COLOR LAYER

a

Tutorial on the Functional Programming Language

HASKELL

June 1991

Paul Hudak
Yale University
Department of Computer Science

Haskell

Haskell is a new, purely functional language, named after the logician Haskell B. Curry, and designed by a 15-member international committee:

Paul Hudak [ed.]	(Yale University)
Philip Wadler [ed.]	(University of Glasgow)
Arvind	(MIT)
Brian Boutel	(Victoria Univ. of Wellington)
Jon Fairbairn	(Cambridge University)
Joseph Fasel	(Los Alamos National Lab)
Kevin Hammond	(University of Glasgow)
John Hughes	(University of Glasgow)
Thomas Johnsson	(Chalmers Univ. of Tech.)
Dick Kieburtz	(Oregon Grad. Inst. of S&T)
Rishiyur Nikhil	(MIT)
Simon Peyton Jones	(UCL / Univ. of Glasgow)
Mike Reeve	(Imperial College)
David Wise	(Indiana University)
Jonathan Young	(Yale / MIT)

(Note: The Haskell 1.1 Report shows a slightly different set of authors.)

Goals in the Design of Haskell

- To have a *common* functional language suitable for:
  - Teaching
  - Research
  - Application
- To capture all of the recent "good ideas" in functional language research.
- To form a basis for further research in language design and implementation.

The design began in October 1987. In December 1988 a preliminary draft of the language was released. The final draft was released in April 1990, and a revised "Haskell 1.1" Report was released in June, 1991.

Key Features of Haskell

	in contrast, SISAL is:
higher-order functions	first-order
polymorphic type system	monomorphic
static type inference	explicitly typed
non-strict (i.e. "lazy")	strict (call-by-value)
user-defined datatypes	none
type-safe modules	none
stream and continuation I/O	streams only
lexical, recursive scoping	same
syntax to support:	
curried functions	none
pattern-matching	none
lists & arrays	arrays only
extensible operators	none

Moral of the story:

Not all functional languages are alike!

Innovative Features of Haskell

- An elegant *overloading* facility, integrated with the polymorphic type system.
- A *purely functional I/O* system, with "dual" *stream* and *continuation* models.
- Non-strict immutable *arrays*, and a notion of *array comprehensions* for defining them.
- An unbundling of ADT's into independent notions of *data abstraction* and *information hiding*.
- A simple *typed kernel language* within which all special syntax is defined.

Acknowledgements

Haskell was influenced by many languages:

McCarthy's Lisp (and its modern-day incarnation, Scheme); Landin's ISWIM; Backus' FP; Gordon, Milner, and Wadsworth's ML; Burstall, MacQueen, and Sannella's Hope; and Turner's series of languages culminating in Miranda.*

*Miranda is a trademark of Research Software Ltd.

Outside Reading

- On Introductory Functional Programming:

  Introduction to Functional Programming, by R. Bird and P. Wadler, Prentice Hall, New York, 1988.

- On Advanced Functional Programming:

  Functional Programming, by A.J. Field and P.G. Harrison, Adison-Wesley, Workingham, England, 1988.

- On History and Theoretical Background:

  "Conception, Evolution, and Application of Functional Programming Languages," by P. Hudak, *ACM Comp. Surveys*, 21(3), 1989, pp. 359-411.

- On Implementing Functional Languages:

  The Implementation of Functional Programming Languages, by S.L. Peyton Jones, Prentice-Hall, Englewood Cliffs, NJ, 1987.

Outline of This Tutorial

1. Introduction (done!)
2. Some Lexical Matters
3. Values, Types, and Other Goodies
4. Abstraction, Abstraction, Abstraction
   - Data Abstraction
   - Higher-order Functions
   - Lazy Evaluation
5. Overloading is not *Ad Hoc*
6. Controlling the World with Functions
7. Comprehending Arrays

Part II

Some Lexical Matters

- Concrete Haskell code will be written in "typewriter font", as in "`f (x+y) (a-b)`".
- Case matters – "`foo`", "`Foo`", and "`FOO`" are different identifiers.
- Bound variables and type variables are denoted by identifiers beginning with a *lowercase* letter; types, constructors, modules, and classes are denoted by identifiers beginning with an *uppercase* letter.
- *Layout* may be used in declaration lists (see following slide).

Layout

Most languages enclose declaration lists (or commands) with explicit syntax, and require delimeters between declarations. In Haskell, this information is (optionally) conveyed via the *layout* of the code. For example, instead of writing:

```
f a + f b where { a = 5; b = 4;
                  f x = x + 1 }
```

one may write:

```
f a + g b where a = 5; b = 4
                f x = x + 1
```

Layout is valid in any declaration list following the keyword "`where`", "`let`", or "`of`".

