# Haskell is Not Not Not ML

Su Lin Blodgett, Sara Burns, and Sravanti Tekumalla
*CS 251 Principles of Programming Languages Final Project*

## I. Introduction

After spending a large part of the semester learning about the functional programming paradigm, we decided for our final project to extend our knowledge of functional programming concepts through studying Haskell. Specifically, our goals were to learn the basic language features of Haskell and compare Haskell to SML, which we learned throughout the semester. Language features we compared between SML and Haskell include typeclasses, laziness, purity, functors and monads. In order to effectively compare these language features, we opted to build an interpreter in Haskell modeled on our SMiLe interpreter, which was for a dynamically-typed subset of the SML language. In addition, we extended our SMiLe interpreter to support lazy evaluation and added a typecheck function, which takes a SMiLe expression and a SMiLe type and checks whether the expression has its corresponding type.

By extending our SMiLe interpreter and building a similar-style interpreter in Haskell, we came away with several insights. Haskell and ML share many commonalities in their language implementation: both languages support pattern matching and use the Hindley-Milner type inference system for static type checking . Both languages are also distinct: we also found that Haskell's purity led us to use a monadic approach in writing our "eval" function in order to handle errors in a pure way. The difference in evaluation semantics between Haskell and SML led us to build a promise datatype in our SML interpreter which we used to implement laziness. In the next section of our paper, we will discuss the specific language differences and implications for each language feature we examined.

## II. Introduction to Haskell

Haskell is a statically typed, purely functional programming language. As discussed in the previous section, there are many similarities between Haskell and SML. Assuming the reader is proficient in SML, we will focus on three key differences between Haskell and SML:

1) Typeclasses and functors

   Haskell solves the issue of overloading with type classes; users have the ability to introduce polymorphism in their program, defining multiple functions with the same name but different types. Somewhat analogous to Java's Interface, a type deriving a typeclass in Haskell means that the type implements the behavior of the typeclass. In SML, a similar mechanism to introduce polymorphism is its module feature.

   SML and Haskell both feature functors, which relate closely to types and typeclasses, although in different ways. In SML, the module system allows us to define structures, which implement new datatypes and functions dealing with these datatypes, as well as signatures, which serve as user-facing specifications of these datatypes and functions. An SML functor is a function that takes one or more structures as a parameter and uses the values and functions in those structures to build a new structure. Thus a functor is a function from structures to structures. In contrast, a Haskell functor aligns closely with the category theoretic notion of a functor; specifically, in Haskell we have categories consisting of types and functions between those types, and functors are maps from categories to categories that satisfy a few laws.

2) Evaluation semantics

   Haskell is a lazy-by-default language. It utilizes both non-strict evaluation semantics and lazy evaluation. Nonstrict semantics delay evaluation until absolutely necessary. When expressions are bound to variables, they are not immediately evaluated, but rather, are delayed until the value is actually needed. In contrast, SML is an eagerly evaluated language, meaning that expressions, when bound to variables, are immediately evaluated, regardless of whether they are used. The route Haskell takes to nonstrict semantics is lazy evaluation. Rather than immediately returning the result of evaluating an expression, it first wraps the expressions in a thunk, a function which takes in nothing and, when evaluated, will evaluate the original expression. Then, when the thunk is evaluated, it simply evaluates to itself, without doing the work of evaluating the expression. Along the same lines, lazy evaluation evaluates an expression at most once.

3) Purity
   A big idea in functional programming is the notion of purity. In a pure world, the result of a function should rely only on the input given, with no capability for causing side effects. A side effect is a broad term that can refer to mutability, I/O, raising exceptions, or any interaction with the world of the program outside of the function. Haskell is a pure language, meaning it strictly prohibits the use of side effects in a program. SML, in

contrast, allows side effects, such as the use of print statements and error handling. Consequently, an SML program can easily raise exceptions and print output. In Haskell, these pursuits require the use of a device called monads, which are discussed in detail in the case study.

Each of these topics are discussed in greater depth below in our case study.

