FP + OOP = Haskell

Emery Berger
emery@cs.utexas.edu
Department of Computer Science
The University of Texas at Austin

December 12, 1991

Abstract

The programming language Haskell adds object-oriented functionality (using a concept known as *type classes*) to a pure functional programming framework. This paper describes these extensions and analyzes its accomplishments as well as some problems.

Contents

1 Introduction

Haskell is the result of an effort undertaken to design a freely-available non-strict, purely functional programming language [3, page v]. The Haskell group decided to incorporate most of the features in wide use in other such languages, like Standard ML[1] and Miranda[1][5].

However, in the design process, it was discovered that a number of fundamental issues dealing with the type system of these languages had been left unresolved. In trying to rectify these, a new abstraction known as *type classes* was created which solves these problems and adds much of the utility of object-orientation to the functional framework.

This paper first presents an overview of the Haskell language and then discusses type classes in detail, including their motivation and implementation, concluding with a discussion of their accomplishments and problems which have appeared with their use.

2 Haskell overview

Haskell is a purely-functional programming language. This means that there are no side effects or imperative features of any kind. The result of evaluating any expression is therefore invariant, which greatly simplifies reasoning about a program's properties.

Haskell uses a lazy, or non-strict, evaluation strategy – no subexpression is evaluated until its value is required. Functions can have defined values even when their arguments are undefined.

Given a definition for a function `cond` as follows:

```
cond b conseq altern | b = conseq
                     |~b = altern
```

The value of `cond True 1 (1/0)` is 1, even though evaluating 1/0 would produce an error.

Another important use of lazy evaluation is infinite data structures. Some Haskell syntax first :

- `[]` is nil, the empty list.
- `:` is an infix cons operator.
- `[a,b,c]` is shorthand for the list `(a:(b:(c:[])))`.
- `[0..5]` is shorthand for the list of elements `[0,1,2,3,4,5]`.
- `[a | a <- xs; f a]` is a *list comprehension*, modeled on Zermelo-Frankel set comprehensions. It can be read as "the list of all a where a is taken in sequence from the list xs and f(a) is true."

Some infinite lists:

```
nats = [0..]    -- all natural numbers
odds = [1,3..]  -- the dotdot syntax handles arithmetic sequences
squares = [n * n | n <- nats]
```

[1]Miranda is a trademark of Research Software Limited.

These structures are treated just like ordinary lists, and can be indexed (e.g., `squares !! 5` returns the fifth square).

Haskell functions may be higher-order, i.e., they may take other functions as arguments. The function `map` defined below takes a function and a list as input, and returns the list of results of applying the function to every item of the input list.

```
map f [] = []
map f (x:xs) = f x : (map f xs)
```

Note the Prolog-like pattern-matching in the function's formal parameters.

Currying (named after the logician Haskell Curry) is a Haskell feature which facilitates the use of higher-order functions. A curried function of n arguments can be called with only m arguments, m < n, resulting in a new function of n-m arguments, with the first m parameters bound. For instance, the function `plus` (`plus x y = x+y`) called as `plus 1` becomes a new unary function which adds one to its argument.

```
map (plus 1) [1,2,3] = [2,3,4]
```

Haskell's types are based on the Hindley-Milner type system[2]. This type system allows static type checking and inference to be performed – the programmer does not need to declare the types of functions or values, since the algorithm is capable of automatically deriving the most general possible type. For instance, the type of `cond` would be inferred as `cond::bool->a->a->a`, where `a` can represent any single type – `map`'s type would be `map::(a->b)->[a]->[b]`. The Hindley-Milner type system implements a kind of polymorphism known as *parametric* polymorphism. This means that functions can be defined over a range of types, performing the same operation for each type.

New types can be added in Haskell via a powerful notion called algebraic datatypes. These have the flavor of Backus-Naur form productions. The name of a new type and its optional type variable parameters are given on the left, and each of its possible expansions, separated by vertical bars, is given on the right. This is powerful enough to represent enumeration types (note - "tags" are in uppercase):

```
data colors = Red | Orange | Yellow | Green | Blue | Violet
```

union types:

```
data boolornum = Boolean bool | Number num
```

and recursive, generic datatypes (which allow user-defined genericity):

```
data tree a = Node a | Tree a (tree a) (tree a)
```

These types can also be statically inferred by the Haskell type system.

