LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

pH Language Reference Manual, Version 1.0—preliminary

Computation Structures Group Memo 369
January 31, 1995

Rishiyur S. Nikhil[00], Arvind[01], James Hicks[11], Shail Aditya[01], Lennart Augustsson[10], Jan-Willem Maessen[01], Yuli Zhou[01]

[00]Digital Equipment Corp, Cambridge Research Lab [01]MIT Lab for Computer Science, [11]Motorola Cambridge Research Center [10]Chalmers University of Technology, Goteborg

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

pH Language Reference Manual, Version 1.0—preliminary

Rishiyur S. Nikhil[00], **Arvind**[01], **James Hicks**[11], **Shail Aditya**[01],
Lennart Augustsson[10], Jan-Willem Maessen[01], Yuli Zhou[01]

Revised January 31, 1995

---

1 Introduction

pH is a parallel language obtained by extending Haskell. This document must be read in conjunction with the Haskell Report [2] since it describes only the extensions.

In Section 2 we present the syntax extensions proper. In Section 3 we present some examples to give a flavor of the language. In Section 4 we present commentary and rationale.

Future versions of this document will address topics such as loop pragmas, data and work distribution, *etc.*

Background

Haskell appears to be achieving its original goal of becoming the standard non-strict functional programming language. Current implementations of Haskell are based on sequential, lazy evaluation (except for the parallel implementation at Glasgow on GRIP).

Id [4] is, and has always been, a parallel language. It is not a purely functional language, but has a non-strict functional language at its core, and the style of programming is predominantly the same as in Haskell. Id has a so-called "lenient" evaluation order rather than lazy evaluation, for two reasons: for more parallelism, and to give meaning to its side-effecting constructs (I-structures, M-structures).

In August 1993, a group of people– Arvind (MIT), Lennart Augustsson (Chalmers), James Hicks (Motorola MCRC), Simon Peyton Jones (Glasgow), Rishiyur Nikhil (DEC CRL), Joe Stoy (Oxford) and John Williams (IBM Research)– met in Cambridge, Mass., to discuss the possibility of merging Id and Haskell into a parallel dialect of Haskell, in order to improve the dialogue between the Id and Haskell communities, to share infrastructure (compilers, systems, application programs), and to facilitate interesting research topics, such as comparing lazy evaluation *vs.* lenient evaluation, parallel programming in a pure functional language *vs.* in a language with I- and M-structures, *etc.*

After extensive discussions on the differences between Id and Haskell, we decided that such a merger was feasible, and we decided to pursue it, calling this new dialect "pH".

---

[00]Digital Equipment Corp, Cambridge Research Lab [01]MIT Lab for Computer Science, [11]Motorola Cambridge Research Center [10]Chalmers University of Technology, Goteborg

2 Haskell Extensions for pH

The extensions here are expressed as changes to the formal syntax in Appendix B of the Haskell Report.

pH is layered (like Id):

- Functional core
- I-structure extensions
- M-structure extensions

These layers are distinguished syntactically and by type rules (like Id), so that it is easy for a compilation system to force programs to be restricted to a desired layer (purely functional, purely functional with I-structures, or unrestricted).

2.1 Functional core of pH: Haskell plus loop notation

The notation for the functional core of pH is identical to Haskell, with a few extensions for loops.

Haskell:

$$
\begin{array}{lcll}
exp^{10} & \rightarrow & \texttt{\textbackslash}\ \ldots & \\
 & | & \texttt{let \{ } decls\ [\texttt{;}]\ \texttt{\} in } exp & \text{(let expression)} \\
 & | & \texttt{if } \ldots & \\
 & | & \texttt{case } \ldots & \\
 & | & fexp & \\
 & & & \\
aexp & \rightarrow & var & \text{(variable)} \\
 & | & \ldots & \\
 & & & \\
apat & \rightarrow & var\ [\ \texttt{@}\ apat\ ] & \\
 & | & \ldots &
\end{array}
$$

Extensions for pH:

$$
\begin{array}{lcll}
exp^{10} & \rightarrow & \ldots & \\
 & | & \texttt{while } exp\ \texttt{do \{ } decls\ [\texttt{;}]\ \texttt{\} finally } exp & \text{(while expression)} \\
 & | & \texttt{for } pat\ \texttt{<- } exp\ \texttt{do \{ } decls\ [\texttt{;}]\ \texttt{\} finally } exp & \text{(for expression)} \\
 & & & \\
aexp & \rightarrow & \ldots & \\
 & | & \texttt{next } var & \text{(loop-carried var)} \\
 & & & \\
apat & \rightarrow & \ldots & \\
 & | & \texttt{next } var & \text{(loop-carried var)}
\end{array}
$$

Semantics of loops

We explain for-loops in terms of a translation to while-loops, and we explain while-loops in terms of a translation to a tail-recursive function definition and invocation.

Given a for-loop:

```
>   for pat <- exp do {
>       decls
>   }
>   finally exp
```

Let `xs` be a new variable. Then the for-loop is equivalent to:

```
>   let
>       xs = exp
>   in
>       while (xs /= []) do {
>           pat:next xs = xs
>           decls
>       }
>       finally exp
```

The *decls* list in a loop is called the "loop body". A `next` var may only be bound in a top-level *decl* of a loop body (in an *apat*). Such variables are called "next-ified", "circulating" or "loop-carried" variables. Their scope, like any other var bound in a top-level *decl* in the loop body, is the entire loop body. A `next` var may also be used in an expression like a normal variable.