More Lexical Matters

- Function application is *curried*, associates to the left, and always has higher precedence than infix operators. Thus "`f x y + g a b`" parses as "`((f x) y) + ((g a) b)`".
- Infix operators may be coerced into prefix functions, and vice versa: "`x + y`" $\equiv$ "`(+) x y`", and "`f x y`" $\equiv$ "`` x `f` y ``".
- Comments are preceded by "`--`" and continue to end of line. For example:

```
succ n = n-1    -- this definition sucks
```

Part III

Values, Types, and Other Goodies

Values are the things we compute with – the integer 5, the identity function, the list "`['a','b','c']`", etc. – and are *first-class*.

First-class values may be passed as arguments to functions, returned as results, placed in data structures, etc.

Values are divided into disjoint sets called *types* – integers, functions, lists, etc.

Types are *not* first-class in Haskell.

Expressions are syntactic terms that denote *values* and thus have an associated *type*.

Haskell's Type System

Haskell is *strongly typed* – every expression has exactly one "most general" type (called the *principal type*).

Types may be *polymorphic* – i.e. they may contain *type variables* which are universally quantified over all types.

Furthermore, it is always possible to statically infer this type; user-supplied type declarations are *optional*.

As a result:

- Strong typing aids reasoning about programs.
- Polymorphism results in more expressive types.
- All type errors are detected at compile-time.
- No run-time type tests are required.

Examples of Pre-Defined Datatypes

`Integer`: the (infinite) set of integers, or "bignums" {... -2, -1, 0, 1, 2, ...}

`Int`: the (finite) set of small integers, or "fixnums" {-max, ..., -2, -1, 0, 1, 2, ..., max}

Note: the lexical tokens "`-1`", "`0`", "`1`", etc. are *overloaded* to denote either an "`Int`" or "`Integer`" (more on this later).

Similarly, there are finite types "`Float`" and "`Double`" containing floating-point numbers, with obvious lexical tokens overloaded to represent both.

`Bool`: the Boolean type containing values "`True`" and "`False`".

`Char`: the character type, with lexical syntax `'a'`, `'b'`, `'c'`, etc.

Pre-Defined Structured Datatypes

Tuples:

$$(e_1,\, e_2,\, \ldots,\, e_n) \qquad n \geq 2$$

If t_i is the type of e_i, then the type of the tuple is $(t_1,\, t_2,\, \ldots,\, t_n)$.

Lists:

$$[e_1,\, e_2,\, \ldots,\, e_n] \qquad n \geq 0$$

Every element e_i must have the same type, say t, and the type of the list is then $[t]$.

The above list is equivalent to:

$$e_1{:}e_2{:}\ldots{:}e_n{:}\,[]$$

i.e., ":" is the infix operator for "cons".

User-Defined Datatypes

The key to good data abstraction is allowing the user to define her own datatypes. This is done via a "`data`" declaration having general form:

$$
\begin{array}{rcl}
\texttt{data}\ T\ u_1\ \cdots\ u_n & = & C_1\ t_{11}\ \cdots\ t_{1k_1} \\
& | & \cdots \\
& | & C_n\ t_{n1}\ \cdots\ t_{nk_n}
\end{array}
$$

where T is a *type constructor*; the u_i are *type variables*; the C_i are *(data) constructors*; and the t_{ij} are the *constituent types* (possibly containing some u_i).

The presence of the u_i implies that the type is *polymorphic* – it may be instantiated by substituting specific types for the u_i (more on this later).

Examples of User-Defined Datatypes

```
data Bool = True | False
data Color = Red | Green | Blue | Indigo | Violet
data Point a = Pt a a
data Tree a = Branch (Tree a) (Tree a) | Leaf a
```

- "`Bool`" is a *nullary type constructor* (because it has no arguments).
- Similarly, "`True`", "`False`", "`Red`", "`Green`", etc. are *nullary data constructors*.
- "`Bool`" and "`Color`" are *enumerations* (because all of their data constructors are nullary).
- "`Point`" is a *tuple* type constructor (because it has only one constructor); the others are *union* types.
- "`Tree`" is a *recursive* type constructor.

Built-in Types are Not Special

Haskell's built-in types can be viewed as that generated by the following (sometimes infinite!) pseudocode (i.e. not legal Haskell):

```
data Integer = ... -2 | -1 | 0 | 1 | 2 ...
data Int = -65532 | ... | -1 | 0 | 1 | ... | 65532
data Char = 'a' | 'b' | 'c' | ...
data [a] = a : [a] | []
data (a,b) = (a,b)
data (a,b,c) = (a,b,c)
data (a,b,c,d) = (a,b,c,d)
...
```

The point being:

Although some built-in types have special syntax, they have no special semantic significance.

