Haskell and HUGS

HASKELL is a lazy, statically scoped, purely functional programming language. Like OCAML, it is statically typed, but most types are automatically deduced by type reconstruction. We will explore HASKELL in the context of HUGS, a HASKELL interpreter. HUGS is only one of many HASKELL implementations; visit `www.haskell.org` for more information on the language and its implementations.

1 Launching HUGS

The HUGS system is launched in Linux by executing `hugs` at the Linux prompt. When HUGS is launched, it displays the following herald:

```
[lyn@jaguar private] hugs
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2001
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: December 2001  _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Reading file "/usr/share/hugs/lib/Prelude.hs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
Type :? for help
Prelude>
```

By default, HUGS loads a large set of HASKELL libraries, known as the "prelude". These libraries are defined in the file `/usr/share/hugs/lib/Prelude.hs`, which you are encouraged to skim. Lots of standard functions are defined in this file, including common list utilities like `map`, `filter`, `length`, `foldr`, `foldr`, `zip`, `unzip`, `take`, and `drop`. The prompt `Prelude>` indicates that any expressions typed in will be evaluated in the context of the `Prelude` module containing the standard library functions.

2 Quitting HUGS

You can quit out of HUGS in one of two ways:

1. Execute the `:quit` directive within the HUGS interpreter.

2. Type `C-d` (i.e., "Control-d").

3 Interacting with HUGS

You interact with the HUGS interpreter in one of two ways:

1. By typing a HASKELL expression to be evaluated. In this case, HUGS displays the value of the expression. See Fig. 1 for some examples. Unlike the OCAML interpreter, HUGS does not display the type of the expression.

```
Prelude> 2*(3+4)
14

Prelude> head [1,2,3,4]
1

Prelude> tail [1,2,3,4]
[2,3,4]

Prelude> map (2*) [1,2,3,4]
[2,4,6,8]

Prelude> take 2 [10,20,30,40,50]
[10,20]

Prelude> drop 2 [10,20,30,40,50]
[30,40,50]

Prelude> foldr (+) 0 [1,2,3,4]
10

Prelude> zip [1,2,3] [10,20,30,40]
[(1,10),(2,20),(3,30)]

Prelude> unzip [(1,10),(2,20),(3,30)]
([1,2,3],[10,20,30])

Prelude> fst (1,2)
1

Prelude> snd (1,2)
2
```

Figure 1: Sample expressions evaluated in HUGS.

2. By typing a HUGS directive, along with its arguments. All directives begin with a colon. For example, the `:cd` directive changes the current working directory to a given directory. For example, if you execute

   `Prelude> :cd /students/`*`your-username`*`/cs251/ps7`

   then HUGS will interpret all following filenames relative to this directory.

   Another important directive is the `:type` directive, which can be abbreviated `:t`. This displays the type of a HASKELL expression. For example:

   ```
   Prelude> :type map
   ```

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

Prelude> :type foldr
foldr :: (a -> b -> b) -> b -> [a] -> bw2

Prelude> :type zip
zip :: [a] -> [b] -> [(a,b)]

Prelude> :type unzip
unzip :: [(a,b)] -> ([a],[b])

Prelude> :type "foo"
"foo" :: String

Prelude> :type "foo" == "bar"
"foo" == "bar" :: Bool

Prelude> :type 1+2
1 + 2 :: Num a => a

Prelude> :type [1,2,3]
[1,2,3] :: Num a => [a]

Prelude> :type 1 == 2
1 == 2 :: Num a => Bool
```

In the last three `:type` examples, the type begins with `Num a => ....` This is a so-called **qualified type**. It turns out that HASKELL has many kinds of numeric types, and integers can have any of these types. A qualified type of the form `Num a => t` specifies a type *t* that is parameterized over any numeric type `a`.

By far the most important directive is the `:load` directive, which can be abbreviated `:l`. This loads the HASKELL declarations in the specified file. For example,

