Advanced Programming 2022

Haskell, Continued

Andrzej Filinski
andrzej@di.ku.dk

Department of Computer Science
University of Copenhagen

September 8, 2022

Today's topics

- Introduction to some more advanced, Haskell-specific features
  - Modules
  - Type classes
  - Laziness
    - Equational reasoning
  - Functional I/O principles
  - List comprehensions
- All useful to know about in own right.
- Provide important background for **monads**, next time.

Haskell's module system

- Relatively simple, compared to, e.g., Standard ML or OCaml modules.
- But quite sufficient for most practical purposes
  - Especially in combination with *type classes*
    - Cover many (but not all) uses of ML's *parameterized modules*
- Two main purposes:
  - Namespace management
    - Using same name for unrelated purposes at different points in big program
  - Abstraction management
    - Preventing unwanted exposure of implementation details
- Fundamental concepts: *imports* and *exports*.

Standard modules

- All Haskell code is type-checked and executed in context of some existing definitions of types and values.
- Some common definitions always visible: "standard Prelude".
  - Saw several examples last time: `pi`, `(+)`, `map`, `[]`, `Maybe`, …
- Large standard library of further functionality available:
  - *Utility* functions and data structures:
    - E.g., formatting, parsing, finite-set operations, …
    - Could in principle by reimplemented by ordinary programmer.
    - But probably not as competently: don't re-invent the wheel!
  - *OS interface* and *evaluation-control* functions:
    - E.g., directory listing, exception handling, threads, …
    - Implementation relies on special support from compiler and/or runtime system.
    - No way to re-implement from scratch in pure Haskell code.
  - Grouped into *modules*.

Importing from modules

- To use all or parts of a module, must explicitly *import* from it.
- "`import ...`" declaration(s) must be at very beginning of file.
- Bulk import:
  - `import System.Directory`
  - Makes everything from module available.
  - Names may clash with own definitions, or other imports.
    - Only get error on attempted *use* of ambiguous name.
  - Normally used for "framework" modules, such as parser combinators
- Selective import
  - `import System.Directory (getCurrentDirectory, doesFileExist)`
  - Only makes explicitly requested names available.
    - Remember to enclose symbolic names (e.g., <>) in *extra* parentheses.
  - Normally preferred if only need a few, unrelated functions from module in question.

Importing from modules, continued

- Qualified import
  - `import qualified Data.Set as S`
  - Like a bulk import, but prefixes all imported names with `S.`
    - `S.map :: Ord b => (a -> b) -> S.Set a -> S.Set b`
  - Avoids clash with (list-based) `map` from standard prelude
- Warning: top-level interactive loop is a bit special.
  - Can refer directly to names from arbitrary modules:

    ```
    ghci> System.Directory.getCurrentDirectory
    "/home/andrzej/teaching/ap2022"
    ```

    - (Aside: how is this even possible in a *purely functional* language?)
  - Will not work in file; need explicit import first.
  - **Tip:** when in a project directory, `stack ghci` (no `exec`!) preloads and opens all project modules.

Creating your own modules

- Start file containing related definitions with
  `module` *ModName* `(`*exports*`) where` *defs*
- *ModName* is the name of the module.
  - Should be the same as source filename (without trailing `.hs`).
  - Beware of Windows' case-insensitive filenames!
- *exports* is comma-separated list of names (types and/or values) to be made available to users (clients) of the module.
  - Often more readable to list one name per line, especially if many.
  - Use *`TypeName`*`(..)` to export a datatype together with all its constructors.
  - Omit export list entirely (including parens) to bulk-export everything defined in module.
- *defs* should start with any needed `import` declarations, as usual.

What to export from a module?

- Not specific to Haskell; general principles for API design.
- Export orthogonal set of functions useful to clients, not any internal “helper” functions you used to define them.
  - If you cannot concisely summarize what a function does, it shouldn’t be exported.
  - Arguably, it probably shouldn’t even have been defined in the first place...
    - Unclear and/or complex specifications for internal functions are a magnet for bugs.
  - Do try to formulate specification (including meanings/roles of all parameters!) in a comment
    - Forces you to consider what the function *should* be doing.
    - Surprisingly feasible in Haskell, because function’s *type* says everything about its possible interactions with rest of program.

Example of API considerations

- Suppose we are defining a module for integer-set operations, with exports:

  ```
  type IntSet
  empty :: IntSet
  singleton :: Int -> IntSet
  union :: IntSet -> IntSet -> IntSet
  member :: Int -> IntSet -> Bool
  ```

- For implementing `union`, may also have defined:

  ```
  insert :: Int -> IntSet -> IntSet
  ```

  Should it be exported? Maybe not.

