Doting on the Dot

Rustom Mody, Anuradha Laxminarayan

September 11, 2012

Dramatis Personae

Pug: speaks 2 languages. Incapable of doing anything of his own initiative. Only executes orders – system-atically.

Funky: Loves Functional Programming and trusts Pug to be an honest and faithful slave

Wonder-kid: Wonders too much...thinks too little.

Lango: Passionately watches and relishes the interplay of language and psychology – or as he would say *languaging* and *thinking*

Sophy: The great...$(great)^{142.857}$ grand daughter of Socrates.

Wonder-Kid (*walks in crying, pointing towards Pug*): That fellow Hugs! I just asked for a function application and he almost bit me!

Funky (*confused*): I don't understand. What happened?

Wonder-Kid: Well, I just found Hugs was available, so I asked him to check the value of $f\texttt{.}3$ for an f which I had defined in my emacs buffer. And he barked at me:

```
Juxtaposition has no meaning. Use dot.
```

Funky (*with a smile*): Ah! I see... So you too mistook Pug to be Hugs. Well, he is actually Hugs' twin Pug. Like all twin pairs, they look very similar - have the same parent –Mark Jones. But one grew to be a little different as he was raised in a different culture of education. And therefore, there are some small differences in the language he understands, one of which is:

What Hugs knows as:

$$\begin{aligned} map\ f\ [] &= [] \\ map\ f\ (x : xs) &= fx : mapfxs \end{aligned}$$

Pug knows as:

$$\begin{aligned} map.f.[] &= [] \\ map.f.(x :: xs) &= f.x :: map.f.xs \end{aligned}$$

FUNKY: Did you observe the difference Kid?
WONDER-KID: Firstly the : has become a ::.
FUNKY Actually, the roles of the : and :: have been swapped. So that, what Hugs understands when we say:

$$map :: (a- > b)- > [a]- > [b]$$

Pug understands as:

$$map : (a- > b)- > [a]- > [b]$$

WONDER-KID: Also, between a function and its argument he likes a . – is that right?
FUNKY: Right. Nice that you have yourself summarised the use of the . !
WONDER-KID: I can see it but what is its significance?
FUNKY: In old maths we used to say $f(x)$ to mean value of the function f at x. Your friend Hugs knows this as fx. The parenthesis has been so heavily loaded with so many meanings that neither Pug nor Hugs use it for function application. But we are students of Dijkstra who insists on the use of the . between a function and its argument. And not just that, we are also functional programmers at the same time. Remember? application is central to our system. So, we give it a formal place with an explicit operator to denote it and what we write is $f.x$
WONDER-KID: I love Dijkstra. The dot needs to be respected. Lets call it the Dijkstra-Dot!
FUNKY: hmnnn...okay! Dijkstra-Dot it shall be.
SOPHY: What awe for Edsgar! – Nevertheless, a cult in the making!
FUNKY: I object to...

LANGO (*Interrupting:*) Funky, don't take this personally. What she says is true. We understand that Application is the essence of your λ-calculus engine, the need for an explicit notation etc. But we don't quite understand the essential significance of its semantics, why does Dijkstra *insist* on the use of such a notation? Would have been great to hear it from him, in his words. Maybe, we should

Before he completes what he is saying Whooosh... A cyclist whizzes past

FUNKY (*calling out loudly*): Hey! Thats Dijkstra

Everyone runs after him forcing him to dismount

ALL (*together:*) Why did you invent the dot?

DIJKSTRA (*twinkling*): Invent? Did not people use the full-stop earlier?

ALL: (*shouting:*) Awwww...

DIJKSTRA: But I tell you the dot is not the main thing. The main thing is equality. If you've understood one you've understood the other

WONDER-KID : Equality? What's there to equality? I don't understand how anyone can not understand equality. Two things are equal if they are the same!

FUNKY (*uneasy:*) Watch your step Kid. This is a dangerous kid@

DIJKSTRA (*gently*): So you say two things are equal if they are the same. I could ask you what is 'same' but I wont do that. Lets instead take some simple examples.

Are 5 and 2+3 the same?

