Experience report: Haskell and mathematics

Henning Thielemann
Martin-Luther-Universität Halle-Wittenberg, Germany
henning.thielemann@informatik.uni-halle.de

Abstract

This report describes experiences with doing mathematics using Haskell in the fields of algebra and signal processing. It discusses advantages of several Haskell features and problems e.g. with respect to type classes and implicit contexts, that arise in mathematical applications. We also propose solutions including references to the Numeric Prelude project.

1. Introduction

With respect to functional programming languages and mathematics there is a paradoxical situation: On the one hand, functional languages are mathematically oriented: They allow formulating algorithms in terms of equations rather than commands, their notion of functions is close to the mathematical one (referential transparency) and allows for equational reasoning, functions of higher order like operators in functional analysis can be used. Pure functional languages allow lazy evaluation, which in turn let us work with infinite data structures like real numbers, continued fractions, time series, sequences, power series. [Hug89] On the other hand, functional languages are not very widespread amongst mathematicians. Computational mathematics is dominated by insecure machine oriented languages like Fortran, C, C++ if speed counts, by numerical scripting packages (MatLab, SciLab, Octave, R) if development time counts, and by Computer algebra systems (MuPAD, Axiom, Mathematica, Maple, Derive) for simplifications of integrals and other expressions.

2. Why Haskell is good for doing mathematics

With respect to strong and statically typed languages like Modula and Ada, MatLab users like to object that they do not allow for interactive program development, that is: enter a command, see the result, if the command was useful, copy it into an editor and create the program step by step this way. Indeed interactive modes for compilers of imperative statically typed languages are a challenge and to the best of our knowledge, they do not exist, or are rarely used. Programmers of statically typed languages complain about the missing types in scripting languages and the resulting fragility of large libraries. Here common Haskell compilers like Hugs and GHC fill a gap: You can interactively edit and run commands, copy useful pieces to a text editor, while the type signature gives an idea what the code is about and static type checking ensures that you can refactor the code later in a safe manner.

Haskell can (still) not compete with specialized software in many fields of mathematics, due to the lack of sophisticated libraries. Common computer algebra systems are great in symbolic integration, expression simplification and symbolic solution. Numerical linear algebra packages achieve high accuracy for imprecise number representations at high speed. However Haskell allows applications that are not possible with those specialized packages. Its strength is the lazy evaluation and the flexibility to combine many applications. With implementations of matrices and lazy computable reals you can immediately do matrix computations over computable reals, which is possible with neither common computer algebra systems, nor numerical linear algebra packages. [TT06, Kar04] With an implementation of power series you can immediately work with polyphase matrices as needed in signal processing, you can solve differential equations by solving fixed point equations on lists. [Kar00] The same applies to time series and recursive filters. [Don03, Thi04, Thi07]

Recursive filters for signal processing can be understood as the solution of a differential equation. To illustrate the elegance of Haskell for this kind of problems, here is a simple example for the approximative solution of an ordinary differential equation of the form $y'(x) = f(x, y(x))$ by the explicit Euler method in terms of a time series. The formulation of the problem is almost the solution in Haskell.

```
integrate :: Num a => a -> [a] -> [a]
integrate = scanl (+)

eulerExplicit :: Num a =>
   (a -> a -> a) -> a -> a -> [a]
eulerExplicit f x0 y0 =
   let x  = iterate (1+) x0
       y  = integrate y0 y'
       y' = zipWith f x y
   in  y
```

The solution of a differential equation in terms of a power series is equally simple.

Another important mathematical application are theorem provers. Haskell cannot replace sophisticated proof assistants, but with QuickCheck [CH00] you can quickly do tests on plausibility.

There are also some minor features of Haskell which nevertheless bring Haskell programs close to mathematics: Number literals can be used for every type where it makes sense. This way we have common numeric literals but yet clean type distinction. If numbers with higher precision than that of `Double` are needed, there is no need to extend the language. The same applies to rational numbers, complex numbers, quaternions, residue classes and other number types.

In comparison to C++ the transparent integration of infix operators in Haskell is certainly another small but useful feature. Although the introduction of new infix operators should be considered thoroughly, it is e.g. certainly better to define new multiplication operators for matrix-vector-multiplication than using the one sign `*` for each operation that is loosely associated with multiplication. Nevertheless it is a pity, that in Haskell 98 operator precedences are defined by numbers, rather than relations like "`(*)` binds more tightly than `(+)`". Relative precedence definitions would also allow to define unrelated infix operators as being unrelated.

