Version 0.8.0

Ánoq of the Sun, Hardcore Processing [*]

October 3, 2007

[*]© 2003-2007 Ánoq of the Sun (alias Johnny Bock Andersen)
Quoted as: Ánoq of the Sun, Ánoq, o., Ánoq, o. t. S. or Ánoq, of the Sun. Not: Sun, Á.
Technically, Ánoq is the "family name", always written and pronounced first.
The given name is "of" (in lower-case). Middle-names: "the" and "Sun".

Contents

Preface

- This document and an implementation is maintained online at:

  `http://www.HardcoreProcessing.com/pro/anoqsmlbasis/`

The current version of this report is a preliminary version. It is intended to end up as a report, which is to be handed in at the Computer Science Department at the University of Copenhagen in Denmark.

Before handing in this report, I am encouraging public feedback, particularly on the issues in section 1.1.

1 Introduction

This project presents some extensions to the Standard ML Basis Library. It is the hope that the extensions presented will be adopted as part of the standard. Alternatively, the extensions are hereby made available to everyone.

The reader is assumed to be familiar with the Standard ML programming language (see: [SML97]) and the Standard ML Basis Library (see: [BasisOnline]).

1.1 Open Issues for Discussion (Temporary)

This section contains a list of known open issues, which should be clarified upon completion of this report. Hence, this section is *temporary.*

1. Should the `tabulate{l|r}` functions be renamed to `unfold{l|r}i` and curried, as in Vesa Karvonen's Basis Library work [VesaBasis]? I have already identified currying of `tabulate{l|r}` as a good thing, and the main reason for not having done so is the existing design of `tabulate`. The sections 4 and 7 are relevant to this question

2. Are the `_TRAV` signatures a good thing? There are 4 of them and 2 of them are redundant. See section 5 (in particular section 5.5)

3. Split the signatures `COLLECTION` etc. into a few more signatures? Concretely, I am considering to split `COLLECTION` into two parts:

   (a) `COLLECTION`, which consists of everything down to and including the "dictionary" functions `lookup` and `substitute`. If the semantics of `insert` is allowed to overwrite an element for certain collection types, then the `insert` function could also be kept in this signature

   (b) `COLLECTION_REDUCIBLE`, which includes `COLLECTION` and adds everything from and including the dictionary functions `insert` and `delete`

   The same would have to be done for `MONO_COLLECTION`, of course. One problem: This gives 4 places to add the `IMPERATIVE_` signatures, resulting in 8 signatures! And subsequently 8 places to add the `_TRAV` signatures, resulting in 16 signatures!

   Regarding the two proposed names above, the first one could also be called `COLLECTION_NON_REDUCIBLE` and the second one just `COLLECTION` (i.e. as it is now). See sections 4 (in particular 4.6) and 5

4. Is the design of `forl` and `forr` good? Right now it takes start index, number of iterations and step size as parameters. It could take e.g. end index or end index minus one as parameters instead? Counting iterations seems simpler though, in particular to extend to e.g. Real, where floating point comparison can give problems. See section 6

5. The issue relating to `platformWin32Windows2000` and `platformWin32WindowsXP`, as described in section 10, should probably be clarified? And it does probably not even belong in this report

2 Project Goals

This section presents the goals of the project.

2.1 Problems Solved

The goals related to the problems being solved are the following:

- Add some often-needed functionality to the Standard ML Basis Library
- Make parts of the Standard ML Basis Library more consistent in design
- Simplify the design, if possible

2.2 Practical Goals

In order to increase the likelihood of the extensions being adopted for the standard, the following goals must also be met:

- Maintain compatibility with existing applications, which use Standard ML Basis Library 2004
- Give a thorough and concrete suggestion for extensions of the Standard ML Basis Library
- Make the extensions available as easily and seamlessly as possible

2.3 Additional Motivation

Another goal, which has been part of the motivation for making this project:

- Provide a foundation for a 'better but simpler' basis library for my CeXL programming language

3 Features of the Supplied Implementation

- Fully backwards compatible with Standard ML Basis Library 2004
- Works off-the-shelf with Standard ML compilers using either the 1997 or the 2004 Basis Library[1]
- Uses native compiler implementations of existing Basis Library functionality, whereever possible
- Introduces new `COLLECTION` and `IMPERATIVE_COLLECTION` signatures and corresponding `MONO` versions of these
- The modules List, Vector, Array, Word8Vector, String and CharVector (which is now the same as String) are all extended to conform to these signatures
- Adds an Array2D module, which also conforms to the above signatures
- Supplies `AnoqUnimplementedCollectionForLazyImplementors`, a structure for easily letting newly written partially complete collection data structures conform to the above signatures
- Adds higher-order looping functions to the module Int, Word and Real
- When using this implementation with a 1997 Basis Library, the functions `TextIO.inputLine` and `OS.Path.mk{Absolute|Relative}` are altered to be 2004 compatible and the function `Substring.full` is added

3.1 Not Implemented in the Supplied Implementation

The following features have not been implemented, even though they probably should be, if this is to become part of the standard:

- The Slice counterparts of the implemented modules
- The Substring module has not been implemented as StringSlice, even though that would be the obvious thing to do
- Wide versions of the String and Char related modules
- `Int<N>`, `Word<N>` and `Real<N>` versions of `forl` and `forr`, since support for the `<N>` modules is compiler dependent
- FIXME: Still not tested: The `map{l|r}[i]` functions and the `Array2D` module
- FIXME: `modify[i]` and `map{l|r}[i]` still not implemented in `Array2D`