Functions

First-class, of course, and therefore "higher-order." They may be defined via declarations, or "anonymously" via *lambda abstractions*.

For example, "`\x -> x+1`" is a lambda abstraction, and is equivalent to the function "`succ`" defined by:

```
succ x = x+1
```

Function definitions and lambda abstractions are "curried", thus facilitating the use of higher-order functions.

> If "`x`" has type t_1 and "`exp`" has type t_2, then "`\x->exp`" has type t_1`->`t_2.

Sections

Haskell also allows *sections* – short-hand notation for partially applying infix operators. For example:

```
(x+)   -- is equivalent to:   \y -> x+y
(+y)   -- is equivalent to:   \x -> x+y
```

Note that this is consistent with the lexical syntax mentioned earlier:

```
(+)   -- is equivalent to:   \x y -> x+y
```

Polymorphism

Functions and datatypes may be *polymorphic*; i.e. universally quantified in certain ways over all types. For example, the "`Tree`" datatype is polymorphic:

```
data Tree a = Branch (Tree a) (Tree a) | Leaf a
```

"`Tree Int`" is the type of trees of fixnums; "`Tree (Char->Bool)`" is the type of trees of functions mapping characters to Booleans, etc. Furthermore:

```
fringe (Leaf x)             = [x]
fringe (Branch left right) = fringe left
                             ++ fringe right
```

"`fringe`" has type "`Tree a -> [a]`", i.e. "for all types `a`, `fringe` maps trees of `a` into lists of `a`."

More Examples of Polymorphism

```
id x   =  x
[]      ++ ys   =  ys
(x:xs) ++ ys   =  x : (xs++ys)
map f  []      =  []
map f (x:xs)   =  f x : map f xs
```

Here "`id`" has type "`a->a`", "`(++)`" (append) has type: "`[a]->[a]->[a]`", and "`map`" has type "`(a->b)->[a]->[b]`".

These types are inferred *automatically*, but may optionally be supplied as *type signatures*:

```
id    :: a -> a
(++)  :: [a] -> [a] -> [a]
map   :: (a->b) -> [a] -> [b]
```

Type Synonyms

For convenience, Haskell provides a way to define *type synonyms* – i.e. names for commonly used types. Type synonyms are created using *type declarations*. Examples include:

```
type String = [Char]
type Person = (Name,Address)
type Name = String
data Address = None | Addr String
```

This definition of `String` is part of Haskell, and in fact the literal syntax `"hello"` is shorthand for:

```
['h','e','l','l','o']
```

Pattern-Matching and Case Expr's

We've already seen examples of pattern-matching in functions ("`fringe`", "`++`", etc.); it is the primary way that elements of a datatype are distinguished.

But function declarations are actually syntactic sugar for something more primitive: *case expressions*. For example, the previous definition of "`++`" is equivalent to:

```
(++) = \xs ys -> case xs of
                   []     -> ys
                   (x:xs) -> x : (xs++ys)
```

Kinds of Patterns

- Variables match any value and are bound to that value.
- "As-patterns" have form "`x@pat`" and allow naming other patterns.
- "Wild-cards" have form `_`, and match anything but bind nothing.
- Data constructors (including nullary ones) we've already seen, and have the obvious semantics.

An example of all four of the above:

```
case e of
  xs@(_:y:rest) -> if y==0 then rest else xs
```

Kinds of Patterns (cont'd)

- "Lazy patterns" have form ~pat. Operationally speaking, matching a value v against ~pat *always succeeds*. If an identifier in pat is later "used" on the right-hand-side, it will be bound to that portion of the value that would result in successful match of v against pat, and $\perp$ otherwise.
- Top-level patterns may also have a *guard*, as in "`case e of [x] | x>0 -> x`". The guard must evaluate to "`True`" for the match to succeed.

Pattern-Matching Semantics

Patterns are matched against values, yielding one of three results: it may *fail*; it may *succeed* (returning a binding for each variable in the pattern); or it may *diverge* (i.e. return $\perp$).

Pattern-matching proceeds left to right, and outside in, according to the rules for each kind of pattern.

Clauses in a case expression or function definition are matched top to bottom.

Examples of Pattern-Matching

Let "`bot`" be an expression with value $\perp$.

If "`[1,2]`" is matched against "`[0,bot]`", then 1 fails to match 0, so the result is a failed match. But if "`[1,2]`" is matched against "`[bot,0]`", then matching 1 against `bot` causes divergence (i.e. result $\perp$).