WONDER-KID: Of course!

DIJKSTRA: Sure? Now consider the insides of Pug (or Hugs or bugs or ...) Would they represent the expressions '5' and '2+3' in the same data-structures?

WONDER-KID: We are talking of math and not the insides of programming language implementations!

DIJKSTRA: I'm sorry, I did not@@@ the context. But could you explain to me a little of these contexts and how they differ?

WONDER-KID: Well if we say that '2' and '3+5' are mathematically the same it means that in any mathematical context they are interchangeable. And when we say that the data-structures or syntax trees or whatever corresponding to '2+3' and '5' are different it means that in some situation in a computer the difference will make it behave differently.@@ reword

DIJKSTRA: Very well and could you now elaborate a little more on 'mathematical context' and 'computer situation?'

WONDER-KID: I guess so... By mathematical context I mean that if we take some expression involving a variable, say x, ...x... and replace x by

2+3 or by 5 it will be the same. Whereas by computer situation I mean that there is (or could be) some piece of code ...x... @@font@@ such that substituting x with '2+3' and with '5' will make it behave differently.

DIJKSTRA: Thank you wonder-kid for the theorem. And in fact that's all we need.

WONDER-KID: With all due respect Professor! I don't know what you are talking about! Where is function application here?

DIJKSTRA: Just give your contexts/situations a name – abstract it.

$$f(x) = ...x...$$

And then what you are saying is that in the maths case

$$x = y => f(x) = f(y)$$

(Think of x = 2+3 and y = 5) Whereas in the programming case:

$$x = y \wedge f(x)! = f(y)$$

FUNKY: I see clearly a relation between the application and the equality. But I dont quite see why we need a dot.

DIJKSTRA: What kind of f are we talking about here. A particular one or any/every one?

FUNKY: Well in the $x = y => f(x) = f(y)$ case it is any and every f And when we negate it we get

$$x = y \wedge (\exists f \bullet f(x)! = f(y))$$

And so wonder-kid's theorem is really:

$$x = y => \forall f \bullet f(x) = f(y)$$

DIJKSTRA: Good! Except that Leibniz noticed that by substituting the identity for f we can even reverse the implication and so we get

$$x = y == \forall f \bullet f(x) = f(y)$$

FUNKY: So wonder-kid's theorem is really Leibniz's theorem!

LANGO (*shivering:*) This is a powerful and amazing statement...

DIJKSTRA: I'm glad you see that Lango. Could you tell us why you find it so?

LANGO: That equality is preserved by function application quite irrespective of what is equal and what is the function.

DIJKSTRA: And isn't it sad that we have – thanks to Roberte@@ Recorde – the equality sign but no one – not even Leibniz – thought of representing the application with an explicit sign.

FUNKY: So equality is preserved by function application

DIJKSTRA: Yes and application preserves equality and thats all we need to know about both of them. And once we have the explicit quantifier we notice that by *interchanging the variables* we get

$$f = g \Rightarrow \forall x \bullet f.x = g.x$$

And Dijkstra jumps on his cycle and pedals off faster than they can all say Extensionality

LANGO: Dijkstra-Dot it is! Having realised the centrality of a notation for $=$, we must likewise appreciate a notation for application.

FUNKY: A notation that makes us conscious of the mathematics as well as how this fellow Pug works.

LANGO: One of those rare times, when the notational need at our reasoning level blends naturally into our need to talk to the machine – this fellow Pug.

SOPHY: (*musing*) In this world of material things, unless you see something, it cannot exist. So you cherish this tool to keep your unconscious lifestyle unconscious?

FUNKY: Ignore the ol' gal and get on to working! Hurry up pug, clean up your mess, collect your garbage and make yourself available for a while. Now, tell me– What is the type of $(.)$?

PUG? $(.) : (\mathtt{a} \to \mathtt{b}) \to \mathtt{a} \to \mathtt{b}$

FUNKY: How about $(.1)$?

