview*

Let `s` be a sequence value.

- If $|s| = 0$, `(showl s)` evaluates to `NIL`.
- If $|s| > 0$, `(showl s)` evaluates to `CONS(`s_0`,`$\langle s_1, \ldots, s_{|s|-1}\rangle$`)`.

2.5.33 `hidel:` α *listview* $\rightarrow$ α *seq*

Let `lv` be a *listview* value.

- If `lv` is `NIL`, then `(hidel lv)` evaluates to $\langle\rangle$.
- If `lv` is `CONS(x,xs)`, then `hidel lv` evaluates to a sequence with length $|xs| + 1$ such that s'_0 is x and s'_i is xs_i for all valid indices i into xs.

Chapter 3

SET Signature

3.1 Overview

We begin by describing an abstract notion of a set representation, which extends the standard mathematical sets. The documentation for the signature that follows states the behavior of implementations in terms of this abstraction.

Like a mathematical set, a *set* S is a finite collection of unique elements of some type and the *size* of S, denoted by $|S|$, is the number of elements in that set. The crucial difference between a set in the mathematical sense and a set is this library is that a set here is always ordered: for enumeration purposes, the implementation gives an implicit ordering of the elements. The *empty set*, denoted by $\emptyset$, is a special set that represents an empty collection, so $|\emptyset| = 0$.

3.2 Signature Definition

```
signature SET =
sig
  type set
  type key
  type t = set
  structure Seq : SEQUENCE
  val empty : set
  val singleton : key -> set
  val size : set -> int
  val equal : set * set -> bool
  val iter : ('b * key -> 'b) -> 'b -> set -> 'b
  val filter : (key -> bool) -> set -> set
  val find : set -> key -> bool
  val union : (set * set) -> set
  val intersection : (set * set) -> set
  val difference : (set * set) -> set
  val insert : key -> set -> set
  val delete : key -> set -> set
  val fromSeq : key Seq.seq -> set
  val toSeq : set -> key Seq.seq
  val toString : set -> string
end
```

3.3 Details of Types

3.3.1 *set*

This is the abstract type representing a set described in Section 3.1.

3.3.2 *key*

This indicates that each element of a set has to have type *key*.

3.4 Details of Values

3.4.1 `empty`: *set*

`empty` represents the empty set $\emptyset$.

3.4.2 `singleton`: $key \rightarrow set$

For a value `x` of type *key*, the expression `singleton x` evaluates to a set containing exactly `x`.

3.4.3 `size`: $set \rightarrow int$

If `s` is a value of type *set*, then `size s` evaluates to $|s|$ (i.e., the number of elements in the set represented by `s`).

3.4.4 `equal`: $set \times set \rightarrow bool$

If `s1` and `s2` are values of type *set*, then `equal (s1,s2)` evaluates to `true` if `s1` and `s2` are identical sets (i.e, they have the exact same set of elements); otherwise, it evaluates to `false`.

3.4.5 `iter`: $(\beta \times key \rightarrow \beta) \rightarrow \beta \rightarrow set \rightarrow \beta$

If `f` is a function, `b` is a value, and `s` is a set value, then `iter f b s` iterates f with left association on s on an implementation-specified ordering, using b as the base case. That is to say, `iter f b s` evaluates to

$$f(f(\ldots f(b, s_{|s|-1}), \ldots s_1), s_0),$$

where $s_0, s_1, \ldots, s_{|s|-1}$ are the elements of s listed in the order that the implementation chooses.

3.4.6 `filter`: $(key \to bool) \to set \to set$

If p is a predicate and s is a set value, then `filter p s` evaluates to the subset s' of s such that an element $x \in s'$ if and only if p holds on x.

3.4.7 `find`: $set \to key \to bool$

If s is a set value and k is a key value, then `find s k` evaluates to a boolean value indicating whether or not k is a member of s.

3.4.8 `union`: $(set \times set) \to set$

If `s` and `t` are set values, `union (s,t)` evaluates to the set $s \cup t$.

3.4.9 `intersection`: $(set \times set) \to set$

If `s` and `t` are set values, `intersection (s,t)` evaluates to the set $s \cap t$.

3.4.10 `difference`: $(set \times set) \to set$

If `s` and `t` are set values, `difference (s,t)` evaluates to the set $s \setminus t$ (i.e. the set $\{x \in s : s \notin t\}$).