More examples:

```
(\ ~(x,y) -> 0) bot               ⇒  0
(\  (x,y) -> 0) bot               ⇒  ⊥

(\ ~[x] -> 0) []                  ⇒  0
(\ ~[x] -> x) []                  ⇒  ⊥

(\ ~[x,~(a,b)] -> x) [0,bot]      ⇒  0
(\ ~[x, (a,b)] -> x) [0,bot]      ⇒  ⊥

(\  (x:xs) -> x:x:xs) bot         ⇒  ⊥
(\ ~(x:xs) -> x:x:xs) bot         ⇒  ⊥:⊥:⊥
```

Other Expressions

- Conditional: $\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$

  Actually shorthand for:

  $$\begin{array}{l} \texttt{case } e_1 \texttt{ of True } \rightarrow e_2 \\ \qquad\qquad\quad\;\; \texttt{False} \rightarrow e_3 \end{array}$$

- Where construct:

  $$\begin{array}{ll} \texttt{fx} & \mid g_1 = e_1 \\ & \ldots \\ & \mid g_n = e_n \\ & \texttt{where } \{\, d_1;\ \ldots;\ d_k \} \end{array}$$

  The declarations $d_1 \ \ldots \ d_k$

  – may be type signatures or value (including function) bindings, but *not* data or class declarations.

  – are mutually recursive and scope over all of e_1 through e_n and g_1 through g_n.

Other Expressions (Cont'd)

- Let expression:

  `let` $\{\ d_1;\ \ldots;\ d_k\}$
  `in` e

  Same restrictions apply to the d_i here as in the where construct, except that they scope only over e, and can thus be nested, whereas a where construct cannot.

Lists

Lists are pervasive in Haskell.

List comprehensions:

```
quicksort  []     = []
quicksort (x:xs) = quicksort [y | y <- xs, y<x ]
                ++ [x]
                ++ quicksort [y | y <- xs, y>=x]
```

Arithmetic sequences:

```
[1..10]    -- yields [1,2,3,4,5,6,7,8,9,10]
[1,3..10] -- yields [1,3,5,7,9]
[1..]      -- infinite list of positive integers
[1,3..]    -- infinite list of positive odds
```

“Infinite” Lists

Come “for free” with lazy evaluation. Examples:

```
ones = 1 : ones
nums_from n = n : nums_from (n+1)
squares = [ x**2 | x <- nums_from 0 ]
fib = 1:1:[ a+b | (a,b) <- zip fib (tail fib) ]
```

Elements are computed “on demand”, thus relieving programmer of specifying “consumer-producer” control flow.

Part IV

Abstraction, Abstraction, Abstraction

This title is the answer to the question: "what are the three most important things in programming."

From Websters:

> "abstract", *vt* (1) remove, separate (2) to consider apart from application to a particular instance.

In programming we do this when we see a "recurring pattern" of some sort, and wish to "separate" that pattern from the "particular instances."

Abstraction in Haskell

Haskell supports abstraction in several ways:

- via where constructs and let expressions
- via function definitions
- via data abstraction
- via higher-order functions
- via lazy evaluation

Familiar Abstractions

Common subexpression elimination:

```
(x+y-z)*(x+y-z)
```

⇒

```
let a=x+y-z in a*a
```

where the recurring pattern is an entire *expression*.

Functional abstraction:

```
(a+b-c)*(x+y-z)
```

⇒

```
let f i j k = i+j-k
in  f a b c * f x y z
```

where the recurring pattern is an expression with "holes" in it (i.e. _+_-_).

Data Abstraction

The benefits of data abstraction are well known.

```
length x1 y1 x2 y2 =
       sqrt ((x2-x1)**2 + (y2-y1)**2)
x_span x1 y1 x2 y2 = x2-x1
```

Here operations are being defined on points and lines. Abstracting the repeating pattern of arguments yields:

```
data Point = Pt Int Int
data Line = Ln Point Point
length (Ln (Pt x1 y1) (Pt x2 y2)) =
             sqrt ((x2-x1)**2 + (y2-y1)**2)
x_span (Ln (Pt x1 y1) (Pt x2 y2)) = x2-x1
```

"Self-documenting", promotes type security, etc.

How Not To Do Data Abstraction

Encoding all data structures as *tuples* or *lists*, say, is an example of *poor* data abstraction! (Or perhaps no abstraction at all.)

User-defined algebraic datatypes encourage good data abstractions.