## III. Laziness

One of the major features we wanted to explore in Haskell was the idea of a lazy-by-default language, which attempts to minimize the computational effort used by a program by refusing evaluation until forced. In order to better understand what laziness entails, and how it separates Haskell from SML, we explicitly extended our SML interpreter to include laziness. In our Haskell interpreter, however, the necessity of adding laziness was not clear. In fact, implementing laziness in our interpreter would have been quite complex. Haskell is nonstrict, so the expressions returned when a pattern is matched in the eval function are already returned as thunks. To test laziness in our Haskell interpreter, we wrote a test case which indicates that arguments are evaluated in a function application even if they are not used in the body of the function. Additionally, our Haskell interpreter can evaluate an expression more than once. Laziness cannot be implemented as it was in SML, because Haskell does not include mutable references.

SML Code:

```
type 'a promise = 'a option ref * (unit->'a)
fun delay thunk = (ref NONE, thunk)
fun force (value, thunk) =
    case !value of
        NONE => let val v = thunk ()
                    val _ = value := SOME v
                in v end
      | SOME v => v
```

Much of our laziness implementation in SML was borrowed from the version implemented in lecture. We included a polymorphic promise type which consists of a mutable option reference and a thunk, as well as two functions delay and force. Delay takes in a thunk and returns a new promise created from that thunk. When a promise needs to be forced, the thunk is applied to unit, saving the value of the delayed expression into the option reference. If the expression is forced again, the result will not be recomputed, but simply pulled from the reference.

```
eval env (Neg e) = delay(fn() =>
                    (case force(eval env e) of
                      IntV i => IntV (~i)
                    | _ => raise TypeError))
```

The results of evaluation are explicitly delayed. When a result is needed, in this case for pattern matching, the delayed expression is forced.

```
factlang =
    LetE ([Fun "fact-broken" x
        (TimesE (VarE "x")
            (ApplyE (VarE "fact")
                (MinusE (VarE "x") (IntE 1)))),
                    (Fun "lazy" "x" (IntE 7))])
         (ApplyE (VarE "lazy")
         (ApplyE (VarE "fact-broken") (IntE 7)))
```

This piece of code, written in our Haskell interpreter, tests laziness by applying a function to an argument that, when evaluated, will enter an infinite recursion. If the Haskell interpreter was lazy, it would note that the argument is never utilized in the body of the function, and would not evaluate it. However, evaluating this expression does cause an infinite loop, so the interpreter is clearly not lazy.

### *Discussion*

Haskell's lazy evaluation model is designed for efficient computation. Only values that are really needed are evaluated, meaning unnecessary computation is eliminated. What's more, an expression is only evaluated once, which also saves computation time. Lazy evaluation allows Haskell to work easily with infinite data structures like streams, because it does not attempt full evaluation. Of course, there are drawbacks to lazy evaluation, and it cannot always be utilized. Converting to thunks requires significant additional overhead, and pattern matching requires eager evaluation. Haskell has a lazy pattern matching feature, but a lazy pattern will make any value it is passed, making it a treacherous endeavour. One of the greatest side effects of laziness has been keeping Haskell pure. Laziness cannot exist without purity, because purity enforces referential transparency  the guarantee that whenever you call a function on a given input, it will give the same result. In a language with side effects, this cannot be guaranteed, so order of evaluation is relevant. This connection between purity and laziness is why functional languages fall into two categories: lazy and pure versus eager and impure. As seen by the difficulty emulating monadic handling in our Haskell interpreter, purity is difficult to maintain. If not absolutely required, it is tempting to allow side effects to creep into a language. However, a pure approach adheres much more closely to the ideals of functional programming, and allows for clean expression of complex ideas.

## IV. Monads

While monads can be implemented in both Haskell and SML, monads are included as a library-level feature in Haskell, so we will focus our discussion of monads on Haskell. Generally speaking, monads are an abstraction that exists to allow programs to perform side effects while preserving purity. In other words, monads distinguish the pure versus impure aspects of a program. Popular monads in Haskell include the Maybe monad for when a program has the possibility of returning Nothing, analogous to SML's option type, and the I/O Monad to perform read and print operations.