3 Type classes

3.1 Motivation

The original impetus behind what became type classes was problems with equality and the definition of arithmetic operators. These problems, which manifest themselves in Standard ML and Miranda, will be examined and their solutions described.

Arithmetic

Standard ML overloads mathematical operators for floating-point numbers and integers, but user-defined functions cannot be overloaded. So, although `3 * 3` and `3.14 * 3.14` are acceptable terms, a function like

```
square x = x * x
```

cannot be used to express `square 3` *and* `square 3.14`.

An apparent easy fix for this would be to allow two overloaded versions of square to be defined with the types `Int->Int` and `Float->Float`. But this is not viable; if the function

```
squares (x,y,z) = (square x, square y, square z)
```

were defined, it would require the creation of eight overloaded versions of `squares` – an exponential growth in code.

Miranda's approach to this problem is to avoid it entirely. Miranda has only one all-inclusive numeric type, called `num`. This type encompasses both floating-point numbers and arbitrary-precision integers, and coerces integers to floating-point "when required" [2] [6, section 11]. One drawback to this approach is that static typing of integer-only operations is not possible.

Equality

In Miranda, equality is fully polymorphic – its type is `a -> a -> Bool`. Equality is automatically defined for new algebraic datatypes as full structural equality. This may be seen to violate the notion of abstraction: for instance, the user might want equality of trees to be defined as equality of lists of leaves. The most serious failing of Miranda's definition of equality, however, is that it is not statically typesafe. If a test for equality is applied to two functions, a runtime error results.

Standard ML's equality, on the other hand, is typesafe, because its application is restricted to types which "admit equality" (so-called "eqtype variables"). Functions and abstract data types are not comparable for equality under this scheme, while built-in types are (like `Int` and `Float`).

3.2 Syntax and semantics

The Haskell designers decided that they wanted the safety of Standard ML's equality, but wanted the user to be able to define equality functions for abstract data types. They also sought to solve the problem of mixing numeric types, rather than sidestepping it as Miranda does. Extending the type system to include type classes was their solution.

A type class is defined by a collection of functions (their names and types) which must be supported by any type within that class. An instance type within this class gives implementation details for these functions (although implementation functions can also be provided within a class declaration – these will be the default implementations unless they are replaced by those of the instance). We can use this framework to solve the problems described with `square` and `squares` above; we will make a type class (called `Num`) which both `Int` and `Float` can belong to (the `a` in `class Num a` is a type variable):

[2] "The two kinds of number, integer and fractional, are both of type 'num', as far as the type-checker is concerned, and can be freely mixed in calculations. There is automatic conversion from integer to fractional when required, but not in the opposite direction."

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

and then declare that types `Int` and `Float` are *instances* of class `Num`:

```
instance Num Int where    -- "Int is an instance of Num"
  (+) = addInt            -- these functions are defined elsewhere
  (*) = mulInt
  (-) = subInt
  negate = negInt

instance Num Float where
  (+) = addFloat
  (*) = mulFloat
  (-) = subFloat
  negate = negFloat
```

Now the type of `square` becomes

```
square :: Num a => a -> a
```

(read, "for all a in class `Num`, `square` has type `a -> a`"), and `squares` will have one type (rather than eight):

```
squares :: (Num a, Num b, Num c) => (a,b,c) -> (a,b,c)
```

Type classes handle the issue of equality nicely, too – types which accept equality can be instances of a class called `Eq`:

```
class Eq a where
  (==) :: a -> a -> Bool
```

A class can be declared so any instance of that class automatically belongs to specified other classes, thus capturing the notion of a class hierarchy. For instance, equality should be defined for any number; the class declaration for `Num` can be modified so that this is the case (in effect, that `Num` is a subclass of `Eq`):

```
class (Eq a) => Num a where
  (+), (*), (-) :: a -> a -> a
  negate :: a -> a
```

3.3 Implementation

Implementation of type classes can be accomplished at compile-time by translating a program containing classes and instances to an equivalent program which does not. As an example, we will detail the translation of the classes and functions used in the `square` example above.

