Designing the Standard Haskell Libraries (Position Paper)

Alastair Reid and John Peterson
Department of Computer Science, Yale University,
P.O. Box 208285, New Haven, CT 06520, USA.
Electronic mail: {reid-alastair,peterson-john}@cs.yale.edu

October 4, 1998

Abstract

Five years after the first Haskell report was published, the Haskell language continues to grow and mature. After five years experience of Haskell programming, we wish to both expand and simplify the Haskell language. Over the years, many Haskell libraries have been developed. The Haskell Committee is expanding the language by standardising a set of libraries to add to the definition of Haskell. Another goal is to simplify Haskell by moving parts of the prelude, a built-in set of types and functions implicitly a part of every Haskell program, into libraries where they can be loaded on demand, This document describes the issues involved in the design of the Haskell libraries and summarises the library modules being considered.

1 Motivation

In the five years since the Haskell programming language was created, Haskell programmers have developed libraries providing many useful functions and datatypes. Each implementation of Haskell currently distributes home grown libraries. Using these libraries speeds development but decreases portability because these libraries are not available on all platforms. We believe that many of the functions and types in these libraries should become a standardised part of Haskell 1.3 (with further libraries being added in later revisions). Standardising these libraries will:

1. Increase the power of the language by providing a much greater level of basic functionality in the standard.
2. Improve the portability of programs by eliminating the need for non-standard libraries.
3. Avoid the splintering of Haskell into different dialects where programmers familiar with the HBC libraries (say) would not be able to understand a program written using the GHC libraries.

The major argument against standardising libraries is that doing so increases the size of language — both the amount new programmers must learn and the size of implementations. We feel that the increased utility of the language outweighs this concern.

The Haskell language has what is essentially a built-in library called the standard prelude. The prelude is special only in that it is implicitly imported into every program. Moving the infrequently-used components of the prelude into libraries has two advantages: it shrinks the core language, making the essential components of Haskell more apparent; and it frees up more of the namespace for the user.

This document describes the issues that arise in turning the existing libraries into a concrete library proposal and briefly summarises each library. This document does not discuss changes to the libraries or prelude which are related to the I/O proposal; these are described in Gordon and Hammond's tutorial paper [3]. Detailed definitions of the libraries are supplied in a companion document [11].

Acknowledgements

We are grateful to members of the Yale, Glasgow and Chalmers functional programming groups (especially Sandra Loosemore, Dan Rabin, Will Partain, Jim Mattson, Andy Gill, Kevin Hammond and Kent Karlsson), Mark Jones, Ian Poole, Nick North, Paul Otto and the other members of the Haskell 1.3 committee for detailed comments on the design criteria and the design of the libraries.

We are especially grateful to the GRASP/AQUA team at Glasgow and to Lennart Augustsson at Chalmers for providing the basis on which much of the library design is built.

2 Design Issues

The main questions to be answered when designing library modules are:

What should be included?
What should go in the libraries and what should go in the prelude?
What should go in each module?
What classes should each new type be an instance of?
What should the type and name of each function be?
How should library functions be defined in a standard?
How do libraries interact with other aspects of Haskell?

We address each of these questions in turn.

2.1 What should be included?

To be included in the standard libraries, an entity (type, type-class or function) must satisfy two criteria:

1. It must be clear what the interface should look like — it is much harder to remove or correct a feature once it has been included in the language standard.
2. The entity must be useful to a "significant number" of programmers.

Since the standard libraries draw heavily on existing libraries, we have some confidence that the interfaces have been tested in real programs.

Entities which cannot be implemented efficiently in standard Haskell because they require special support in the compiler or runtime system deserve special consideration since users have no portable alternative.

Since the primary goal of having standard libraries is to improve portability, Haskell implementations are required to provide all of the standard libraries and are not permitted to add, modify, or omit entities. Implementors are encouraged to provide optimised versions of library functions *provided that the optimised version has the same external behaviour (type, strictness, error conditions, etc) as in the specification.*

2.2 What should go in the libraries and what should go in the prelude?

The *only* operational difference between entities defined in the prelude and those defined in a library is that the prelude is automatically imported into every Haskell module whereas libraries must be explicitly imported. This division into prelude and libraries does not imply that the libraries are optional (they're not) or that the prelude cannot import library modules (it can). On the other hand, entities in the prelude can be considered "more essential" to the language. Students of Haskell would be expected to learn about the prelude before looking into the libraries.