3.4.11 `insert`: $key \to set \to set$

If `k` is a key value and `s` is a set, `insert k s` evaluates to the set $s \cup \{k\}$.

3.4.12 `delete`: $key \to set \to set$

If `k` is a key value and `s` is a set, `delete k s` evaluates to the set $s \setminus \{k\}$.

3.4.13 `fromSeq`: $keySeq.\ seq \to set$

If `s` is a sequence value of type *key Seq.seq*, then `fromSeq s` evaluates to the set containing the elements $s_0, s_1, \ldots, s_{|s|-1}$. The ordering in the set representation may differ from the ordering in the sequence representation.

3.4.14 `toSeq`: $set \to keySeq.\ seq$

If `s` is a set value where the elements have type *key*, then `toSeq s` evaluates to the sequence of type *key Seq.seq* containing all $|s|$ elements of s appearing in the order of the implementation's choosing.

3.4.15 `toString`: *set* $\rightarrow$ *string*

If `s` is a set, `toString s` evaluates to a string representation of *s* listing the elements of *s*, interleaved with ",".

Chapter 4

TABLE Signature

4.1 Overview

Abstractly, a *table* is a set of key-value pairs where the keys are unique. For this reason, we often think of it as a mapping that associates each key with a value. Since tables are sets, standard set operations apply on them. We denote by $\emptyset$ an empty table. The size of a table S is the number of keys in S and is rendered $|S|$ in mathematical notation. Furthermore, a table of size n can be written as follows

$$\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\},$$

where $k_1, \ldots, k_n$ are n distinct keys and each key k_i maps to v_i for $i \in [n]$. For concreteness, we say that a key k is present in a table T, written as $k \in_m T$, if there exists a value v such that $(k, v) \in T$. The documentation that follows states the behavior of operations on this abstraction.

4.2 Signature Definition

```
signature TABLE =
sig
  type 'a table
  type 'a t = 'a table
  structure Key : EQKEY
  type key = Key.t
  structure Seq : SEQUENCE
  type 'a seq = 'a Seq.seq
  structure Set : SET where type key = key
  type set = Set.set
  val empty : unit -> 'a table
  val singleton : key * 'a -> 'a table
  val size : 'a table -> int
  val map : ('a -> 'b) -> 'a table -> 'b table
  val mapk : (key * 'a -> 'b) -> 'a table -> 'b table
  val tabulate : (key -> 'a) -> set -> 'a table
  val domain : 'a table -> set
  val range : 'a table -> 'a seq
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a table -> 'a
  val filter : (key * 'a -> bool) -> 'a table -> 'a table
  val iter : ('b * (key * 'a) -> 'b) -> 'b -> 'a table -> 'b
  val iterh : ('b * (key * 'a) -> 'b) -> 'b -> 'a table -> ('b table * 'b)
  val find : 'a table -> key -> 'a option
  val merge : ('a * 'a -> 'a) -> ('a table * 'a table) -> 'a table
  val mergeOpt : ('a * 'a -> 'a option) -> ('a table * 'a table) -> 'a table
  val extract : 'a table * set -> 'a table
  val extractOpt : ('a * 'b -> 'c option) -> 'a table * 'b table -> 'c table
  val erase : 'a table * set -> 'a table
  val insert : ('a * 'a -> 'a) -> (key * 'a) -> 'a table -> 'a table
  val delete : key -> 'a table -> 'a table
  val fromSeq : (key*'a) seq -> 'a table
  val toSeq : 'a table -> (key*'a) seq
  val collect : (key*'a) seq -> 'a seq table
  val toString : ('a -> string) -> 'a table -> string
end;
```

4.3 Details of Types

4.3.1 α *table*

This is the abstract type representing a table with key type *key* (see below) and value type α.

4.3.2 αt

This type is a shorthand for the abstract type α *table* representing a table.

4.3.3 *key*

This indicates that the type of keys in a table has to have type *key*.

4.4 Details of Values

4.4.1 `empty`: $unit \rightarrow \alpha\ table$

`empty` represents the empty collection $\emptyset$.

4.4.2 singleton: $key \times \alpha \to \alpha\ table$

If k is a value of type key and v is a value of type α, the expression singleton (k,v) evaluates to the collection $\{(k, v)\}$.

4.4.3 size: $\alpha\ table \to int$

If T is a value of type $\alpha\ table$, then size T evaluates to $|T|$ (i.e., the number of keys in the collection T).

