Fun with Functional Dependencies

or (Draft) Types as Values in Static Computations in Haskell

Thomas Hallgren

December 29, 2000

Abstract

This paper illustrates how Haskell's type class system can be used to express computations. Since computations on the type level are performed by the type checker, these computations are static (i.e., performed at compile-time), and, since the type system is decidable, they always terminate. Haskell thus provides a means to express static computations, and has a clear distinction between static and dynamic computations.

Instance declarations define predicates over types, or in the case of multi-parameter classes, relations between types. With functional dependencies, multi-parameter classes directly specify functions, and thanks to them you can get the type checker to compute the values of function applications, rather than just checking that the result of an application is what you say it is.

This way of expressing computation gives us the power of a small, first-order functional programming language, with pattern matching and structural recursion. We can easily define things like booleans, natural numbers, lists, and functions over these types. We give some examples of completely static computations, the most elaborate one being an implementation of insertion sort. We also give examples where static and dynamic computations are mixed.

1 Introduction

Concepts, such as *programs*, *programming languages*, *computations*, *values* and *types*, are probably familiar to most readers of this paper. But, to make a long story short, programming languages are used to express computations. Computations manipulate values. Typed programming languages distinguish between *types* and *values*. Types are related to values by a typing relation that says what values *belong* to what types, so one usually think of types as sets of values. Expressions, and other program parts, can be assigned types too, to indicate what kind of values the produce or manipulate. Types can thus be used do document programs (to clarify what kind

of values are involved in a certain part of the program) and to help detect programmer mistakes.

In statically typed languages, the types are not seen as something that take part in computations, but rather something that allows a compiler to check that a program is *type correct* without actually running the program.

Seeing types as a way to organize values, one can ask the question if it would be meaningful to have a similar way to organize types? The answer is yes, and different programming languages have different ways to organize types. Most widely known is probably the way types are organized in class hierarchies in object oriented programming languages.

Haskell [Pet97] also has a class system [WB89] to organize types, originally introduced to allow a systematic treatment of overloading. Haskell classes are not quite like classes in object oriented languages: the relation between types and classes is similar to the relation between values and types, i.e., types can *belong* to classes. For example, the class `Eq` is used to group types that allow their values to be tested for equality and the `Show` class contain types whose values can be converted to strings.

The interesting observation for the following is that in Haskell we have three levels on which things are described. On the ground level we have values. The values belong to types, which form the second level, and the types belong to classes, which form the third level. We thus have two relations, one between values and types and one between types and classes. In the next section, we make some reflections on the similarities and differences between these two relations.

2 Values and types vs types and classes

Haskell has, unsurprisingly, ways to introduce values, types and classes, and to create relations between them.

Values and types are introduced together in data-type declarations. For example, the definition

```
data Bool = False | True
```

simultaneously introduces the values `False` and `True`, the type `Bool` and states that `False` and `True` belong to the type `Bool`.

Classes and their relations to types are introduced in a slightly different way. Classes are introduces by stating their names and parameters and giving the types of the overloaded operations that types belonging to the class should support. As an example, a class for types that support equality could be introduced with the following declaration:

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

Types are declared as belonging to a class, often referred to as being an instance of the class, in separate instance declarations. This means that the definition of what types belong to a particular class is left open, allowing a class to be extended with new instances at arbitrary points in the program. In contrast, data-type definitions are closed.

To declare that booleans can be tested for equality we would give the following declaration:

```
instance Eq Bool where (==) = ...
```

where `...` is a suitable implementation of equality for booleans.

Type definitions can be parameterized. A typical example is the definition of the list type, where the type parameter give the type of the elements of a list:

```
data List a = Nil | Cons a (List a)
```

When parameterized types are declared as instances of classes, it is often useful to make some assumptions about the parameter types. For example, to define how lists are tested for equality, we need to refer to the equality test for the elements of the list. Instance declarations of this kind look like this:

```
instance (Eq a) => Eq (List a) where (==) = ...
```

An instance relation like this can be seen as a computation rule, that given an equality test for an arbitrary type, for example `Bool`, gives us an equality test for lists containing values of that type, for example `List Bool`. As we will see later, this gives us a way to express computations on the type level.