- Client could themselves define nominally equivalent function:

  ```
  myInsert x s = singleton x `union` s
  ```

  Should be almost as efficient, uses only core operations.

  - If `myInsert` significantly slower than `insert`, maybe should improve performance of `union` in general.
    - E.g., always add elts of *smaller* set to *larger*, not vice versa.

Preventing leakage of implementation details

- Suppose we implement `IntSet` as *unsorted, duplicate-free* lists.
- Could just make definition in module:

  ```
  type IntSet = [Int]
  ```

  But that exposes to clients that an `IntSet` is actually a list.
  - In particular, this could evaluate to `False`:

    ```
    singleton 3 `union` singleton 4  ==
      singleton 4 `union` singleton 3
    ```
- Better: in implementation, define a *new* type, equivalent to `[Int]`.

  ```
  newtype IntSet = IS {unIS :: [Int]}
  ```
  - Almost same as `data` with a single constructor.
  - Note: *did not* include `deriving Eq` in definition!
- Export type `IntSet`, but *not* constructor `IS`, nor projection `unIS`
  - Only use internally in module, to define `empty`, `union`, etc.
- Clients can neither create new `IntSet` values, nor inspect existing ones, except through exported API functions.
  - But then, API should probably also include an equality test.

Overloading in Haskell

- Have already seen (sometimes implicit) examples of restricted polymorphic functions:

```
(==) :: Eq a => a -> a -> Bool
(+) :: Num a => a -> a -> a
show :: Show a => a -> String
```

- Haskell's type inferencer automatically keeps track of restrictions:

```
ghci> twice x = x + x
ghci> :t twice
twice :: Num a => a -> a
```

- In general, may have multiple constraints:

```
foo :: (Num a, Show a) => a -> String
foo x = show (x + x)
```

- Captures a uniform notion of *overloading*, where computation to be performed depends materially on types of operands and/or result.

Type classes

- A Haskell *type class* is an (open-ended) collection of types supporting a fixed set of operations.
  - Not entirely unlike *interfaces* in Java or C#.
- Declared with `class` *`ClassName typevar`* `where` *`decls`*
  - As usual, the *decls* should align vertically
- Several predefined classes, including (all slightly simplified):

```
class Show a where
  show :: a -> String

class Eq a where
  (==), (/=) :: a -> a -> Bool

class Num a where
  (+), (-), (*) :: a -> a -> a
  fromInteger :: Integer -> a
```

- Use `:info` *`ClassName`* in GHCi (or Hoogle) to see full list of operations.

Declaring class membership

- To include a (new or previously defined) type in a class, must add an *instance declaration*.
- Simply need to supply all the required operations of the class.
- Example (of course, better version exists in standard library):

```
data Complex = Complex {re, im :: Double}

instance Num Complex where
  (Complex r1 i1) + (Complex r2 i2) = Complex (r1+r2) (i1+i2)
  -- Defs of (-), (*), ...
  fromInteger n = Complex (fromInteger n) 0.0
```

  - **Note:** The `fromInteger n` on the RHS is *not* a recursive call, but an invocation of `fromInteger :: Integer -> Double` !
- Likewise,

```
instance Show Complex where
  show c = show (re c) ++ "+" ++ show (im c) ++ "i"
```

Numeric types in Haskell

- Actually, whole hierarchy of numeric type classes
  - `Num a`, for types `a` that have operations `(+)`, `(-)`, `(*)`
    - Mathematically: $\sim$ *rings*
  - `Fractional a`, for `Num`-types `a` that also have `(/)`
    - Mathematically: $\sim$ *fields*
  - `Integral a`, for types `a` that have `div`, `mod`
    - instances: `Int`, `Integer`, `Word` ($\approx$ unsigned `Int`), ...
  - ...
- Main oddity: even *literals* are overloaded!
  - Plain `42` actually behaves like `fromInteger (42::Integer)`,
- Therefore:
  - **OK:** `pi + 1                  -- 1 can have type Double`
  - **Not OK:** `pi + length "x"  -- length s has only type Int`
  - **OK:**
    `pi + (fromIntegral $ length "x")  -- explicit coercion`
    - Aside: `$` often useful to avoid deeply nested parentheses
    - Just a right-assocociative infix application operator: `f $ a = f a`

More type-class constructions

- Class inheritance
  - Can also constrain type variable in class declaration

    ```
    class Bar a => Foo a where ...
    ```

  - Only allowed to declare a type to be instance of `Foo`, if it's already an instance of `Bar`.
  - Ex: `class Eq a => Ord a where (<) :: a -> a -> Bool; ...`
- Default implementations
  - Can include a *default* definition of a class operation:

    ```
    class Eq a where
      (==), (/=) :: a -> a -> Bool
      x /= y = not (x == y)  -- and vice versa
    ```

  - In instance declaration, if we omit definition for `(/=)`, the default one is used
  - Note: default implementation may use operations of superclass.
- Both features a bit esoteric, but recent-ish API change for `Monad` class makes them unavoidable...