[1]Tested with SML/NJ 110.0.7, MLton 20040227 and MLton 20020923. Which basis library and compiler you use is configurable in the file AnoqSMLBasis/srcSML/thisdirsources.cm

4 The Collection Signatures

4.1 The Collection Signature

FIXME: Most of the included source code should probably be moved to an appendix

```
(* This is supposed to match ALL polymorphic collection modules in the
   Standard ML Basis library, except for (the hopefully obsolete) Array2.
   I.e. Array, Vector, List, Array2D etc.
   It should also match most other polymorphic collections. *)
signature COLLECTION =
  sig
    type index
    type dimensions
    type 'a t

    (* Construction operations *)
    val empty      : unit -> 'a t
    val create     : dimensions * 'a -> 'a t
    val tabulate   : dimensions * (index -> 'a) -> 'a t

    (* tabulate{l|r} are new and badly missing in SML Basis. Could be curried? *)
    val tabulatel  : dimensions * (index * 'b -> 'a * 'b) * 'b -> ('a t) * 'b
    val tabulater  : dimensions * (index * 'b -> 'a * 'b) * 'b -> ('a t) * 'b

    (* Read operations *)
    val sub        : 'a t * index -> 'a
    (* Dynamic array and dictionary are really good examples of
       difference between dimensions and count.
       count returns exact number of elements. dimensions returns what
       makes sense for data structure dimensions, e.g. the list of all
       used keys in a dictionary or the allocated array size of a dynamic
       array. The type dimensions must make sense for tabulate etc. *)
    val count      : 'a t -> int
    val dimensions : 'a t -> dimensions

    (* Read-only traversal operations *)
    val foldl     : ('a * 'b -> 'b) -> 'b -> 'a t -> 'b
    val foldli    : (index * 'a * 'b -> 'b) -> 'b -> 'a t -> 'b
    val foldr     : ('a * 'b -> 'b) -> 'b -> 'a t -> 'b
    val foldri    : (index * 'a * 'b -> 'b) -> 'b -> 'a t -> 'b

    val foldsepl  : ('b -> 'b) -> ('a * 'b -> 'b) -> 'b -> 'a t -> 'b
    val foldsepli : ('b -> 'b) -> (index * 'a * 'b -> 'b) -> 'b -> 'a t -> 'b
    val foldsepr  : ('b -> 'b) -> ('a * 'b -> 'b) -> 'b -> 'a t -> 'b
    val foldsepri : ('b -> 'b) -> (index * 'a * 'b -> 'b) -> 'b -> 'a t -> 'b

    val exists    : ('a -> bool) -> 'a t -> bool
    val existsi   : (index * 'a -> bool) -> 'a t -> bool
    val find      : ('a -> bool) -> 'a t -> 'a option
    val findi     : (index * 'a -> bool) -> 'a t -> 'a option
    val app       : ('a -> unit) -> 'a t -> unit
    val appi      : (index * 'a -> unit) -> 'a t -> unit

    (* Functional operations *)
    val rev       : 'a t -> 'a t
    val append    : 'a t * 'a t -> 'a t
    val concat    : 'a t list -> 'a t
    val map       : ('a -> 'b) -> 'a t -> 'b t
    val mapi      : ((index * 'a) -> 'b) -> 'a t -> 'b t
    (* map{l|r}[i] are new and badly missing in Standard ML Basis. *)
    val mapli     : ((index * 'a * 'b) -> ('c * 'b)) -> 'b -> 'a t -> ('c t * 'b)
    val mapri     : ((index * 'a * 'b) -> ('c * 'b)) -> 'b -> 'a t -> ('c t * 'b)
    val mapl      : (('a * 'b) -> ('c * 'b)) -> 'b -> 'a t -> ('c t * 'b)
    val mapr      : (('a * 'b) -> ('c * 'b)) -> 'b -> 'a t -> ('c t * 'b)

    (* Conversion to and from lists *)
    val toList    : 'a t -> 'a list
    val fromList  : 'a list -> 'a t

    (* Dictionary functions, which do not alter collection dimensions. *)
    val lookup    : 'a t -> index -> 'a option
    val substitute : index -> ('a -> 'a) -> 'a t -> 'a t
    (* The name update is used by the imperative collection signature,
       so we call it substitute. Unlike update, this is a functional. *)

    (* Dictionary functions. delete diminishes collection dimensions
       and insert may enlarge dimensions for some collections.
       This may not make sense for all kinds of collections. *)
    val insert    : index -> 'a -> 'a t -> 'a t
    val delete    : index -> 'a t -> 'a t

    (* More functional operations: These diminish the collection dimensions.
       This may not always make sense for all collections, e.g. Array2D. *)
    val filter   : ('a -> bool) -> 'a t -> 'a t
    val filteri  : ((index * 'a) -> bool) -> 'a t -> 'a t

    val splitAt  : ('a t) * index -> ('a t) * ('a t)

    val splitl   : ('a -> bool) -> 'a t -> ('a t) * ('a t)
    val splitli  : ((index * 'a) -> bool) -> 'a t -> ('a t) * ('a t)
    val splitr   : ('a -> bool) -> 'a t -> ('a t) * ('a t)
    val splitri  : ((index * 'a) -> bool) -> 'a t -> ('a t) * ('a t)
    val takel    : ('a -> bool) -> 'a t -> 'a t
    val takeli   : ((index * 'a) -> bool) -> 'a t -> 'a t
    val taker    : ('a -> bool) -> 'a t -> 'a t
    val takeri   : ((index * 'a) -> bool) -> 'a t -> 'a t
    val dropl    : ('a -> bool) -> 'a t -> 'a t
    val dropli   : ((index * 'a) -> bool) -> 'a t -> 'a t
    val dropr    : ('a -> bool) -> 'a t -> 'a t
    val dropri   : ((index * 'a) -> bool) -> 'a t -> 'a t

    (* triml/r require that dimensions is "negatable" in some sense,
       which should be the case for almost any sensible dimensions type. *)
    val triml    : dimensions -> 'a t -> 'a t  (* comparable to dropl *)
    val trimr    : dimensions -> 'a t -> 'a t  (* comparable to dropr *)

    val keepl    : dimensions -> 'a t -> 'a t  (* comparable to takel *)
    val keepr    : dimensions -> 'a t -> 'a t  (* comparable to taker *)
  end
```