Given a while-loop:

```
>   while ePred do {
>       decls
>   }
>   finally eFinal
```

Let x_1, ..., x_N be the next-ified variables bound in *apat*'s in the top-level of *decls*. For each x_i, let *next_*x_i be a corresponding new variable. Also, let `loop` be a new variable. Then the while-loop is equivalent to:

```
>   let
>       loop x1 ...  xN = if ePred then
>                                 let
>                                     decls'
>                                 in
>                                     loop next_x1 ...  next_xN
>                             else
>                                 eFinal
>   in
>       loop x1 ...  xN
```

where *decls'* is identical to *decls* except that each occurrence of `next` x_i is replaced by *next_*x_i, both in *apat*'s and in *exp*'s.

2.2 I- and M-structure extensions

2.2.1 I- and M-structure semantics

An I- or M-structure field in a data structure is an updateable field. However, unlike the "raw" reads and writes in conventional languages, the accesses are combined with synchronization, so that they are relatively easy to use in a parallel environment.

For both I- and M-structure fields, the field may be in one of two states: Full and Empty.

An I-field is initially in the Empty state. When assigned, it goes to the Full state. If a second assignment to an I-field is attempted, the entire program is in error. An expression that reads an I-field simply returns the value when it becomes Full. The use of I-structure fields does not compromise the Church-Rosser property—programs are still guaranteed to be deterministic.

An M-field is initially in the Empty state. When assigned, it is in the Full state. An expression that reads an M-field performs two actions atomically: it returns the value in the field when it becomes Full, and resets it to the Empty state. Thus, amongst multiple expressions attempting to read a Full M-field, only one of them can succeed in reading the value (it is non-deterministic as to which one succeeds). The other reads must wait until the field is assigned again. M-fields allow one-at-a-time access to its contents. Assignments are only allowed on Empty fields; if a second assignment to a Full M-field is attempted, the entire program is in error. The Church-Rosser property no longer holds when M-structure fields are used.

M-fields have two additional operations that are convenient compositions of normal M-field reads and writes. The *examine* operation returns the value of the field when Full, and leaves it Full. This is equivalent to a read and a write that just writes back the value. The *replace* operation replaces the value in a Full field by a new value. This is equivalent to a read followed by a write that writes back the new value. Note that these are not conventional "raw" reads and writes— they both respect the synchronization semantics of M-fields, and will both wait on an Empty M-field.

Referential transparency no longer holds with the use of I-structures and/or M-structures. Further, the polymorphism of I- and M-fields is restricted in exactly the same way as "`ref`" types in SML (although SML's updateable cells do not have any synchronization semantics).

2.2.2 Single I- and M-structure cells

The following primitives are for constructing, reading from, and writing to single cells with I-structure and M-structure semantics:[1]

```
>    -- New prelude primitives
>
>    data ICell a
>
>    iCell    :: a        -> ICell a                -- Constructor
>
>    iFetch   :: ICell a  -> a
>
>    iStore   :: ICell a  -> a        -> ()
>
>
>    data MCell a
>
>    mCell    :: a        -> MCell a                -- Constructor
>
>    mFetch   :: MCell a  -> a
>    mStore   :: MCell a  -> a        -> ()
>
>    mExamine :: MCell a  -> a
>    mReplace :: MCell a  -> a        -> ()
```

[1] Single assignable cells are reminiscent of `ref` cells in SML.

Empty cell construction The `iCell` and `mCell` constructors each take a single argument v and produce an I-structure object and M-structure object, respectively, containing v.

Optionally, the cells may be constructed in an "empty" state by applying them to the special expression "_", described by the following syntactic extension.

Haskell:

aexp	$\rightarrow$	*var*	(variable)
	$\vert$	...	

Extension for pH:

aexp	$\rightarrow$	...	
	$\vert$	_	(Empty field)

2.2.3 I- and M-structure array construction

There are two new "array comprehension" contructor functions in the prelude, for I- and M-structure arrays:

```
>    data    (Ix a) => Iarray a b
>    data    (Ix a) => Marray a b
>
>    iarray         ::  (Ix a) => (a,a) -> [Assoc a b] -> Iarray a b
>    marray         ::  (Ix a) => (a,a) -> [Assoc a b] -> Marray a b
```

An empty I- or M-structure array can be constructed simply by applying these constructors to an empty list of index-value associations.

The semantics of the `array` "array comprehension" constructor function is changed for pH:

In Haskell:

- If there is more than one index-value pair for the same index, that element goes to $\perp$
- If any index-value pair has its index out of bounds, the entire array goes to $\perp$

In pH:

- If there is more than one index-value pair for the same index, the entire program goes to $\top$
- If any index-value pair has its index out of bounds, then that assignment goes to $\perp$

2.2.4 I- and M-structure array component selection

New prelude infix operators for array component selection:

```
>    infixl  9  !.         -- I-structure fetch
>    infixl  9  !^         -- M-structure fetch
>    infixl  9  !^^        -- M-structure examine
>
>    (!.)    :: Iarray a b  ->  a  ->  b
>    (!^)    :: Marray a b  ->  a  ->  b
>    (!^^)   :: Marray a b  ->  a  ->  b
```

2.2.5 I- and M-structure array component assignment

New prelude functions for array component assignment:

```
>    iastore   :: Iarray a b  ->  a  ->  b  ->  ()
>    mastore   :: Marray a b  ->  a  ->  b  ->  ()
>    mareplace :: Marray a b  ->  a  ->  b  ->  ()
```

2.2.6 Syntax extensions for I- and M-structure array component assignment

One frequently finds an I- or M-structure assignment as part of a "dummy" binding in a declaration:

```
>   let
>        decls
>        ...
>        _ = iastore eA  eJ  eV
>        ...
>        decls
>   in
>        e
```

The following syntax extension allows us to omit the annoying "_ =" prefix.