It is slightly more convenient to use entities from the standard prelude than from a library. But, each entity placed in the prelude "steals" a potentially useful name from programmers. To avoid name clashes, programmers must give their own entities different names, or use hiding or renaming.

In order to justify "stealing a name" from the user, each entity in the prelude must satisfy one or more of the following criteria:

1. It is very heavily used by all programmers. For example, the existing functions `map` and `show` are so heavily used that programmers are unlikely to use the name for any other purpose.
2. It occurs in introductory functional programming courses/textbooks. For example, `interact` is not heavily used in large programs but including it in the prelude avoids or delays the need to teach students about Haskell's module system.

Experience with Haskell 1.2 suggests that arrays, rational numbers and complex numbers are not used heavily enough to justify their inclusion in the prelude. Therefore `PreludeArray`, `PreludeRatio` and `PreludeComplex` will be libraries in Haskell 1.3. Similarly, the functions `ord`, `chr`, `isControl`, `isPrint`, etc. in the module `Prelude` are rarely used and will be moved to module `LibCharType`.

2.3 What should go in each module?

Each library module comprises one of:

1. A single (abstract) data type. (Or a family of types and corresponding type class in the case of `LibWord`.)
2. A single type class.
3. A set of closely related utility functions.

   For example, operations on lists are divided into modules `LibLength`, `LibDuplicates`, `LibScan`, `LibSubsequences`, etc. rather then being grouped into a single module. Our goal is to keep libraries small and self-contained so that programmers can import those functions they need without importing many functions they don't need.

2.4 What classes should each new type be an instance of?

In Haskell 1.2, an instance of a class must be defined in either the module that defines the class or in the module that defines the instance. This rule makes it impossible for the programmer to add an instance if it has been omitted from the prelude or libraries. Therefore care must be taken to define every possible instance of every possible class to avoid leaving the programmer high and dry.

At the time of writing, it seems likely that this rule will be relaxed in Haskell 1.3 to allow a given instance to be defined anywhere in the program — provided there is at most one instance. Nevertheless, it is important to provide every possible instance for all abstract data types — and most reasonable instances for all other types.

The question of where to define an instance is also important. If instances are defined in the modules that defines the classes, importing a class might cause a large amount of unwanted code (associated with types that the programmer is not using) to come into scope. If instances are defined in the modules that define the types, importing a type might cause a large number of unwanted code (associated with classes the programmer is not using) to come into scope. Both seem to result in a very cluttered name space and large compiled programs. We don't have a good solution at the moment.

2.5 What should the type and name of each function be?

The primary goal in choosing names is that it should be possible to guess the purpose of a function from its name and type signature. In some cases it may be appropriate to change the names of existing prelude functions to achieve this goal (e.g. a better name for `null` would be `isEmpty`).

Secondary considerations include:

1. Consistency with the existing Haskell prelude.

   For example, modules `LibSet`, `LibBag` and `LibFiniteMap` all provide a function analogous to the prelude function `filter` to "select" values from a collection. We use the names `filterSet`, `filterBag` and `filterFM` for these functions.

2. Consistency between different libraries.

   For example, modules `LibSet`, `LibBag` and `LibFiniteMap` all provide a function to combine sets, bags or datatypes (respectively). We use similar names (`unionSet`, `unionBag` and `unionFM`) for all three functions.

   At the time of writing, it seems likely that Haskell 1.3 will provide a form of "qualified names" allowing one to import several entities with the same name and using the module name as a qualifier to resolve any ambiguity. If this proposal is adopted, both the `Lib` prefixes on the module names and the type suffixes on the function names would be dropped — the functions being called `Set.union`, `Bag.union` and `FiniteMap.union` respectively.

3. Consistency with the existing Haskell naming conventions.

   For example, identifiers formed by the concatenation of several words use capitalisation rather than underscores to separate the words.

4. We try to avoid using a name if a programmer might reasonably use the name for some other purpose.

In a language that encourages use of partial application and allows any binary function to be used as an infix operator, it is important to consider possible uses of a function when choosing the order of arguments. For example, a function which modifies an object of type `T` should take this object as the last argument. Thus `add` would have type `a -> Set a -> Set a` instead of `Set a -> a -> Set a`.