Automatically deriving instances

- Haskell can automatically construct *certain* boilerplate instance declarations for newly defined types.
  - Only for a few built-in classes (need compiler support)
- `data MyType = ...  deriving (Eq, Show, Read, ...)`
- Derived `Show`:
  - Displays values in a format parseable as source code.
  - E.g., `"Complex {re = 3.0, im = 4.2}"`
  - Whereas our custom `show` would return `"3.0+4.2i"`
- Derived `Eq`:
  - Structural equality (assuming all constituent types have `Eq` instances).
  - Usually fine, but sometimes want a coarser notion of equality.
  - E.g., in our module implementing `IntSet` as unsorted lists:

```
instance Eq IntSet where
  (IS xs) == (IS ys) = all (\x -> x `elem` ys) xs &&
                       all (\y -> y `elem` xs) ys
```

Monoids (in the good old days)

- Another common class: types with notion of "accumulation"

```
class Monoid a where
  mempty :: a
  (<>) :: a -> a -> a  -- aka. mappend

instance Monoid String where  -- or just Monoid [a]
  mempty = "" ; (<>) = (++)

instance Monoid Int where
  mempty = 0 ; (<>) = (+)      -- one possible choice

instance (Monoid a, Monoid b) => Monoid (a,b) where
  mempty = (mempty {-of type a-}, mempty {-of type b-})
  (a1, b1) <> (a2, b2) = (a1 <> a2, b1 <> b2)
```

- All `Monoid` instances a *should* satisfy, for all $x, y, z :: a$

  `mempty <>` $x \simeq x$, x `<> mempty` $\simeq x$,

  x `<>` (y `<>` z) $\simeq$ (x `<>` y) `<>` z

Monoids (in our brave new world)

- Trend to maximally subdivide class functionality:

```haskell
class Semigroup a where
  (<>) :: a -> a -> a  -- should be associative

class Semigroup a => Monoid a where
  mempty :: a    -- should be neutral elt. for mappend
  mappend :: a -> a -> a
  mappend = (<>) -- "default" implementation
```

- To declare new Monoid instance, need to split up the operations:

```haskell
instance Semigroup MyType where
  x <> y = ...

instance Monoid MyType where
  mempty = ...
```

- Or stick it to the Haskell powers-that-be:

```haskell
instance Monoid MyType where
  mempty = ...
  x `mappend` y = ...

instance Semigroup MyType where (<>) = myappend
```

Constructor classes

- Can also classify *type constructors* (parameterized types).
- Example: *functors*, for “container-like” type constructors

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where  -- type [a] is sugar for ([] a)
  fmap = map

data Tree a = Leaf a | Node (Tree a) (Tree a)
instance Functor Tree where
  fmap f (Leaf a) = Leaf (f a)
  fmap f (Node tl tr) = Node (fmap f tl) (fmap f tr)
```

  Then, `fmap odd $ Node (Leaf 2) (Node (Leaf 3) (Leaf 5))` evaluates to `Node (Leaf False) (Node (Leaf True) (Leaf True))`

- All `Functor` instances *should* satisfy (where `.` is function composition):
  `fmap id` $\simeq$ `id`, `fmap (`g `.` f`)` $\simeq$ `fmap` g `. fmap` f

Laziness

- Uncommonly, Haskell has a *lazy* ($\approx$ *non-strict*) semantics.
  - Subexpressions not evaluated until their values actually needed.
- To illustrate behavior, `undefined` is a (polymorphic!) predefined constant that causes a runtime error when evaluated.
- Sample interaction:

```
ghci> let x = undefined in x + 1
*** Exception: Prelude.undefined
ghci> let x = undefined in 3
3
```

- Even if everything terminates (eventually), lazy evaluation may avoid wasting work: `let x =` *`bigExp`* `in 0`
- But in `let x =` *`bigExp`* `in x*x`, Haskell will *memoize* ($\approx$ *cache*) value of `x` after first use, to avoid recomputation.
  - Only safe because *`bigExp`* cannot have side effects!
- Same behavior for function arguments ("call-by-need")