4.2 The Imperative Collection Signature

```
(* This is supposed to match ALL polymorphic imperative collection
   modules in the Standard ML Basis library, except for
   (the hopefully obsolete) Array2. E.g. Array, Array2D.
   It should also match most other polymorphic imperative collections. *)
signature IMPERATIVE_COLLECTION =
  sig
    include COLLECTION

    (* Destructive traversal updates *)
    val modify  : ('a -> 'a) -> 'a t -> unit
    val modifyi : (index * 'a -> 'a) -> 'a t -> unit

    (* Other destructive updates *)
    val update  : 'a t * index * 'a -> unit
  end
```

4.3 The Mono Collection Signature

```
(* This is supposed to match ALL monomorphic collection modules in the
   Standard ML Basis library.
   I.e. Word8Vector, String, IntVector etc.
   It should also match most other monomorphic collections. *)
signature MONO_COLLECTION =
  sig
    type index
    type dimensions
    type t
    type elem

    (* Construction operations *)
    val empty      : unit -> t
    val create     : dimensions * elem -> t
    val tabulate   : dimensions * (index -> elem) -> t

    (* tabulatel/r are new and badly missing in SML Basis. Could be curried? *)
    val tabulatel  : dimensions * (index * 'b -> elem * 'b) * 'b -> t * 'b
    val tabulater  : dimensions * (index * 'b -> elem * 'b) * 'b -> t * 'b

    (* Read operations *)
    val sub        : t * index -> elem
    (* Dynamic array and dictionary are really good examples of
       difference between dimensions and count.
       count returns exact number of elements. dimensions returns what
       makes sense for data structure dimensions, e.g. the list of all
       used keys in a dictionary or the allocated array size of a dynamic
       array. The type dimensions must make sense for tabulate etc. *)
    val count      : t -> int
    val dimensions : t -> dimensions

    (* Read-only traversal operations *)
    val foldl     : (elem * 'b -> 'b) -> 'b -> t -> 'b
    val foldli    : (index * elem * 'b -> 'b) -> 'b -> t -> 'b
    val foldr     : (elem * 'b -> 'b) -> 'b -> t -> 'b
    val foldri    : (index * elem * 'b -> 'b) -> 'b -> t -> 'b

    val foldsepl  : ('b -> 'b) -> (elem * 'b -> 'b) -> 'b -> t -> 'b
    val foldsepli : ('b -> 'b) -> (index * elem * 'b -> 'b) -> 'b -> t -> 'b
    val foldsepr  : ('b -> 'b) -> (elem * 'b -> 'b) -> 'b -> t -> 'b
    val foldsepri : ('b -> 'b) -> (index * elem * 'b -> 'b) -> 'b -> t -> 'b

    val exists    : (elem -> bool) -> t -> bool
    val existsi   : (index * elem -> bool) -> t -> bool
    val find      : (elem -> bool) -> t -> elem option
    val findi     : (index * elem -> bool) -> t -> elem option
    val app       : (elem -> unit) -> t -> unit
    val appi      : (index * elem -> unit) -> t -> unit

    (* Functional operations *)
    val rev       : t -> t
    val append    : t * t -> t
    val concat    : t list -> t
    val map       : (elem -> elem) -> t -> t
    val mapi      : ((index * elem) -> elem) -> t -> t

    (* map{l|r}[i] are new and badly missing in Standard ML Basis. *)
    val mapli     : ((index * elem * 'b) -> (elem * 'b)) -> 'b -> t -> (t * 'b)
    val mapri     : ((index * elem * 'b) -> (elem * 'b)) -> 'b -> t -> (t * 'b)
    val mapl      : ((elem * 'b) -> (elem * 'b)) -> 'b -> t -> (t * 'b)
    val mapr      : ((elem * 'b) -> (elem * 'b)) -> 'b -> t -> (t * 'b)

    (* Conversion to and from lists *)
    val toList    : t -> elem list
    val fromList  : elem list -> t

    (* Dictionary functions, which do not alter collection dimensions. *)
    val lookup    : t -> index -> elem option
    val substitute : index -> (elem -> elem) -> t -> t
    (* The name update is used by the imperative collection signature,
       so we call it substitute. Unlike update, this is a functional. *)

    (* Dictionary functions. delete diminishes collection dimensions
       and insert may enlarge dimensions for some collections.
       This may not make sense for all kinds of collections. *)
    val insert    : index -> elem -> t -> t
    val delete    : index -> t -> t

    (* More functional operations: These diminish the collection dimensions.
       This may not always make sense for all collections, e.g. Array2D. *)
    val filter   : (elem -> bool) -> t -> t
    val filteri  : ((index * elem) -> bool) -> t -> t

    val splitAt  : t * index -> t * t

    val splitl   : (elem -> bool) -> t -> t * t
    val splitli  : ((index * elem) -> bool) -> t -> t * t
    val splitr   : (elem -> bool) -> t -> t * t
    val splitri  : ((index * elem) -> bool) -> t -> t * t
    val takel    : (elem -> bool) -> t -> t
    val takeli   : ((index * elem) -> bool) -> t -> t
    val taker    : (elem -> bool) -> t -> t
    val takeri   : ((index * elem) -> bool) -> t -> t
    val dropl    : (elem -> bool) -> t -> t
    val dropli   : ((index * elem) -> bool) -> t -> t
    val dropr    : (elem -> bool) -> t -> t
    val dropri   : ((index * elem) -> bool) -> t -> t

    (* triml/r require that dimensions is "negatable" in some sense,
       which should be the case for almost any sensible dimensions type. *)
    val triml    : dimensions -> t -> t  (* comparable to dropl *)
    val trimr    : dimensions -> t -> t  (* comparable to dropr *)

    val keepl    : dimensions -> t -> t  (* comparable to takel *)
    val keepr    : dimensions -> t -> t  (* comparable to taker *)
  end
```

