Pug Haskell syntax differences

Rusi Mody
Changed 13th March 2022

1. Introduction

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

Why?

Some may argue 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 rather, computer programmers. I however believe that just as the mathematics of the last four centuries arose from physics, the mathematics of the future will be shaped by computer programming.

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.

There is a widely dispersed trend in computer science today of searching for simple, general mathematical formalisms unifying diverse branches of mathematics and computer science. Examples are seen in APL [Iver], to the Bird-Meertens formalism, [Bird], [Meer] to specification languages like Z [Spivey]. Unfortunately this trend is blocked by various ad-hoceries of mathematicians that die very hard.

In this note I discuss the notations for function application, function composition, 'cons' versus 'has type' and sections.

2. 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, ML don't require the parenthesis, i.e. fx is the norm though of course $f(x)$ is allowed, but then so is $(f)x$. In short, whereas Euler (and following 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 easy.

2.1 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 notation) in the Eulerian form:

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

   and then in the modern functional form:

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

   When we see the Eulerian notation we realize why mathematicians are so uneasy with higher order functions!
3. This use of parenthesis represents an overloading of parentheses. If we also allow elision of operators to indicate multiplication then $x(y)$ is ambiguous — it could be either multiplication or application.
4. This brings us to the main reason to question $f(x)$. Euler rarely needed anything more complex than f 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 $f\ x$.

2.2 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 $f\ \ x$

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 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 notation 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 Snobol)
5. Apply function x to argument y (in modern functional languages)

The developers of each of these theories find elision internally convenient *but any merger of theories is hampered by mutual inconsistency.*

2.3 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 is desirable.

2.3.1 Typing and Typesetting The amount of typing required for $f\ x$ and $f.x$ is identical and so is the space taken, 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 layout sometimes application.

2.3.2 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 but 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 would not be possible with Dijkstra's dot.

2.3.3 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.

2.3.4 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 $a \to b$ 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.

2.3.5 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.

2.4 Implementation Considerations

It may at first sight appear that my suggestion is one that would complicate the Haskell syntax. This is not true.

Our local modifications to gofer that incorporate the Dijkstra-dot are in fact *reductions* in the grammar. In the original gofer we need the productions

appExp : *appExp atomic*
| *atomic*

to describe the application syntax. Now we don't have appExp at all because syntactically the dot is just an infix operator.

2.5 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. [Joos] claim 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.

2.6 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?

3. Function Composition

Generally function composition is defined as follows:

$$(g \circ f).x = g.(f.x)$$

We however define it as:

$$(f \circ g).x = g.(f.x)$$

Although our notation may seem more convoluted, we prefer it because

1. Our definition has the obvious intuition of "and then".
2. The type is

   $$(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$$

   rather than

   $$(\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

   The first is obviously more intuitive than the second.

It is curious that most modern 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 cardinal 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 its origin which they seek to forget.

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

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

4. Type vs. Cons

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. We consider the Haskell view to be misguided. Even if programs contain few type declarations, discussions **about** programs — ranging from machines that compile to humans that prove theorems — contain a lot more. Here again the notation may produce an unnecessary rift between the programmer and the mathematician.

5. Sections

Binary functions are so common in mathematics that they are typically given a more convenient infix syntax. Examples are '+' for numbers, '++' for lists etc. Sections make the benefits of currying equally available to both the parameters of an infix binary operator. In addition, they also provide a concise way to notate the operator itself as a first-class value. Although they are not strictly necessary, they are indispensable if higher order functions are to become normal parlance.

In the immediate sequel, we temporarily forget about the Dijkstra-dot. That is, we will us the functional language notation for application, $f\ x$ rather than $f.x$

When dealing with an operation such as '+' , if we want to use it as a functional value we could define a function of 2 variables that had identical effect to '+'. i.e.

$$plus\ x\ y = x + y$$

However this name is quite unnecessary. We could use $\lambda xy \bullet x + y$ but this too is excessively verbose when all we want to say is '+'! '+' however is difficult to use because of problems of syntactic ambiguity. Is$f + x$ a call to f with 2 parameters, + and x or is it the sum of f and x? In order to resolve the conflict we decree that $f + x$ is the sum of f and x (even if that is

meaningless). If we want to pass or return the operation '+' itself then we do $f\ (+)\ x$. This brings us to the definition of sections.

Given a binary operation '*', we define the following:

Let $x : a, \quad y : b$

$$
\begin{array}{ll}
(*) & : a \to b \to c \\
(*) & = \lambda x \bullet \lambda y \bullet x * y \\
(x\ *) & : b \to c \\
(x\ *) & = \lambda y \bullet x * y \\
(\ *\ y) & : a \to c \\
(\ *\ y) & = \lambda x \bullet x * y
\end{array}
$$

In short $x * y = (x*)\ y = (\ *\ y)\ x = (*)\ x\ y$
(*) is called a full section of '*', (x*) is called a left-section of '*' by x, (*y) is called a right-section of '*' by y.

The syntactic problems described above are ameliorated by the Dijkstra-dot. If we wanted to add f and x we would write $f + x$ whereas if we wanted to pass two parameters to the curried function f we would write $f.+.x$. This may lead us to conclude that the Dijkstra-dot obviates the need for sections. However consider (+×). Is it a right-section of + or a left section of ×?

We see therefore that we need the Dijkstra-dot. Our defining line becomes
$x * y = (x*).y = (\ *\ y).x = (*).x.y$

When we add the Dijkstra-dot into our notation we see that it is itself an infix binary operator and may hence be sectioned! The following is a cute result of this. $f = (f.) = ((f.).) = \cdots$

We give here some small examples demonstrating the expressive power of sections.

$$
\begin{array}{ll}
\mathit{reciprocal} & = (1\ /\ ) \\
\mathit{halve} & = (/\ 2)
\end{array}
$$

As another example, consider the statement of the commutativity of '+'. This statement can be stated in 3 ways:

1. $\forall x, y : R \bullet\ x + y = y + x$
2. $\forall x : R \bullet\ (x+) = (\ + x)$
3. $\mathit{flip}.(+) = (+)$

These three statements though logically equivalent are structurally distinct. Let us think of the table for '+' as a matrix M. The first statement says that $M_{x,y} = M_{y,x}$. The second says that $Row_x = Column_x$ The third says that$M$ is identical to its transpose.

6. References

[Bird]

[Joos]

[Iver]

[Meer]

[Spivey]