Haskell to Pug: Motivations and Syntax

Rusi Mody
June 1995
Revised 13 March 2022

1. Introduction

In its underbelly pug is nearly identical to Mark Jones' gofer. And therefore Haskell. On the visible side however there are marked differences.

Why?

This note is about twiddle-dee and twiddle-dum — at least that is what a self-respecting mathematician would say. But we are not mathematicians, we are computer scientists or should we say rather, computer programmers?

Any discussion on notation would naturally address issues like readability, expressivity, conciseness, etc.. We however give paramount importance to those attempts that close the gap between mathematics and programming. This arises from the belief that mathematics and programming are not disparate activities and members of both parties would immensely benefit from having a common core notation.

In fact, there is a trend in computer science today of searching for simple, general mathematical formalisms unifying diverse branches of mathematics and computer science. Functional programming languages (FPLs) represent one of the best bets that we have for effecting this integrating because they sit in a centrist position between mathematics and CS. This kind of goal is clearly long-term and more educational and cultural than narrowly scientific. In fact the (original@) Haskell report makes a clear commitment to a language that is as much for educational purposes as it is for software engineering.

Thirteen years of teaching programming with FPLs as vehicles – Scheme(88), Miranda TM(89), and gofer(92) has given me data for saying that some changes need to be made to gofer to improve the bandwidth of programming education. The fact that a gofer (or Haskell or ML or Scheme or Clean) programmer has an enormous advantage over one trained with Pascal or C or C++ or some other arcane OO language is not enough. We need to judge on internal considerations also whether we could do better, whether we are not being unnecessarily tripped up by small things.

Some years ago I made some small changes to gofer towards this end. In this note I discuss the issues of four syntactic aspects of gofer – all of which are identical to Haskell – and try to explain the reasons for my changes. The four syntaxes are: function application, function composition, the 'cons' vs the 'has type' operators and concrete types.

2. ':'   :   '::'

ML uses ':' for 'has type' and '::' for 'cons'. Haskell uses the opposite convention presumably because cons is far more frequent in functional programs than declarations. I consider the Haskell choice to be misguided for the following reasons:

The single colon is conventionally used for 'has type' by mathematicians, eg '$f : R \rightarrow R$'. Now when a convention is unhealthy it should certainly be broken but to flout it needlessly seems to be uncalled for. The Haskell violation of the mathematical convention is gratuitous. Even if programs contain few type declarations, discussions **about** programs — ranging from machines that compile to humans that prove theorems — contain a lot more.

The strongest argument against the flipped colon and the double-colon can be seen by analogy with similar misguided choices in C.

In C, && is logical-and whereas & is bitwise-and. Presumably, this was done because in the earliest days when C was used to write Unix, it was expected that uses of bitwise operations would predominate. Clearly this has not been true thereafter and the choice, in hindsight, is a mistake.

Even worse is the case of = and == in C. Again, one may presume that the choice of these two operators is based on statistical considerations: assignment occurs far more frequently than equality. And again, in hindsight the choice is clearly a mistake. Not only does the use of = for assignment confuse students– the y actually think it is equality – but even experienced software developers sometimes slip up: they forget the second = in == in a condition like `if (x = 1) {` and then ...

Languages should be designed on primarily psychological considerations and not on blind statistics.

3. Function Application

Euler is the one who introduced the modern notation for function application − $f(x)$. Most of today's functional languages such as Haskell and ML don't require the parenthesis, i.e. $f\ x$ is the norm though of course $f(x)$ is allowed, but then so is $(f)x$. In short, whereas Euler (and follo wing him, modern mathematicians), denote function application with parenthesis, FPLs denote it by nothing i.e. juxtaposition. Dijkstra recognising that application operates on a function and an argument to produce a result, explicitly shows it as an operator. His notation is$f.x$.

Euler's notation is ambiguous about the structure of $(f)(p)(q)$. Is it $((f)(p))(q)$ or $(f)((p)(q))$? Hence disambiguating parenthesis are required. FPLs and Dijkstra posit application to be left-associative so that currying becomes natural.