4.4 The Imperative Mono Collection Signature

```
(* This is supposed to match ALL monomorphic imperative collection
   modules in the Standard ML Basis library.
   E.g. IntArray.
   It should also match most other monomorphic imperative collections. *)
signature IMPERATIVE_MONO_COLLECTION =
  sig
  include MONO_COLLECTION

  (* Destructive traversal updates *)
  val modify  : (elem -> elem) -> t -> unit
  val modifyi : (index * elem -> elem) -> t -> unit

  (* Other destructive updates *)
  val update  : t * index * elem -> unit
end
```

4.5 Examples of Other Collection Data Structures

Apart from the modules in the Standard ML Basis Library, several other data structures could be made to conform to these signatures:

- A dictionary data structure with strings as keys. The `dimensions` type would be `string list` and the `index` type would be `string`
- The signatures can also support symbol table implementations, like the one found in section 5 of the book Modern Compiler Implementation in ML ("The Tiger Book" - FIXME: Make the litterature reference). It can be done by letting the `index` type be a hash or an integer (possibly made opaque by signature mathching) and by having two extra functions like these (which are not part of the general signatures):

  ```
  val symbol : string -> index
  val name : index -> string
  ```

  So now, hopefully the signatures also satisfy many compiler writers :-)

4.5.1 Lazy Implementors Writing New Collection Data Structures

If you are writing a new collection data structure and you want it to conform to one of the `COLLECTION` signatures, but you don't want to bother implementing everything, you can quickly implement it as follows:

```
structure MyNewCollection
  : COLLECTION (* Or one of the other signatures *) =
struct
  open AnoqUnimplementedCollectionForLazyImplementors

  (* Add the types and functions here,
     which you want to bother implementing *)
end
```

Hence, being lazy is not even an excuse for not using these signatures :-)

FIXME: Some functions can easily be implemented in terms of other functions. E.g.: `foldl` in terms of `foldli`, all the `take` and `drop` functions in terms of `split{l|r}[i]` etc. I am considering to supply functors for doing just that.

4.6 Functions Altering Data Structure Dimensions

4.6.1 Functions Reducing Dimensions

The functions mentioned last in the `COLLECTION` signatures differ from the other functions, in that they reduce the dimensions of the data structure. This goes for the functions which are variants of `filter`, `split`, `take`, `drop`, `trim` and `keep`. The same also holds for the function `delete`.

This kind of reduction of dimensions may not make sense for all collection data structures. Hence, I am considering to split the `COLLECTION` signatures into two signatures.

Also note that, for some of the above functions (in particular `triml` and `trimr`), their `dimensions` parameter is the dimensions in some negated sense. This should make sense for almost any sensible `dimensions` type, but it is important to notice.

4.6.2 The Functions insert and delete

As mentioned in the previous section, the function `delete` can be put into the category of reducing data structure dimensions.

The function `insert` also alters data structure dimensions, but this is by increasing the dimensions. This may not make sense for all data structures either. However, in the case of `insert`, a work-around can be made for most data structures here, by allowing its semantics to differ slightly from data structure to data structure. In particular, for data structures like lists, vectors and arrays, a new index can be inserted, while shifting all other indices. For a data structure like a dictionary, this function is more likely to just overwrite the given index, without shifting any other indices. For a data structure like `Array2D`, it is also not really possible to insert a new index and shift all other indices, so overwriting would be the behaviour of choice here.

I am not sure of this differing in behaviour is a bad thing, but some data structures work more naturally with one of these behaviours, while others work more naturally with the other.

5 Traversal Mode Additions for the Collection Signatures

FIXME: I am not sure if having these signatures is a good idea? There are 4 signatures and 2 of them are even redundant.