3 Computations

3.1 Dynamic computation

In Haskell, computations are usually expressed as functions from values to values. For example, if we define natural numbers (and an abbreviation for a sample number) as

```
data Nat = Zero | Succ Nat

three = Succ (Succ (Succ Zero))
```

we can define functions that tell if a number is even or odd as follows:

```
even Zero = True
even (Succ n) = odd n
odd Zero = False
odd (Succ n) = even n
```

In an interactive Haskell system, such as Hugs [Jon00a], we can then ask for expressions to be computed:

```
> odd three
True
```

3.2 Static computation

As mentioned earlier, some instance declarations in Haskell can be seen as computation rules. Since Haskell is statically typed, the computations expressed in this way will be static, i.e., performed at compile-time.

3.2.1 Predicates

To define what even and odd numbers are, a Prolog programmer could define the following predicates:

```
even(zero).
even(succ(N)):-odd(N).
odd(succ(N)):-even(N).
```

where names beginning with lowercase letter denote constants, which need not be declared before they are used. We can make a rather direct transcription of this program using Haskell type classes, but we first need to declare the constants involved:

```
data Zero
data Succ n

type Three = Succ (Succ (Succ Zero))   -- Just a sample number.

class Even n
class Odd n
```

Note that the Prolog constants and predicates become types and classes, respectively. (Here, since we are not interested in values, but only types and classes, we have defined data types without any constructors and classes without any overloaded operations.) We then define the predicates using instance declarations:

```
instance           Even Zero
instance Odd n  => Even (Succ n)
instance Even n => Odd (Succ n)
```

The question now is: how do we ask the Haskell system to check if a number is even? The computations are performed by the type checker, and in Hugs, the only way to make the type checker work for us is to ask it to compute the type of an expression, or to check that an expression has a given type. Although the definitions given above are enough to express the desired computation, for practical reasons we have to make a small addition to them:

```
class Even n where isEven :: n
class Odd  n where isOdd  :: n
```

We are now saying that, if n is a type representing an even number, then there is an element of n, which can be referred to by the name `isEven`. The instance declarations can be left unchanged.

We can now ask Hugs to check if a number is even or odd:

```
> :type isEven :: Three
ERROR: Illegal Haskell 98 class constraint in inferred type
*** Expression : isEven
*** Type       : Odd Zero => Three
```

We got a type error because three is not an even number. An interpretation of the last line is that if zero were odd, then three would be even.

```
> :type isOdd :: Three
isOdd :: Three
```

The absence of a type error means that three is an odd number.

3.2.2 Relations

If a Prolog programmer wanted to define a relation corresponding more directly to the functions `even` and `odd` in section 3.1, the result would probably be the following:

```
even(zero,true).
even(succ(N),B):-odd(N,B).
odd(zero,false).
odd(succ(N),B):-even(N,B).
```

Using *multi-parameter classes* [JJM97] we can again make a rather direct Haskell transcription. We start by declaring the constants we haven't used before:

```
data True
data False

class Even n b where even :: n -> b
class Odd  n b where odd  :: n -> b
```

And again, for practical reasons, we have included overloaded operations in the classes, although we are only interested in the types.

The Prolog relations can now be transcribed as:

```
instance              Even Zero     True
instance Odd n b  => Even (Succ n) b
instance              Odd  Zero     False
instance Even n b => Odd  (Succ n) b
```

and we can ask Hugs to check if a number is even or odd:

```
> :type odd (undefined::Three) :: True
odd undefined :: True
> :type odd (undefined::Three) :: False
odd undefined :: Odd (Succ (Succ (Succ Zero))) False => False
> :type even (undefined::Three) :: False
even undefined :: False
```

The queries now look a bit more complicated. In the first example, we asked if `Three` is related to `True` by the relation `Odd`, and Hugs replied that, indeed, that is the case. In the second example, we ask in the same way if three is false, and Hugs says that this would have be the case, if the program had contained an instance declaration like

```
instance Odd (Succ (Succ (Succ Zero))) False
```

but the program doesn't. There is nothing that prevent us from adding such an instance, but then `Even` and `Odd` would no longer correspond to the functions `even` and `odd` in section 3.1. In fact, they would not be functions anymore, but some other kind of relations.