Haskell:

decl	→	*vars* :: *[context* => *] type*	
	|	*valdef*	

Extensions for pH:

decl	→	*vars* :: *[context* => *] type*	
	|	*valdef*	
	|	*exp*	for effect only

The following syntax extension goes further and makes the assignment really look like an assignment.

exp	→	*lval sel* = *exp*	
lval	→	*lval sel*	
	|	*exp*	
sel	→	`!.` *exp*	Array I-str
	|	`!^` *exp*	Array M-str store
	|	`!^^` *exp*	Array M-str replace

In particular,

`eA!.eJ  = eV`	is equivalent to	`iastore`	`eA`	`eJ`	`eV`
`eA!^eJ  = eV`	is equivalent to	`mastore`	`eA`	`eJ`	`eV`
`eA!^^eJ = eV`	is equivalent to	`mareplace`	`eA`	`eJ`	`eV`

2.3 Sequencing in decl lists

Haskell:

decls	$\rightarrow$	*decl*$_1$; ... ; *decl*$_n$	$n \geq 0$
decl	$\rightarrow$	*vars* :: [*context* =>] *type*	
	$\mid$	*valdef*	

Extension for pH:

decl	$\rightarrow$	*vars* :: [*context* =>] *type*
	$\mid$	*valdef*
	$\mid$	`par {` *decls* `}`
	$\mid$	`seq {` *decls* `}`

`seq` and `par` are only control constructs; they do not introduce any new scope.

By default, the top-level decl lists (in `let` and `where` blocks and loop bodies) are parallel.

2.4 Layout ("off-side" rule)

The layout rule of Haskell allows one to omit braces and semicolons in a decl list. Decl lists follow the `where`, `let` and `of` reservedids.

Extension for pH: The lay-out rule also applies to the decl lists that follow these new reservedids:

`do`	(in loops)
`seq`	(sequenced declarations)
`par`	(parallel declarations)

3 Examples

This section shows a number of program fragments so the reader can get a flavor of the syntax extensions.

Loops

Integration of a function `f` in the range `x1` to `x2`:

```
>    let s = 0.0
>        x = x1
>    in
>        while (x <= x2) do
>           next s = s + f(x)
>           next x = x + delta_x
>        finally s
```

Factorial of `n`

```
>    let  f = 1
>    in
>       for j <- [1..n] do
>          next f = f * j
>       finally f
```

I-structure cells

"Open lists" (related to "difference lists" in Prolog):

```
>    data  olist a  = ONil | OCons    a    (iCell (olist a))
>
>    is_OCons             :: olist a -> Bool
>    is_OCons  ONil        = False
>    is_OCons  (OCons x y) = True
>
>    grow_olist                   :: olist a -> olist a -> olist a
>    grow_olist  (OCons x y)  ol2  = let
>                                      iStore  y  ol2
>                                    in
>                                      ol2
```

Tail-recursive "map", using open lists:

```
>    omap                 ::  (a -> b) -> (olist a) -> (olist b)
>
>    omap  f  ONil          =  ONil
>    omap  f  (OCons x ixs)  =  let
>                                ys     = OCons  x  (ICell _)
>                                xs     = iFetch  ixs
>                                lastys = while (is_OCons xs) do
>                                           OCons x ixs = xs
>                                           next xs     = iFetch ixs
>                                           next ys     = grow_olist  ys
>                                                                    (OCons x (iCell _)
>                                         finally ys
>                                grow_olist  lastys  ONil
>                              in
>                                ys
```

M-structure cells

Updateable lists

```
>    data  mlist a  = MNil | MCons    a    (mCell  (mlist a))
```

Destructively insert an integer into a sorted list

```
>    insert                            :: a -> mlist a -> mlist a
>
>    insert  y  MNil                    = MCons  y  (mCell MNil)
>    insert  y  (MCons x mxs) | y <= x    = MCons  y  (MCons x mxs)
>                             | otherwise = let
>                                             xs'  =  mFetch mxs
>                                             mStore mxs (insert y xs')
>                                           in
>                                             Mcons x mxs
```

I-structure arrays

Given an Iarray A of integers, produce an Iarray B containing all negative numbers followed by all non-negative numbers:

```
>    let
>       (l,u) = ibounds a
>       b     = iarray (l,u) []
>       kl    = l
>       ku    = u
>       for j <- [l..u] do
>          (j1, next kl, next ku)  = if (a!.j < 0) then (kl, kl+1, ku)
>                                                else (ku, kl, ku-1)
>          b!.j1                   = a!.j
>       finally ()
>    in
>       b
```

M-structure arrays

Construct a histogram of a tree of numbers in the range 1 to 10:

```
>    data  tree = Empty | Node int tree tree
>
>    hist  t   =  let
>                    h = marray (1,10) [ (j := 0) | j <- [1..10]]
>                    traverse  t  h                                   -- for effect
>                 in
>                    h
>
>    traverse  Empty         h  =  ()
>    traverse  (Node j l r)  h  =  let
>                                    traverse  l  h                   -- for effect
>                                    traverse  r  h                   -- for effect
>                                    h!^j = h!^j + 1                  -- atomic incr
>                                  in
>                                    ()
```

Note: all three statements in the last `let` block execute in parallel.

Sequencing

In the histogram program above (M-structure arrays), there is a problem: we should not return the histogram until the traversal is completed, else a consumer may read an intermediate value of the histogram. This can be accomplished using sequencing.

```
>    hist  xs  =  let
>                    h  = marray (1,10) [ (j := 0) | j <- [1..10]]
>                    seq  traverse  xs  h
>                         h1 = h
>                 in
>                    h1
```

4 Commentary

This section is a collection of comments on various aspects of pH, in no particular order of importance.