PUG? $(.\,\mathtt{1}) : \mathtt{Float}$

FUNKY: Oops! I meant $(.1)$

PUG? $(.\ \mathtt{1}) : (\mathtt{Int} \to \mathtt{b}) \to \mathtt{b}$

FUNKY: And $((+1).)$

PUG? $((+\mathtt{1}).) : \mathtt{Int} \to \mathtt{Int}$

FUNKY: So $(.)$ is almost a regular binary operator. As you have already seen, it can be used in partial application contexts as well. $f = (f.) = ((f.).) = (((f.).).)$

WONDER-KID: What a curious identity!

FUNKY: Not all that curious. Don't feel too much. Think! Just look at the type of $(.)$. Can we see that again Pug?

PUG? $(.) : (\mathtt{a} \to \mathtt{b}) \to \mathtt{a} \to \mathtt{b}$

FUNKY: If we remember that $\to$ in the world of types is right associative, we have $(a \to b) \to (a \to b)$. Now, is that not a familiar type?

WONDER-KID: Familiar... Er

FUNKY: Remember? $id : (a \to b) \to (a \to b)$ Same as $(a \to a)$ with the substitution $a := (a \to b)$

WONDER-KID: Now, now... you need to be careful with that.

FUNKY: Sure, and I'll tell you why – There could be other functions with that type. For example@@@ A type could denote many values... you cannot pick up an unique value by just looking at the type unless its a single value type like () for instance? But we also have the identity, I described above. That should be enough.

LANGO: But there are other real questions about your *operator*. For example: $f.x = 2$ Is the use of `.` here as an operator? Is it the value (`.`)? Is the Dot used on the left of the =, the same as the one on the right?

WONDER-KID: They look like 2 entirely different usages.

LANGO: Our friend Hugs has a similar inconsistency with 2 different meanings to the use of the space-key or *nothing*, doesn't he? Sometimes the space is just a space– nothing – and sometimes a fundamental semantic object – application. Since in both cases, it is syntactically non-existent, nobody notices the inconsistent usages.

SOPHY: How can something which *isn't* be noticed? And that too as inconsistent? Also, whats almost there but not there and is still everywhere!

LANGO: Ol'gal's talking sense. Its everywhere and therefore not noticed as special in this context...

FUNKY: In any case, coming back to definitions Pug, add the following definition from the file buffer

$$f.x = 3$$

And tell me the type of f

PUG? `f : a → Int`

FUNKY: And now lets reload the buffer with the following definition–

$$f + x = 3$$

And after that check whats the type of f?

PUG? `redefinition of +`

FUNKY: Now, we are in trouble...the file could not even be loaded – no question of checking types. Whats the problem here? What is being defined here? Not f or x but $(+)$ of course!

WONDER-KID: I am getting confused. In one case f is defined, in another $(+)$ with the argument variables f and x. How would I know which and when? Why do we not read $f.x = 3$ as a definition of (`.`)?

FUNKY: Think Wonder-kid, think! Every function definition will have a **.** between the function and its argument. If we take it as a definition of **.** how many different definitions of **.** are we producing? Don't you see a problem here?

And how would we define anything else? Even $x + y$ is short for $(+)\mathbf{.}\,x\mathbf{.}\,y$ So thats your exception to the rule: if we have **.** on the left of a definition, then . is not being defined, its just a placeholder. Its definition is internal to Pug because application is central to his calculus. But, if we have another binary operator, say + as above, then it is + that is being defined.

Therefore **.** on the left is indeed special. Quite like :: on the left is different from the one on the right. On the left side it is a destructor and does not denote the value (::), on the other a constructor. Likewise, **.** has a different special meaning on the left. It is visible, but in some sense does not quite exist! Syntax, if you wish.

SOPHY: Ah! so, for Mr. Hugs its not there but its there, and for dear Pug, it is sometimes there when not quite there!

WONDER-KID: If it has no purpose, just throw it out. Why make rules, and then make exceptions to them? This is all so confusing – I don't like it.

FUNKY: Cmon fellow, this is programming. Its all about level-headed thinking. Nothing to do with emotions – likes and dislikes.

SOPHY: (*chuckling*) likes and dislikes eh? Feeling with the mind?