Abstracting Higher-Order Functions

```
sum   []     = 0
sum  (x:xs) = x + sum xs

prod  []     = 1
prod (x:xs) = x * prod xs
```

Note repeating pattern – parameterizing the variable elements `0/1` and `+/*` using names "`init`" and "`op`", resp., and calling the new function "`fold`", we get:

```
(fold op init)  []     = init
(fold op init) (x:xs) = x `op` (fold op init) xs
```

(The parens around `fold op init` are used only for emphasis, and are otherwise superfluous.)

Using "fold" we can derive these new definitions for "sum" and "prod":

```
sum  = fold (+) 0
prod = fold (*) 1
```

Of course now that the fold abstraction has been made, many other useful functions can be defined, even something as seemingly unrelated as append:

```
append xs ys = fold (:) ys xs
```

Indeed "fold" is so useful that several versions of it are predefined in Haskell.

Lazy Evaluation Enhances Abstraction

Why lazy evaluation?

Hype #1: Allows infinite data structures (whoop-de-do!).

A more pragmatic claim:

> Promotes *modularity* by separating *data* (e.g. a list) from *control* (e.g. when the list elements are evaluated).

Hype #2: Programs terminate more often (yahoo!).

A more pragmatic claim:

> Frees programmer from concerns about *evaluation order* (termination being just a special case!).

For example, this seemingly harmless transformation:

```
if p then (x+y)*(x+y)
     else z
```

$\Rightarrow$

```
if p then a*a else z
where a=x+y
```

is simply *not valid* without lazy evaluation (because the expression "`x+y`" may result in error or even non-termination), thus preventing a simple application of abstraction.

Abstract Data Types

Information hiding is done via *modules*. For example:

```
module Stack (StackType, push, pop, top, empty)
where
data StackType a = Empty | Stk a (StackType a)
push x s = Stk x s
pop (Stk _ s) = s
top (Stk x _) = x
empty = Empty
```

The constructors "`Empty`" and "`Stk`", which comprise "the implementation," are not exported, and thus hidden outside of the module. To make the datatype *concrete*, one would write:

```
module Stack (StackType(Empty,Stk), push, ...)
...
```

Part V

Overloading is not *Ad Hoc*

The polymorphism shown so far is *parametric* polymorphism; in contrast, there is *ad hoc* polymorphism, or *overloading*:

- The literals 1, 2, etc. represent both fixnums and bignums.
- Numeric operators such as "+" work on all kinds of numbers.
- The equality operator "==" works on numbers and many other (but not all) kinds of datatypes.

The point being:

These overloaded behaviors are *different* for each type (in fact sometimes undefined), whereas in parametric polymorphism the type truly does not matter.

Consider, for example:

```
member x  []     = False
member x (y:ys) = x==y || member x ys
```

The type of member "ought" to be: "`a->[a]->Bool`", implying that "==" has type "`a->a->Bool`". But in fact "==" is *not* defined on all types (consider, for example, functions). Also, how do we know which version of "==" to use inside of "`member`"?

What to do?

(Miranda and ML use rather *ad hoc* techniques...)

Type Classes

Say that we wish to overload the operations "`+`" and "`negate`" on types "`Int`" and "`Float`". To do so, we introduce a new *type class* called "`Num`":

```
class Num a  where
  (+)    :: a -> a -> a
  negate :: a -> a
```

which may be read "a type `a` is an instance of the class "`Num`" if there are (overloaded) operations "`+`" and "`negate`", of the appropriate types, defined on it."

"`+`" actually has type "`(Num a) => a->a->a`";
the "`(Num a) => `" part is called a *context* and places *constraints* on the type variables – i.e. "`+`" is not defined for *all* types, just those in the class "`Num`".

Type Classes, cont'd

We may then declare "`Int`" and "`Float`" to be *instances* of "`Num`", as follows:

```
instance Num Int  where
  x + y        =  addInt x y
  negate x     =  negateInt x

instance Num Float  where
  x + y        =  addFloat x y
  negate x     =  negateFloat x
```

where "`addInt`", "`negateInt`", etc. are assumed to be primitive functions, but in general could be any user-defined function. The first declaration above may be read "`Int` is an instance of the class `Num` as witnessed by these definitions (i.e. *methods*) for "`+`" and "`negate`"."

