Deep Strictness: Milestone 2

Kenny, Hengchu

November 14, 2017

We aim to extend QuickCheck with the ability to probabilistically verify the laziness/strictness behavior of Haskell functions. That is, we will allow users to write a specification of the strictness behavior of a particular function, and by fuzzing inputs to the function, determine if the function is *exactly* as strict as the user has specified.

Before describing in detail how our proposed tool will work, we give several definitions regarding laziness in Haskell:

A lazy function takes an *input* to an *output*. For now, we consider only inputs and outputs restricted to single algebraic datatypes which do not contain functions (that is, all functions we can speak about are unary first-order functions). We plan to extend this to multi-argument and higher-order functions in later stages of this project.

When a lazy function is applied to an input, it is done so in a *context* which makes use of some part of its output—that is, the context may not use the entire output data structure to continue its own computation. We call the portion of the input which is used by the context the *demand* on the output. The semantics of lazy evaluation dictate that any portion of the input which is not required to compute the demanded portion of the output are not evaluated in the first place—that is, evaluation only occurs as much as is necessary to satisfy the demand of the calling context.

For example, let us consider a function `f :: Int -> Int -> Int` where `f x n = fst (x, fib n)`. Although computing the second component of the pair in `f 1 100000000` would be very expensive, this computation never occurs because the demand placed upon the pair (extracting the first component) only requires the first component to be evaluated. We say that `f` is *strict* in its first parameter (`x`) and *lazy* in its second parameter (`n`).

We can capture this notion formally as a function from a given demand of the output to a demand of the input. We call this a *demand specification*. Here, we can concretely represent a demand as a *subshape* of the input/output—that is, a data structure corresponding to one of these data types, but which can be truncated at any arbitrary position in the structure of the data type. We call this derived data type the *demand type* corresponding to a data type.

For example, the corresponding demand type for the type `(Int, Int)` would be:
`Demand (Demand Int, Demand Int)`
where

```
data Demand a = ~    -- ^ Not demanding
              | ! a  -- ^ Demanding the constructor at this level
              | *    -- ^ Demanding a primitive or nullary constructor
```

All the values of these types are: `~`, `(~, ~)`, `! (*, ~)`, `! (~, *)`, and `! (*, *)`. These represent all of the possible demands which a context could place on the type `(Int, Int)`.

Suppose we have a function `g :: (Int, Int) -> Int`. The corresponding demand specification for g would be a function `gSpec :: Demand Int -> Demand (Demand Int, Demand Int)`.

Concretely, consider the function `fst` (for integers):

```
fst :: (Int, Int) -> Int
fst (a, b) = a
```

An accurate demand specification for `fst` is:

```
fstSpec :: Demand Int -> Demand (Demand Int, Demand Int)
fstSpec ~ = ~
fstSpec * = ! (*, ~)
```

This states that if we demand nothing from the result of `fst`, we do not need to evaluate its argument at all. (This is true for *every* function in a lazy language!) However, if we demand the resultant integer from `fst`, we must evaluate the first integer in the input pair, but we do not need to evaluate the second element of the pair.

For functions like `fst`, the demand on input is a *static demand*: that is, for each demand on the output, any input to the function will be evaluated to the same degree.

This is not true in general, and not merely in pathological examples. Many (perhaps most!) useful functions have dynamic demand behavior—that is, their demand specification depends on the values of the input. We say that such functions have *dependent demand*: that is, the demand specification of these functions must consider the input values of the function as well.

```
take :: (Int, [Int]) -> [Int]
take (n, xs) =
  case (n, xs) of
    (0, _)     -> []
    (n',x:xs') -> x:(take ((n'-1), xs'))
    (n', [])   -> []
```

Just given some demand on the output list, we can't guess the demand on the second element of the input pair, although we do know that the first element of the pair will always be evaluated.

Suppose we have demand the first 2 elements of some result of take, there is no one input demand that corresponds to this output demand. However, if we know what the first argument to `take` is, we can determine the degree to which the input list must be evaluated. Note that the demand on the output still is another factor in determining the demand on the input, since if we demand nothing on the output, then nothing on the input will be demanded.

The demand specification of `take` could be

```
takeSpec :: (Int, [Int]) -> Demand [Demand Int] -> Demand (Demand Int, Demand [Demand Int])
takeSpec _        ~    =  ~
takeSpec (0, _)  ![]   =  !(*, ~)
takeSpec (n, xs) !ds   =
  case (ds, xs) of
    ([],     _)    -> !(*, ![])
    (d:ds', x:xs') ->
      let !(_, !ds'') = takeSpec (n-1, xs') !ds' in
      !(*, !(d:ds''))
```

Determining statically whether a function meets its demand spec is undecidable due to the Halting Problem. So, instead of statically approximating these behavior, we choose to apply runtime instrumentation with randomized testing, this gives a high level of confidence.