Can we ask Hugs to compute function applications? We can try:

```
> :type odd (undefined::Three)
odd undefined :: Odd (Succ (Succ (Succ Zero))) a => a
```

Hugs' reply means that the result of applying `Odd` to `Three` can be any type a, provided the program contains instance declarations allowing us to derive that `Odd (Succ (Succ (Succ Zero)))` a holds. Hugs does not try to enumerate possible values of a, like a Prolog system would. With the given instance declarations, the only possible value for a is `True`, but since the instance relation is open, it is seen as a premature commitment to say that a must be `True`.

3.2.3 Functions

With the definitions given in the previous section, Hugs has no idea that we intend for `Even` and `Odd` to be functions, rather than arbitrary relations. However, recent work has added the possibility to declare *functional dependencies* between the parameters of a multi-parameter class [Jon00b]. We can redefine `Even` and `Odd` as follows:

```
class Even n b | n -> b where even :: n -> b
class Odd  n b | n -> b where odd :: n -> b
```

This says that the relation `Even` n b is actually a *function* from n to b. This prevents us from at the same time declaring both `Even Zero True` and `Even Zero False`, and allows b to be computed if n is a known number:

```
> :type even (undefined::Three)
even undefined :: False
> :type odd (undefined::Three)
odd undefined :: True
```

Now, having seen that these strange looking definitions actually can be used to compute something, we perhaps feel more motivated to go on and define some more functions on natural numbers. The following dynamic ones,

```
add Zero b = b
add (Succ a) b = Succ (add a b)

mul Zero b = Zero
mul (Succ a) b = add b (mul a b)
```

have the following static counterparts:

```
class Add a b c | a b -> c where add :: a -> b -> c
instance              Add Zero b b
instance Add a b c => Add (Succ a) b (Succ c)

class Mul a b c | a b -> c where mul :: a -> b -> c
instance                           Mul Zero b Zero
instance (Mul a b c,Add b c d) => Mul (Succ a) b d

u=undefined
```

Note that we also introduced `u` as a convenient abbreviation of `undefined`. We can try some static additions and multiplications:

```
> :type add (u::Three) (u::Three)
add u u :: Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
> :type mul (u::Three) (u::Three)
mul u u :: Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ
Zero)))))))
```

Note that the command `:type` asks Hugs to just infer the *type* of an expression, not to compute its value. No ordinary, dynamic Haskell computations are performed in the above examples.

3.3 Mixing static and dynamic computations

We have now seen that Haskell allows us to define dynamic functions (section 3.1), i.e., computations to be performed at run-time, and static functions (section 3.2.3), i.e., computations to be performed at compile-time. Can we mix the two, and define functions that are computed partly at compile-time, partly at run-time? The answer is: yes, definitely. It actually happens all the time, when overloaded functions are used in ordinary Haskell programs. Or, to be more precise, the compiler has the opportunity to perform some computations at compile-time, but can also choose to delay most of the work until run-time [Jon94].

A common example used to illustrate static vs dynamic computations is the power function. The dynamic version could be defined as

```
pow b Zero = one
pow b (Succ n) = mul b (pow b n)
```

and a completely static version could be defined as

```
type One = Succ Zero

class Pow a b c | a b -> c where pow :: a -> b -> c

instance                            Pow a Zero One
instance (Pow a b c,Mul a c d) => Pow a (Succ b) d
```

Using the Haskell type `Int` for the dynamic part of the computation, we can define a version of the power function, where the base is dynamic and the exponent is static, as follows:

```
class Pred a b | a -> b where pred :: a->b
instance Pred (Succ n) n

class Power n where power::Int->n->Int
instance            Power Zero where power _ _ = 1
instance Power n => Power (Succ n) where
   power x n = x*power x (pred n)
```

An example computation is

```
> power 2 (mul (u::Three) (u::Three))
512
```

This simple example might seem a bit pointless in an interactive environment where compile-time and run-time coincide. The computation proceeds roughly as follows:

- The type checker computes nine from three times three.
- The application of `Power` to nine is reduced by the type checker, generating a version of `power` that for a given n computes n^9.
- Finally the dynamic function is applied to 2, and the result 2^9 is computed by the interpreter.