4.1 Comment: compatibility with Haskell

Will all Haskell programs parse *as pH programs?*

Yes, except for the following situation. Even though functional pH uses Haskell notation with no changes, the addition of new *reservedid*'s means that a normal Haskell program that happened to use one of these as an ordinary identifier/symbol would have to be modified.

New *reservedid*'s:

for while do finally next
par seq

Will all Haskell programs run *as pH programs?*

This is mainly a termination issue, *i.e.,* when a program terminates under both Haskell and pH, both implementations will produce the same answer.

The termination issue is complicated, and we'd rather not get into it here. Briefly:

- For a strict implementation of pH: some Haskell programs will deadlock or not terminate.
- For a non-strict implementation of the functional core of pH with a fair scheduler: all Haskell programs will terminate with the same answers.
- For non-strict implementations of pH without a fair scheduler: some Haskell programs will not terminate.

4.2 Comment: sequencing

Why have sequencing at all? Aren't they meaningless in a functional language?

In the functional core and the I-structure layer, sequencing is only important if you wish to control the space complexity (for which, of course, you have to know something about the underlying implementation).

With M-structures, sequencing is sometimes necessary to control determinacy (and even absence of visible errors).

Why sequencing in decl lists, rather than expr lists?

In Scheme, Lisp and ML, there are typically two kinds of sequencing constructs. First, there is sequencing in expression lists: `(begin e1 ... eN)` in Scheme, `(progn e1 ... eN)` in Lisp and `(e1;...;eN)` in SML; each contains a list of expressions to be executed in sequence, and the value `eN` is returned. Second, there is sequencing in declarations: `let*` in Scheme and Lisp, and semicolon-separated declarations in SML.

Both forms are useful, although we have generally found the latter to be more so, as we often want to bind and use the results of intermediate expressions in the sequence. In a sense, our proposal unifies the two: sequencing is done in decl lists, but a bare expression is allowed in place of a decl when we don't wish to bind its result to anything. However, note that unlike sequenced declarations in Scheme, Lisp and SML, our proposal does not involve any new scope rules, *i.e.,* scoping is entirely orthogonal (controlled only by the initial `let`).

4.3 Comment: Parseability

The grammar, as shown in the proposals above, is almost certainly not LALR(1). However, the only place where parsing difficulties may arise is in decl lists, where we now allow bare expressions in addition to decls. We repeat the proposed rule here, for reference:

decl	$\rightarrow$	*vars* :: *[context* => *] type*	
	$\mid$	*valdef*	
	$\mid$	*exp*	for effect only

In Id, we have been able to handle this as follows. First, we use the following grammar instead, which collapses the last three productions above into one:

decl	$\rightarrow$	*vars* :: *[context* => *] type*
	$\mid$	*exp*

and treats "=" initially as an ordinary infix operator (of lowest precedence). When the *decl* $\rightarrow$ *exp* rule is reduced, the associated semantic action examines *exp* to see if it has a "=" at the toplevel, and fixes up the parse tree accordingly.

4.4 Comment: Strictness and semantics of array constructors

The Haskell `array` constructor function is strict in all the indices of the array. This is because `array` has to

- Check for out-of-bounds indices, which cause the whole array to go to $\perp$
- Check for duplicate indices, which cause that component to go to $\perp$

We have changed the semantics of `array` in the interest of parallelism. The proposed semantics allows the implementation to return the array before *any* index has been computed.

We expect very little practical difference to the programmer. For an out-of-bounds index, the programmer will always see an error, both in Haskell and in pH. For duplicate indices, the pH programmer will always see an error. The Haskell programmer will always see an error if that component of the array is demanded, and in some implementations may see that error even if that component of the array is not demanded.

4.5 Comment: Loops

Why introduce loop notation? Isn't tail recursion enough?

We make an analogy with list comprehensions, which do not add any fundamental power to Haskell, but are convenient for certain common computations on lists involving maps and filters.

Similarly, loops do not add any fundamental power to Haskell/pH, but they are convenient for certain common computations involving reductions (folds on lists, folds on arrays, and other reductions).

4.6 Comment: Polymorphism and I- and M-structures

Is the Hindley-Milner type system used by Haskell still sound, with the addition of I- and M-structures?

No. I- and M-structures introduce the problem of "ref types", which is also seen in ML. There are a variety of solutions in the literature, in existing ML implementations, and in the existing Id implementation, any of which should carry over directly. sectionModules

The module system in pH is identical to the one used in Haskell.

5 Basic Types

Initial commentary

The existing basic types in Haskell exist essentially unchanged in pH. This section will briefly discuss the differences which do exist, and which have not been mentioned in preceding sections on syntax. The ultimate reference on such matters continues to be the Standard Prelude to pH, which is presented (skeletally) in Appendix A.

Implementation decisions are not implied

Note that there are a few places where Haskell has made design decisions which might overly constrain the implementation of the resulting functions. For example, the Haskell report gives the following definition of the boolean and operation `(&&)`:

```
> True  && x              =  x
> False && x              =  False
```

This definition implies (though it is not explicitly stated) that `(&&)` is strict in its first argument and non-strict in its second. This has implications for the error behavior of programs. In this case, we cannot assume from the above definition that the following expression has the value False, since x may never be well defined:

```
x && False
```

In general, overspecifying the behavior of such functions is to be avoided. Thus, the behavior of the code in exceptional cases should always be suspect, and if several correct implementations will give different results then the programmer should assume that any such implementation might be used. Thus, in the above example, the programmer cannot assume that either of the following are False unless she knows that x will produce a value:

```
x && False
False && x
```

Conversely, the implementor is free to choose any correct implementation of the functions given in the Prelude (for example, a parallel *and* which would cause *both* of the above expressions to terminate).