3.1 Criticism of the Eulerian Notation

1. Of all the notations $f(x)$ is the longest.
2. Experience with Lisp indicates that excessive parenthesis causes problems for human parsers — if not for automatic ones. If the parenthesis in $f(x)$ seem to be innocuous, look at the definition of B (function composition in λ calculus) in the Eulerian form:

   $$((B(f))(g))(x) = f(g(x))$$

   and then in the modern functional form:

   $$B\ f\ g\ x = f\ (g\ x)$$

   We see here that the Eulerian notation is unsuitable for conducting higher-order business and we get a clue why mathematicians typically find higher-order concepts difficult — the problem is with their notation not the concepts.[Whorf], [Wittgenstein]
3. This use of parenthesis to denote function application is an overloading of the parentheses 'operator'. If we also allow elision of operators to indicate multiplication − as is common in traditional mathematics − then $x(y)$ is ambiguous because it could be either multiplication or application.
4. This brings us to the main reason to question $f(x)$. Euler rarely needed an ything more complex than a literal 'f' or 'J' or 'sin' etc in the function position although he often needed complex expressions in the argument position. As soon as we start having more complex expressions in the function position such as $(f \circ g)(x)$, we see that we need parenthesis. So in the general case we are likely to find ourselves writing $(f)(x)$. Instead we may as well drop all parenthesis other than the ones used for grouping and we obtain the modern functional language alternative $f\ x$.

3.2 Criticism of the functional language alternative

In a `typewriter font` the difference between the 2-character identifier `fx` and the 2 juxtaposed identifiers `f x` is obvious. However, where between
fx and $f x$ and $f\, x$ and $f\ x$ and $f\quad x$
we stop having a single identifier and begin to have two, is not clear. Mathematicians typically avoid the problem by using one-character names — a solution inconceivable to us programmers.

A more serious problem is that it blocks mathematical unification. The FP alternative amounts to elision of a symbol. Eliding a symbol means writing less. But in different notations the elision denotes different things. For example $x\ y$ could mean:

1. Multiplication (in school algebra)
2. Application of an arbitrary operation (in Group Theory)
3. Application of the multiplicative operation (in Ring Theory)
4. List concatenation (in awk)
5. Apply function x to argument y (in modern functional languages)

Perhaps sometimes we need to sit back and look at all the different theories we use and ask ourselves: Are we paying too high a global price at the altar of internal convenience?

3.3 The Benefits of the explicit Dot

Below I discuss some of the points of view – in bottom-up order of significance – from which the application dot (henceforth the Dijkstra-dot) is desirable.

1. Typing and Typesetting
The amount of typing required for $f\ x$ and $f.x$ is identical and so is the space tak en, and both are better than $f(x)$ on both counts. As far as readability is concerned the FP alternative must rate lower than Dijkstra's because space gets a context sensitive meaning, or equivalently, the semantics of space is overloaded – sometimes denoting nothing other than layout and sometimes denoting the very significant operation of application.

2. Pedagogy
The above problem is not merely academic but pedagogical as well. Students find it hard to understand that the space between f and x in $f\ x$ is not nothing b ut denotes something of great importance. This problem is compounded by the fact that in most other places (eg. surrounding operators, =, also around [] etc) the space is merely a lexeme separator and actually denotes nothing.

I have found that students see $(p\ q)$ as the tuple $(p,\ q)$ with the comma missing! Such confusion w ould not be possible with the Dijkstra dot.

3. First Classness
Functional programming is characterized by all values being first-class. In other words a function is no more or less special than say an integer. So just as integers can be combined using '+', '×' etc., functions can be combined using '.', '∘' etc.. Any irregularity of notational conventions would mean a compromise of first-classness.

4. Category Theory
There has been an increasing interest in applying category theory to computer science. An important category for computer scientists is the Cartesian Closed Category – one in which if a and b are two 'objects' (sets) then so is $a \rightarrow b$ and it comes equipped with an *apply* operator. Being able to write$f.x$ instead of $apply(f, x)$ should make Cartesian Closed Categories easier to handle.

5. Mathematical Progress
The fact that function application could be viewed as an operator itself, is an important but very recent mathematical discovery. The sooner it is incorporated into the mathematical mainstream, the better.

3.4 Implementation Considerations

It may at first sight appear that my suggestion is one that would complicate the Haskell syntax. This is not true. The local modifications I've made to gofer that incorporate the Dijkstra-dot have a reduced grammar. In the original gofer we need the productions

$$appExp \rightarrow appExp\ atomic \mid atomic$$

to describe the application syntax. Now we don't need *appExp* at all because syntactically the dot is just an infix operator like any other.

3.5 Similar Ideas Elsewhere

Others with diverse requirements have seen the need for an application operator. On the theoretical side, Barendregt uses a strange integral-like sign. Peyton-Jones uses @ to denote application nodes in FP graphs. Joosten asserts that many problems in teaching first year students is connected with priority, associativity rules and the placing of parenthesis, especially with the 'invisible' function application operator.

The above shows that an application operator is found necessary by many people.

One should concede here that Lisp and its descendents like Scheme with its *Apply* function has been systematic and consistent for 40 years.