3. How Haskell becomes better for doing mathematics

3.1 Numeric Prelude

The numerical type class hierarchy is certainly one of the most criticized points of Haskell 98. There are several counterproposals like those of MECHVELIANI [Mec06] and THURSTON [TT06], where we want to focus on the latter one here. The Numeric Prelude project initiated by DYLAN THURSTON provides a more mathematically oriented replacement for the numerical type class hierarchy of Haskell 98. It was also extended by modules for several mathematical objects, which allow to study whether the hierarchy is useful.

3.2 Number literals

As noted, polymorphic number literals are good thing for doing mathematics, but when it comes to Numeric class replacements its design leads to a problem: Since `(2::Integer)` will be expanded to `fromInteger (2::Integer)` there is the danger of getting lost in infinite recursion. If you don't use the standard Prelude and thus have two different versions of `fromInteger`, then there is a conflict. It seems more reasonable for us to have two types of literals: Monomorphic literals for `Integer`, say `#2`, and polymorphic literals for all numeric types, like `2`. This way `myFromInteger #2` could be written in a context, where the globally visible `fromInteger` is not the required one.

3.3 Powers

It is a good thing, that there are different operators for the power in Haskell 98. Although mathematical notation does not distinguish them, strict mathematics need the distinction. E.g. cf. to the discussion whether $\sqrt[3]{-1}$ should be defined or not. It must also be thought about replacing

```
(**) :: Floating a => a -> a -> a
```

by a power function like

```
(^?) :: (PositiveReal a, Exponent b) => a -> b -> a
```

where `b` can be even a matrix type. In Haskell 98 `(**)` is defined for `Float`, `Double`, and their complex counterparts, where the complex power is used only rarely. Actually, we needed a power with both complex basis and complex exponent only once so far, namely for the CAUCHY wavelet $t \mapsto (1-i\cdot k\cdot t)^{(-1/2+\mu_2/k+i\cdot\mu_1)}$, and `(**)` could not be used due to its discontinuities.

3.4 Algebraic structures vs. type classes

In Haskell algebraic structures are represented by type classes. However, those two concepts do not match exactly. Firstly, type classes can not enforce laws on their methods. This feature would be certainly easy to add, if the laws could be formulated as compiler optimization rules or if extended static checking becomes available ([Xu06]). But the common arithmetic laws do not apply to imprecise numbers (Section 3.7) and deferred computations (Section 3.8). Secondly, we lay some more interpretation in operator symbols. E.g. both $(\mathbb{Q} \setminus \{0\}, 1, \cdot)$ and $(\mathbb{Q}, 0, +)$ are groups, but we prefer different symbols for the neutral elements and the group operations. So it arises the question, whether a `Group` type class should use `+`, `*` or a different symbol for the group operation, and whether it should be super class to say the `Ring` type class.

3.5 Implicit contexts

The referential transparency of Haskell makes it necessary to reveal all data dependencies. This is an important thing for doing mathematics, e.g. for equational reasoning, but sometimes it would be nice to hide some information which is the same for a bundle of operations. Computations with residue classes are usually all performed with respect to the same divisor (which can also be a polynomial or a GRÖBNER basis), computations with fixed point numbers are performed with the same precision, computable reals in a positional number representation all have the same basis and addition of finite vectors is only possible for matching dimensions. Imperative languages could hide such informations in global variables but this in turn is error-prone, not thread-safe and makes it difficult to run different computations in different contexts in an interleaved way.

There are several possible solutions, but none of the existing ones is satisfying:

- Objects are defined as reader monads.

  ```
  newtype T context a = Cons (context -> a)
  ```

  Then sharing of results is difficult and equality tests cannot have type `T context a -> T context a -> Bool`.

- The reader monads could provide operations that are adapted to the context.

  ```
  do (zero',(+~),(-~),negate') <- getAdditiveOps
     return (1 +~ 2 -~ 3)
  ```

  This is cumbersome and disallows using functions that rely (in this example) on methods of the `Additive` class (part of Numeric Prelude).