Type Classes, cont'd

So now the following types are valid:

```
double :: (Num a) => a -> a
double x = x+x

doubles :: (Num a, Num b) => (a,b)->(a,b)
doubles (x,y) = (double x, double y)

(double 2)          :: Int
(double 3.14)       :: Float
(doubles (2,3.14)) :: (Int,Float)
```

Haskell's Numeric Structure

In Haskell the class mechanism is used to describe a rather rich set of numeric types and classes, using a hierarchy that amounts to multiple inheritance:

```
           Eq
          /  \
       Ord  Num
          \  / \
          Real Fractional
          /  \   /  \
 Integral RealFrac Floating
                \  /
               RealFloat
```

"`Integral`" has instances of "`Int`" and "`Integer`", "`RealFrac`" has "`Ratio`", "`RealFloat`" has "`Float`", and "`Floating`" has "`Complex`".

```
class Eq a  where
    (==), (/=)  ::  a -> a -> Bool
class (Eq a) => Ord a  where
    (<), (<=), (>=), (>) ::  a -> a -> Bool
    max, min             ::  a -> a -> a
class (Eq a) => Num a  where
    (+), (-), (*)        ::  a -> a -> a
    negate, abs, signum  ::  a -> a
    fromInteger          ::  Integer -> a
class (Num a, Ord a) => Real a  where
    toRational           ::  a -> Rational
class (Real a) => Integral a  where
    ...
class (Num a) => Fractional a  where
    ...
class (Fractional a) => Floating a  where
    ...
class (Real a, Fractional a) => RealFrac a
    ...
```

Overloading Numeric Literals

An integer literal is considered to be an `Integer`, but with the function `fromInteger` implicitly applied to it (whose type is "`Num a => Integer->a`"). Thus integer literals effectively have type: "`Num a => a`".

Similarly, floating literals are `Rational` with `fromRational` implicitly applied, yielding type "`Fractional a => a`".

For example, the function:

```
fac 0 = 1
fac n = n*fac(n-1)
```

has type "`Num a => a->a`".

Derived Instances

Haskell provides a way to have certain commonly used operations automatically defined when a new datatype is defined – things like equality, comparison, printing, etc. Specifically, *derived instances* of the classes "`Eq`", "`Ord`", "`Ix`", "`Enum`", "`Text`", and "`Binary`" are defined if listed in the "`deriving`" form of a *data* declaration. For example:

```
data Bool  =  False | True  deriving (Eq, Ord)
```

implicitly generates an instance declaration for "`Bool`" in "`Eq`" and "`Ord`", thus allowing use of "`==`", "`>`", etc.

Instances are derived only for those classes explicitly requested.

Part VI

Controlling the World With Functions

```
+--------+        +-------------+        +--------+
| merge  |------->|  operating  |------->| split  |
+--------+        |   system    |        +--------+
 ^     ^          +-------------+         |     |
 | ... |                                  | ... |
 |     |           +-----------+          |     |
 |     +-----------| program1  |<---------+     |
 |                 +-----------+                |
 |stream of             ...           stream of|
 |requests         +-----------+      responses|
 +-----------------| programn  |<---------------+
                   +-----------+
```

Thus a program must have type:

```
[Response] -> [Request]
```

Key Points of I/O System

- Programs have type `[Response] -> [Request]`.
- The nth response is the operating system's response to the nth request.
- Non-strict semantics (i.e. lazy evaluation) is critical to making this work.
- Non-determinism is confined to the *operating system*, and thus referential transparency is maintained *within* a Haskell program.
- This *stream* model of I/O has an equivalent *continuation* model, and Haskell provides it as well.

Haskell's I/O Design

```
type Dialogue = [Response] -> [Request]
type Name = String
data Request =
     -- file system requests:
                ReadFile      Name
              | WriteFile     Name String
              | AppendFile    Name String
              | DeleteFile    Name
              | StatusFile    Name
     -- channel system requests:
              | ReadChan      Name
              | AppendChan    Name String
              | StatusChan    Name
     -- environment requests:
              | Echo Bool
              | ...
```

Haskell's I/O Design, cont'd

```
data Response = Success
              | Str String
              | Bn  Bin
              | Failure IOError

data IOError = WriteError  String
             | ReadError   String
             | SearchError String
             | FormatError String
             | OtherError  String

stdin   = "stdin"
stdout  = "stdout"
stderr  = "stderr"
stdecho = "stdecho"
```

As an example, consider the task:

Prompt the user for the name of a file, then look up and display the contents of the file on standard-output. Include explicit echoing of the filename.

Using streams (note the use of lazy patterns):

```
main ~(Success : ~((Str userInput) :
     ~(Success : ~(r4 : _)))) =
 [ AppendChan stdout "type filename\n",
   ReadChan stdin,
   AppendChan stdout name,
   ReadFile name,
   AppendChan stdout
     (case r4 of Str contents    -> contents
                 Failure ioerror -> "can't open")
 ] where (name : _) = lines userInput
```

The Continuation Model

Here I/O occurs via *transactions* which are commands to the OS that contain (possibly several) *continuations* which the OS selectively invokes. Each request "`Req`" has a corresponding transaction "`req`".