FUNKY: (*unmindful of Sophy*) I'll explain why it should be there...

LANGO: (*interrupting*) Friends, this is getting to far...when too many exceptions begin to appear, its not a good idea to push them beneath the carpet by saying:" It is so because I say so!". It is a call to ponder, to look closely at the languages in question! Firstly Kid, the problem does not go because you throw away the operator. The same problem exists in Hugs as well. When notations become explicit, hidden issues begin to come to the surface and we begin to see (in all senses) what we are doing! Just *sense* this...is this not an issue with Hugs? Are we not having an inconsistency there as well?

Programming is less about thinking and more about understanding our thinking process – we want Pug who cannot think to clone our thought process, don't we? And our thinking is not separate from the language we think in – our thoughts are made up of language – as we give ourselves a vocabulary to represent the variety of thoughts, we can formalise our thinking process more and more...

I'll give Funky the opportunity to complete what he has to say before we dive into a discussion of the many languages– the real cause of this confusion. Sorry, Funky...

FUNKY: Thanks Lango, for that elucidatory interrupt. I like to add some comments that a programmer seeing the mathematics understands. So, for the moment, forget about pug and the files he reads. We have the following mathematical equation:

$$f.x = 3$$

This is not just a definition of f but a mathematical equation.

LANGO: Funky, I suggest that when we are discussing mathematics, you use a different font so that it is not confused with Pug's definitions. Quite like you have been using a different font when you exchange something with Pug. Also I noticed how nicely you use dot **.** in formulas and dot . in sentences – just to avoid confusions. How about writing the maths as:

$$\mathbf{f.x = 3}$$

FUNKY: Sure, thats a good idea. Don't tell me that, with different fonts I am beginning to speak different languages...

WONDER-KID: Hey! that helps. I know when you are talking maths and I know when you are getting Pug-friendly.

LANGO: Thats the power of notation!

FUNKY: And the definition we are interested in, is in fact, the solution of the above equation in f. In other words, we may read it as:

$$\exists \mathbf{f} \bullet \forall \mathbf{x} \bullet (\mathbf{f.x = 3})$$

And the function which we are trying to define is a construction of such an f.

WONDER-KID: One more Dot?

FUNKY: Look carefully! Thats not a **.**, its a fat bullet and is different from a **.** – it is a separator for the quantifier and term in predicates.

So what we have here is a *Specification* of the function. In principle, there could be many such **f**. Let there be 2 such, **f** and **g**. So we have:

$$\begin{array}{lll} & (\forall \mathbf{x} \bullet \mathbf{f}.\mathbf{x} = \mathbf{3}) \wedge (\forall \mathbf{x} \bullet \mathbf{g}.\mathbf{x} = \mathbf{3}) & \\ = & (\forall \mathbf{x} \bullet \mathbf{f}.\mathbf{x} = \mathbf{g}.\mathbf{x}) \wedge (\forall \mathbf{x} \bullet \mathbf{g}.\mathbf{x} = \mathbf{3})) & \text{“}\textit{extensionality}\text{”} \\ = & \mathbf{f} = \mathbf{g} \wedge (\forall \mathbf{x} \bullet \mathbf{g}.\mathbf{x} = \mathbf{3}) & \end{array}$$

So, we could choose any such $\mathbf{f}$ We have:

$$\begin{array}{rcll} & \mathbf{f.x} = \mathbf{3} & & \textit{“extensionality”} \\ = & \boldsymbol{\lambda}\mathbf{x} \to \mathbf{f.x} = \boldsymbol{\lambda}\mathbf{x} \to \mathbf{3} & & \textit{“}\eta\textit{ rule”} \\ = & \mathbf{f} = \boldsymbol{\lambda}\mathbf{x} \to \mathbf{3} & & \end{array}$$

Remember the $\forall$ which has vanished? So what we really have is:

$$\forall \mathbf{x} \bullet \mathbf{f.x} = \mathbf{3}$$
$$\forall \mathbf{x} \bullet \boldsymbol{\lambda}\mathbf{x} \to \mathbf{f.x} = \boldsymbol{\lambda}\mathbf{x} \to \mathbf{3}$$
$$\forall \mathbf{x} \bullet (\mathbf{f} = \boldsymbol{\lambda}\mathbf{x} \to \mathbf{3})$$
$$\mathrm{f} = \forall \mathrm{x} \bullet \lambda \mathrm{x} \to 3$$

LANGO: **Freak out on fonts**! With that strange mix of fonts, are you indicating an intuitive un-mathematical expression in the last one there? Is that a half-mathematical language?