- Phantom types: It would be nice to add context information to the type.

  ```
  newtype T basis a = Cons [a]

  class Basis basis where
     getBasis :: T basis a -> Int

  data Ten = Ten

  instance Basis Ten where
     getBasis _ = 10

  add :: T basis a -> T basis a -> T basis a
  add x y = reallyAdd (getBasis x) x y
  ```

  This is a good solution, if the context has a simple structure, a fixed type and does not depend on IO actions. More sophisticated solutions need type hacks or locally declared type class instances [KS04]. However, the latter one is not implemented anywhere, and not much is known about its interaction with other type extensions.

3.6 Vector Spaces vs. Vectors

For vector computations Numeric Prelude started with a multi-parameter type class `VectorSpace a v` (to be precise, the main class definition is for `Module a v` and `VectorSpace` extends this from rings to fields), where `a` is the scalar type and `v` the associated vector type. This type class is very good for studying the advantages and disadvantages of multi-parameter type classes.

The `VectorSpace` type class cannot use functional dependencies and thus type inference often fails and the user has to attach type hints. If `a` is of the `Field` type class, one usually also expects a `VectorSpace a a` instance, which has to be implemented individually for each type `a`. Then for each algorithm for fields you have to decide, whether to implement that algorithm also for `VectorSpace a v`. Since the general implementation for vector spaces cannot replace the basic implementation over fields, you usually end up implementing the algorithm twice. This was done sometimes in Numeric Prelude, but is obviously unsatisfying.

An advantage of this approach is, that if you have instances for `VectorSpace a [v]` and `VectorSpace a (v0,v1)` then you have automatically instances for `VectorSpace a [(v0,v1)]`. The big drawback is, that the scalar type `a` can hardly be a complex type, like complex numbers. Otherwise you risk overlapping and undecidable instances.

A simpler approach which works with Haskell 98 is a `Vector` type constructor class for type constructors like `[]`. However, then all restrictions on the element type `a` have to be put in the signatures of `Vector` methods. These are sometimes too weak. Imagine polynomials where leading zero coefficients shall be eliminated after addition. This requires an `Eq` instance and adding this to the `Vector` addition method signature, excludes vectors of functions, because functions cannot be of class `Eq`. You have also to duplicate the addition and subtraction methods from the `Additive` class in the `Vector` class.

3.7 Imprecise number types vs. exact number types

In the Haskell 98 type class hierarchy and also in the current Numeric Prelude, the same type classes are used for both precise types like `Rational` and their approximate counterparts like `Double`. However numerical algorithms need optimization for accuracy and speed, while these optimizations are often not possible for precise types. There are complications even for simple types: For the division of complex numbers, you have to compute the expression $\sqrt{a^2 + b^2}$, which can cause an intermediate overflow, say for $a = 10^{300}$ and $b = 10^{300}$ for `Double` values, although the result can be presented as `Double`. It is simple to avoid the problem for floating point types, but it is not simple for general types. Another example are determinants, which are usually implemented using matrix factorizations for floating point types (cubic run-time), but need asymptotically slower algorithms for rings (fourth power run-time, [Rot01]).

In Numeric Prelude we solved such problems by new type classes which work around these problems. This means that for declaring one instance (say `Field` on complex numbers) you need to instantiate helper type classes, whose implementations essentially choose an appropriate default implementation. A radical approach would be the maintenance of two type hierarchies: One for imprecise number types and one for exact number types, which guarantee some algebraic laws.

3.8 Immediate computations vs. computation planning

Embedded domain specific languages are a very popular application of Haskell and it is very common to map numerical operators to their counterparts in the wrapped language, see for instance MetaPost [Hob92] (wrapper functionalMetaPost [Kor98]), CSound [Ver] (wrapper Haskore [HMGW96]), SuperCollider [McC96] (wrapper HSC [Dra]). However, this way computations are not performed immediately but are deferred to the wrapped language. This way, sub-expressions cannot be shared using `let`, the wrapped language may introduce side-effects and fundamental laws do not hold. E.g. the Haskell expressions `a+b` and `b+a` don't denote the same, because the generated foreign expressions are not the same. One might argue, that they *mean* the same, however, writing an equality test for these types means implementing a full blown computer algebra system, and in the whole the problem is undecidable. Cf. to the missing `(==)` definition in the situation of implicit contexts in Section 3.5.

In EDSLs usually new relations and `if` constructs are defined that defer the computation to the wrapped language. Again, the question shall be raised, whether the mentioned problems justify another parallel type class hierarchy.