2.6 How should library functions be defined in a standard?

With the exception of primitive arithmetic and I/O-related functions, all functions in the Haskell 1.2 prelude are described in English in the body of the report and defined in Haskell in an appendix. Providing a Haskell definition avoids ambiguity but can be very verbose and hard to understand (see, for example, `PreludeText`).

For some functions, we might wish to be deliberately ambiguous: all a programmer needs to know about a sort function is whether it is stable and for which inputs it behaves efficiently — details about the choice of algorithm are best left to the implementor. For these functions, it is more appropriate to provide an English description of the function backed up by mathematical identities, error conditions and strictness properties as appropriate.

For all other functions (i.e., those simple enough that they are best specified in Haskell), the required semantics is precisely that of the definition — implementors are *not* free to change the semantics to improve performance.[1]

Some types such bitsets and random states we wish to leave the implementor free to choose an efficient representation but wish to constrain the behaviour sufficiently to guarantee portability. In these cases, a reference implementation is provided in Haskell but the representation is not exported from the defining module.

Dealing with the strictness properties of library functions is a particular problem. In many cases, the strictness of a function depends on the order in which tests for errors or special conditions are made. For some functions, laziness is crucial to the program; for others, it makes little difference. The standard can either be very precise about strictness or it can allow implementations to choose whatever strictness properties lead to the best implementation. We propose to explicitly mark functions for which the implementation is free to alter the strictness properties. Most of the others can be defined in Haskell, which precisely defines the strictness. To avoid introducing subtle portability problems, we plan to keep the number of functions with undefined strictness properties as small as possible.

As in the existing prelude, implementations are free to alter calls to the `error` function in library functions. Error messages may be changed freely to make them more useful. Library functions should detect errors as early as possible and report them clearly.

The libraries introduce a number of new types which are similar to existing types (e.g. `PackedString`s are similar to `String`s, `FiniteMap`s to association lists and arrays, `Set`s to lists). The relationship between two types can often be described by a pair of functions ($f :: U \to V, g :: V \to U$) such that $g \circ f = id$. By abuse, we call such pairs "retraction pairs" and write $(f, g) : U \leftrightarrow V$. (The complete definition of "retraction pairs" would require that $f \circ g \sqsubseteq id$ for an appropriate choice of domain.) When a retraction pair exists, it is natural to use the pair to define the semantics of functions over the new types in terms of corresponding functions over the old types. For example, the retraction pair for packed strings (`unpackPS`, `packString`) : `PackedString` $\leftrightarrow$ `String` can be used to define the functions

```
headPS :: PackedString -> Char
headPS = head . unpackPS

tailPS :: PackedString -> PackedString
tailPS = packString . tail . unpackPS

nullPS :: PackedString -> Bool
nullPS = null . unpackPS
```

When accompanied by a definition of the composition `unpackPS . packString`, this completely specifies the required behaviour of these functions — though one would hope for more efficient implementations!

[1]Many Haskell compilers break this rule for prelude functions. Since this reduces portability, we recommend that they provide a compiler option to force the use of a correct (but possibly slower) implementation — as far as is possible.

2.7 How do libraries interact with other aspects of Haskell?

Although the Haskell report is silent on exactly how the components of a program are gathered for compilation, we expect that the user should not have to do anything special to specify the location of modules in the standard Haskell library. Simply mentioning a library name in an `import` declaration ought to be sufficient to link the appropriate library into the Haskell program.

Libraries may define derivable classes. The names of such classes must be explicitly imported into the modules that define types which derive them. It is the responsibility of the compiler to correctly expand a `deriving` clause involving such a class.

It is also possible that the compiler-generated code for a `deriving` clause may reference entities defined in libraries. It it is not necessary for the programmer to import these implicitly-referenced entities, although the compiler must arrange for them to be linked into the resulting program.

3 An Overview of the Proposed Libraries

This section summarises those libraries which will be included in Haskell 1.3. (A complete definition may be found in document [11].) We omit `LibCharType` which provides character operations removed from `Prelude`, and `LibArray`, `LibComplex` and `LibRatio` which are just old prelude modules turned into libraries.