Structurally, monads have three components: a monad type, a return function and a bind function. The return and bind (denoted $>>=$ in Haskell) functions are of the following format, where m is the monad type[1]:

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
```

The return function takes an item and returns the item wrapped in a monad type. The bind function takes a monad type and a transformation to apply to the input to produce a monad type with the transformed input. The bind function bears some resemblance to the map function, $(a- > b)- > [a]- > [b]$. The bind transformation happens sequentially and allows the programmer to chain actions step by step, much like in the style of imperative programming. More specifically, bind uses the value of a monad type to bind to the argument of the next lambda expression in the sequence of actions. This is supported in Haskell through the do statement, which is syntactic sugar for multiple calls of bind. In this way, monads can be thought of as a way to support a series of actions in a program.[2]

Notably, the monad structure implies that no value can be extracted from a monad once the return function is called on it. That is, one cannot call return, bind, and then receive the transformed value without the monad type wrapper. The reason why this makes sense is because monads allow for side effects, or for actions to take place within a computational context. This computational context is independent of the program; to allow the programmer to be able to extract the resulting value would allow a mixing of pure and impure aspects of the program, thus violating Haskell's purity.

To experience how monads work in Haskell and to program with them first hand, we decided to implement our eval function in our interpreter to handle exceptions by using the Maybe monad. As noted above, the Maybe monad allows the program to return Nothing or Just if there was no error with the computation. To implement the Maybe monad in Haskell, we had each of our eval cases return a Maybe value instead of a value. We also changed our Value datatype (TupleV [(Maybe Value)], TagV Sym (Maybe Value), ClosureV Environment (Maybe Sym) Sym Expr) to reflect the return values of the eval function.

The structure of our eval function before we used Monads relied on raising an error if the type wasn't the expected type:

```
eval env (NotE e) =
      (case (eval env e) of
       BoolV b  -> BoolV (not b)
       _   -> error "TypeError")
```

After implementing the Maybe monad, the structure of our eval function changed to returning Nothing if the type wasn't the expected type:

```
eval env (NotE e) =
     (case (eval env e) of
         Just (BoolV b)  -> Just (BoolV (not b))
          _  -> Nothing)
```

By replacing error TypeError with Just or Nothing values, we are able to stop the evaluation of eval if its argument is Nothing, with the function returning Nothing instead of the program crashing completely.

### *Discussion*

In an earlier implementation of our interpreter, instead of using the Maybe monad, our interpreter simply declared error TypeError. Why, then, did we choose to implement monads in our interpreter if we could simply declare an error? First, to use monads is consistent with an idiomatic use of Haskell. Secondly, monads are consistent with the modular principle of functional programming. By fully discerning the pure versus impure aspects of a program and separating actions from computations, computations can be composed of other computations instead of also having to take into account the side effects. Lastly, Monads allow the user to implement computations in an imperative style with the use of Haskell's do statement. For a purely functional programming language like Haskell, monads are a great fit for a library-level feature because of its emphasis on purity. However, for a language like SML, which is impure and allows the use of side effects, monads are not as frequently used, which makes sense  unless the programmer is interested in separating out the side effects, it is simply easier to raise an exception or to use the I/O libraries provided by SML.

## V. Typeclasses and Functors

As mentioned in the introduction, SML features a module system as a mechanism for introducing polymorphism. This module system, which permits us to create abstract data types, is one of SML's most powerful tools. We can create structures, which permit us to collect together values, exceptions, and functions into a logical unit. The type for a structure, which specifies the types of these values and functions, is a signature, and the implementation of these values and functions in the structure must obey the specification given in the signature. Together, this is a powerful way both to organize code into logical collections of values and functions, but also to abstract away the implementation of these values and functions away from the user, offering only a promise of their ultimate behavior. In this way, we can rely on the signature's guarantee of the behavior of the type and its functions, and every structure of this signature must implement everything promised by the signature.

In contrast, Haskell has a system of typeclasses, and a type deriving from a typeclass implements that typeclass's behavior. Haskell comes with a standard set of typeclasses, such as Eq and Show. A type that is an instance of the Eq class, for example, has an overloaded operator $==$ defined on it, while a type that is an instance of the Show class is one that can be converted to character strings. This is Haskell's mechanism for introducing polymorphism, by permitting multiple definitions of a function according to type.

Though functors are a language feature of both SML and Haskell, in some ways they represent a naming collision, since only in Haskell are they deeply rooted in category theory. We will examine functors in both languages.

Functors, along with structures and signatures, are a third component of the module system and allow us to connect structures in a logical fashion. Essentially, a functor is a function from structures to structures, taking as its input one or more structures and returning another structure. Thus, a functor permits us to construct a structure using values and functions from other structures, facilitating reuse of code and the ability to reason about a structure's behavior by understanding the behavior of the functor's input structures.

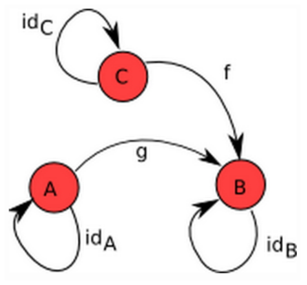
For example, suppose that we have defined a Stack structure. It is then very natural to want to build a Queue structure in terms of this Stack structure, rather than building it from scratch, because the two structures share many behaviors. Thus, we can create a function QueueFn which takes in the Stack structure as a parameter and uses the values and functions from Stack to construct a Queue[3].