With an optimizing compiler, and the same function is used repeatedly, the possibility to move computations to compile-time could of course give a considerable speed-up.

3.4 A larger example of static computation

In the above sections we have presented a way to express static computations in Haskell, using the class system. We now show that this way of expressing static computations is not limited to the rather simple algorithms we have seen so far. We start with a representation of lists and conclude with an implementation of insertion sort.

First, the constructors of the list type:

```
data Nil = Nil
data Cons x xs = Cons
```

Generating a descending sequence of numbers:

```
class DownFrom n xs | n -> xs where downfrom :: n -> xs
instance DownFrom Zero Nil
instance DownFrom n xs => DownFrom (Succ n) (Cons n xs)
```

Comparing numbers:

```
class Lte a b c | a b -> c where lte :: a -> b -> c
instance Lte Zero b T
instance Lte (Succ n) Zero F
instance Lte a b c => Lte (Succ a) (Succ b) c
```

Insertion sort:

```
class Insert x xs ys | x xs -> ys where insert :: x -> xs -> ys
instance Insert x Nil (Singleton x)
instance (Lte x y b, InsertCons b x y ys) => Insert x (Cons y ys) r

class InsertCons b x1 x2 xs ys | b x1 x2 xs -> ys
instance InsertCons T x1 x2 xs (Cons x1 (Cons x2 xs))
instance Insert x1 xs ys => InsertCons F x1 x2 xs (Cons x2 ys)

class Sort xs ys | xs -> ys where sort :: xs -> ys
instance Sort Nil Nil
instance (Sort xs ys,Insert x ys zs) => Sort (Cons x xs) zs
```

To test the above definition we define

```
l1 = downfrom (u::Three)
```

and make some tests in Hugs:

```
> :type l1
l1 :: Cons (Succ (Succ Zero)) (Cons (Succ Zero) (Cons Zero Nil))

> :type sort l1
sort l1 :: Sort (Cons (Succ (Succ Zero)) (Cons (Succ Zero) (Cons
Zero Nil))) (Cons Zero a) => Cons Zero a

> sort l1
ERROR: Unresolved overloading
*** Type       : (Sort Nil a, Insert Zero a b, Insert (Succ Zero) b
c, Insert (Succ (Succ Zero)) c (Cons Zero d)) => Cons Zero d
*** Expression : sort l1
```

Unfortunately, Hugs' type checker doesn't reduce the types as far as expected. The reason for this is at the time of this writing unknown...

4 Concluding remarks

The particular use of type classes explored in this paper are perhaps of the more esoteric kind, and probably not what they were intended for. But, as many people have already discovered, multi-parameter classes with functional dependencies can be very useful for more conventional programming tasks as well.

Haskell 98 [JHe+99], the most recent version of Haskell, does not include multi-parameter classes and functional dependencies. GHC [ghc00] and Hugs [Jon00a] support these extensions to varying degree, though.

It appears that the limits of what can be done within Haskell-like type systems are yet to be found. Two recent examples of other tricks that seem to stretch the limits are [Oka99] and [Wei00].

References

[ghc00] The Glasgow Haskell Compiler. http://www.haskell.org/ghc/, 2000.

[JHe+99] Simon Peyton Jones, John Hughes, (editors), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Haskell Workshop*, 1997.

[Jon94] Mark P. Jones. Dictionary-free Overloading by Partial Evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.

[Jon00a] Mark P. Jones. Hugs 98. http://www.haskell.org/hugs/, February 2000.

[Jon00b] Mark P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposiumon Programming, ESOP 2000*, number 1782 in LNCS, Berlin, Germany, March 2000. Springer-Verlag.

[Oka99] Chris Okasaki. From Fast Exponentiation to Square Matrices: An Adventure in Types. In *International Conference on Functional Programming*, pages 28–35, Paris, France, September 1999.

[Pet97] J. Peterson. The Haskell Home Page. `http://haskell.org`, 1997.

[WB89] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings 1989 Symposium Principles of Programming Languages*, pages 60–76, Austin, Texas, 1989.

[Wei00] Stephanie Weirich. Type-safe cast. In *International Conference on Functional Programming*, Montréal, Canada, September 2000.