5.1 The Collection Traversal Signature

```
(* This is supposed to match all polymorphic collection modules in the
   Standard ML Basis library which support modes of traversal.
   E.g. a string dictionary with alphabetical vs. arbitrary traversal. *)
signature COLLECTION_TRAV =
  sig
    include COLLECTION

    type traversal

    (* Supposed to be the default and thus most efficient
       traversal order for the collection data structure. *)
    val defaultTraversal   : traversal

    (* Setting and getting the traversal mode (always a persistent setting) *)
    val setTraversal       : traversal -> 'a t -> 'a t
    val getTraversal       : 'a t -> traversal

    (* Construction operations *)
    val tabulateTraversal  : traversal -> dimensions * (index -> 'a) -> 'a t
    val tabulatelTraversal : traversal -> dimensions * (index * 'b -> 'a * 'b) * 'b -> ('a t) * 'b
    val tabulaterTraversal : traversal -> dimensions * (index * 'b -> 'a * 'b) * 'b -> ('a t) * 'b
  end
```

5.2 The Imperative Collection Traversal Signature

```
(* This is supposed to match all imperative polymorphic collection modules
   in the Standard ML Basis library which support modes of traversal.
   E.g. Array2D. It should also match other polymorphic collections,
   like an Array3D module and a string dictionary with alphabetical
   vs. arbitrary traversal. *)
signature IMPERATIVE_COLLECTION_TRAV =
  sig
    include IMPERATIVE_COLLECTION

    type traversal

    (* Supposed to be the default and thus most efficient
       traversal order for the collection data structure. *)
    val defaultTraversal   : traversal

    (* Setting and getting the traversal mode (always a persistent setting) *)
    val setTraversal       : traversal -> 'a t -> 'a t
    val getTraversal       : 'a t -> traversal

    (* Construction operations *)
    val tabulateTraversal  : traversal -> dimensions * (index -> 'a) -> 'a t
    val tabulatelTraversal : traversal -> dimensions * (index * 'b -> 'a * 'b) * 'b -> ('a t) * 'b
    val tabulaterTraversal : traversal -> dimensions * (index * 'b -> 'a * 'b) * 'b -> ('a t) * 'b
  end
```

5.3 The Mono Collection Traversal Signature

```
(* This is supposed to match all monomorphic collection modules in the
   Standard ML Basis library which support modes of traversal. *)
signature MONO_COLLECTION_TRAV =
  sig
    include MONO_COLLECTION

    type traversal

    (* Supposed to be the default and thus most efficient
       traversal order for the collection data structure. *)
    val defaultTraversal : traversal

    (* Setting and getting the traversal mode (always a persistent setting) *)
    val setTraversal       : traversal -> t -> t
    val getTraversal       : t -> traversal

    (* Construction operations *)
    val tabulateTraversal  : traversal -> dimensions * (index -> elem) -> t
    val tabulatelTraversal : traversal -> dimensions * (index * 'b -> elem * 'b) * 'b -> t * 'b
    val tabulaterTraversal : traversal -> dimensions * (index * 'b -> elem * 'b) * 'b -> t * 'b
  end
```

5.4 The Imperative Mono Collection Traversal Signature

```
(* This is supposed to match all monomorphic collection modules in the
   Standard ML Basis library which support modes of traversal. *)
signature IMPERATIVE_MONO_COLLECTION_TRAV =
  sig
    include IMPERATIVE_MONO_COLLECTION

    type traversal

    (* Supposed to be the default and thus most efficient
       traversal order for the collection data structure. *)
    val defaultTraversal : traversal

    (* Setting and getting the traversal mode (always a persistent setting) *)
    val setTraversal       : traversal -> t -> t
    val getTraversal       : t -> traversal

    (* Construction operations *)
    val tabulateTraversal  : traversal -> dimensions * (index -> elem) -> t
    val tabulatelTraversal : traversal -> dimensions * (index * 'b -> elem * 'b) * 'b -> t * 'b
    val tabulaterTraversal : traversal -> dimensions * (index * 'b -> elem * 'b) * 'b -> t * 'b
  end
```

5.5 Some Examples of Modules with Traversal Modes

The following are some examples of modules, which could have traversal modes:

- A dictionary data structure with strings as keys could use this traversal datatype:

  ```
  datatype traversal =
    Arbitrary
  | Alphabetical
  ```

- An `Array3D` module could use this traversal datatype:

  ```
  datatype traversal =
    XYZ
  | YXZ
  | XZY
  | ZXY
  | YZX
  | ZYX
  ```

- A suggestion for traversals of a binary tree data structure:

  ```
  datatype traversal =
    Inorder
  | Preoder
  | Postorder
  ```

  Maybe only `Inorder` and `Preorder` are needed, since `Postorder` would be like using `{fold|map|tabulate}r[i]` instead of `{fold|map|tabulate}l[i]` for the traversal

I hope that this is enough to convince the reader that the traversal modes are useful for more than just `Array2D`.

5.6 A More Drastic Idea

If one wouldn't mind introducing incompatibilities with the current SML Basis, one could even replace the `{fold|map|tabulate}{l|r}[i]` functions with only versions `{fold|map|tabulate}[i]` (i.e. without the `{l|r}` part), and supporting forward and backward traversal modes for everyting by letting all the functions have a traversal parameter.