```
Prelude> :l Test.hs
```

loads the declarations in the file `Test.hs`. Note that the filename need not be delimited by double quotes, although they are allowed. The `:reload` directive, abbreviated `:r`, re-executes the most recent `:load` directive. For example, if `Test.hs` has been loaded as shown above, then `:r` will load the contents of `Test.hs` again. The `:reload` directive is commonly used after editing a file to add or fix a declaration.

The `:quit` directive exits the HUGS interpreter. The `:?` displays a list of all directives.

4 HASKELL Declarations

Unlike in the OCAML and MIT-SCHEME interpreters, in HUGS it is not possible to enter a declaration directly to the interpreter. Instead, all declarations must be written in files, and the `:load` and `:reload` directives are used to communicate these declarations to the HUGS interpreter.

Fig. 2 shows some representative HASKELL declarations, which we can imagine are in the file `Test.hs`. We will discuss these declarations in the context of some sample expressions that will be evaluated in HUGS after executing the directive `:load Test.hs`. Because the file `Test.hs` does not have any module declarations, the declarations in the file are interpreted relative to the default `Main` module.

```
Prelude> :load Test.hs
Reading file "Test.hs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
Test.hs
Main>
```

A line comment in HASKELL is introduced via the double dashes, `--`, and goes until the end of the line. Various comments are sprinkled throught `Test.hs` in Fig. 2.

In HASKELL, a name may be attached to any value via the syntax I `=` E, as in `a = 2 + 3`. Because HASKELL is a lazy language, the definition expression E (in this case, `2 + 3` is not evaluated until the the name I is required later (if ever). If it is evaluated later, the value is memoized so that it will be computed at most once. Evaluating a variable in the HUGS interpreter forces its value to be computed in order to print the value.

```
Main> a
5
```

The HASKELL syntax for abstractions is `\` I_{formal} `->` E_{body}, where the slash mark `\` was chosen because it resembles a Greek λ symbol. So `\ x -> x*x` is the HASKELL notation for a squaring function. The notation

`\` I_1 ... I_n `->` E_{body}

is sugar for the curried function

`\` I_1 `->` `\` I_2 `->` ... `\` I_n `->` E_{body}.

The declaration

I_{name} I_1 ... I_n `=` E_{body}

is syntactic sugar for the curried function declaration

I_{name} `=` `\` I_1 `->` `\` I_2 `->` ... `\` I_n `->` E_{body}

For example,

```
avg x y = x+y/2
```

is syntactic sugar for

```
avg = \ x -> \ y -> (x+y)/2
```

Function application is denoted by juxtaposition of function and argument(s). For example, `sq a` denotes the result of applying the squaring function to the value of `a`. Function application is left-associative, which is consistent with curried functions. For example, `avg 3 8` is parsed as `(avg 3) 8`.

```
Main> sq a
25

Main> avg 3 8
5.5
```

```
a = 2 + 3 -- declare variable a

sq = \ x -> x * x -- sugared form: sq x = x * x

fact 0 = 1     -- Recursive factorial
fact n = n * fact (n-1)

factIter n = loop n 1 -- Iterative factorial
  where loop 0 ans = ans
        loop num ans = loop (num-1) (num*ans)

isEven 0 = True  -- Mutually recursive functions isEven and isOdd
isEven m = isOdd (m - 1)

isOdd 0 = False
isOdd n = isEven (n - 1)

sumList = foldr (+) 0 -- list summation function

nats = 0 : (map (1+) nats) -- infinite list of natural numbers

twos = 1 : (map (2*) twos) -- infinite list of powers of two

fibs = 0 : 1 : (zipWith (+) fibs (tail fibs)) -- infinite list of Fibonacci numbers

data Tree a = Leaf | Node (Tree a, a, Tree a) -- tree datatype declaration
  deriving (Show, Eq) -- show and equality (==) functions on trees

testTree = Node(Node(Node(Leaf, 4, Leaf),
                     1,
                     Node(Node(Leaf,5,Leaf),
                          2,
                          Leaf)),
                6,
                Node(Leaf, 3, Node(Leaf, 7, Leaf)))

value (Node(_,v,_)) = v -- accessor functions for tree nodes
left (Node(l,_,_)) = l
right (Node(_,_,r)) = r

height Leaf = 0
height (Node(l,_,r)) = 1 + max (height l) (height r)

treeSum Leaf = 0
treeSum (Node(l,v,r)) = (treeSum l) + v + (treeSum r)

treeMap f Leaf = Leaf
treeMap f (Node(l,v,r)) = Node(treeMap f l, f v, treeMap f r)