New Basic Types

Two new (and useful) basic types have been added to the pH prelude. They express common "either/or" values. Both types are slated to appear in Haskell 1.3, and the pH prelude will incorporate the operations on both types which appear in the new Haskell prelude. At the moment, the provided functionality is a best guess based on existing implementations and upon what was useful in actual code.

```
In PreludeCore:
> data Maybe a    = Nothing | Just a   deriving (Eq, Ord, Text)
> data Either a b = Left a  | Right b  deriving (Eq, Ord, Text)

In Prelude:
> errMaybe                      :: String -> Maybe a -> a
> errMaybe msg Nothing          =  error msg
> errMaybe _   (Just r)         =  r
```

```
>
> lift                        :: (a -> b) -> (Maybe a -> Maybe b)
> lift _ Nothing              =  Nothing
> lift f (Just x)             =  Just (f x)
>
> lift2                       :: (a -> a -> a) -> (Maybe a -> Maybe a -> Maybe a)
> lift2 _ Nothing r           =  r
> lift2 _ l Nothing           =  l
> lift2 f (Just l) (Just r)   =  Just (f l r)
>
> maybe                       :: b -> (a -> b) -> (Maybe a -> b)
> maybe b _ Nothing           =  b
>
> either                      :: (a -> c) -> (b -> c) -> (Either a b -> c)
> either f _ (Left a)         =  f a
> either _ g (Right b)        =  f b
```

5.1 Lists

In the interests of efficiency, pH includes a number of functions on lists which did not exist in Haskell; in the interests of correctness, several functions leading to infinite computations have been omitted.

Because pH is a parallel programming language, and because lists are inherently linear data structures, a large proportion of list operations are not terribly parallel. Several functions are provided which are intended to make it simpler to write parallel list operations by placing the construction and traversal of lists under wraps. List comprehension syntax makes it particularly easy to generate efficient code when using lists. Implementations should use the most efficient means available to compile code with lists, and should eliminate them whenever possible, most notably in the case of array comprehensions.

Of the new list functions, the most confusing is `reduce`. It has the following signature:

```
> reduce                        :: (a -> a -> a) -> a -> [a] -> a
```

`reduce` is in effect a directionless fold—its first argument must be an associative function, and the secod argument must be a zero of that function. The elements of the list are combined so that the following identities hold:

```
reduce f z (a ++ b)      ==     (reduce f z a) `f` (reduce f z b)
reduce f z [e]           ==     e
reduce f z []            ==     z
```

The implementation is permitted to perform the reduction in any order (thus the requirement that the function provided be associative) and is permitted to introduce the zero at any point in the computation, or may ignore it entirely (thus the requirement that the zero be an actual zero of the given function). These restrictions permit a large number of compiler optimizations. The programmer should be able to assume that `reduce` is always *at least* as efficient as the equivalent code written using `foldl` or `foldr`.[2]

The remaining list functions are straightforward. The `someOrder` function returns a (not necessarily random) permutation of its argument and is essentially a declaration to permit compiler optimization and to make ordering dependencies (or the lack thereof) obvious to human readers. `reduce1` is a version of `reduce` which works on non-empty lists. `unfold` produces a list systematically from left to right. Definitions for all of these functions are given in Appendix A. A detailed discussion of these functions, their importance, and the ability of the compiler to optimize them can be found in [3].

[2]Note that a naive implementation may perform no optimization at all and simply define `reduce` to be the faster of these operations.

All of the omitted functions produce infinite lists. Of these, the most frequently used are `enumFrom` and `enumFromBy`. When porting existing Haskell code to pH, upper bounds should be given in uses of these constructs. In reality, implementations are free to implement either of these functions on finite types, in which case they should halt when the reach the largest (or smallest as appropriate) element of the type. The remaining function on infinite lists is `iterate`. The functions `repeat` and `cycle` can be defined using circular lists, and were thus left in for compatibility with Haskell.

5.2 Binary Datatype

The `Bin` datatype is not required by pH—indeed, very few Haskell compilers implement it as specified. Its primary purpose is to enable efficient I/O; a later section describes the current state of I/O in pH.

5.3 Numbers

The Haskell type system specifies a large numeric hierarchy. While the functionality provided is laudable, it should be recognized that existing implementations of pH do not support all of it. However, implementations must provide the types Int and Float.

5.4 Arrays

The semantics of arrays in pH were discussed earlier. Again it is only when errors occur that pH and Haskell take different positions on the behavior of arrays. The guiding principle is to *make as much of the array available as is possible despite the presence of errors.* This means that while undefined, multiply defined, or out of range assignments should signal an error, the values of the remaining elements of the array ought to be well-defined.

5.4.1 The Class Ix

A new method, `sizeRange`, has been added to the Ix class in pH. It has the following definition:

```
>          sizeRange :: (a,a) -> Int
```

Note that the behavior of the `Ix` class is somewhat different in pH. In particular, the following identity **does not necessarily hold**:

```
range (l,u) !! index (l,u) i == i
```

This permits implementations of arrays that are NOT mapped to storage in a perfectly linear fashion. Instead, the following identities take its place:

```
inRange (l,u) i == index i < rangeSize (l,u)
inRange (l,u) i == 0 <= index i
rangeSize (l,u) == length (range (l,u))
```

5.4.2 Array Construction

The `Assoc` type does not exist in pH. Instead, tuple notation is used for the construction of arrays, so the Haskell expression `i := v` would be written in pH as `(i,v)`. Note that Haskell 1.3 will almost certainly use the latter notation.