4.4.4 map: $(\alpha \to \beta) \to \alpha\ table \to \beta\ table$

If f is a function of type $\alpha \to \beta$ and T is a value of type $\alpha\ table$ with entries

$$\{(k_1, v_1), \ldots, (k_n, v_n)\},$$

then map f T evaluates to $\{(k_1, f v_1), (k_2, f v_2), \ldots, (k_n, f v_n)\}$. That is, it creates a new collection with the same keys by applying f on each value.

4.4.5 mapk: $(key \times \alpha \to \beta) \to \alpha\ table \to \beta\ table$

This function generalizes the map function. If f is a function of type $key \times \alpha \to \beta$ and T is a value of type $\alpha\ table$ with entries $\{(k_1, v_1), \ldots, (k_n, v_n)\}$, then mapk f T evaluates to $\{(k_1, f(k_1, v_1)), (k_2, f(k_2, v_2)), \ldots, (k_n, f(k_n, v_n))\}$.

4.4.6 tabulate: $(key \to \alpha) \to set \to \alpha\ table$

If f is a function of type $key \to \alpha$ and S is a value of type set with elements

$$\{k_1, \ldots, k_n\},$$

then tabulate f S evaluates to $\{(k_1, f k_1), (k_2, f k_2), \ldots, (k_n, f k_n)\}$.

4.4.7 domain: $\alpha\ table \to set$

For a table T, the function domain T returns the domain of T as a set.

4.4.8 range: $\alpha\ table \to \alpha\ seq$

For a table T, the function range T returns the range of T as a sequence. In particular it is equivalent to Seq.map (fn (k,v) => v) (toSeq T).

4.4.9 reduce: $(\alpha \times \alpha \to \alpha) \to \alpha \to \alpha\ table \to \alpha$

The function reduce f init T returns the same as Seq.reduce f init (range(T))

4.4.10 `filter`: $(key \times \alpha \to bool) \to \alpha\ table \to \alpha\ table$

If `p` is a predicate and `T` is an α *table* value, then `filter p T` evaluates to the collection T' of T such that $(k, v) \in T$ if and only if p evaluates to `true` on (k, v).

4.4.11 `iter`: $(\beta \times (key \times \alpha) \to \beta) \to \beta \to \alpha\ table \to \beta$

If `f` is a function, `b` is a value, and `T` is a table value, then `iter f b s` iterates f with left association on T on an implementation-specified ordering, using b as the base case. That is, `iter f b T` evaluates to

$$f(f(\ldots f(b, (k_{|T|}, v_{|T|})), \ldots (k_2, v_2))), (k_1, v_1)),$$

where $(k_1, v_1), (k_2, v_2), \ldots, (k_{|T|}, v_{|T|})$ are members of T listed in the order that the implementation chooses.

4.4.12 `iterh`: $(\beta \times (key \times \alpha) \to \beta) \to \beta \to \alpha\ table \to (\beta\ table \times \beta)$

If `f` is a function, `b` is a value, and `T` is a table value, then `iterh f b s` iterates f with left association on T on an implementation-specified ordering, using b as the base case. Unlike `iter`, `iterh` also stores intermediate results in a table. That is, if the implementation orders T as $(k_1, v_1), (k_2, v_2), \ldots, ..., (k_{|T|}, v_{|T|})$ and we let r_i denote the result of the partial evaluation up to the i-th pair (i.e., $r_i = f(f(\ldots f(b, (k_i, v_i)), \ldots (k_2, v_2))), (k_1, v_1)))$, then `iterh` evaluates to the pair

$$(\{(k_i, r_i) : i = 1, \ldots, |T|\}, r_{|T|}),$$

where $r_{|T|} =$ `iter f b T` by definition.

4.4.13 `find`: $\alpha\ table \to key \to \alpha\ option$

If `T` is a table value and `k` is a key value, then `find T k` evaluates to `SOME v` provided that k is present in T and is associated with the value v; otherwise, it evaluates to `NONE`.

4.4.14 `merge`: $(\alpha \times \alpha \to \alpha) \to (\alpha\ table \times \alpha\ table) \to \alpha\ table$

`merge` is a generalization of set union in the following sense. If `f` is a function of type $\alpha \times \alpha \to \alpha$ and `S` and `T` are α tables, then `merge f (S, T)` evaluates a table with the following properties: (1) it contains all the keys from S and T and (2) for each key k, its associated value is inherited from either S or T if k is present in *exactly* one of them. But if k is present in both tables, i.e., $(k, v) \in S$ and $(k, w) \in T$, then the value is $f(v, w)$.