```
let f x = 42 in f (1 `div` 0) -- immediately returns 42
let f x = x*x in f (fac 10)   -- only computes fac 10 once
```

Lazy evaluation, continued

- Even when result of subexpression is used, it will only get evaluated just enough to allow computation to proceed:

```
let p = (undefined, 3) in snd p  --  returns 3

case Just undefined of
  Nothing -> False ; Just _ -> True  -- returns True
```

- In general, evaluation of all constructor arguments (including tuples and list nodes, but *not* `newtype`-constructors) is delayed.
  - Can inadvertently construct "booby-trapped" values that only explode when accessed.
  - Commonly: only when being printed as results.

```
ghci> let l = [10,20,undefined,40] in (length l, show l)
(4,"[10,20,*** Exception: Prelude.undefined
```

  - The top-level printer is *forcing* evaluation.
  - Apocryphal lecture by Simon Peyton Jones (Haskell pioneer): "This is a talk about lazy evaluation. Any questions?"

Streams

- In most practical situations, lazy vs. eager evaluation of functional program makes no difference.
  - Rare to evaluate a non-trivial subexpression, then never actually use its result (dead code).
- But lazy evaluation makes it particularly simple and natural to work with *infinite* lists (≈ *streams*).
- Just like functions can be recursively defined, so can list values:

```
ones, nats :: [Int]
ones = 1 : ones
nats = 0 : map (\x -> x+1) nats
```

- `ghci> take 5 nats` prints `[0,1,2,3,4]`
- `ghci> drop 5 nats` prints `[5,6,7,8,9,10,11,`... until interrupted.
- Again, the top-level printer drives the actual computation.

- Much formal and semi-formal reasoning about Haskell programs is about *equivalence* of expressions.
- When e_1 and e_2 are of same type, will write $e_1 \simeq e_2$ when they are equivalent.
- Equivalent expressions *mean* the same thing.
  - `x + y` $\simeq$ `y + x` (for `x,y :: Int`),
  - `[x, 3] ++ xs` $\simeq$ `x : 3 : xs` (for `x :: Int`, `xs :: [Int]`)
  - `map g . map f` $\simeq$ `map (g . f)` (for `f :: a -> b`, `g :: b -> c`)
- Special case: *evaluation* of complete expressions to values
  - E.g. `2+2` $\simeq$ `4`, `reverse [1,2,3]` $\simeq$ `[3,2,1]`
- Fundamental principle: can replace equivalent subexpressions for each other, without affecting meaning of program
  - E.g., `let y = 2 + 2 in map (\x -> x + y)` $\simeq$
    `let y = 4     in map (\x -> y + x)`

Equational reasoning, continued

- How to argue that two expressions are equivalent?
- Small collection of general principles, including:
  - $\simeq$ is *reflexive*, *symmetric*, *transitive*, and a *congruence*.
    - Can compactly write reasoning chains $e_1 \simeq e_2 \simeq \cdots \simeq e_n$
    - If $e_1 \simeq e_1'$ and $e_2 \simeq e_2'$, then $f\ e_1\ e_2 \simeq f\ e_1'\ e_2'$, for any f.
  - When definition `x =` e (local or global) is in scope, then $x \simeq e$.
    - E.g., `let x = 2+3 in x*x` $\simeq$ `(2+3)*(2+3) -- note parens`
    - E.g., `let x = undefined in 0` $\simeq$ `0`
    - Also works for *patterns* p on LHS of = (with caveat for _)
  - If $C\ e_1\ e_2 \simeq C\ e_1'\ e_2'$ (C a constructor), then $e_1 \simeq e_1'$ and $e_2 \simeq e_2'$.
    - E.g. if `([x], y)` $\simeq$ `([3], undefined)`, then `[x]` $\simeq$ `[3]` (and hence `x` $\simeq$ `3`) and `y` $\simeq$ `undefined`.
  - `(\`p `->` e_1`)` e_2 $\simeq$ `let` p `=` e_2 `in` e_1
    - E.g., `(\x -> x+1) (y*2)` $\simeq$ `let x = y*2 in x+1` $\simeq$ `y*2+1`
  - Usual arithmetic equalities (but beware of potentially undefined subexpressions)
    - `x + x` $\simeq$ `2 * x`; `x * 0` $\not\simeq$ `0` (consider `x = undefined`)

Introduction to Haskell I/O

- Haskell is a completely pure language, no side effects allowed.
- So how can we possibly write Haskell programs that interact with the real world?
  - File system, terminal, network, other OS services,....
- Answer: top-level result printer need not *itself* be pure!
- Can have pure program compute a lazy list (stream) of *I/O requests* (aka. *actions*) for top-level printer to perform.
  - *Producing* the list itself is effect-free; *obeying* it is not.
  - The list is inspected *incrementally*, as and when the program produces it.
    - So earlier I/O actions performed even if program later diverges
- Actually need a datatype slightly more complicated than a list, to allow program to also receive *input* from real world.

A SimpleIO type constructor

- Simplified version of actual Haskell `IO` type constructor.
- Three-way choice in *interaction tree*:

```
data SimpleIO a = Done a
                | PutChar Char (SimpleIO a)
                | GetChar (Char -> SimpleIO a)
```

- Sample value of type `SimpleIO Int`, ready to be performed:

```
PutChar '?' (GetChar (\x -> PutChar (toUpper x) (PutChar '!' (Done (ord x)))))
```

- REPL has following conceptual structure:
  - If top-level expression has an "ordinary" (non-`SimpleIO`) type, just evaluate it and print the result (incrementally).
  - If expression has type `SimpleIO a`, evaluate it just enough to expose top constructor. Then:
    1. If of the form `Done` x: evaluate and print x, like in previous case
    2. If of the form `PutChar` c s: output c, and continue evaluating s.
    3. If of the form `GetChar` f: input a c, and continue evaluating $f\ c$.
- But how do we write a big program of type, say, `SimpleIO ()`?
  - Seems awkward to generate all IO requests in functional style.
  - Next time: monads to the rescue!

List comprehensions

- Cute Haskell feature, allows many list-processing functions to be written clearly and naturally.
  - Subsequently adopted by many other languages, e.g., Python
- Inspired by mathematical notation for *set comprehensions*:
  - subset: $\{x \mid x \in \{2,3,5,7\} \wedge x > 4\} = \{5,7\}$
  - direct image: $\{x+1 \mid x \in \{2,3,5,7\}\} = \{3,4,6,8\}$
  - Cartesian product: $\{(x,y) \mid x \in \{2,3\} \wedge y \in \{\top,\bot\}\} = \{(2,\top),(2,\bot),(3,\top),(3,\bot)\}$
  - general union: $\{x \mid s \in \{\{2,3\},\emptyset,\{5\}\} \wedge x \in s\} = \{2,3,5\}$
- Can write Haskell expressions with almost same notation:
  - `[x | x <- [2,3,5,7], x > 4]` $\simeq$ `[5,7]`
  - `[x + 1 | x <- [2,3,5,7]]` $\simeq$ `[3,4,6,8]`
  - `[(x,y) | x <- [2,3], y <- [True,False]]` $\simeq$ `[(2,True), (2,False), (3,True), (3,False)]`
  - `[x | s <- [[2,3],[],[5]], x <- s]` $\simeq$ `[2,3,5]`

List comprehensions, continued

- Can even use all idioms on previous page together.

```
> [100 * x + y | x <- [1..4], x /= 3, y <- [1..x]]
[101,201,202,401,402,403,404]
```

- General shape: `[` *exp* `|` $qual_1, \ldots, qual_n$ `]`, where each $qual_i$ is:
  - a *generator*, *x* `<-` $lexp_i$, where $lexp_i$ is a list-typed expression; or
  - a *guard*, $bexp_i$, which must be a `Bool`-typed expression.
- Qualifiers considered in sequence, from left to right:
  - For each generator, bind variable to successive list elements, and process next qualifiers (∼ `foreach`-loop in imperative language)
  - For each guard, check that it evaluates to `True`; otherwise, return to previous generator (∼ conditional `continue` in imperative).
  - When all qualifiers successfully considered, evaluate *exp* and add its value to result list.
- Aka. depth-first search, backtracking, generate-and-test
  - Will see again in Prolog, parsing
  - Also an instance of programming with monads!

What now?

- Attend exercise sessions today after lunch, from 13:00(ish)
  - Check rooms on Absalon; not same as on Tuesday
    - If needed, do ad-hoc load balancing
  - No need to come on time: no scheduled activities
- Get started on Assignment 1, **due 22:00 on Friday, Sept. 16**
  - Talk to a fellow student about forming an assignment group (two is max)
  - Detailed (group-)submission instructions coming,
- Work on (remainder of) Exercise Set 1 as and when time permits
- Use Absalon/Discord fora for questions after exercise hours
- Next lecture: monads!
  - Recommended reading materials on Absalon
    - "The Essence of Functional Programming" may be much more relatable *after* working on Assignment 1 (main part).