I would not recommend such a drastic change to the Standard ML Basis Library though. It might also give a little run-time overhead for dispatching on the traversal parameter, for modules not requiring more traversal modes than left vs. right, at least if the Basis Library interface is made available through an FFI interface, but see section 7.2 for my discouragement from doing that in the first place.

6 Number Looping Functions

It is my experience that one often writes a snippet of ML code, which starts by using `List.tabulate` to make a list of integers, which is subsequently used for `List.foldl`, to fold a function over the indices. This can be quite inefficient, if the compiler is unable to optimize away the intermediate list. On these occasions, the `forl` and `forr` functions presented here would undoubtedly be better.

6.1 Int Looping Functions

```
(* An integer module with forl and forr functions. *)
structure Int =
  struct
    open Int

    (* The forl and forr functions could be curried, but one
       may be likely to forget the last init parameter,
       if using it for a side-effecting loop! *)
    fun forl (starti, iters, stepi, f, init) =
          if iters >= 0 then
            let
              val endi = starti + stepi * iters

              (* The inner loop, which should be efficient.
                 Only one live variable for counting: i. *)
              fun loopUp (i, acc) =
                    if i < endi then
                      loopUp (i + stepi, f (i, acc))
                    else
                      acc

              (* The inner loop for when stepi < 0 *)
              fun loopDown (i, acc) =
                    if i > endi then
                      loopDown (i + stepi, f (i, acc))
                    else
                      acc
            in
              if stepi > 0 then
                loopUp (starti, init)
              else if stepi < 0 then
                loopDown (starti, init)
              else
                raise Subscript
            end
          else
            raise Subscript

    fun forr (starti, iters, stepi, f, init) =
          if iters >= 0 then
            let
              (* In forr the loop just counts backwards,
                 starting at the last iteration. Notice: iters - 1 *)
              val endi = starti + stepi * (iters - 1)

              (* The inner loop, which should be efficient.
                 Only one live variable for counting: i. *)
              fun loopUp (i, acc) =
                    (* Notice: i >= starti *)
                    if i >= starti then
                      loopUp (i - stepi, f (i, acc))
                    else
                      acc

              (* The inner loop for when stepi < 0 *)
              fun loopDown (i, acc) =
                    (* Notice: i <= starti *)
                    if i <= starti then
                      loopDown (i - stepi, f (i, acc))
                    else
                      acc
            in
              if stepi > 0 then
                loopUp (endi, init)
              else if stepi < 0 then
                loopDown (endi, init)
              else
                raise Subscript
            end
          else
            raise Subscript
  end
```

6.2 Word Looping Functions

```
(* A word module with forl and forr functions. *)
structure Word =
  struct
    open Word

    (* The forl and forr functions could be curried, but one
       may be likely to forget the last init parameter,
       if using it for a side-effecting loop! *)
    fun forl (starti, iters, stepi, f, init) =
          (* Negative Word values do not exist, so stepi is never negative *)
          if stepi > 0w0 andalso iters >= 0w0 then
            let
              val endi = starti + stepi * iters

              (* The inner loop, which should be efficient.
                 Only one live variable for counting: i. *)
              fun loop (i, acc) =
                    if i < endi then
                      loop (i + stepi, f (i, acc))
                    else
                      acc
            in
              loop (starti, init)
            end
          else
            raise Subscript

    fun forr (starti, iters, stepi, f, init) =
          (* Negative Word values do not exist, so stepi is never negative *)
          if stepi > 0w0 then
            if iters > 0w0 then
              let
                (* In forr the loop just counts backwards,
                   starting at the last iteration. Notice: iters - 1 *)
                val endi = starti + stepi * (iters - 0w1)
                (* The inner loop, which should be efficient.
                   Only one live variable for counting: i. *)
                fun loop (i, acc) =
                      (* (print (String.concat ["loop ", Word.toString i]); *)
                      if i > starti then
                        loop (i - stepi, f (i, acc))
                      else if i = starti then
                        (* We have to have this case, otherwise i wraps around *)
                        f (i, acc)
                      else
                        acc
              in
                loop (endi, init)
              end
            else if iters = 0w0 then
              (* We have to have this case, otherwise iters - 0w1 wraps around. *)
              init
            else
              raise Subscript
          else
            raise Subscript
  end
```

6.3 Real Looping Functions

```
(* A real module with forl and forr functions. *)
structure Real =
  struct
    open Real

    (* The forl and forr functions could be curried, but one
       may be likely to forget the last init parameter,
       if using it for a side-effecting loop! *)
    fun forl (starti : real, iters : int, stepi : real, f, init) =
          if Int.>=(iters, 0) then
            let
              (* The inner loop, which should be efficient.
                 We need two live variables just for counting though.
                 iters controls the number of iterations, without floating
                 point inaccurracy problems, and i is the index. *)
              fun loop (i : real, iters : int, acc) =
                    if Int.>(iters, 0) then
                      loop (i + stepi, Int.-(iters, 1), f (i, acc))
                    else
                      acc
            in
              if stepi > 0.0 orelse stepi < 0.0 then
                loop (starti, iters, init)
              else
                raise Subscript
            end
          else
            raise Subscript

    (* forr is not tail-recursive and thus uses stack space!
       This is necessary to guarantee that the same index values are
       traversed, in the presence of floating point inaccuracy problems. *)
    fun forr (starti : real, iters : int, stepi : real, f, init) =
          if Int.>=(iters, 0) then
            let
              (* The inner loop, which should be efficient.
                 We need two live variables just for counting though
```

```
               and it is not tail-recursive.
               iters controls the number of iterations, without floating
               point inaccurracy problems, and i is the index. *)
            fun loop (i : real, iters : int) =
                  if Int.>(iters, 0) then
                    f(i, loop (i + stepi, Int.-(iters, 1)))
                  else
                    init
          in
            if stepi > 0.0 orelse stepi < 0.0 then
              loop (starti, iters)
            else
              raise Subscript
          end
        else
          raise Subscript

end
```

7 Software and API Design Guidelines

This section discusses some API design principles. The main focus is for Standard ML and its Basis Library, but some principles can be taken as being more general than that, which makes this section valuable to programmers in general.