FUNKY: Yes, there's a reason, we need to understand the role of the $\boldsymbol{\lambda}$, and we are just hopping there. Do you know whats un-mathematical about it? Its not a type-correct statement – the term in a quantified predicate should be of a Boolean type, right? And what term do we have there? – $\boldsymbol{\lambda}\mathbf{x} \to \mathbf{3}$. Certainly not a Boolean. So, the construction of f above, is not even *well-formed*.

But, thats a step that one is tempted to take, even mechanically.

SOPHY: Mechanically?

WONDER-KID: But it is anything but mechanical. Its as though the symbols are getting a life of their own!

LANGO: You are right Kid. Maybe, we should distinguish mechanical from *formal* – when the form itself becomes part of the content...

LANGO: Thats right, we took the cue from its form: We have $\mathbf{f}$ free from any $\mathbf{x}$, so, we push the quantifier into the seeming non-boolean term. How else do we get rid of the $\forall$?

LANGO: I see another option, we could notationally push the quantifier into the $\boldsymbol{\lambda}$ notation once and for all... How about that?

FUNKY: Exactly! How come, you thought of it? Thats why the typed $\boldsymbol{\lambda}$, $\boldsymbol{\lambda_t}$ comes in. Let me explain: Given that whenever, we have $\boldsymbol{\lambda}\mathbf{x} \to \mathbf{e}$, we are talking of $\mathbf{x} \in \mathbf{Z}$ or $\mathbf{x} \in \mathbf{N}$ etc. But, thats when we work with sets. However, we are now moving to a mathematics of types...

LANGO: Oh! I see... You are getting into a language framework where Pug is to figure out questions of set membership etc...And you spare him the trouble by inventing a world of types.

FUNKY: So that $x : t$ means, $x \in S$ where the type t corresponds to the set S or is *denoted by* the set S. Right? Also, its easy for Pug to check types based entirely on form and structure alone. Sometimes, its not just a matter of convenience but one of *do-ability*

WONDER-KID: What a lovely idea to keep out a whole lot of unwanted paradoxes and computability issues without hiding them! And it seems that you like your own idea so much that you adapt it back into your language of mathematics?

LANGO: Thats right! See...how languages evolve

FUNKY: Is that how it goes? I just know the what of it, not the how. Getting back to the un-mathematical equation above, we have:

$$\mathtt{f} = \forall \mathtt{x} \bullet \lambda \mathtt{x} \rightarrow 3$$
$$\mathtt{f} = \forall \mathtt{x} \bullet \mathtt{x} : \mathtt{a} \bullet \lambda \mathtt{x} \rightarrow 3$$

And then we push the $\forall$ into the $\boldsymbol{\lambda}$ to get $\boldsymbol{\lambda_t}$ and $\boldsymbol{\lambda_t}$ begins to play the role of the new binding construct. So $\forall$ goes and we have:

$$\mathbf{f} = \boldsymbol{\lambda}_\mathbf{t}(\mathbf{x} : \mathbf{a}) \rightarrow \mathbf{3}$$

$\forall$ and $\boldsymbol{\lambda_t}$ are therefore in relation. And thus we have the typed $\boldsymbol{\lambda}$ calculus. We never work with the untyped $\boldsymbol{\lambda}$ and so we can shorten $\boldsymbol{\lambda_t}$ to $\boldsymbol{\lambda}$ with the full awareness that this is in fact $\boldsymbol{\lambda_t}$.

WONDER-KID: But doesn't Pug always keep information about all types. Why then do we need that explicit type in the above definition? Why this redundancy? I sense some kind of mess here.