5.4.3 Incremental Array Updates

The introduction of I-structure and M-structure arrays include incremental update-in-place operations directly analogous to the Haskell's incremental update operations, (\\) and accum. They are:

```
>   (//=)    :: (Ix a) => IArray a b -> [(a,b)] -> ()
>   (//^=)   :: (Ix a) => MArray a b -> [(a,b)] -> ()
>   (//^^=)  :: (Ix a) => MArray a b -> [(a,b)] -> ()
>   mAccum   :: (Ix a) => (b -> c -> b) -> MArray a b -> [(a,c)] -> MArray a b
```

5.5 Errors

Not all errors in pH are semantically equivalent to $\perp$. Indeed, things like multiply defined array elements and multiply written I-structures are semantically equivalent to $\top$, the opposite extreme of the value domain. Nonetheless, an application of `error` must still display the provided string and should terminate the program.

6 Input/Output

I/O in pH is based upon the new monadic definition from Haskell 1.3. At the moment only a stripped-down version of this definition exists. The implementation is based on the more general notion of a state transformer, which is discussed in the next section.

7 State Transformers

State transformers can be used to manipulate state—in particular, I- and M-structure cells and vectors, and external files. Programs which manipulate state using state transformers rather than raw I- or M- structures, and which are written without `par` and `seq` constructs, can be run under Haskell if libraries are provided to simulate the state transformer functions. This will permit state-manipulating code to be written portably.

A Changes to the Standard Prelude

This appendix lists the portions of the pH prelude which are different from the corresponding Haskell definitions.

```
module Prelude(
    PreludePHArray.., PreludePH..,
    errMaybe, lift, lift2, maybe,
```

The remaining Prelude exports have not changed.

```
import PreludePHArray
import PreludePH
```

The remaining imports are the same, as is the code with the following addition:

```
thenMaybe                     :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing `thenMaybe` _         = Nothing
Just a  `thenMaybe` f         = f a

errMaybe                  :: String -> Maybe a -> a
errMaybe msg Nothing      = error msg
errMaybe _   (Just r)     = r

lift                          :: (a -> b) -> Maybe a -> Maybe b
lift _ Nothing                = Nothing
lift f (Just x)               = Just (f x)

lift2                         :: (a -> a -> a) -> Maybe a -> Maybe a -> Maybe a
lift2 _ Nothing r             = r
lift2 _ l Nothing             = l
lift2 f (Just l) (Just r)     = Just (f l r)

maybe                         :: b -> (a -> b) -> (Maybe a -> b)
maybe b _ Nothing             =  b

either                        :: (a -> c) -> (b -> c) -> (Either a b -> c)
either f _ (Left a)           =  f a
either _ g (Right b)          =  f b
```

A.1 Prelude `PreludeCore`

```
module PreludeCore (
    Maybe(..), Either(..),
    Ix(range, index, inRange, sizeRange),
    Enum(enumFromTo, enumFromThenTo),
```

Remaining exports as before, omitting Assoc.

```
import PreludeArray(Array)
```

Do not import `Assoc` *from* `PreludeArray`*. Remaining imports are unchanged.*

```
data Maybe a    = Nothing | Just a   deriving (Eq, Ord, Text)
data Either a b = Left a  | Right b  deriving (Eq, Ord, Text)


class  (Ord a) => Ix a  where
    range             :: (a,a) -> [a]
    index             :: (a,a) -> a -> Int
    inRange           :: (a,a) -> a -> Bool
    sizeRange         :: (a,a) -> Bool

    sizeRange         = length . range
```

```
class  (Ord a) => Enum a    where
    enumFromTo          :: a -> a -> [a]
    enumFromThenTo      :: a -> a -> a -> [a]
```

omit class `Binary`

A.2 Prelude `PreludeList`

```
module PreludeList where
```

Again much the same as before, with the following additions:

The following identities must hold for reduce; its implementation is otherwise unconstrained:

```
reduce f z (a ++ b)     ==     (reduce f z a) `f` (reduce f z b)
reduce f z [e]          ==     e
reduce f z []           ==     z

reduce          :: (a -> a) -> a -> [a] -> a
```

Note that many of the functions in the prelude can be defined using `reduce` rather than `foldl`, `foldr`, or explicit recursion. For example, we can define `sum` as follows:

```
sum = reduce (+) 0
```

`someOrder` returns a permutation (perhaps fixed) of its argument.

```
someOrder       :: [a] -> [a]
```

`unfold` generates a list systematically given a function.

```
unfold :: (a->Bool) -> (a->(b,a)) -> a -> [b]
unfold p f v = h v
         where h v | p v       = []
                   | True      = case f v of (a,b) -> a:h b

concatMap :: (a->[b]) -> [a] -> [b]
concatMap f l =
    let h1 [] = []
        h1 (y:ys) =
            let h2 [] = h1 ys
                h2 (x:xs) = x : h2 xs
            in  h2 (f y)
    in  h1 l
```

A.3 Prelude `PreludeArray`

This portion of the prelude much the same as in Haskell; however, the semantics of some array operations is different in pH. See the main text for more details. The Haskell report gives an implementation which "specifies the semantics of arrays only". Such a definition for pH will eventually appear here. Note that the Assoc type has been replaced with 2-tuples throughout.

```
interface  PreludeArray where

infixl 9  !
infixl 9  //
```