3.9 Typesetting Haskell programs as mathematical formulas

It would be very nice to use Haskell as the language in LaTeX for type-setting formulas. It would then be possible to actually use the mathematics described in a paper (if the source code is available) and it would be possible to track simple mistakes by Haskell's syntax parser, type checker and eventually QuickCheck. Remember, that LaTeX even does not check for matching parentheses.

The preprocessor lhs2TeX actually already supports typesetting Haskell code in math mode. However the capabilities of converting prefix functions into infix operators and vice-versa including automatic insertion of parentheses and automatic layout are very limited.

3.10 Subtypes

The Pascal family of languages and Ada support subranges, theorem provers support even more general subtypes. This would be great to have in Haskell in order to express conditions on the operands, like "divisor must be non-zero", "cardinality is non-negative", "operands of division must be divisible", "vector component index must fit vector dimension". Today we can define distinct types with `newtype` whose constructing functions verify certain conditions. However these conditions cannot be checked statically, and you have to convert numbers of such types frequently, because e.g. `a*b` with `a :: Double` and `b :: PositiveDouble` is not allowed (and it is generally a good thing that multiplication enforces equal operand types in order to let type inference work). Again, extended static checking ([Xu06]) would help here.

4. Conclusion

All problems sketched above should not make us forget that problems in other languages are even worse. Even in more recent imperative languages like Java it is difficult to statically enforce type constraints that can be done with type variables and type classes in Haskell. Or consider C++, which is widely used for mathematics, although you even have to cope with memory management and segmentation faults if you want to test new mathematical ideas.

That is, Haskell performs already good in doing mathematics, but, as always, it could still do better.

References

[CH00] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming, ICFP 2000*, Montreal, Canada, 2000.

[Don03] Matthew Donadio. Haskell DSP. `http://haskelldsp.sourceforge.net/`, 2003.

[Dra] Rohan Drape. HSC3: Haskell interface to Super Collider 3. `http://slavepianos.org/rd/f/207949/`.

[HMGW96] Paul Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3), June 1996.

[Hob92] John D. Hobby. A user's manual for MetaPost. Computing Science Technical Report no. 162, AT&T Bell Laboratories, Murray Hill, New Jersey, 1992.

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[Kar00] Jerzy Karczmarczuk. Lazy processing and optimization of discrete sequences. Technical report, Dept. of Computer Science, University of Caen, France, 2000.

[Kar04] Jerzy Karczmarczuk. Most unreliable technique in the world to compute π. Technical report, Dept. of Computer Science, University of Caen, France, 2004.

[Kor98] Joachim Korittky. functional METAPOST: Eine Beschreibungssprache für Grafiken. Master's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, December 1998.

[KS04] Oleg Kiselyov and Chung-chieh Shan. Functional pearl: Implicit configurations – or, type classes reflect the values of types. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.

[McC96] James McCartney. Super Collider. `http://www.audiosynth.com/`, March 1996.

[Mec06] Serge D. Mechveliani. Haskell and computer algebra: Basic algebra library proposal. Technical report, Programm Systems Institute, Pereslavl-Zalessky, Russia, 2006.

[Rot01] Günter Rote. Division-free algorithms for the determinant and the pfaffian: Algebraic and combinatorial approaches. In H. Alt, editor, *Computational Discrete Mathematics*, LNCS 2122, pages 119–135. Springer-Verlag Berlin Heidelberg, 2001.

[Thi04] Henning Thielemann. Audio processing using Haskell. In Gianpaolo Evangelista and Italo Testa, editors, *DAFx: Conference on Digital Audio Effects*, pages 201–206. Federico II University of Naples, Italy, October 2004.

[Thi07] Henning Thielemann. *Haskell Communities and Activities Report*, chapter Audio processing. http://www.haskell.org/, 12th edition, June 2007.

[TT06] Dylan Thurston and Henning Thielemann. *Haskell Communities and Activities Report*, chapter Numeric Prelude. http://www.haskell.org/, 10. edition, June 2006.

[Ver] Barry Vercoe. CSound. `http://www.bright.net/~dlphilp/linux_csound.html`.

[Xu06] Dana N. Xu. Extended static checking for haskell. In *Haskell Workshop 2006*, Portland, Oregon, USA, September 2006.