We choose to implement this as an extension to QuickCheck (CITATION). We will use generic programming to derive the demand types of functions under consideration. We will use existing QuickCheck generator infrastructure to fuzz inputs to the function, and demands on the output of the function. Because only certain shapes of demand are realizable on a given output, we specialize the demand generators to produce only sensible demands on each invocation of the function. We then run the function on the random input to obtain un-evaluated output, and place a random demand on this output while instrumenting the function to determine the resultant demand on the input. We can then use the demand specification to check whether the demand specification exactly matches what the actual demand behavior of the function was in this trial.

Because we integrate with the existing functionality of QuickCheck, we can employ its *shrinking* abilities to produce minimal counterexamples of input/demand pairs which fail to satisfy a given demand specification, if this specification is faulty.

Note that there are two ways in which a specification may be faulty: it may fail to produce an input demand which exactly matches the real observed input demand, or it may fail to cover all possible realizable combinations of input and demand.

To avoid unnecessary metaprogramming, we choose not to calculate an instrumented version of the function under consideration. We do not need to resort to such strategies because we can instead instrument the input data structure to report to us how it is evaluated by the function. As such, we may treat our functions as a black box.

In order to instrument a lazy data structure, we must use so-called "unsafe" features of Haskell—in particular, the function `unsafePerformIO` which allows us to attach a callback to a particular lazy value so that the callback is run when that value is evaluated. We may traverse the generated input data structure to add such a callback to every node in that structure. By tracking which of these callbacks are invoked when we demand a particular part of the output of some function applied to the instrumented structure, we produce an exact representation of the input demand of this function given a particular output demand and input value.

Once we have obtained a pair of generated output demand and observed input demand, it's trivial to run the demand specification on the output demand and check whether it exactly matches the observed demand. If it does not, we may repeat this process by shrinking inputs to the function to compute a minimal example exhibiting the detected violation of the demand specification.

We need not only to consider how to *check* such specifications, but also how to specify them in the first place! To that end, we will use Haskell's generic programming features to provide a syntax for users to succinctly write these demand specifications without worrying about the details of our implementation. Ideally, this syntax would look very similar to the syntax we use above, but we may need to make some concessions due to the limitations of the implementation strategy we choose for this interface.

Moving forward, we want to allow users to check demand specifications of multi-argument functions, higher-order functions, and functions on data structures which themselves contain functions.

1 Progress as of Second Milestone

As of now, we have successfully implemented a proof-of-concept for the instrumentation of functions over lists. By making use of a mutable pointer-based data structure, we managed to implement this instrumentation such that the execution of the instrumented function is a constant multiplicative factor slower than the execution of the uninstrumented function. We have also (mostly) implemented the module which will enable us to handle instrumenting functions of an arbitrary number of arguments. Tomorrow, we plan to meet with José Manuel Calderón Trilla tomorrow, who's done quite a bit on strictness analysis. We are also anticipating collaborations with Leo Lampropoulos (expert in property based random testing) and Antal Spector-Zabusky (well versed in generic programming in Haskell).

Here are some examples of functions that operate on the `List` data type. Each of these functions demonstrate different strictness on the input list. For example, the lazy `Context` doesn't inspect the input list at all, and thus is completely lazy. By contrast, `spineStrict` evaluates the structure of the list without forcing its contents to be evaluated. A more complex example of a demand context is `evenStrict` which, like `spineStrict`, evaluates the entire structure of the list, but which also evaluates every other element contained in the list.

```
type Context a = a -> ()

lazy :: Context [a]
lazy = const ()

whnf :: Context [a]
whnf = flip seq ()

nthStrict :: Int -> Context [a]
nthStrict n = flip seq () . (!! n)
```

```
spineStrict :: Context [a]
spineStrict = flip seq () . foldl' (flip (:)) []

allStrict :: NFData a => Context [a]
allStrict = rnf

oddStrict :: NFData a => Context [a]
oddStrict []           = ()
oddStrict [x]          = rnf x
oddStrict (x : _ : xs) = rnf x `seq` oddStrict xs

evenStrict :: NFData a => Context [a]
evenStrict []           = ()
evenStrict [_]          = ()
evenStrict (_ : x : xs) = rnf x `seq` evenStrict xs
```