Each class declaration defines a "method dictionary" for that class, which wraps all the methods (class functions) for the class into an algebraic data structure, and defines extractor functions to select the appropriate method. For class `Num`, then, we get the following translation:

```
data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a -> a) (a -> a)

add (NumDict a s m n) = a  -- the first dictionary function is addition
sub (NumDict a s m n) = s  -- and so on
mul (NumDict a s m n) = m
neg (NumDict a s m n) = n
```

For each instance of `Num`, we must return a different version of the dictionary with the appropriate functions for each type.

```
numDInt :: NumD Int
numDInt = NumDict addInt subInt mulInt negInt

numDFloat :: NumD Float
numDFloat = NumDict addFloat subFloat mulFloat negFloat
```

Now we translate `square` and `squares` so that they pass around the dictionaries and access the instance functions via the dictionary extractor functions:

```
square' :: NumD a -> a -> a
square' numDa x = mul numDa x x

squares' :: (NumD a, NumD b, NumD c) -> (a, b, c) -> (a, b, c)
squares' (numDa, numDb, numDc) (x, y, z)
  = (square' numDa x, square' numDb y, square numDc z)
```

Each application of `square` or `squares` in the original program must be translated to pass in the appropriate dictionary. For instance, `square 3` would become `square' numDInt 3` and `square 3.14` would become `square' numDFloat 3.14`. The overhead of looking up functions in the dictionary can be optimized away by the compiler (so `mul numDInt 3 4` becomes `mulInt 3 4`).

3.4 Accomplishments

In addition to solving the problems encountered in Standard ML and Miranda, type classes successfully integrate object-oriented programming with functional programming. In fact, the functional framework actually can be seen to enhance object-oriented programming in these ways:

- The notion of classes is a higher-level abstraction than objects, since it separates the interface from the representation.
- Since there is no concept of internal state, and therefore no slots, multiple inheritance cannot cause slot-name clashes, a serious problem with most object-oriented programming languages. (Note that types can be instances of more than one type class – Haskell prevents function name clashes by enforcing the restriction that two classes in scope at the same time may not share function names.)
- Type checking can be done statically, making message sends typesafe and providing the opportunity for compiling away the overhead of method dispatching.
- Type classes provide a structured means of obtaining *ad hoc* polymorphism, i.e., function overloading (same function name, different code for different types of arguments). Overloaded functions must fit into the class hierarchy rather than being truly *ad hoc* as in C++[4] (these can be declared at any time and are independent of subtyping relationships).

3.5 Problems

The introduction of type classes into the Miranda-like framework has caused some unexpected, and occasionally quite problematic, interactions with other aspects of the language. Some of these problems have been solved by adding restrictions to the language, and others by requiring user annotations. Still others remain unsolved.

3.5.1 Ambiguity

Adding classes to Haskell adds a greater potential for ambiguity to the type system; we may know the class of an object, but not its type. An ambiguous typing is a static error, and can only be resolved by the user explicitly adding type information. For example, in the expression below we return the textual representation of the user input:

```
let x = read "..." in show x -- this is illegal
```

This would be typed as `Text a => String` because of the types of `show` and `read`[3]. Since `a` occurs in the class restrictions but not in the type, the expression is said to be *ambiguously overloaded*; if `Int` and `Bool` are instances of `Text`, then the type checker cannot decide if `x` is supposed to be a `Bool` or an `Int`. By explicitly giving the type of `x`, though, it can be processed:

```
let x = read "..." in show (x::Bool) -- this is OK
```

The expression type now becomes `String`.

The most frequent ambiguities occur in the class `Num`, especially because numeric constants (e.g., 1, 2, 3) are overloaded (of type `Num a => a`). Because such ambiguities are so common, Haskell provides a special mechanism to resolve them, called the *default declaration*. This default declaration simply states which numeric types may be considered default types for items in class `Num`. The "default" default declaration is `default (Int, Double)`, so that the type of `let x = 1` would be `Int`; we try each item in the default declaration in turn, and the first one which does not cause a type error is accepted.

[3] `show :: Text a => a -> String; read :: Text a => String -> a`

3.5.2 Restricting polymorphism

The addition of type classes causes some problems with polymorphism which have led the Haskell group to impose the so-called *monomorphism restriction*. This restriction defines when full polymorphism is not allowed without explicit type signatures, and the type is restricted to one form. The rules are rather subtle and will not be explained in detail here, but the following examples give the flavor of why it was needed and show its advantages and disadvantages.