Obligatory definitions of classes `Eq`*,* `Ord`*, and* `Ix` *must appear in a proper interface file, but are omitted for clarity.*

```
data  (Ix a)    => Array a b

array           :: (Ix a) => (a,a) -> [(a, b)] -> Array a b
listArray       :: (Ix a) => (a,a) -> [b] -> Array a b
(!)             :: (Ix a) => Array a b -> a -> b
bounds          :: (Ix a) => Array a b -> (a,a)
indices         :: (Ix a) => Array a b -> [a]
elems           :: (Ix a) => Array a b -> [b]
assocs          :: (Ix a) => Array a b -> [(a, b)]
accumArray      :: (Ix a, Eq a) => (b -> c -> b) -> b -> (a,a) -> [(a, c)] -> Array a b
(//)            :: (Eq a, Ix a) => Array a b -> [(a, b)] -> Array a b
accum           :: (Ix a, Eq a) => (b -> c -> b) -> Array a b -> [(a, c)] -> Array a b
amap            :: (Ix a) => (b -> c) -> Array a b -> Array a c
ixmap           :: (Ix a, Ix b) => (a,a) -> (a -> b) -> Array b c -> Array a c

instance  (Ix a, Eq b)  => Eq (Array a b)

instance  (Ix a, Ord b) => Ord (Array a b)

instance  (Ix a, Text a, Text b) => Text (Array a b)
```

A.4 Prelude `PreludeIO`

This module is based on the Haskell 1.3 IO proposal[1]. It is thus monadic in structure. It is built on a state transformer substrate (see the discussion of `PreludeST`, below). This enables operations on state and IO operations to be freely intermixed. The implementation of IO should, of course, be hidden from view. The extensions described here permit parallel I/O (beyond simple parallel file read/write) with some loss of safety.

The function `doST` changes an arbitrary state transformer computation (whose state is constrained to type `IOState`, rather than being fully polymorphic) into a computation of type `IO`.

```
doST :: ST IOState a -> IO a
```

A.5 Prelude `PreludePH`

Primitive functions and types which implement I- and M- structures. These functions will not behave correctly under Haskell.

```
interface PreludePH where
```

Again, obligatory definitions of classes `Eq`*,* `Ord`*, and* `Ix` *must appear in a proper interface file, but are omitted for clarity.*

Dummy class to express the fact that a type variable is imperative. The compiler knows about this class.

```
class Imperative a
```

The `unboxed` declaration is used in many compilers to keep a datatype from being used polymorphically. This permits `ICell` and `MCell` to be implemented at no additional space cost.

```
data unboxed ICell a
iCell :: (Imperative a) => a -> ICell a
iFetch :: ICell a -> a
iAssign :: ICell a -> a -> ()

data unboxed MCell a
mCell :: (Imperative a) => a -> MCell a
mFetch :: MCell a -> a
mExamine :: MCell a -> a
mStore :: MCell a -> a -> ()
mReplace :: MCell a -> a -> ()
```

A.6 Prelude `PreludePHArray`

Functions and types to implement I- and M-structure arrays. The non-updating functions should have the same observable behavior as the equivalent functions on ordinary arrays.

These are canonical definitions for I- and M- structure arrays in terms of arrays of `ICell` and `MCell`. Places where implementations can be more flexible are duly noted.

```
module  PHArray (IArray(..), MArray(..),
                 iArray, mArray,
                 iBounds, mBounds,
                 (!.), iAStore, (//=)
                 (!^), mAStore, (//^=)
                 (!^^), mAReplace, (//^^=)
                 iToArray, mToArray
                 ) where

import PreludeCore
import PreludeUtil
import Prelude(error, and)
import PreludePHVector
```

```
import PreludePH(Imperative(..))
import PreludeArray(Array(..))

infixl 9 !.
infixl 9 !^
infixl 9 !^^
infixl 9 //=
infixl 9 //^=
infixl 9 //^^=

data (Ix a) => IArray a b = MkIArray (Array a (IVar b))
data (Ix a) => MArray a b = MkMArray (Array a (MVar b))

iArray          :: (Ix a, Imperative b) => (a,a) -> [(a, b)] -> IArray a b
iBounds         :: (Ix a) => IArray a b -> (a,a)
(!.)            :: (Ix a) => IArray a b -> a -> b
iAStore         :: (Ix a) => IArray a b -> a -> b -> ()
(//=)           :: (Ix a) => IArray a b -> [(a, b)] -> ()
iToArray        :: (Ix a) => IArray a b -> Array a b

mArray          :: (Ix a, Imperative b) => (a,a) -> [(a, b)] -> MArray a b
mBounds         :: (Ix a) => MArray a b -> (a,a)
(!^)            :: (Ix a) => MArray a b -> a -> b
(!^^)           :: (Ix a) => MArray a b -> a -> b
mAStore         :: (Ix a) => MArray a b -> a -> b -> ()
mAReplace       :: (Ix a) => MArray a b -> a -> b -> ()
(//^=)          :: (Ix a) => MArray a b -> [(a, b)] -> ()
(//^^=)         :: (Ix a) => MArray a b -> [(a, b)] -> ()
mAccum          :: (Ix a) => (b -> c -> b) -> MArray a b -> [(a,c)] -> MArray a b
mToArray        :: (Ix a) => MArray a b -> Array a b
```

IArray functions:

```
iArray b ivs = MkIArray a
        where a = array b [(i,iCell _) | i <- range b]
              _ = a //= ivs

iBounds (MkIArray a) = bounds a

(MkIArray a)!.i = iFetch (a!i)

iAStore (MkIArray a) i x = iStore (a!i) x

(MkIArray a) //= ivs =
        -- Do vector writes into the appropriate places.
        for (i,s) <- ivs do
            _ = iAStore a i x             -- store the values
        finally ()
```

This version will not have the constructed array shadow the value of the IArray. The function will deadlock if any of the cells of the IArray remain empty. It is perfectly legal to allow this function to terminate regardless of the state of the IArray.

```
iToArray (MkIArray a) = amap iFetch a
```