-- infinite tree of integers in which every node has
-- its binary address as its value.
intTree = makeTree 0
  where makeTree n = Node(makeTree (2*n), n, makeTree ((2*n)+1))
```

Figure 2: Sample HUGS declarations in the file `Test.hs`.

Like OCAML, HASKELL has a case-based pattern-matching construct, which has the form:

```
case E_discriminant of
  P_1 -> E_1
    ⋮
  P_n -> E_n
```

As an example, here is the definition of a `swapList` function that swaps the first two elements of a list with at least two elements:

```
swapList = \ xs -> case xs of
                     [] -> []
                     [x] -> [x]
                     x:y:zs -> y:x:zs
```

Note that the clauses of the `case` construct are not separated by any sort of syntax (like the vertical bar that separates `match` clauses in OCAML). This is because HASKELL, unlike almost every other modern language, actually uses indedentation and whitespace to as a disambiguation aid in parsing.

It is rare to see explicit `case` constructs in HASKELL programs, because they are usually written in a sugared form as a sequence of function definitions with different patterns in the parameter position(s). For example, the sugared form of the above `swapList` function is:

```
swapList [] = []
swapList [x] = [x]
swapList (x:y:zs) = (y:x:zs)
```

All names declared in a file are defined in a single recursive scope. In Fig. 2, `fact` is an example of a recursive function definition, `isEven` and `isOdd` are mutually recursive functions, and `nats`, `twos`, and `fibs` are recursively defined infinite lists. Mutually recursive definitions – especially of non-function values – are much easier to handle in a lazy language than in a strict one. Local recursive bindings are introduced in HASKELL via the `where` clause, which appears in the `factIter` and `intTree` declarations in Fig. 2. The `where` clause is HASKELL's version of Scheme's `letrec`, OCAML's `let rec`, and HOILIC's `bindrec`. Interestingly, the concrete syntax of `where` has the local declarations *following* the body rather then *preceding* it.

```
Main> fact 5
120

Main> factIter 6
720

Main> isEven 10
True

Main> isOdd 10
False

Main> sumList [1,2,3, 4]
10

Main> take 20 nats
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]

Main> take 15 fibs
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]
```

The portions after this point are still under construction!

Haskell supports Ocaml-like sum-of-product data types via the `data` declaration. Here is a binary tree data type:

```
data Tree a = Leaf | Node (Tree a, a, Tree a)
  deriving (Show, Eq)
```

There are two tree constructors: the nullary `Leaf` constructor, and the unary `Node` constructor, which takes a triple of the left subtree, the root value, and the right subtree. The declaration `deriving (Show, Eq)` tells Haskell that string representations of trees (via the `show` function) and equality on trees (via the `==` function) should be automatically defined in a structural way. Here is a sample tree:

```
testTree = Node(Node(Node(Leaf, 4, Leaf),
                     1,
                     Node(Node(Leaf,5,Leaf),
                          2,
                          Leaf)),
                6,
                Node(Leaf, 3, Node(Leaf, 7, Leaf)))
```

Tree operations can be defined via pattern matching. For example, we can define functions that access the three parts of a tree node:

```
value (Node(_,v,_)) = v -- accessor functions for tree nodes
left (Node(l,_,_)) = l
right (Node(_,_,r)) = r
```

For example:

```
Main> testTree
Node (Node (Node (Leaf,4,Leaf),1,Node (Node (Leaf,5,Leaf),2,Leaf)),6,Node (Leaf,3,Node (Leaf,7,Leaf

Main> value testTree
6

Main> left testTree
Node (Node (Leaf,4,Leaf),1,Node (Node (Leaf,5,Leaf),2,Leaf))

Main> right testTree
Node (Leaf,3,Node (Leaf,7,Leaf))

Main> height testTree
0
Main> :r
Reading file "Test.hs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
Test.hs
Main> height testTree
4
Main> treeSum testTree
28
```

```
Main> treeSum (treeMap (2*) testTree)
56
Main>
```