```
signature STACK =
   sig
     type 'a stack
     val empty : 'a stack
     val isEmpty : 'a stack -> bool
     val push : ('a * 'a stack) -> 'a stack
   end

signature QUEUE =
   sig
      type 'a queue
      val empty : 'a queue
      val isEmpty : 'a queue -> bool
      val enqueue : ('a * 'a queue) -> 'a queue
   end

functor QueueFn (S:STACK) =
   struct
     type 'a queue = 'a S.stack * 'a S.stack
     val empty : 'a queue = (S.empty, S.empty)
     fun isEmpty ((s1,s2):'a queue) =
       S.isEmpty s1 andalso S.isEmpty s2
    fun enqueue (x:'a,
                (s1,s2):'a queue):'a queue =
       (S.push (x,s1), s2)
 end
```

Having created the functor QueueFn, which used the values and functions of the Stack structure to create a structure Queue, we can easily instantiate such a structure with the following:

```
structure s1 = Stack
structure q1 = QueueFn(s1)
```

In contrast, functors in Haskell relate categories together. We therefore take a quick detour into the basics of category theory.[4] Informally, a category is a collection that obeys three properties. First, it contains a collection of objects. Second, it contains a collection of morphisms, which take one object to another object. (These morphisms are shown below as arrows (and in fact in category theory they are generally referred to as arrows), while the objects are shown in red.) Morphisms are frequently functions, although they need not be. Third, morphisms must be composable with one another.



In addition, there are three laws that categories must obey: associativity under morphism composition, closure of the category under composition, and the existence of an identity morphism for every object which is an identity under composition with any other morphism. (For details, see the Wikibooks page).

One of the most natural categories we can imagine is the category whose objects are sets and whose morphisms are simply all functions between sets. For Haskell, the category we are interested in is Hask, whose objects are all Haskell types and whose morphisms are Haskell functions. Thus a Haskell morphism is an ordinary function $f :: A \to B$. From this definition we can easily check that the three category laws are satisfied (the second through Haskell's (.) function composition and the third through Haskell's id function).

We can now define functors, which take categories to categories. In particular, a functor must act on both a category's objects and its morphisms; a functor must take any object in its domain to an object in its codomain, and it must take any morphism in its domain to a morphism in its codomain. In addition, $F$ must take an identity function of an object to the identity function of its image object, and $F(f \circ g)$ must equal $F(f) \circ F(g)$ for any morphisms $f, g$ in the domain.

Despite the relatively lengthy introduction so far, we will see that these properties are critical to understanding functors in Haskell, since they are defined in close alignment with this functor definition. In particular, in Haskell a functor is a map from the category Hask to one of its subcategories. Recalling that Hask is the category containing all Haskell types and functions between those types, we can now define a functor as a map between Hask and a subset of its types (and the functions between those types). For example, the list functor is a map from Hask to Lst, the subcategory containing just list types, as well as the associated functions which act on list types.

Perhaps a more interesting Haskell functor is the Maybe functor. How is this a functor? From our definitions, we know that to be a functor, it need only do two things: first, take the objects of the Hask category  that is, types  to other types, and second, to take functions between types to other functions between types. Looking at the code for the Maybe functor below,

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

instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap _ Nothing  = Nothing
```