3.1 Non-overloaded prelude functions

Functions such as `elem` use the `Eq` class to supply the `==` operation. There are situations in which the operation defined by overloading is not appropriate. For example, one might wish to use a case-insensitive comparison when operating on strings. It is straightforward to define versions of these functions which are not overloaded (but are polymorphic) by adding an extra argument which provides an explicit equality or comparison predicate. For example, we have `nubBy :: (a -> a -> Bool) -> [a] -> [a]`. While it is trivial to define overloaded functions in terms of non-overloaded ones, the reverse is not possible.

The prelude provides several functions which are overloaded with respect to `Eq` and `Ord`: `nub`, `elem`, `notElem`, `min`, `max`, `maximum`, `minimum` and `(\\)`. Rather than creating a module containing a random assortment of such functions, we place the non-overloaded version in the same module as the original definition. The overloaded version can be defined using the new function; for example, `nub` can be defined by `nub = nubBy (==)`.

3.2 Packed Strings

Haskell represents strings by lists of characters. While this interacts well with lazy evaluation and allows many prelude functions (such as `map` and `length`) to be used on strings, typical Haskell implementations consume 20–40 bytes per character to represent such strings.

The module `LibPackedString` provides a new type `PackedString` which is evaluated more strictly to allow a more compact representation (as low as 1 byte per character plus a small constant overhead).

The retraction pair (`unpackPS`, `packString`) : `PackedString` $\leftrightarrow$ `String` is not quite an isomorphism since `packString` completely evaluates its argument. These functions are used to define `PackedString` versions of the prelude constructors `[]` and `(:)` and the prelude functions `head`, `tail`, `init`, `last`, `null`, `length`, `append`, `map`, `filter`, `foldl`, `foldr`, `take`, `drop`, `splitAt`, `takeWhile`, `dropWhile`, `span`, `break`, `lines`, `words`, `reverse`, `concat`, `elem` and `(!!)`.

This module is based on the `PackedString` module distributed with GHC.

3.3 List Operations

As one of the most heavily used data structures in Haskell, it is not surprising that the last five years use has produced a host of useful new functions over lists.

`LibSort`

This module provides a *stable* sorting function `sort :: Ord a => [a] -> [a]`. (`sortBy` is also provided.)

`LibDuplicates`

This module provides functions for manipulating lists with duplicate values: `group :: Eq a => [a] -> [[a]]` and `uniq :: Eq a => [a] -> [a]` which group together adjacent equal elements in a list and eliminate adjacent equal elements. (The function `LibSort.sort` can be used to bring duplicates together.) (`groupBy` and `uniqBy` are also provided.)

`LibLength`

This module provides functions such as `lengthLe, lengthEq :: [a] -> Int -> Bool` to test the length of a list without evaluating the entire list.

`LibScans`

This module provides unidirectional and bidirectional generalisations of `fold` and `scan` based on functions provided in HBC's ListUtils module, GHC's Utils module and O'Donnell's paper on bidirectional fold and scan [9].

`LibSubsequences`

This module provides functions to generate the list of all subsequences, prefixes, suffixes or permutations of a list and to test whether one list is a subsequence, prefix, suffix or permutation of another list. These are based on functions defined by Bird and Wadler [2].

3.4 Collections

Lists are very heavily used in Haskell programs. Sadly, lists can be quite inefficient (in time) for storing large collections of data and it is possible to do significantly better using data structures based on binary trees or hash tables.

The modules `LibBag`, `LibSet`, `LibFiniteMap` and `LibHashTable` define types `Bag`, `Set`, `FiniteMap` and `HashTable`, retraction pairs relating them to lists or association lists and functions over these types (mostly based on prelude functions over lists and arrays).

`Bag` α

The type `Bag` α is isomorphic to $[\alpha]$ but provides constant time appending and concatenation functions and logarithmic time `head` and `last` functions.

The retraction pair for bags (`toList`,`fromList`) : `Bag` $\alpha \leftrightarrow [\alpha]$ forms an isomorphism. That is

```
toList . fromList = id
```

`Set` α

The type `Set` represents (ordered) collections with no duplicates.

The retraction pair for sets (`toList`,`fromList`) : `Ord` $\alpha \Rightarrow$ `Set` $\alpha \leftrightarrow [\alpha]$ satisfies the identity

```
toList . fromList = uniq . sort
```

`FiniteMap` α β and `HashTable` α β

The types `FiniteMap` α β and `HashTable` α β represent lookup tables with keys of type α and elements of type β. `FiniteMap`s behave like balanced binary trees (logarithmic access time and insertion time) and `HashTable`s behave like (functional) hash tables (near-constant access time and linear insertion time).