LANGO: Can't question that feeling.

SOPHY: Feeling? What a world! Feelings are not approved but sensing is called feeling and it is allowed to drive mathematical notation...

FUNKY: Okay Sophy, you've made your point – one that I feel (!) too.

LANGO: Redundancy in a language means a lot of work to keep checking consistency in the many places that the attribute occurs. Mess – is that what you called it? You can see that in natural languages too: @@example@@

FUNKY: So Kid, since that redundancy costs us in terms of our own thinking load as well as Pug's cross-checking load, we disallow such spurious inconsistent information by dropping the explicit type from the above definition. And all type properties are therefore attributes of our system – Pug. Now the function looks like:

$$f = \boldsymbol{\lambda} x \rightarrow 3$$

WONDER-KID: So now, we speak one language of mathematics with $\forall$ another with $\boldsymbol{\lambda_t}$. And when we speak to Pug, we use $\boldsymbol{\lambda}$?

FUNKY: Lango, our man seems to be getting a hang of your point now!

LANGO: Indeed! But I recommend that we don't use too many more languages here, there are other hidden ones coming up too...why not just remember the type as Pug does, and use his language?

FUNKY: (*chuckling*) Makes my life easy!

LANGO: You may say it jest, but major decisions in programming languages, for that matter in any notation are tightly coupled with ease of use, intuitiveness,ergonomics[1] etc I suggest that when we talk programs, we use Pug's language, otherwise we talk mathematics.

FUNKY: Sounds good!

WONDER-KID: That was brilliant. Thank-you. I am beginning to understand the *relation between* $\forall$ *and* $\boldsymbol{\lambda}$. The $\forall$ disappears giving $\boldsymbol{\lambda}$ the responsibility of being a binding construct.

FUNKY: Thats not all. There's a little more to it. Shall we go on?

WONDER-KID: Please do!

FUNKY: Remember from school, 2 ways of reading the function definition $\mathbf{f(x) = e}$

$$\forall \mathbf{x} \bullet \mathbf{f(x) = e} \quad (\textit{declarative}) \tag{1}$$

$$\text{f takes any x to e} \quad (\textit{imperative}) \tag{2}$$

$\boldsymbol{\lambda}$ represents the second expression and the equation $\mathbf{f} = \boldsymbol{\lambda}\mathbf{x} \rightarrow \mathbf{e}$ becomes a definition or construction. Also good to notice that (**1**) is an implicit equation whose solution is the function of interest $\mathbf{f}$ whereas the $\boldsymbol{\lambda}$-ized form is an explicit construction–one in which the name is decoupled from the value.

WONDER-KID: That seems to be the obvious the relation between $\boldsymbol{\lambda}$ and the **.**?

LANGO: *obvious*? With explicit notations, we need not take recourse to that word. We can get clearer and spell out the relation, *calculate* it, I'm sure. Funky?

FUNKY: Lets *calculate*. Pug as we know, never needs to enumerate the extension of f – Remember what extension is? In mathematics, we sometimes describe a function as a set of $(\mathbf{x}, \mathbf{y})$ pairs of domain and range values. He is not capable of giving us short and sweet answers (even when they exist!) and he is so sincere that he may keep going on for ever computing it. In short, our only reference to function values is as themselves which means *do nothing*, or in the computation of its value at a given argument. So:

[1] And saves the author typesetting headaches

$$
\begin{array}{lll}
 & \mathbf{f} = \boldsymbol{\lambda}\mathbf{x} \to \mathbf{e} & \text{“by extensionality”} \\
= & \forall \mathbf{x} \bullet \mathbf{f}.\mathbf{x} = (\boldsymbol{\lambda}\mathbf{x} \to \mathbf{e}).\mathbf{x} & \text{“}\boldsymbol{\beta}\text{ reduction”} \\
= & \forall \mathbf{x} \bullet \mathbf{f}.\mathbf{x} = \mathbf{e} &
\end{array}
$$

Note that one direction of extensionality causes $\boldsymbol{\lambda}$-fication and the other causes $\boldsymbol{\beta} - \mathbf{reduction}$ or dot-tification. In this sense, $\mathbf{.}$ and $\boldsymbol{\lambda}$ invert each other or *reciprocate*. And if we take it that $\forall$ is always there *implicitly*, we can just drop it (not from our minds) to write the implicit form as $\mathbf{f}.\mathbf{x} = \mathbf{e}$

SOPHY: All that work, just for nothing!