The monomorphism restriction prevents computations from being unexpectedly repeated to satisfy different overloadings. For instance, given the standard function `genericLength` with the type below:

```
genericLength :: Num a => [b] -> a
```

Now given the below expression:

```
let { len = genericLength xs } in (len, len)
```

One would think that `len` would only be computed once. However, without the monomorphism restriction, it could be computed twice, each time to satisfy a different overloading of `Num`. The monomorphism restriction ensures that `len` will only have one type, and therefore will only be computed once. If the programmer actually wishes to have `len` computed at different overloadings, an explicit type signature can be added which overrides the monomorphism:

```
let { len :: Num a => a; len = genericLength xs } in (len, len)
```

The monomorphism restriction has the added bonus of limiting certain types of ambiguity, but it comes with the cost of some subtle differences in the way Haskell types an expression. For the function

```
f x y = x + y
```

the function `f` will be fully overloaded for class `Num`. However, for the "equivalent" function definition (using Haskell's lambda expression syntax, as $f = \lambda x : (\lambda y : x + y)$)

```
f = \x -> \y -> x + y
```

the function `f` requires an explicit type declaration for it to be overloaded. These curious subtleties are considered undesirable, and for this reason the monomorphism restriction is viewed as a provisional fix. A search for a more satisfying solution to these difficulties is in progress.

3.5.3 Pattern Matching

The pattern-matching feature of Haskell can cause some problems when dealing with type classes. Suppose we have the function `fact` as follows:

```
fact :: Num a => a -> a
fact 0 = 1
fact (n+1) = (n+1) * fact n
```

Haskell will internally translate away the pattern-matching to this function:

```
fact :: Num a => a -> a
fact n = if (n == 0) then 1 else (n * fact (n-1))
```

The addition in the `(n+1)` pattern in the original function is replaced with subtraction on the right-hand side of the translated function.

Let's say that a careless user defines an instance of class `Integral` (`n+k` patterns, as in `fact (n+1)` above, can only be matched with members of this predefined Haskell class), called `DumNum`:

```
instance Integral DumNum where
    (+) = addInt
    (-) = addInt -- instead of subInt
    (*) = mulInt
    negate = negInt
```

The definition of subtraction in `DumNum` is not the inverse of addition, as Haskell assumes in its pattern translation. Evaluation of `fact n`, where n is a `DumNum` and $n > 0$, will diverge rather than returning $n!$ as expected – a subtle bug. The semantics of Haskell's pattern transformation make an implicit assumption which is not guaranteed.

This particular example could be fixed by restricting `n+k` patterns to the (pre-defined) type `Int` rather than allowing its use for any member of the `Integral` class, or even eliminating `n+k` patterns altogether. But problems are actually deeply-rooted in pattern-matching. If equality were redefined to be asymmetric, for instance, to know what the program does, the user must know which of the following translations Haskell actually uses (the order of comparison in the equality differs):

```
fact n = if (n == 0) then 1 else (n * fact (n-1))
fact n = if (0 == n) then 1 else (n * fact (n-1))
```

Preventing the user from defining a function inappropriately, like equality which does not act like an equivalence relation, is undecidable in the general case, so Haskell is stuck with these nasty pitfalls. The very subtle bugs which pattern matching can produce seem to imply that the entire concept needs some rethinking.

4 Conclusion

The addition of type classes to Haskell adds an elegant abstraction mechanism, giving the features of object-oriented programming and clearing up some ugly problems. Although difficulties remain which need to be addressed, Haskell has become a more powerful language than any of its predecessors.

References

[1] R. Harper, R. Milner, and M. Tofte, The definition of Standard ML, version 2. Report ECS-LFCS-88-62, Edinburgh University, Computer Science Dept., 1988.

[2] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29-60, December 1969.

[3] P. Hudak, S. Peyton-Jones, and P. Wadler, editors. Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.1). Yale University, Department of Computer Science Technical Report YALEU/DCS/RR777, August 1991.

[4] B. Stroustrop. *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

[5] D. A. Turner. Miranda : a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Springer-Verlag, Nancy, France, September 1985.

[6] D. A. Turner. Miranda system manual. Miranda version 2, August 1989.

[7] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, January 1989.