For example, instead of "`ReadFile name`", which induces either "`Failure msg`" or "`Str contents`", we would instead use:

```
readFile name (\ msg -> errorTransaction)
              (\ contents -> successTransaction)
```

where the 3rd and 4th args are the *failure continuation* and *success continuation*, respectively. Upon failure, the error continuation is applied to the error message; upon success the success continuation is applied to the contents of the file.

The Continuation Model, cont'd

We can define all this, perhaps surprisingly, in a simple way in terms of the stream model:

```
type Dialogue    =  [Response] -> [Request]
type SuccCont    =                Dialogue
type StrCont     =  String     -> Dialogue
type FailCont    =  IOError    -> Dialogue

strDispatch fail succ (resp:resps) =
   case resp of Str val     -> succ val resps
                Failure msg -> fail msg resps
succDispatch fail succ (resp:resps) =
    case resp of Success     -> succ resps
                 Failure msg -> fail msg resps
```

The Continuation Model, cont'd

```
done      :: Dialogue
readFile  :: Name ->
                 FailCont -> StrCont  -> Dialogue
writeFile :: Name -> String ->
                 FailCont -> SuccCont -> Dialogue

done resps = []
readFile name fail succ resps =
   ReadFile name  :  strDispatch fail succ resps
writeFile name contents fail succ resps =
   WriteFile name contents  :
             succDispatch fail succ resps
```

Similarly for the other requests.

Useful I/O Functions

```
infixr 0 $
f $ a    = f a
let x k = k x

abort, exit :: FailCont
abort err = done
exit  err = appendChan stdout msg abort done
    where msg =
              case err of ReadError   s -> s
                          WriteError  s -> s
                          ...
print x = appendChan stdout (show x) abort done

interact  ::  (String -> String) -> Dialogue
interact f = readChan   stdin abort $ \x ->
             appendChan stdout (f x) abort done
```

As an example, consider the task:

> Prompt the user for the name of a file, then look up and display the contents of the file on standard-output. Include explicit echoing of the filename.

Using continuations:

```
main =
  appendChan stdout "type filename\n" abort $
  readChan stdin abort  $ \userInput ->
  let (lines userInput) $ \(name : _) ->
  appendChan stdout name abort $
  readFile name
    ( \ioerror -> appendChan stdout
                  "can't open" abort done)
    $ \contents ->
  appendChan stdout contents abort done
```

Part VII

Comprehending Arrays

Haskell's arrays are *monolithic* (i.e. defined as whole, as opposed to incrementally), and *non-strict* (i.e. they may contain $\perp$ elements). Pragmatically, this means that arrays may be *recursive*, often resembling the recurrence equations from which they are derived.

A note on efficiency:

Providing arrays as a primitive abstract type implies that an implementation should be able to implement them efficiently. Constant-time access is easily achieved, although to achieve par with, say, Fortran, will require various optimizations to overcome, for example, the (normal) overheads of lazy evaluation.

Array Primitives

Arrays are created using the primitive function "`array`":

```
array :: (Ix a) => (a,a)->[Assoc a b]-> Array a b
data  Assoc a b  =  a := b
```

So "`array`" takes a pair of bounds (which must be in the index class "`Ix`") and an association list of index/value pairs. The resulting array has two primitive operations defined on it:

```
(!)    :: (Ix a) => Array a b -> a -> b
bounds :: (Ix a) => Array a b -> (a,a)
```

"`!`" takes an array and an index and returns the array's value at that index, and "`bounds`" simply returns the index bounds.

Array Comprehensions

Since "`array`" takes a list argument, it is most convenient to define arrays in a style resembling list comprehensions – thus the name "array comprehensions." Some examples:

```
array (1,n)                  -- vector sum
  [ i := a!i + b!i | i<-[1..n] ]

array ((1,1),(n,n))          -- matrix mult
  [ (i,j) := inner_prod a b i j
      | i<-[1..n], j<-[1..n] ]

array ((1,1),(n,n))          -- unit diagonal
  ( [ (i,i) := 1 | i<-[1..n] ] ++
    [ (i,j) := 0 | i<-[1..n], j<-[1..n], i/=j ] )
```

Arrays are Non-strict

As with other data structures, arrays being non-strict means that one need not worry about *evaluation order*; thus arrays may be *recursive*. For example:

```
array ((1,1),(n,n))
( [ (1,1) := 1 ] ++
  [ (i,1) := 1 | i<-[2..n] ] ++
  [ (1,j) := 1 | j<-[2..n] ] ++
  [ (i,j) := a!(i,j-1) + a!(i-1,j) + a!(i-1,j-1)
                | i<-[2..n], j<-[2..n] ] )

          1  1  1  1  1  1
          1  2  4  6
          1  4 10
          1  6    "wavefront"
          1        evaluation
          1
```

Final Example: LU Decomposition

Given a matrix A, we wish to decompose it into L and U such that:

$$A = L * U$$

where L and U are $n \times n$ lower- and upper-triangular matrices, respectively. Without loss of generality we will assume that U is unit-diagonal.