The retraction pair for finite maps (`toList`,`fromList`) : `Ord` $\alpha \Rightarrow$ `FiniteMap` α $\beta \leftrightarrow [(\alpha,\beta)]$ satisfies the identity

```
toList . fromList = uniqBy eq . sortBy cmp
 where
  (x,_) `cmp` (y,_) = x <= y
  (x,_) `eq` (y,_) = x == y
```

Hash tables require a new type class `Hashable`. The following laws are satisfied by the retraction pair for hash tables (`toList`,`fromList`) : `Hashable` $\alpha \Rightarrow$ `HashTable` α $\beta \leftrightarrow [(\alpha,\beta)]$

```
map fst xs `isPermutationOf` map fst (toList (fromList xs))
lookup xs = lookup (toList (fromList xs))
```

All four modules provide functions corresponding to the prelude constructors `[]` and `(:)` and of the prelude functions `++`, `\\`, `length`, `genericLength`, `map`, `partition`, `filter`, `foldl` and `foldr`. `LibSet` and `LibBag` also provide versions of `elem` and `notElem`; `LibBag` provides versions of `head`, `tail`, `init`, `last`, `(!!)` and `reverse`; and `LibFiniteMap` and `LibHashTable` provides versions of `indices`, `elems`, `accum`, `(//)`, `(!)`, `amap` and `ixmap`. (All functions are defined using the retraction pairs.)

The module `LibHash` is provided to help support `LibHashTable`. It provides a new abstract type `Hash`, a new type class `Hashable` defining a method `hash :: Hashable a => a -> Hash` and instances for Haskell's primitive types (e.g., `Int`, `Integer`). Instances of `Hashable` may be derived.

Modules `LibBag, LibSet` and `LibFiniteMap` are based on modules distributed with GHC; `LibHash` is loosely based on a module distributed with HBC (the type `Hash` is just `Int` in the HBC version.)

3.5 Monads

Since their first use in pure functional programming [13], monads have revolutionised the way programmers perform input/output, update-in-place, parsing, exception handling and many other tasks.

If constructor classes [6] are added to Haskell 1.3, it would be possible to define constructor classes representing monads, monads with a zero element and monads with a choice operator. Such classes could be defined in a library but are sufficiently important to justify their addition to the prelude. Instances would include lists, `Maybe`, `Either`, `IO` and `Parser`. There are a number of useful monadic functions which can be defined using these operations. The current interface is based on monads used within GHC and on examples distributed with Gofer.

3.6 Mutable Structures

Peyton Jones, Wadler and Launchbury [7, 8] describe how monads may be used to provide mutable variables and mutable arrays without losing referential transparency.

The module `LibMutable` provides both mutable variables and mutable arrays. The major unresolved question is whether the operations to create, read and write mutable structures should be part of the `IO` monad, part of a state thread monad or whether the `IO` monad should be an instance of state thread monads as in Peyton Jones and Launchbury's elegant lazy state threads paper [8]. The problem is that their approach requires the addition of a new language construct `runST` with special type-checking rules. It is not yet clear whether the extra power justifies complicating the language. This module is based on the `PreludeGlaST` module distributed with GHC.

3.7 Printing and Parsing

The prelude provides the `Text` class for printing and parsing values. Derived methods have the desirable property that `read . show = id` (for non-functions and ignoring strictness). However, the output from `show` can be rather ugly and it is awkward to construct good parsers using `read`.

The module `LibPretty` provides a new abstract type `Pretty` representing a pretty-printed block of text and functions for combining values of type `Pretty` in various ways useful when printing programs and data structures. The current interface is based on Hughes' pretty-printing library as distributed with HBC and GHC.

The module `LibParse` provides a new abstract type `Parser` α β of backtracking recursive-descent parsers which consume a token stream of type α and produces "parse trees" of type β. The current interface is based on Hutton's parsing library [5] as distributed with HBC.

3.8 Binary Files

The class `Text` provides a limited form of persistence: most built in and user-defined types can be printed to text files and subsequently read back in. However, the process of converting values to strings and using a backtracking lexer and parser to read them back in is notoriously inefficient. Haskell 1.2 provided the `Bin` datatype and the `Binary` type class for efficiently writing values to files in a more concise (implementation-specific) manner.