LANGO: We need to be careful here, by *just drop it*, you perhaps mean that you want to use it as a short-form

FUNKY: Well, I have a practical problem here. Pug only understands the world of ascii. $\forall$ is hard to represent in that world. So when I talk to him, $\mathbf{f}.\mathbf{x} = \mathbf{e}$ is the syntactic–shortform suitable to him whereas, I keep remembering that $\forall$ in my mind. From Pug's point of view, we are telling him how to *use* f whereas from our point of view, this is a *Mnemonic style definition* or is a description of its use.

Therefore, strictly speaking, the explicit form is a construction of the value of f. And the implicit form, for us, is a reminder that f is a solution to an equation – and for pug its only a syntactic short form for the explicit form.

WONDER-KID: I notice that you are still not talking Pug's language. I can make out from the funny font that you are using.

FUNKY: Good! but you didn't notice that I slinked in the word *definition* where I was earlier using *equation*. The reason is that = in our world of mathematics does not quite correspond to the = in Pug.

LANGO: Good place for me to point out that Pug understands a language of definitions and one of expressions. What equations you describe in Mathematics would perhaps move in smoothly into Pug's expression language. Right?

FUNKY: Yes. I'll come back to that soon in full detail. We will talk of= == etc But just now, lets just get a sense of the *relation between* $\mathbf{.}$ *and* =.Look at the following equations:

$$f.x = e \tag{3}$$

$$f = \lambda x \to e \tag{4}$$

In the above two above equations, the = are different type. In (3)

$$(=) : b \to b$$

whereas in (4)

$$(=) : (a \to b) \to (a \to b)$$

This in fact clarifies the relation between **.** and =. When the **.** crosses from one side to become a λ on the other, = in the equation is of a different type or raised type.

WONDER-KID: Raised type?

FUNKY: Yes, raised from type a to functions over a. This is *almost* true of Pug's =

In fact, even with Hugs, the raising of = happens. This property remains unnoticed due to the fact which Lango raised: Pug , for that matter even Hugs, speaks 2 languages. One is that of expressions that sit on the right of an =

In fact, expressions can by themselves, be given to Pug for evaluation. Pug, just lets demonstrate. Whats 2+ 3

PUG? `3`

FUNKY: And f **.** x = 3?

PUG? `Syntax error`

FUNKY: So Pug does not understand definitions unless they are given in a file.

LANGO: The expression language is part of the definition language when it comes from a file, but it can be stand-alone too, right?

FUNKY: thats right...so coming back to the = In maths, when we say $\mathbf{x} = \mathbf{y} + \mathbf{3}$, we are free to read it as an equation so that $\mathbf{x}$ may be substituted for $\mathbf{y} + \mathbf{3}$ or $\mathbf{y} + \mathbf{3}$ for $\mathbf{x}$ in an expression which contains free occurrences of $\mathbf{x}$ or $\mathbf{y}$. In the former use, we could read this equation as a definition of $\mathbf{x}$.

However, thats not true in Pug's language. The equational reasoning which we do in mathematics, can be done only *one-way* by him. He can replace an lhs with an rhs, with appropriate substitution of parameters, but not the right with the left. The 2 sides are not quite *equal* for him. So when we write an equation for Pug, he sees it almost as a definition which gives a re-writing rule for replacing an lhs with an rhs – he effects an equat-or for the equat-ion.

LANGO: Thats the real issue with **.** on the left and **.** on the right

FUNKY: Yeah... There cannot be fat things on the left, only single things being *defined*

LANGO: But thats true of mathematics too. If there is a single thing on the left of the equation, like in $\mathbf{x} = \mathbf{y} + \mathbf{3}$, I could view it as a definition of x.