Using the def'n of matrix multiplication:

$$A_{i,j} \;=\; \textstyle\sum_{k=1}^{n} L_{i,k} * U_{k,j} \quad \text{for } i, j \in 1 \ldots n$$

But since L and U are triangular, we know that:

$$\begin{aligned} L_{i,k} &= 0 \text{ if } i < k \\ U_{k,j} &= 0 \text{ if } k > j \end{aligned}$$

thus leading to:

$$A_{i,j} = \textstyle\sum_{k=1}^{j} L_{i,k} * U_{k,j} \text{for } i \in 1 \ldots n, \;\; j \in 1 \ldots i \qquad (1)$$

$$A_{i,j} = \textstyle\sum_{k=1}^{i} L_{i,k} * U_{k,j} \text{for } i \in 1 \ldots n, \;\; j \in i+1 \ldots n \qquad (2)$$

To solve for L and U, we first expand (1) recalling that U is unit-diagonal:

$$\begin{aligned} A_{i,j} &= L_{i,j} * U_{j,j} + \textstyle\sum_{k=1}^{j-1} L_{i,k} * U_{k,j} \\ &= L_{i,j} + \textstyle\sum_{k=1}^{j-1} L_{i,k} * U_{k,j} \end{aligned}$$

and solving for L yields:

$$\begin{array}{ll} L_{i,j} = A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} * U_{k,j} & \text{for } i \in 1...n,\ \ j \in 1...i \\ L_{i,j} = 0 & \text{for } i \in 1...n,\ \ j \in i+1...n \end{array}$$

Similarly, expanding (2):

$$A_{i,j} = L_{i,i} * U_{i,j} + \textstyle\sum_{k=1}^{i-1} L_{i,k} * U_{k,j}$$

and solving now for U:

$$\begin{aligned} U_{i,j} &= (A_{i,j} - \textstyle\sum_{k=1}^{i-1} L_{i,k} * U_{k,j}) \ / \ L_{i,i} \\ & \qquad\quad \text{for } i \in 1...n,\ j \in i+1...n \\ U_{i,j} &= 0 \qquad \text{for } i \in 1...n,\ j \in 1...i-1 \\ U_{i,i} &= 1 \qquad \text{for } i \in 1...n \end{aligned}$$

Note the highly recursive nature of the derived specifications. In most languages one would not be able to simply forge ahead and convert this specification into an array, since the elements would need to be computed in a particular order; in other words the recursive dependencies would need to be unravelled by the programmer.

In contrast, using Haskell's array comprehensions, we can write out a solution almost immediately.

In addition, we can take advantage of the fact that, except for the diagonal, L is 0 whenever U is not (and vice-versa), so that we can represent both arrays as one, with L's diagonal stored on the diagonal of the result and U's unit diagonal being "implicit." Calling this array `lu`, the resulting program is shown on the next slide.

Final Program for LU-Decomposition

```
lu = array ((1,n),(1,n))
     ( [ (i,j) := do_l i j
                  | i<-[1..n], j<-[1..i]   ] ++
       [ (i,j) := do_u i j
                  | i<-[1..n], j<-[i+1..n] ] )

do_l i j =  a!(i,j) - sum (1,j-1)
                          (\k->lu!(i,k)*lu!(k,j))
do_u i j = (a!(i,j) - sum (1,i-1)
                          (\k->lu!(i,k)*lu!(k,j))
           / lu!(i,i)

sum (a,b) f | a>b  = 0
                   = f a + sum (a+1,b) f
```

Note the great similarity to the mathematical spec given earlier; except for syntax, they are identical.

Status oF Haskell

- Haskell Report 1.1 released June 1991.
- Active Haskell mailing list (requests to join may be sent to `haskell-request@cs.yale.edu`).
- Yale Haskell implementation available on Sparc-Stations and DecStations (for information, mail to `haskell-request@cs.yale.edu`).
- Glasgow Haskell implementation also available (mail to `haskell-request@dcs.glasgow.ac.uk`).