This feature of Haskell 1.2 was essential to some programs but rarely used and so is being moved into a library. At the same time, the adoption of monadic I/O makes it possible to provide operations to read and write values to a binary file directly — eliminating the need to create an intermediate `Bin` value.

To ensure portability of generated files, the specification of this module should define the precise external representation for each datatype in class `Binary`. Some care is required to avoid overly restricting the possible range of certain datatypes such as `Int` and `Float`.

Instances of `Binary` may be derived. The current interface is based on the Native module distributed with HBC with extensions based on the current Yale implementation.

3.9 Random Numbers and Splittable Supplies

While functional programming languages are valued because their results are deterministic and easy to predict, many kinds of programs such as simulations and games need to appear non-deterministic — they need a supply of random numbers.

The module `LibRandom` provides a new abstract type: `RandomState` together with a function `nextRandomState :: RandomState -> RandomState`; and a new type class `Random` which provides a method `fromRandomState :: Random a => RandomState -> a`. There are also functions for initialising random states, and generating lists of random values and `Text` and `Binary` instances for reading and writing random states to files.

Although the type `RandomState` is abstract, the definition will precisely specify the algorithm used to generate random numbers to ensure consistent results across all implementations.

The current interface and semantics is loosely based on the random number operations in Common Lisp.

The module `LibSupply` provides a new abstract type `Supply` α which represents a splittable supply of values of type α. This module is included to support supplies of random numbers but has also proved useful within compilers. The current interface is based on Augustsson, Rittri and Synek's splittable supply library [1] as distributed with HBC.

The price paid for an efficient splittable supply implementation is that programs using the splittable supply module might produce different results when compiled with different compilers, with different optimisation options or even after small semantics-preserving changes to the program. In practice, this does not present any problems. For example, in a type-checker, it is important to have an efficient supply of distinct type variables but it is irrelevant where each type variable is used.

3.10 BitSets

Bitwise operations are useful for two purposes: 1) they provide fast, compact operations on sets of small integers; and 2) they are useful in communicating with hardware devices, network protocols, etc.

The module `LibBitSet` provides types `Word8`, `Word16`, `Word32`, `Word64`, `Word128` and `Word` (unbounded words) and a type class `BitSet` which provides the usual bit-manipulation operations (including both arithmetic and logical shifting operations). The new types are instances of `Integral` (and all superclasses); negative numbers are interpreted as though they are represented in two's-complement notation. The current interface is loosely based on the Common Lisp logical operations [12] and the `Word` library distributed with HBC.

3.11 Future Work

There is a considerable amount of work ahead designing the exact interface to these modules, defining the precise semantics of these operations and documenting the resulting modules.

We are also considering libraries to support matrix operations, regular expressions, Hughes' lazy memo functions [4], Johnsson's lazy arrays, the language independent arithmetic standard [10] and the X11 graphics protocol.

References

[1] L Augustsson, M Rittri, and D Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, January 1994.

[2] RS Bird and PL Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[3] AD Gordon and K Hammond. Monadic I/O in Haskell. In *Proceedings of Haskell Workshop*, June 1995.

[4] John Hughes. Lazy memo-functions. In *Functional Programming and Computer Architecture*, pages 129–146, September 1985.

[5] G Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[6] MP Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Functional Programming and Computer Architecture*, 1993.

[7] SL Peyton Jones and PL Wadler. Imperative functional programming. In *Principles of Programming Languages*, pages 71–84. ACM, January 1993.

[8] J Launchbury and SL Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, 1994.

[9] JT O'Donnell. Bidirectional fold and scan. In *Draft Proceedings of Glasgow Functional Programming Workshop*, pages XX1 – XX6. Glasgow Functional Programming Group, July 1993.

[10] M Payne, C Schaffert, and BA Wichmann. The language compatible arithmetic standard. *ACM SIGPLAN Notices*, 25(1):59–86, January 1990.

[11] AD Reid and JC Peterson. A proposal for the standard Haskell libraries. 1995. In preparation for distribution at Haskell Workshop.

[12] GL Steele. *Common Lisp — The Language.* Digital Press, 2nd edition, 1994.

[13] PL Wadler. Comprehending monads. In *Proc ACM Conference on Lisp and Functional Programming, Nice.* ACM, June 1990.