3.6 A possible Haskell objection

Haskell uses a dot for composition thereby precluding Dijkstra's usage. In an ASCII context where $\circ$ is unavailable, I feel that the semicolon is quite appropriate for composition (see next section) thereby freeing the dot for application. A few Haskellers may object that the semicolon itself has a usage in Haskell – it is available as an alternate separator for those who do not use layout. Why not provide assignment, pointers and registers for these antiquated users?

4. Function Composition

Generally function composition is defined as follows:
$(g \circ f).x = g.(f.x)$
We instead use ';' and define it as follows:
$(f; g).x = g.(f.x)$
Although our notation may seem more convoluted, we prefer it because firstly our definition has the obvious intuition of ''and then''. More significantly, the type is
$(a \to b) \to (b \to c) \to (a \to c)$
rather than
$(b \to c) \to (a \to b) \to (a \to c)$
The first is obviously more intuitive than the second.

It is curious that many texts on category theory choose the old order rather than the one recommended here even though many accept that this one is preferable. This makes it necessary to read all diagrams backwards. This is sad because the manifesto of category theory is to abstract away from elements of sets to sets themselves. Yet the category theorists burden themselves with a notation whose only benefit lies in the origin which they seek to forget.

Comment: It was pointed out to me that if $f; g$ is preferable to $g \circ f$ on essentially 'operational' or 'diagrammatic' considerations, then similarly $f.x$ should be replaced with $x.f$ (read as 'gi ve the value x to f').

We don't make this switch-over because we are lazy!

5. Concrete Types

Starting with ML, concrete types have been providing a powerful framework for defining user-defined data types, a framework that subsumes records, unions, enumerations, recursive types and generic types in one elegant setting.

However the syntax of the concrete type declaration in Haskell is unwholesome, resulting in painful confusions for students and unnecessary problems for teachers. I should mention that this is being written after 12 years of using Bird & Wadler notation, a notation that is almost identical to Haskell in this aspect.

To better understand the problem let us try to simulate the psychology of the programmer, in particular the programmer doing functional programming. The key difference between the programmer doing functional programming and one following the imperative or OO styles is that whereas the latter thinks in terms of state, pointers, assignments, objects etc, the former thinks mostly with mathematical values. In philosophical terms the imperative or object-oriented programmer is an empiricist, the functional programmer by comparison has a platonic outlook. When designing a type he starts of with the intention of describing a certain *set of values.* This then refines to the fact that different elements of the type may be differently constituted or *constructed* whence we come to different *constructors* with their respective types.

Ideally, therefore the language should allow the programmer to specify *what* these constructors are whose closure is the type being defined. Haskell instead asks the programmer to specify *how* these constructors look when used. In short it emphasises *syntax* at the cost of *semantics.*

To elucidate the problem let me present my suggested alternative notation for concrete types. This will help us to more easily pinpoint the problems with the Haskell syntax.

5.1 The Alternative Notation

The Haskell declaration:
  data *Tree a = Lf a | Br a (Tree a) (Tree a)*
becomes
  ctype *Tree.a* ***where***
    Lf : a → Tree.a
    Br : a → Tree.a → Tree.a → Tree.a

5.2 Problems

1. Non-Uniform Syntax:
The ***class***, ***instance*** and ***data*** declarations in Haskell are a related trio. However the ***class*** and ***instance*** syntax is uniform but the ***data*** syntax is not. The ***ctype*** syntax has been chosen to be consistent with that of ***instance*** and ***class***.

2. Return Type not Evident:
All the arguments that a constructor takes are present next to the constructor. However the return type is not immediately evident but must be found from the declaration head. This contributes its own share of confusion.

3. Terminology not Suggestive:
I guess that the history behind the keyword ***data*** is that the Haskell committee took the word ***datatype*** from ML and then shortened it. Unfortunately the word ***data*** does not suggest the notion of type. I have experienced this once when I asked students to distinguish between data and type in Haskell. A number of students, completely disregarding the Haskell which they all knew, gave philosophical answers distinguishing values (data) from types.

4. Multiple notions of Application:
This is perhaps the single biggest problem, most of the others being corollary to it. In the example above, there is an application between *Tree* and *a* and another application between *Lf* and *a*. In addition, there are the 2 standard applications, as in *Lf 2* – a simple value-generating constructor use, and in
  f (Lf x) = ...
ie a pattern usage. When constructors introduce **four** new applications in addition to the usual *f x* kind, it should be no surprise that students find it heavy going. To make it clear let me tabulate the 6 kinds of applications that a Haskell programmer must consciously distinguish.

function call	*f x*
constructor as constructor – an rhs value	*Lf 2*
constructor as destructor – an lhs pattern	*Lf x*
type constructor	*Tree a*
class constructor	*Eq a*
constructor definition – in a ***data*** rhs	*Lf a*

Now it should be obvious that imposing so many subtly different applications, is heavy going on the hapless beginner. One could perhaps argue that the first 5 above are applications of some sort. But for the last there is no such excuse. Why must we have these kind of arcane confusions in Haskell wherein clarity and elegance are revered? Is it because Haskell is an *applicative* language that the Haskell programmer must face a barrage of applications?