4.4.15 `mergeOpt`: $(\alpha \times \alpha \to \alpha\ option) \to (\alpha\ table \times \alpha\ table) \to \alpha\ table$

`mergeOpt` further generalizes set union, allowing values to cancel out each other and eliminate the presence of a key in manner similar to set symmetric difference. If `f` is a function of type $\alpha \times \alpha \to \alpha$ *option* and `S` and `T` are α tables, then `merge f (S, T)` evaluates a table with the following properties: (1) it contains all the keys from S and T and (2) for each key k, its associated value is inherited from either S or T if k is present in *exactly* one of them. But if k is present in both tables, i.e., $(k, v) \in S$ and $(k, w) \in T$, then the following outcomes are possible: in the case that $f(v, w)$ evaluates to `NONE`, the key k will not be present in the output table; otherwise, $f(v, w)$ evaluates to `SOME r` and the key k will be associated with the value r.

4.4.16 `extract`: $\alpha\ table \times set \to \alpha\ table$

`extract` is a generalization of set intersection in the following sense. If `T` is an α table and `S` is a set, then `extract (T,S)` evaluates to $\{(k, v) \in T : k \in_m S\}$.

4.4.17 `extractOpt`: $(\alpha \times \beta \to \gamma\ option) \to \alpha\ table \times \beta\ table \to \gamma\ table$

`extractOpt` is a further generalization of set intersection. If `f` is a function $\alpha \times \beta \to \gamma$ *option*, `T` is an α table, and `S` is a β table, then `extractOpt f (T,S)` evaluates to $\{(k, w) : (k, v) \in T, (k, v') \in S, \text{ and } w = f(v, v')\}$.

4.4.18 `erase`: $\alpha\ table \times set \to \alpha\ table$

This operation extends set difference. If `T` is an α table, and `S` is a set, then `erase (T,S)` evaluates to $\{(k, v) \in T : (k, v) \in T, k \notin_m S\}$.

4.4.19 `insert`: $(\alpha \times \alpha \to \alpha) \to (key \times \alpha) \to \alpha\ table \to \alpha\ table$

For a function `f` of type $\alpha \times \alpha \to \alpha$, a key-value pair `(k, v)`, and a table `T`, `insert f (k, v) T` evaluates to $T \cup \{(k, v)\}$ provided that $k \notin_m T$; otherwise, if $(k, v') \in T$, it evaluates to $(T \setminus \{(k, v')\}) \cup \{(k, f(v, v'))\}$ (i.e., it replaces the value associated with k with the result of applying f on the old value v' and the new value v).

4.4.20 `delete`: $key \to \alpha\ table \to \alpha\ table$

If `k` is a value of type *key* and `T` is an α *table*, then `delete k T` evaluates to $\{(k', v') \in T : k' \neq k\}$.

4.4.21 `fromSeq`: $(key \times \alpha)\ seq \to \alpha\ table$

If `s` is a $key \times \alpha$ sequence such that

$$s = \langle (k_1, v_1), (k_2, v_2), \ldots (k_n, v_n) \rangle,$$

then `fromSeq s` evaluates to $\{(k_1, v_1), (k_2, v_2), \ldots (k_n, v_n)\}$.

4.4.22 `toSeq`: $\alpha\ table \to (key \times \alpha)\ seq$

If `T` is an α table representing $\{(k_1, v_1), (k_2, v_2), \ldots (k_n, v_n)\}$, then `toSeq T` evaluates to $\langle (k_1, v_1), (k_2, v_2), \ldots (k_n, v_n) \rangle$, where the ordering is determined by the implementation.

4.4.23 `collect`: $(key \times \alpha)\ seq \to \alpha\ seq\ table$

This function groups values of the same key together as a sequence of values that respects the original sequence ordering. Specifically, if `s` is a $key \times \alpha$ sequence representing $\langle (k_1, v_1), (k_2, v_2), \ldots (k_n, v_n) \rangle$, then `collect s` evaluates to $\{(\ell_1, s_1), (\ell_2, s_2), \ldots, (\ell_m, s_m)\}$, where the ℓ_i's are unique keys belonging to $\{k_1, \ldots, k_n\}$ and for $i \in [m]$, s_i is the sequence of values in s with the key ℓ_i (i.e., $s_i = \langle v_j : k_j = \ell_i \rangle$).