FIXME: Structure this section into a discussion and the resulting guidelines / principles.

7.1 When To Use Currying and When Not To

There are pros and cons to using curried function parameters. Some important arguments are listed here:

Pros: Every programmer of a functional programming language know that curried function parameters are very convenient on many occasions.

Cons: There are many potential downsides to consider, when a choice between currying or not is to be made:

1. It is possible to forget one or more of the last arguments for a curried function call (at a call-site for the function), without the type-checker complaining.

   If the nature of a function is such that it returns a useful value that one is typically interested in using - e.g. the result of a lookup function in a data structure, then the type-system of ML will complain if an argument is forgotten.

   However, if a function does not return a useful value, or if it returns a value that one might not always want to use, then it is *quite* easy to forget an argument. If the purpose of such a function is to give a side-effect and it just returns the value of type unit, then this is an obvious place where using record parameters or tuple parameters may be in favour of currying.

2. The type errors that one has to deal with for curried functions are usually a little more complex than non-curried counterparts

3. It is not always possible to determine an ordering of the curried parameters, such that this particular ordering is typically the most useful.

4. Currying may have a computational price, especially if it is exported though a foreign function interface (FFI) or similar.

Which of the above downsides are important as design parameters?

- As discussed in section 7.2, I would strongly discourage wrapping the Standard ML Basis library into an FFI. Hence the computational price of curring (”con 4” in the above list) should not be an important design parameter for the interfaces of the basic data structures. However, they might be relevant for modules accessing the operating system.

All in all, I think that the SML Basis Library has made quite sensible choices of when to use currying and when not to. There are a few places, where one could consider using currying, mostly the function sub I guess, but I would call that a small flaw :) One could also consider using records more frequently than tuples, but at least the tuples are very small and simple in the SML Basis, and records are not even better than tuples for all uses either.

7.2 Foreign Function Interfaces (FFIs) and External Libraries

Something about FFIs, when to use external libraries and when not to:

- Certain things make very good sense to put into a "library" (meaning here an external library accessed via an FFI). This is mostly things which both have a substantial size and a well-defined interface, preferably with small amounts of data transfer. This also goes for interfaces to external hardware and the likes. Examples: RenderMan interface, OpenGL interface, graphical user interface toolkits etc.

  Certain parts of the SML Basis Library might make sense to have in an external library. E.g. the modules which access the operating system. But then again, the lower-level POSIX interface might be even better.

- Certain things should never be put into (external) libraries. In particular functionality which is simple or has large amounts of data transfer through the interface, compared to how much computation is done on those data.

  Canonical examples: The basic data structures of the Standard ML Basis library(!) and other small utility functions. The best is of course for the compiler to be able to optimize (e.g. inline, specialize, monomorphize etc.) on such things.

  To take the argument of never passing through an FFI further, if the Basis Library has bugs, an application using the Basis might have corresponding work-around bugs, so just upgrading a .dll or .so with a new basis might actually crash applications using it - so not even from a maintainance point-of-view is it a good thing to e.g. isolate it in a .dll or .so file.

  A related real-world example: Typically, applications for Linux link dynamically with the `libc` library. Binary incompatibilities of libc can be one of the big headaches of installing Linux applications today! The SML Basis Library should not become a similar obstacle for ML programs. In fact, an ML-compiler producing Linux binaries without using `libc` could give some interesting competition to C-programs on Linux :-)

8 Existing Basis Library Functionality To Deprecate and Eventually Remove

The following parts of the Standard ML Basis Library are what I would recommend to deprecate, and eventually, in a few years or so, remove:

- Many of the `MONO_` modules could be deprecated. In particular `IntArray`, `Int<N>Array`, `IntVector`, `Int<N>Vector`, `RealArray`, `Real<N>Array`, `RealVector`, `Real<N>Vector`, `WordArray`, `Word<N>Array` (except those mentioned below), `WordVector`, `Word<N>Vector` (except those mentioned below) and the `Slice` counterparts of all these.

  Reasons: A good compiler should be able to infer as efficient compilation of these as for their polymorphic counterparts. Programmers should also be discouraged from even considering to use these modules, since they are not type-wise compatiple with their polymorphic counterparts, which probably results in needless conversions at runtime when using them.

  However, the modules `CharArray`, `CharVector`, `String`, `StringSlice`, `Substring`, `Word8Array`, `Word8Vector`, `BoolArray`, `BoolVector` and their corresponding `Slice` modules should probably be kept. One could also consider keeping `Word32Array`, `Word32Vector`, `Word64Array`, `Word64Vector` and their `Slice` counterparts, since these conform to natural computer memory layouts

- All `Array2` variants, in favor of `Array2D`.