FUNKY: Coming to think of it, its not all that different. Other than the

fact that it is purely syntax, maybe the other difference is that, in mathematics you can view it as an equation if you wish and substitute for either expression – left or right. for eg $\mathbf{z} = (\mathbf{y} + \mathbf{3}) * \mathbf{x}$ could become $\mathbf{z} = \mathbf{x} * \mathbf{x}$or $\mathbf{z} = (\mathbf{y} + \mathbf{3}) * (\mathbf{y} + \mathbf{3})$ based on which direction we want to go.

WONDER-KID: But Pug can go only one way, the $\mathbf{z} = (\mathbf{y} + \mathbf{3}) * (\mathbf{y} + \mathbf{3})$ way, right?

FUNKY: Yes. And the whole world on the left and the = remain as syntax and have no existence after Pug understands whats to be done. Jargon says that it has no *run-time existence* – like Pug is *running* with our commands after he internalises them.

Also called by some people as the world of *values* rather than the world of *syntax*

LANGO: That is clarifying. The world of expressions which populate the right hand side, denote values which exist as entities in Pug's work- environment. The world of definitions has a left side, a right side and a connecting =. The right is the expressions we have talked of, the = is a syntax for connecting the left with the right and the left is syntax to explain to Pug whats happening.

FUNKY: And Pug works with remarkable simplicity – Given an expression to evaluate, he looks around for suitable definitions, each time rewriting left hand sides with right hand sides until no further rewriting is possible.

WONDER-KID: I don't think that Pug is all that simple. Did you not tell me before that you can give him fat breakable values which he compares with the left sides of definitions. I think that its quite a lot of work to *destruct* the value into components based on its structure.

LANGO: The left side is very convenient for my thinking – I can think of how values look –structurally, I think Funky called it Mnemonic style syntax.

FUNKY: Yes, thats what it is for us. What Pug needs to do to break up these values is called Pattern matching.

LANGO: Thats a lot of work for him. Looks like we can have the privilege of simplicity because Pug here puffs away, doing a lot of complex hard work...

WONDER-KID: Funky, why then do you call him declarative? The way he executes commands, I would call him imperative.

FUNKY: Thats my very question!

(*Turning towards Lango with a look of being out of depth*)

Perhaps, I would have put it differently: *Is the Dot declarative or imperative?*

LANGO (*meditating for a while with closed eyes*): The essence of all programming – *languaging*. We describe Pug as an interpreter – sometimes, computer. This makes him out to be different from us – in fact, he is

exactly what we are – language-ers.

SOPHY: We think with language...feel with language...speak with language (karatas@@@...)

LANGO (*continuing as if there was no interruption:*) We have a thinking language – the language of mathematics. Then we have a language to talk to Pug – lets call it Puggish. Pug of course, follows Puggish – the language that we have wired into him. Further, since he has to carry our message to the machine, he talks to the machine – yet another language. And the machine is made up of hardware components that understand a language of signals...

SOPHY (*with a twinkle*): And what about this language we are using now to talk about languages?

LANGO (*smiling*): This cascading sequence of languages is the vertebral column of our thinking organism. It provides a clean separation of concerns: We stay with our understanding which is often declarative, just bending enough to talk to Pug in Puggish and leave it to him to *get it done*.

And we call him an *interpreter* because like we are *language-ing* between our 2 languages, he *language-es* or *imperates* Puggish by translating it to Machine language...

FUNKY: And a special case is the Dot! Like all else, so too with the Dot – Our understanding of $\mathbf{f.x}$ as the value of $\mathbf{f}$ at $\mathbf{x}$ becomes the action $\beta - reduction$ for his machinery. In short:

Pug imperates while we declare-ate

Turning to Sophy with a naughty look

Sophy! Are we languag-ing about *languaging*?

SOPHY:(*chirping*) While we are dote-ing on Pug's dot-ting, my stomach's calling *Its time for lunch!*

WONDER-KID: Yes I am famished, lets go.

LANGO: And while we eat, Wonder-Kid, do wonder: *Its time for lunch* – Is it declarative or imperative?

---

notes: brackets in proofs