MArray operations:

```
mArray b ivs = MkMArray a
        where a = array b [(i,mCell _) | i <- range b]
              _ = a //^= ivs


mBounds (MkMArray a) = bounds a

(MkMArray a)!^i = mFetch (a!i)

(MkMArray a)!^^i = mExamine (a!i)

mAStore (MkMArray a) i x = mStore (a!i) x

mAReplace (MkMArray a) i x = mReplace (a!i) x
```

Note that the following will probably need to be followed by a barrier whenever it is used!! It behaves like mAReplace—so the function can return a value immediately, before all the replacements have taken place.

```
(MkMArray a) //^^= ivs =
        -- Do vector writes into the appropriate places.
        for (i,s) <- ivs do
            _ = mAReplace a i x             -- store the values
        finally ()
```

Note that `mAccum` actually returns a value of type `MArray`. It should be the same value as was passed in, but must not be made available until all fetches have taken place. Thus, uses of the result will be properly synchronized by simple data dependency.

```
mAccum f inp@(MkMArray a) ivs =
      let
        -- Do updates into appropriate places.
        flag = ()
        flag' = for (i,s) <- ivs do
                  seq val = mAFetch v i
                      par _   = mAStore v i val'
                          next flag = flag
                  val' = f val s
                finally flag
        seq _ = flag'
            res = inp
      in res
```

This particular implementation of mToArray produces an array whose value is fixed as the value of the mArray as the array is being constructed. Actual implementations are encouraged to have the value of the array shadow the current value of the mArray if that is possible. In any case, this function screws referential transparency no matter how you slice it. MArrays just aren't refer- entially transparent to begin with. Note also that this implementation can deadlock if any element of the MArray is unfilled.

```
mToArray (MkMArray a) = amap iExamine a
```

A.7 State Transformers

State transformers can be used in the implementation of I/O. In addition, they can be used to encapsulate any state-manipulating function—this means that programs manipulating state using only the functions in `PreludeST` and `PreludeMutable` can be run in both Haskell and pH.

```
interface PreludeST where

infixr 1 >#>, >#., <>=

data ST s a
```

`ST s a` is a State Transformer computation returning a result of type `a`. The type variable `s` is used to represent the state being manipulated.

`gate` is similar to the `strict` construct in many languages. It evaluates its second argument to weak head normal form, then returns its first argument.

```
gate :: a -> b -> a
```

Monadic binding, non-strict and strict in the state.

```
(<>=) :: ST a b -> (b -> ST a c) -> ST a c
(>#>) :: ST a b -> (b -> ST a c) -> ST a c
```

Sequencing, non-strict (actually runs computations in parallel) and strict in the state.

```
(<||>) :: ST a b -> ST a c -> ST a c
(>#.)  :: ST a b -> ST a c -> ST a c
```

Encapsulating state manipulations. Note that in order to gaurantee safety, `runST` must actually possess the type $\forall \mathtt{a}.(\forall \mathtt{b}.\mathtt{ST\ a\ b}) - > \mathtt{b}$.

```
runST             :: ST a b -> b
_unsafePerformST :: ST a b -> b
```

Fixed point of the value portion of the state.

```
fixST :: (a -> ST b a) -> ST b a
```

Arbitrary parallel state manipulation. Results are combined using the provided function.

```
joining :: (a -> b -> c) -> ST d a -> ST d b -> ST d c
```

Sequence a large number of computations.

```
seqST :: [ST a ()] -> ST a ()
```

The functions shown here produce state transformers which can manipulate mutable objects, either strictly or non-strictly.

```
interface PreludeMutable where

import PreludeST
```

The type `MutVar` describes a memory location which always contains a meaningful value. Note that using `writeVar` in parallel with itself introduces non-strictness into the program, and either value written may be available to later computations.

```
data MutVar s a

newVar   :: (Imperative s, Imperative a) => a -> ST s (MutVar s a)
readVar  :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

The type `MutVector` describes a vector of `MutVars`. The function `_unsafeFreezeMutVector` permits `MutVector` to be used to implement arrays.

```
data MutVector s a = MutVector (MVector a)
data Vector a = Vec (MVector a)

newMutVector           :: (Imperative s) => Int -> a -> ST s (MutVector s a)
readMutVector          :: MutVector s a -> Int -> ST s a
writeMutVector         :: MutVector s a -> Int -> a -> ST s ()
_unsafeFreezeMutVector :: MutVector s a -> ST s (MVector a)
readVector             :: Vector a -> Int -> a
```

I-structure variables.

```
data IVar s a = IVar (ICell a) {-# STRICT #-}

newIVar   :: ST s (IVar s a)
readIVar  :: IVar s a -> ST s a
writeIVar :: IVar s a -> a -> ST s ()
```

M-structure variables.

```
data MVar s a = MVar (MCell a) {-# STRICT #-}

newMVar     :: ST s (MVar s a)
takeMVar    :: MVar s a -> ST s a
putMVar     :: MVar s a -> a -> ST s ()
examineMVar :: MVar s a -> ST s a
replaceMVar :: MVar s a -> a -> ST s ()
```

References

[1] A. Gordon, K. Hammond, A. Gill, I. Poole, and J. Mattson. The Definition of Monadic IO for Haskell 1.3. Available as `http://www.dcs.gla.ac.uk/~kh/Haskell1.3/IO.html`.

[2] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language haskell, a non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[3] J.W. Maessen. Simplifying Parallel List Traversal Technical Report CSG Memo 370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, January 30, 1995.

[4] R. S. Nikhil. Id (Version 90.1) Reference Manual. Technical Report CSG Memo 284-2, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 15 1991.