We see that the functor satisfies those two conditions. It takes any type T to a Maybe T, and fmap takes any function $A \to B$ and to a function $MaybeA \to MaybeB$. We can also check that the two additional axioms for functors are satisfied. Thus Maybe, as improbable as it may seem, is a functor.

Thus, in contrast to SML functors, which map structures to structures[5], Haskell functors map types to types (that preserves their corresponding functions), providing us with a notion of which types can be mapped over. In particular, a Haskell functor such as Maybe tells us that for any function between objects that we might have, we are provided a route to a corresponding function over the Maybe's of those objects.

### *Discussion*

In replicating the SML interpreter in Haskell we defined datatypes, and quickly realized that in defining Haskell datatypes we needed to also declare from which typeclasses our datatypes derived. For each datatype, therefore, we had to make a conscious decision about whether it belonged to the typeclass of things that could be compared for equality (Eq), whether it belonged to the typeclass of things that can be converted to character strings for I/O (Show), and so on. We found that the requirement of making these deliberate decisions to be illuminating, since in SML we generally did not have to think about these properties for datatypes except when creating modules.

In SML, we built a signature and a structure which define and implement a datatype called MyArray, which predictably implements an array-like data structure using SML lists. We then defined a signature for a linkedList type and built a functor which took the MyArray structure as a parameter and used the MyArray functions to construct the linkedList functions. One thing which after some time became very clear to us was the strength of the abstraction provided by the signature interface; while it was very tempting to access the list implementation of MyArray in order to create the list implementation of linkedList, it was impossible to do so, since the signature abstraction guaranteed us the behavior of the MyArray datatype without allowing us access to the implementation. Since the construction of the linkedList functions therefore required us to use the MyArray functions (implemented as lists), we ended up creating inefficient code whose writing process was nevertheless highly instructive.

## VI. Conclusion

Through our exploration of Haskell, we realized that while Haskell shares many commonalities with SML, the two programming languages are distinct in several ways, as outlined in our case study. Some of the strengths of Haskell are its purity and lazy evaluation. By separating the pure versus impure components of a program, code in Haskell becomes more compositional. In addition, Haskell's lazy evaluation means that expressions are evaluated once, at most, lending itself nicely to using infinite data structures. Haskell is also useful for concurrent programming due to its reliance on monads.
Through this project, we walked away not only with a better understanding of Haskell as a programming language, but with a greater understanding of the functional programming paradigm.

## Acknowledgment

The authors would like to thank Ben Wood for his help and insight throughout the development and implementation of this project.

## References

[1]Learn You a Haskell, the Monad Type Class,
http://learnyouahaskell.com/a-fistful-of-monads
[2]Learn You a Haskell, do notation,
http://learnyouahaskell.com/a-fistful-of-monads
[3]Example code gratefully borrowed from
http://www.cs.cornell.edu/courses/cs312/2006fa/recitations/rec08.html
[4]This detour borrows very heavily from the Wikibooks page Haskell/Category Theory at:
http://en.wikibooks.org/wiki/Haskell/Category_theory.
[5]An interesting discussion on SML functors can be found at http://cs.stackexchange.com/questions/9769/what-is-the-relation-between-functors-in-sml-and-category-theory; essentially, it treats SML structures as algebras and considers whether these functors are functors in the category theoretic sense. Obviously there is a map between objects (structure to structure), but is there a map between morphisms? One response suggests not, as long as the structures contain higher-order operations, which category theory does not deal with.

## Resources

http://en.wikibooks.org/wiki/Haskell/Laziness
https://wiki.haskell.org/Functional_programming#Purity_and_effects
https://wiki.haskell.org/wikiupload/0/0a/TMR-Issue10.pdf