  Reasons: Consistency of signatures

9 The Wall of Shame

9.1 An Alternative Collection Traversal Signature

An alternative way of defining the traversal functionality would be to add new functions with an extra curried parameter. This requires addition of many new redundant functions. Putting that many redundant functions in a general signature, just to achieve compatibility with a single module, Array2, is something which I consider to be a bad idea.

```
(* This is supposed to match all polymorphic collection modules in the
   Standard ML Basis library which support modes of traversal.
   E.g. Array2D. It should also match other polymorphic collections,
   like an Array3D module and a string dictionary with alphabetical
   vs. arbitrary traversal. *)
signature COLLECTION_TRAV_OBSOLETE =
  sig
    include COLLECTION

    type traversal

    (* Supposed to be the default and thus most efficient
       traversal order for the collection data structure. *)
    val defaultTraversal : traversal

    (* Construction operations *)
    val tabulateTraversal : traversal -> dimensions * (index -> 'a) -> 'a t
    (* tabulate{l|r}Traversal could be defined too *)

    (* Read-only traversal operations *)
    val appTraversal    : traversal -> ('a -> unit) -> 'a t -> unit
    val appiTraversal   : traversal -> (index * 'a -> unit) -> 'a t -> unit
    val foldlTraversal  : traversal -> ('a * 'b -> 'b) -> 'b -> 'a t -> 'b
    val foldliTraversal : traversal -> (index * 'a * 'b -> 'b) -> 'b -> 'a t -> 'b
    val foldrTraversal  : traversal -> ('a * 'b -> 'b) -> 'b -> 'a t -> 'b
    val foldriTraversal : traversal -> (index * 'a * 'b -> 'b) -> 'b -> 'a t -> 'b

    (* Functional operations *)
    val mapTraversal    : traversal -> ('a -> 'b) -> 'a t -> 'b t
    val mapiTraversal   : traversal -> ((index * 'a) -> 'b) -> 'a t -> 'b t

    (* More functions, like map{l|r}[i]Traversal, could be defined too *)
  end
```

9.1.1 An Alternative Mono Collection Traversal Signature

The above signature alone would not be the complete story of how many redundant signature functions would have to be added. A mono version of the signature would also be needed, which would look as follows.

```
signature MONO_COLLECTION_TRAV_OBSOLETE =
  sig
    include MONO_COLLECTION

    type traversal

    (* Supposed to be the default and thus most efficient
       traversal order for the collection data structure. *)
    val defaultTraversal : traversal
    (* Construction operations *)
    val tabulateTraversal  : traversal -> dimensions * (index -> elem) -> t
    val tabulatelTraversal : traversal -> dimensions * (index * 'b -> elem * 'b) * 'b -> t * 'b
    val tabulaterTraversal : traversal -> dimensions * (index * 'b -> elem * 'b) * 'b -> t * 'b

    (* Read-only traversal operations *)
    val appTraversal    : traversal -> (elem -> unit) -> t -> unit
    val appiTraversal   : traversal -> (index * elem -> unit) -> t -> unit
    val foldlTraversal  : traversal -> (elem * 'b -> 'b) -> 'b -> t -> 'b
    val foldliTraversal : traversal -> (index * elem * 'b -> 'b) -> 'b -> t -> 'b
    val foldrTraversal  : traversal -> (elem * 'b -> 'b) -> 'b -> t -> 'b
    val foldriTraversal : traversal -> (index * elem * 'b -> 'b) -> 'b -> t -> 'b

    (* Functional operations *)
    val mapTraversal    : traversal -> (elem -> elem) -> t -> t
    val mapiTraversal   : traversal -> ((index * elem) -> elem) -> t -> t
  end
```

10 Mistakes Made In The Current (2004) Standard ML Basis Library

This section points out a few mistakes, which are currently present in the Standard ML Basis Library. (FIXME: This does probably not belong in the report...?)

- In the history log for Standard ML Basis, I found an entry where `platformWin32Windows2000` and `platformWin32WindowsXP` have been removed and an accompanying claim that `platformWin32WindowsNT` is identical to `platformWin32Windows2000`.

  Windows NT and Windows 2000 are *not* entirely identical, not even from a programmer's point of view. Proof:

  Try running the following Standard ML program on NT and 2000, respectively:

  ```
  val str = TextIO.openOut "testLinefeeds.txt"
  val _ = TextIO.output (str, "Hello\013\010\013\010World")
  val _ = TextIO.close str
  ```

  The output files are not identical on the OSes.

  Hence, `platformWin32Windows2000` and `platformWin32WindowsXP` should probably be added back in? Maybe along with some way to determine the "type" of operating system? Or perhaps just an ordering comparison on platform, to be able to ask "is this 'at least' Windows NT"?

References

[SML97] Robin Milner, Mads Tofte, Robert Harper, David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[BasisOnline] John Reppy et. al. *The Standard ML Basis Library.* `http://www.standardml.org/Basis/`

[VesaBasis] Vesa Karvonen. *An Extended Basis Library.* `svn://mlton.org/mltonlib/trunk/com/ssh/extended-basis/unstable/`

[SMLNJ] David MacQueen et. al. *Standard ML of New Jersey.* `http://www.smlnj.org/`

[MLton] Stephen Weeks et. al. *MLton.* `http://www.mlton.org/`