5. Lessons from C not learnt:
In C, a variable declaration is mnemonic ie the declaration looks like the usage and C is notorious for its difficult declaration syntax. eg the declaration for an array of pointers to functions taking char and returning int is: `int (*a[])(char)`

The Haskell ***data*** follows the same design principle as C – the declaration mirrors the usage. This is unfortunate because although the type world is totally different from the value world, the language introduces needless confusions in this regard.

It should be noted here that although the problem of so many different types of applications is significantly alleviated by making the applications explicit with the Dijkstra-dot, the Scheme alternative is logically simpler and more consistent: everything that exists in the language can be combined uniformly, types simply do not exist.

6. Pseudo Precedence Clashes:
In the example given above, it is seen that *Tree a* needs to be parenthesized in the Haskell original. Otherwise we would get a sub-expression of the form (*Br a Tree*). Clearly this is a manifestation of the above problem: two clashing kinds of application. This problem is not there with the ***ctype*** formulation.

7. Confusion with Grammars:
A typical student being introduced to CS is likely to be exposed to context free grammars either in this course or in some other concurrent course. Now we, as computer scientists, can see that

Loosely: a data declaration comes close to a grammar though it is not quite that.

Precisely: if we replace each type expression in a data declaration by a corresponding non-terminal for the the grammar of expressions of that type, we get a grammar for the data type being defined.

Now as a teacher, I can give the precise statement above and frighten the hapless student, or I can give the loose statement and confuse him. Which shall I choose?

6. Conclusion

I do not like these choices. They are not fundamental or genuine dilemmas; rather they are a consequence of Haskell's design. If I can change Haskell – if necessary locally as I have done for gofer – then I would certainly do that. Standardization must be sacrificed at the altar of clarity and simplicity.

7. Problems Remaining

1. The type constructor for lists '[]' used as [*Int*] while seemingly suggestive is unduly confusing by mixing up the singleton list value containing a: [*a*] with the generic list of a, also [*a*]. I would like to go back to the ML-like *List*.*a* which should be easy enough to do.
2. The tuple constructor has the same problem but is a tougher nut to crack.
3. I would have preferred to use the keyword ***type*** instead of ***ctype*** but since type is already in use for defining type synonyms Ive not disturbed it.
4. The emacs interface is not quite up to the mark
5. What Ive done to gofer should probably be redone for hugs. Honestly though, I have reasons to prefer to gofer to Haskell: My preference, both personally and as a teacher, is for a calculator of Turing-complete power – like APL – and not a software engineering behemoth in the Ada, C++, Java traditions.

8. Appendix

The changes that distinguish pug from standard gofer and Haskell are as follows. Note that one can always switch syntaxes on the fly by using the options `:s +S` or `:s -S`

1. Application must be explicitly shown with a dot (Dijkstra philosophy)
2. This applies to other non first class applications as well viz. type constructors (*Tree.a*) and class constructors (*Eq.a*)
3. *bind* is reversed and called with a double dot (so that the generalization of simple application to monadic bind is apparent)
4. syntax of ***data*** is changed to ***ctype*** – concrete type – so that the syntaxes (is it syntices?) of ***class***, ***instance*** and ***ctype*** are uniform.

5. : and :: are flipped

   To allow for the above changes the following changes are also there

6. enumFrom etc use triple dot instead of double dot
7. composition is named ; and is in left-to right order

   And to make allowances for the above

8. layout is the only way to denote nesting. Use of {} gives the error
   Layout imperative in functional programming

9. Bibliography

1. Henk Barendregt; Lambda Calculus; North Holland
2. Simon L Peyton-Jones; The Implementation of Functional Languages; Prentice Hall 1986
3. R Bird and P Wadler; Introduction to Functional Languages; Prentice Hall
4. S Joosten, Klaas van den Berg, Gerrit van der Hoeven; Teaching Functional Programming to First Year Students; Journal of Functional Programming 3(1), Jan 1993
5. L. Meertens
6. A.J.M. van Gasteren; On the Shape of Mathematical Arguments; Springer Verlag
7. Haskell Report
8. E.W. Dijkstra and Carel Scholten; Predicate Calculus and Program Semantics; Springer Verlag 1990
9. Brian Kernighan and Dennis Ritchie; The C Programming Language Prentice Hall
10. Benjamin Lee Whorf; Language Thought and Reality;
11. L. Wittgenstein; Tractacus Logico Philosophicus;