To observe the behavior of these functions, we used the `demandList` function on each of the contexts defined above. We evaluate the same function (`take 6`) on the same data (`[1..5]`), but with differing demands on the result.

```
mapM_ printDemand_primList $
   map (\context -> demandList context (take 6) [1..5]) $
      [lazy,
       whnf,
       spineStrict,
       nthStrict 2,
       allStrict,
       oddStrict,
       evenStrictt]
```

We then print a representation of the demand on the *input* which is incurred by this evaluation. An "O" indicates that the data at that list cell was forced in the corresponding context, and an underscore indicates means that the data at its list cell was not evaluated. An ellipsis indicates that the rest of the list was not evaluated, while a closing square bracket indicates that the list was evaluated all the way down to its end.

```
lazy:         ...
whnf:         [_, ...
spineStrict:  [_, _, _, _, _, _]
nthStrict 2:  [_, _, 0, ...
allStrict:    [0, 0, 0, 0, 0, 0]
oddStrict:    [0, _, 0, _, 0, _]
evenStrict:   [_, 0, _, 0, _, 0]
```

The process of *currying* takes a function of one argument which returns another function of one argument and converts it into a function taking two arguments and returning a result. Conversely, *uncurrying* takes such a function of two arguments and converts it into a function of one argument which returns another function of one argument.

With clever type-level programming, we can generalize currying to functions with arbitrary numbers of arguments. This is necessary to enable our library to handle testing such "curried" functions, which are ubiquitous in Haskell. As such, we have implemented the following (non-trivial) functions, which will be a necessary part of our internal infrastructure.

```
curryAll   :: function -> (Tuple (Args function) -> Result function)
uncurryAll :: (Tuple (Args function) -> Result function) -> function
```

There is a mechanical transformation between the type of a function and the corresponding type of its demand specification. Similarly, there is a mechanical transformation from the type of a value to the type of the representation of the demand upon it.

In order to perform these transformations, we use a feature in Haskell called Type Families. Type Families allow us to write recursive functions over the structure of types to compute other types. The Haskell compiler GHC has another feature called Generics, which transforms any type to a generic representation that uses sums, products and arrows. We have named our two Type Families `Spec` and `Demand` respectively, and here is their definition.

```
data Thunk a = T | E a

Demand (a :+: b) = Demand a :+: Demand b
Demand (a :*: b) = Thunk (Demand a) :*: Thunk (Demand b)
Demand (a -> b)  = FuncDemand

Spec (a :+: b)   = Spec a :+: Spec b
Spec (a :*: b)   = Spec a :*: Spec b
Spec (a -> b)    = (a, Spec a) -> Demand b -> Thunk (Demand a)
```

We use the data type `Thunk` to introduce an additional layer of indirection. A value of type `Demand a` can only ever be a prefix of its corresponding value at the type `a`, and thunks represent the locations where the remaining structure can be chopped off by a lack of further evaluation of the data structure.

With higher-order functions, uncurrying all of the arguments of the higher order function before applying `Spec` is critical to generate the most fine-grained type of its demand specification. Consider the example `map` which maps a function over a list.

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

If we don't uncurry `map`, then its specification will have type

```
(a -> b, (a, Spec a) -> Demand b
                     -> Thunk (Demand a))
-> Demand ([a] -> [b])
-> Thunk (Demand (a -> b))
```

Figure 1: curried `map` specification type

This type just simplifies to

```
(a -> b, (a, Spec a) -> Demand b
                     -> Thunk (Demand a))
-> FuncDemand
-> Thunk FuncDemand
```

`FuncDemand` only captures whether the function was called or not. However, in a typical implementation of `map`, the usage of the input function depends on whether the input list is empty or not. It's impossible to specify the demand behavior of Haskell standard library's implementation of `map` using this type signature.

However, consider the uncurried `map`, which has type

```
(a -> b, [a]) -> [b]
```

Applying `Spec` to this type produces

```
((a -> b, [a]), ((a, Spec a) -> Demand b
                             -> Thunk (Demand a), Spec [a]))
-> Demand [b]
-> Thunk (Demand [a])
```

Figure 2: uncurried `map` specification type

Here, we see that this type in fact takes the input list `[a]` as an input to the specification as well! This allows a precise and correct specification of `map`. This observation applies to general higher-order functions.

We will integrate the `Spec` and `Demand` Type Families with GHC's generics mechanism to automatically derive specification types of algebraic data types. We will then write generic generators for representations of specifications and demands, and then we will develop a presentable API for programmers to write property based test cases to check the demand behaviors of Haskell programs.